## Features

- High Performance, Low Power AVR® 8-Bit Microcontroller
- Advanced RISC Architecture
  - 131 Powerful Instructions – Most Single Clock Cycle Execution
  - 32 × 8 General Purpose Working Registers
  - Fully Static Operation
  - Up to 16MIPS Throughput at 16MHz
  - On-chip 2-cycle Multiplier
- High Endurance Non-volatile Memory Segments
  - 4/8/16K bytes of In-System Self-Programmable Flash Program Memory
  - 256/512/512 bytes EEPROM
  - 512/1K/1K bytes Internal SRAM
  - Write/Erase Cycles: 10,000 Flash/100,000 EEPROM
  - Optional Boot Code Section with Independent Lock Bits
    - In-System Programming by On-chip Boot Program
    - True Read-While-Write Operation
  - Programming Lock for Software Security
- Peripheral Features
  - Two 8-bit Timer/Counters with Separate Prescaler and Compare Mode
  - One 16-bit Timer/Counter with Separate Prescaler, Compare Mode, and Capture Mode
  - Real Time Counter with Separate Oscillator
  - Six PWM Channels
  - 8-channel 10-bit ADC
    - Temperature Measurement
  - Programmable Serial USART
  - Master/Slave SPI Serial Interface
  - Byte-oriented 2-wire Serial Interface (Philips I$^2$C compatible)
  - Programmable Watchdog Timer with Separate On-chip Oscillator
  - On-chip Analog Comparator
  - Interrupt and Wake-up on Pin Change
- Special Microcontroller Features
  - Power-on Reset and Programmable Brown-out Detection
  - Internal Calibrated Oscillator
  - External and Internal Interrupt Sources
  - Six Sleep Modes: Idle, ADC Noise Reduction, Power-save, Power-down, Standby, and Extended Standby
- I/O and Packages
  - 23 Programmable I/O Lines
  - 32-lead TQFP, and 32-pad QFN
- Operating Voltage:
  - 2.7V to 5.5V
- Temperature Range:
  - –40°C to +125°C
- Speed Grade:
  - 0 to 8MHz at 2.7V to 5.5V, 0 to 16MHz at 4.5V to 5.5V
- Power Consumption
  - Active Mode: 1.4mA at 4MHz 3V 25°C
  - Power-down Mode: 0.8µA

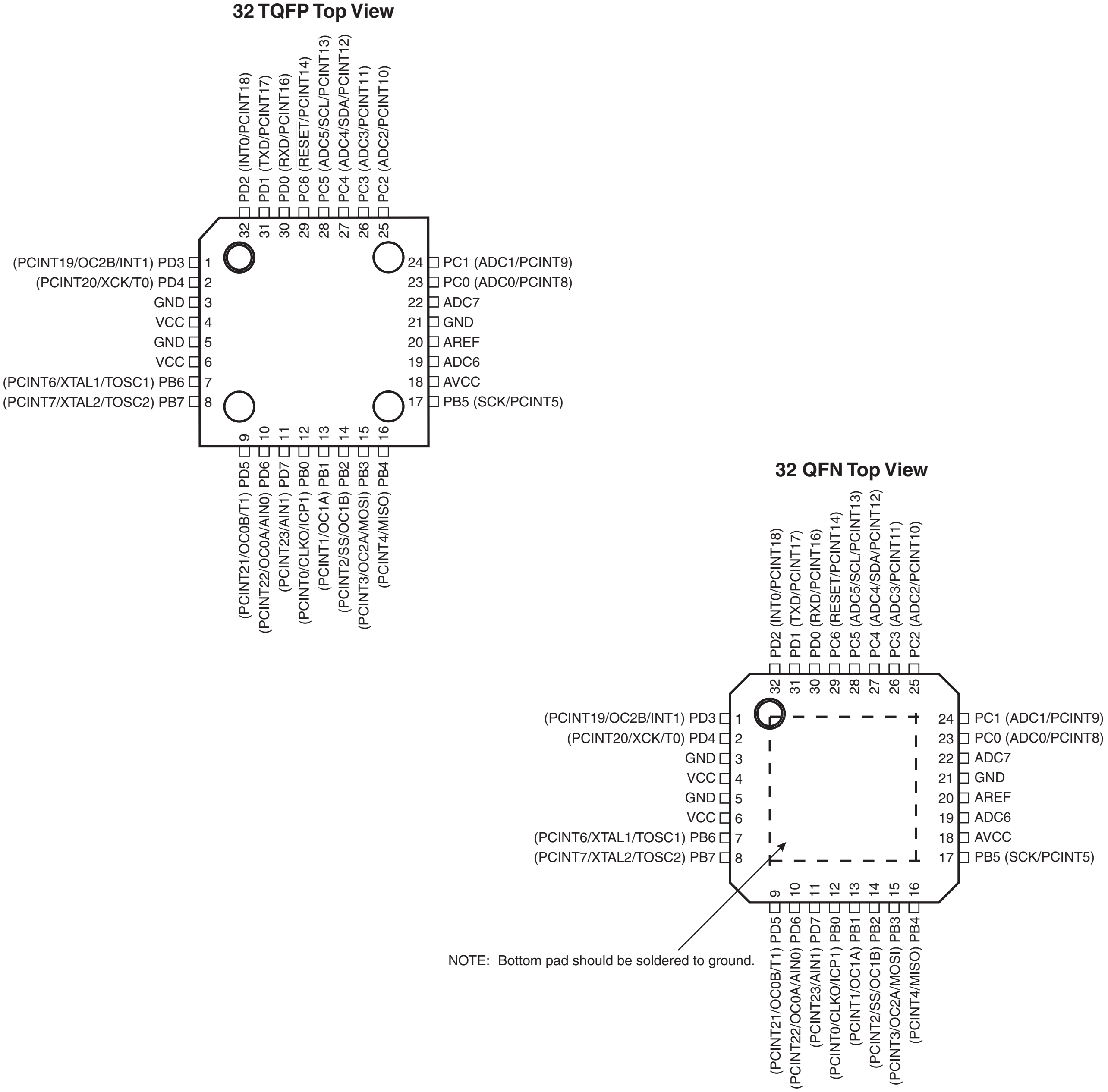
# 8-bit AVR® Microcontroller with 4/8/16K Bytes In-System Programmable Flash

# Atmel ATmega48PA ATmega88PA ATmega168PA

# Automotive

# 1. Pin Configurations

**Figure 1-1.** Pinout Atmel® ATmega48PA/88PA/168PA



32 TQFP Top View

32 QFN Top View

NOTE: Bottom pad should be soldered to ground.

## 1.1 Pin Descriptions

### 1.1.1 VCC

Digital supply voltage.

### 1.1.2 GND

Ground.

### 1.1.3 Port B (PB7:0) XTAL1/XTAL2/TOSC1/TOSC2

Port B is an 8-bit bi-directional I/O port with internal pull-up resistors (selected for each bit). The Port B output buffers have symmetrical drive characteristics with both high sink and source capability. As inputs, Port B pins that are externally pulled low will source current if the pull-up resistors are activated. The Port B pins are tri-stated when a reset condition becomes active, even if the clock is not running.

Depending on the clock selection fuse settings, PB6 can be used as input to the inverting Oscillator amplifier and input to the internal clock operating circuit.

Depending on the clock selection fuse settings, PB7 can be used as output from the inverting Oscillator amplifier.

If the Internal Calibrated RC Oscillator is used as chip clock source, PB7...6 is used as TOSC2...1 input for the Asynchronous Timer/Counter2 if the AS2 bit in ASSR is set.

The various special features of Port B are elaborated in "Alternate Functions of Port B" on page 81 and "System Clock and Clock Options" on page 26.

### 1.1.4 Port C (PC5:0)

Port C is a 7-bit bi-directional I/O port with internal pull-up resistors (selected for each bit). The PC5...0 output buffers have symmetrical drive characteristics with both high sink and source capability. As inputs, Port C pins that are externally pulled low will source current if the pull-up resistors are activated. The Port C pins are tri-stated when a reset condition becomes active, even if the clock is not running.

### 1.1.5 PC6/$\overline{\text{RESET}}$

If the RSTDISBL Fuse is programmed, PC6 is used as an I/O pin. Note that the electrical characteristics of PC6 differ from those of the other pins of Port C.

If the RSTDISBL Fuse is unprogrammed, PC6 is used as a Reset input. A low level on this pin for longer than the minimum pulse length will generate a Reset, even if the clock is not running. The minimum pulse length is given in Table 29-5 on page 317. Shorter pulses are not guaranteed to generate a Reset.

The various special features of Port C are elaborated in "Alternate Functions of Port C" on page 85.

### 1.1.6 Port D (PD7:0)

Port D is an 8-bit bi-directional I/O port with internal pull-up resistors (selected for each bit). The Port D output buffers have symmetrical drive characteristics with both high sink and source capability. As inputs, Port D pins that are externally pulled low will source current if the pull-up resistors are activated. The Port D pins are tri-stated when a reset condition becomes active, even if the clock is not running.

The various special features of Port D are elaborated in "Alternate Functions of Port D" on page 88.

### 1.1.7 AV$_{CC}$

AV$_{CC}$ is the supply voltage pin for the A/D Converter, PC3:0, and ADC7:6. It should be externally connected to V$_{CC}$, even if the ADC is not used. If the ADC is used, it should be connected to V$_{CC}$ through a low-pass filter. Note that PC6...4 use digital supply voltage, V$_{CC}$.

### 1.1.8 AREF

AREF is the analog reference pin for the A/D Converter.

### 1.1.9 ADC7:6 (TQFP and QFN Package Only)

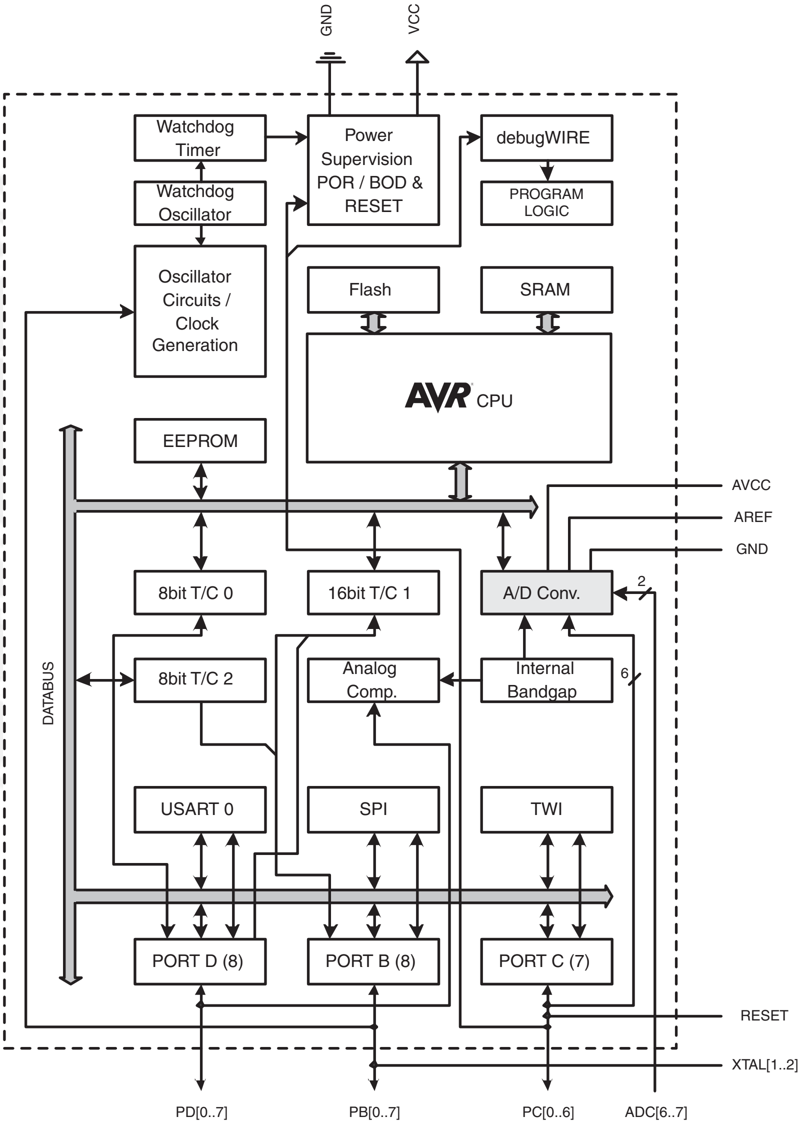
In the TQFP and QFN package, ADC7:6 serve as analog inputs to the A/D converter. These pins are powered from the analog supply and serve as 10-bit ADC channels.

# 2. Overview

The Atmel® ATmega48PA/88PA/168PA is a low-power CMOS 8-bit microcontroller based on the AVR enhanced RISC architecture. By executing powerful instructions in a single clock cycle, the Atmel ATmega48PA/88PA/168PA achieves throughputs approaching 1 MIPS perMHz allowing the system designer to optimize power consumption versus processing speed.

## 2.1 Block Diagram

**Figure 2-1.** Block Diagram



The AVR core combines a rich instruction set with 32 general purpose working registers. All the 32 registers are directly connected to the Arithmetic Logic Unit (ALU), allowing two independent registers to be accessed in one single instruction executed in one clock cycle. The resulting architecture is more code efficient while achieving throughputs up to ten times faster than conventional CISC microcontrollers.

The Atmel® ATmega48PA/88PA/168PA provides the following features: 4K/8K bytes of In-System Programmable Flash with Read-While-Write capabilities, 256/512/512 bytes EEPROM, 512/1K/1K bytes SRAM, 23 general purpose I/O lines, 32 general purpose working registers, three flexible Timer/Counters with compare modes, internal and external interrupts, a serial programmable USART, a byte-oriented 2-wire Serial Interface, an SPI serial port, a 8-channel 10-bit ADC, a programmable Watchdog Timer with internal Oscillator, and five software selectable power saving modes. The Idle mode stops the CPU while allowing the SRAM, Timer/Counters, USART, 2-wire Serial Interface, SPI port, and interrupt system to continue functioning. The Power-down mode saves the register contents but freezes the Oscillator, disabling all other chip functions until the next interrupt or hardware reset. In Power-save mode, the asynchronous timer continues to run, allowing the user to maintain a timer base while the rest of the device is sleeping. The ADC Noise Reduction mode stops the CPU and all I/O modules except asynchronous timer and ADC, to minimize switching noise during ADC conversions. In Standby mode, the crystal/resonator Oscillator is running while the rest of the device is sleeping. This allows very fast start-up combined with low power consumption.

The device is manufactured using Atmel's high density non-volatile memory technology. The On-chip ISP Flash allows the program memory to be reprogrammed In-System through an SPI serial interface, by a conventional non-volatile memory programmer, or by an On-chip Boot program running on the AVR core. The Boot program can use any interface to download the application program in the Application Flash memory. Software in the Boot Flash section will continue to run while the Application Flash section is updated, providing true Read-While-Write operation. By combining an 8-bit RISC CPU with In-System Self-Programmable Flash on a monolithic chip, the Atmel ATmega48PA/88PA/168PA is a powerful microcontroller that provides a highly flexible and cost effective solution to many embedded control applications.

The Atmel ATmega48PA/88PA/168PA AVR is supported with a full suite of program and system development tools including: C Compilers, Macro Assemblers, Program Debugger/Simulators, In-Circuit Emulators, and Evaluation kits.

## 2.2 Comparison Between Processors

The Atmel ATmega48PA/88PA/168PA differ only in memory sizes, boot loader support, and interrupt vector sizes. Table 2-1 summarizes the different memory and interrupt vector sizes for the devices.

**Table 2-1.** Memory Size Summary

| Device | Flash | EEPROM | RAM | Interrupt Vector Size |
|---|---|---|---|---|
| Atmel ATmega48PA/ | 4K Bytes | 256 Bytes | 512 Bytes | 1 instruction word/vector |
| Atmel ATmega88PA | 8K Bytes | 512 Bytes | 1K Bytes | 1 instruction word/vector |
| Atmel ATmega168PA | 16K Bytes | 512 Bytes | 1K Bytes | 2 instruction words/vector |

The Atmel ATmega48PA/88PA/168PA support a real Read-While-Write Self-Programming mechanism. There is a separate Boot Loader Section, and the SPM instruction can only execute from there. In the Atmel ATmega48PA there is no Read-While-Write support and no separate Boot Loader Section. The SPM instruction can execute from the entire Flash.

## 3. Automotive Quality Grade

The Atmel® ATmega48PA/88PA/168PA have been developed and manufactured according to the most stringent requirements of the international standard ISO-TS-16949. This data sheet contains limit values extracted from the results of extensive characterization (Temperature and Voltage).

The quality and reliability of the Atmel ATmega48PA/88PA/168PA have been verified during regular product qualification as per AEC-Q100 grade 1 (–40°C to +125°C).

**Table 3-1.** Temperature Grade Identification for Automotive Products

| Temperature (°C) | Temperature Identifier | Comments |
|---|---|---|
| -40; +125 | Z | Full Automotive Temperature Range |

## 4. Resources

A comprehensive set of development tools, application notes and datasheets are available for download on http://www.atmel.com/avr.

## 5. Data Retention

Reliability Qualification results show that the projected data retention failure rate is much less than 1 PPM over 20 years at 85°C.

## 6. About Code Examples

This documentation contains simple code examples that briefly show how to use various parts of the device. These code examples assume that the part specific header file is included before compilation. Be aware that not all C compiler vendors include bit definitions in the header files and interrupt handling in C is compiler dependent. Please confirm with the C compiler documentation for more details.

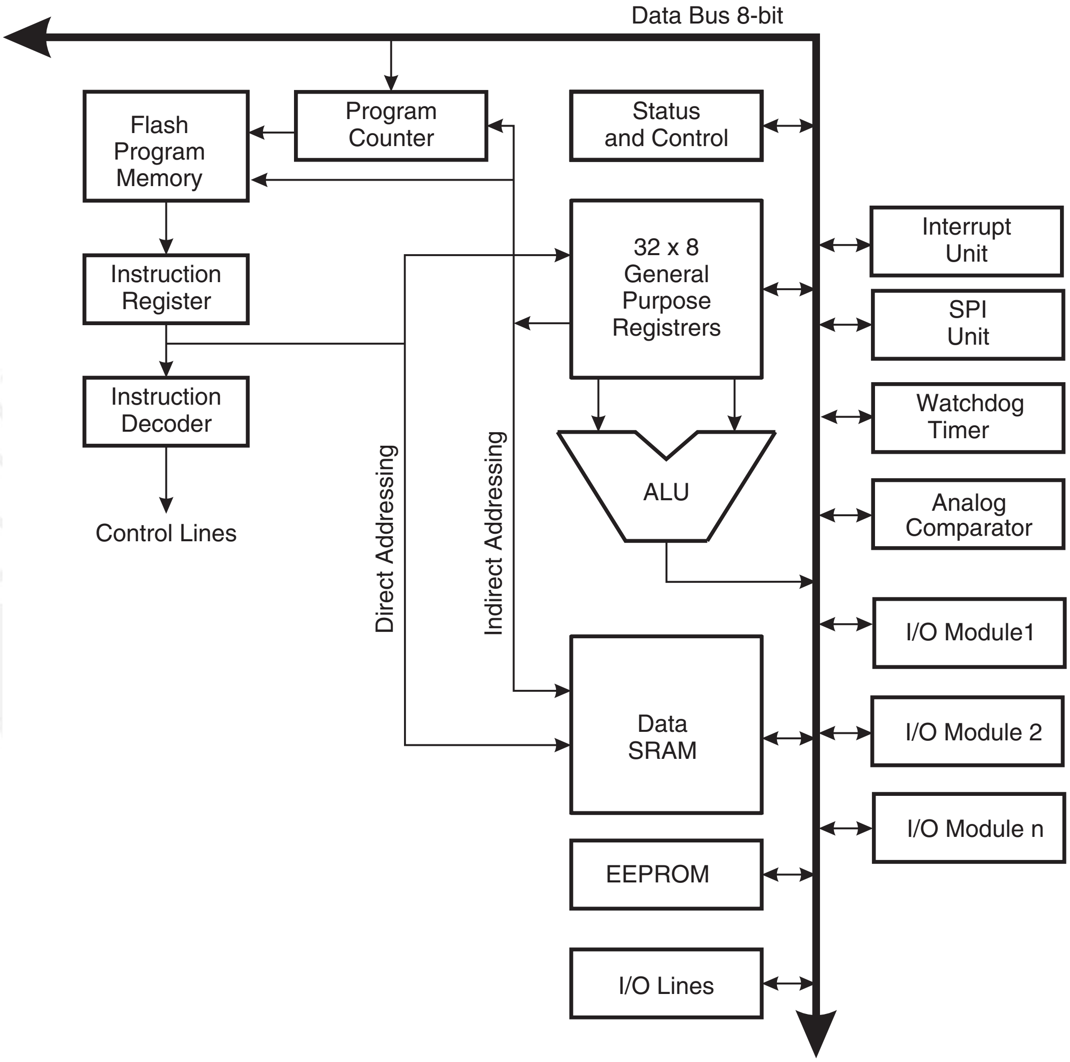
For I/O Registers located in extended I/O map, “IN”, “OUT”, “SBIS”, “SBIC”, “CBI”, and “SBI” instructions must be replaced with instructions that allow access to extended I/O. Typically “LDS” and “STS” combined with “SBRS”, “SBRC”, “SBR”, and “CBR”.

# 7. AVR CPU Core

## 7.1 Overview

This section discusses the AVR core architecture in general. The main function of the CPU core is to ensure correct program execution. The CPU must therefore be able to access memories, perform calculations, control peripherals, and handle interrupts.

**Figure 7-1.** Block Diagram of the AVR Architecture



In order to maximize performance and parallelism, the AVR uses a Harvard architecture – with separate memories and buses for program and data. Instructions in the program memory are executed with a single level pipelining. While one instruction is being executed, the next instruction is pre-fetched from the program memory. This concept enables instructions to be executed in every clock cycle. The program memory is In-System Reprogrammable Flash memory.

The fast-access Register File contains 32 x 8-bit general purpose working registers with a single clock cycle access time. This allows single-cycle Arithmetic Logic Unit (ALU) operation. In a typical ALU operation, two operands are output from the Register File, the operation is executed, and the result is stored back in the Register File – in one clock cycle.

Six of the 32 registers can be used as three 16-bit indirect address register pointers for Data Space addressing – enabling efficient address calculations. One of the these address pointers can also be used as an address pointer for look up tables in Flash program memory. These added function registers are the 16-bit X-, Y-, and Z-register, described later in this section.

The ALU supports arithmetic and logic operations between registers or between a constant and a register. Single register operations can also be executed in the ALU. After an arithmetic operation, the Status Register is updated to reflect information about the result of the operation.

Program flow is provided by conditional and unconditional jump and call instructions, able to directly address the whole address space. Most AVR instructions have a single 16-bit word format. Every program memory address contains a 16- or 32-bit instruction.

Program Flash memory space is divided in two sections, the Boot Program section and the Application Program section. Both sections have dedicated Lock bits for write and read/write protection. The SPM instruction that writes into the Application Flash memory section must reside in the Boot Program section.

During interrupts and subroutine calls, the return address Program Counter (PC) is stored on the Stack. The Stack is effectively allocated in the general data SRAM, and consequently the Stack size is only limited by the total SRAM size and the usage of the SRAM. All user programs must initialize the SP in the Reset routine (before subroutines or interrupts are executed). The Stack Pointer (SP) is read/write accessible in the I/O space. The data SRAM can easily be accessed through the five different addressing modes supported in the AVR architecture.

The memory spaces in the AVR architecture are all linear and regular memory maps.

A flexible interrupt module has its control registers in the I/O space with an additional Global Interrupt Enable bit in the Status Register. All interrupts have a separate Interrupt Vector in the Interrupt Vector table. The interrupts have priority in accordance with their Interrupt Vector position. The lower the Interrupt Vector address, the higher the priority.

The I/O memory space contains 64 addresses for CPU peripheral functions as Control Registers, SPI, and other I/O functions. The I/O Memory can be accessed directly, or as the Data Space locations following those of the Register File, 0x20 - 0x5F. In addition, the Atmel® ATmega48PA/88PA/168PA has Extended I/O space from 0x60 - 0xFF in SRAM where only the ST/STS/STD and LD/LDS/LDD instructions can be used.

## 7.2 ALU – Arithmetic Logic Unit

The high-performance AVR ALU operates in direct connection with all the 32 general purpose working registers. Within a single clock cycle, arithmetic operations between general purpose registers or between a register and an immediate are executed. The ALU operations are divided into three main categories – arithmetic, logical, and bit-functions. Some implementations of the architecture also provide a powerful multiplier supporting both signed/unsigned multiplication and fractional format. See the “Instruction Set” section for a detailed description.

## 7.3 Status Register

The Status Register contains information about the result of the most recently executed arithmetic instruction. This information can be used for altering program flow in order to perform conditional operations. Note that the Status Register is updated after all ALU operations, as specified in the Instruction Set Reference. This will in many cases remove the need for using the dedicated compare instructions, resulting in faster and more compact code.

The Status Register is not automatically stored when entering an interrupt routine and restored when returning from an interrupt. This must be handled by software.

### 7.3.1 SREG – AVR Status Register

The AVR Status Register – SREG – is defined as:

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| 0x3F (0x5F) | **I** | **T** | **H** | **S** | **V** | **N** | **Z** | **C** | **SREG** |
| Read/Write | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

- **Bit 7 – I: Global Interrupt Enable**

The Global Interrupt Enable bit must be set for the interrupts to be enabled. The individual interrupt enable control is then performed in separate control registers. If the Global Interrupt Enable Register is cleared, none of the interrupts are enabled independent of the individual interrupt enable settings. The I-bit is cleared by hardware after an interrupt has occurred, and is set by the RETI instruction to enable subsequent interrupts. The I-bit can also be set and cleared by the application with the SEI and CLI instructions, as described in the instruction set reference.

- **Bit 6 – T: Bit Copy Storage**

The Bit Copy instructions BLD (Bit LoaD) and BST (Bit STore) use the T-bit as source or destination for the operated bit. A bit from a register in the Register File can be copied into T by the BST instruction, and a bit in T can be copied into a bit in a register in the Register File by the BLD instruction.

- **Bit 5 – H: Half Carry Flag**

The Half Carry Flag H indicates a Half Carry in some arithmetic operations. Half Carry Is useful in BCD arithmetic. See the “Instruction Set Description” for detailed information.

- **Bit 4 – S: Sign Bit, S = N ⊕ V**

The S-bit is always an exclusive or between the Negative Flag N and the Two’s Complement Overflow Flag V. See the “Instruction Set Description” for detailed information.

- **Bit 3 – V: Two’s Complement Overflow Flag**

The Two’s Complement Overflow Flag V supports two’s complement arithmetic. See the “Instruction Set Description” for detailed information.

- **Bit 2 – N: Negative Flag**

The Negative Flag N indicates a negative result in an arithmetic or logic operation. See the “Instruction Set Description” for detailed information.

- **Bit 1 – Z: Zero Flag**

The Zero Flag Z indicates a zero result in an arithmetic or logic operation. See the "Instruction Set Description" for detailed information.

- **Bit 0 – C: Carry Flag**

The Carry Flag C indicates a carry in an arithmetic or logic operation. See the "Instruction Set Description" for detailed information.

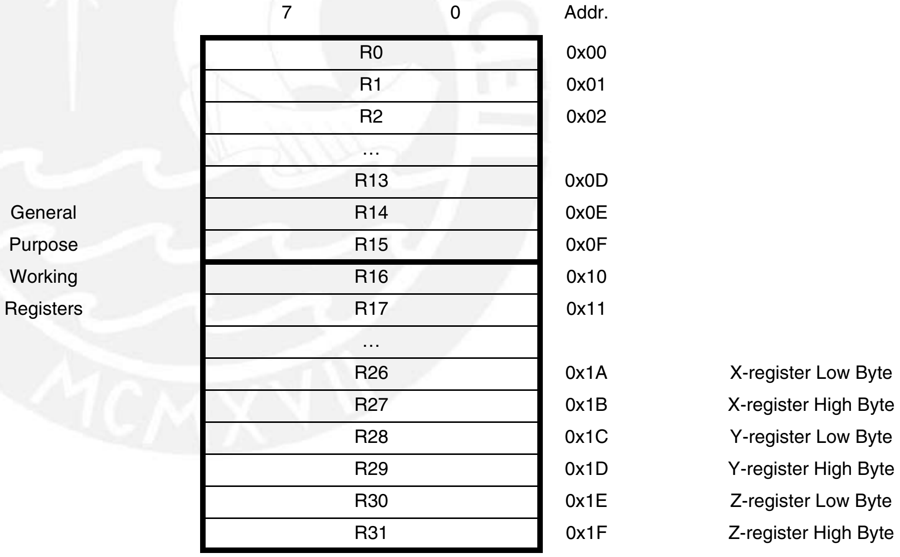
## 7.4 General Purpose Register File

The Register File is optimized for the AVR Enhanced RISC instruction set. In order to achieve the required performance and flexibility, the following input/output schemes are supported by the Register File:

- One 8-bit output operand and one 8-bit result input
- Two 8-bit output operands and one 8-bit result input
- Two 8-bit output operands and one 16-bit result input
- One 16-bit output operand and one 16-bit result input

Figure 7-2 shows the structure of the 32 general purpose working registers in the CPU.

**Figure 7-2.** AVR CPU General Purpose Working Registers



| | 7 0 | Addr. | |
|---|---|---|---|
| | R0 | 0x00 | |
| | R1 | 0x01 | |
| | R2 | 0x02 | |
| | … | | |
| | R13 | 0x0D | |
| General | R14 | 0x0E | |
| Purpose | R15 | 0x0F | |
| Working | R16 | 0x10 | |
| Registers | R17 | 0x11 | |
| | … | | |
| | R26 | 0x1A | X-register Low Byte |
| | R27 | 0x1B | X-register High Byte |
| | R28 | 0x1C | Y-register Low Byte |
| | R29 | 0x1D | Y-register High Byte |
| | R30 | 0x1E | Z-register Low Byte |
| | R31 | 0x1F | Z-register High Byte |

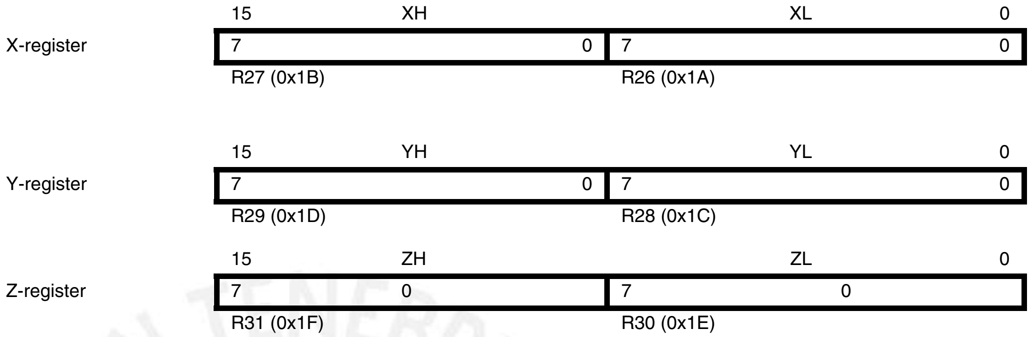
Most of the instructions operating on the Register File have direct access to all registers, and most of them are single cycle instructions.

As shown in Figure 7-2, each register is also assigned a data memory address, mapping them directly into the first 32 locations of the user Data Space. Although not being physically implemented as SRAM locations, this memory organization provides great flexibility in access of the registers, as the X-, Y- and Z-pointer registers can be set to index any register in the file.

### 7.4.1 The X-register, Y-register, and Z-register

The registers R26...R31 have some added functions to their general purpose usage. These registers are 16-bit address pointers for indirect addressing of the data space. The three indirect address registers X, Y, and Z are defined as described in Figure 7-3.

**Figure 7-3.** The X-, Y-, and Z-registers



In the different addressing modes these address registers have functions as fixed displacement, automatic increment, and automatic decrement (see the instruction set reference for details).

## 7.5 Stack Pointer

The Stack is mainly used for storing temporary data, for storing local variables and for storing return addresses after interrupts and subroutine calls. Note that the Stack is implemented as growing from higher to lower memory locations. The Stack Pointer Register always points to the top of the Stack. The Stack Pointer points to the data SRAM Stack area where the Subroutine and Interrupt Stacks are located. A Stack PUSH command will decrease the Stack Pointer.

The Stack in the data SRAM must be defined by the program before any subroutine calls are executed or interrupts are enabled. Initial Stack Pointer value equals the last address of the internal SRAM and the Stack Pointer must be set to point above start of the SRAM, see Table 8-3 on page 18.

See Table 7-1 for Stack Pointer details.

**Table 7-1.** Stack Pointer Instructions

| Instruction | Stack pointer | Description |
|---|---|---|
| PUSH | Decremented by 1 | Data is pushed onto the stack |
| CALL<br>ICALL<br>RCALL | Decremented by 2 | Return address is pushed onto the stack with a subroutine call or interrupt |
| POP | Incremented by 1 | Data is popped from the stack |
| RET<br>RETI | Incremented by 2 | Return address is popped from the stack with return from subroutine or return from interrupt |

The AVR Stack Pointer is implemented as two 8-bit registers in the I/O space. The number of bits actually used is implementation dependent. Note that the data space in some implementations of the AVR architecture is so small that only SPL is needed. In this case, the SPH Register will not be present.

### 7.5.1 SPH and SPL – Stack Pointer High and Stack Pointer Low Register

| Bit | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | |
|---|---|---|---|---|---|---|---|---|---|
| 0x3E (0x5E) | **SP15** | **SP14** | **SP13** | **SP12** | **SP11** | **SP10** | **SP9** | **SP8** | **SPH** |
| 0x3D (0x5D) | **SP7** | **SP6** | **SP5** | **SP4** | **SP3** | **SP2** | **SP1** | **SP0** | **SPL** |
| | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
| Read/Write | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | |
| | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | |
| Initial Value | RAMEND | RAMEND | RAMEND | RAMEND | RAMEND | RAMEND | RAMEND | RAMEND | |
| | RAMEND | RAMEND | RAMEND | RAMEND | RAMEND | RAMEND | RAMEND | RAMEND | |

## 7.6 Instruction Execution Timing

This section describes the general access timing concepts for instruction execution. The AVR CPU is driven by the CPU clock $\text{clk}_{\text{CPU}}$, directly generated from the selected clock source for the chip. No internal clock division is used.

Figure 7-4 shows the parallel instruction fetches and instruction executions enabled by the Harvard architecture and the fast-access Register File concept. This is the basic pipelining concept to obtain up to 1 MIPS perMHz with the corresponding unique results for functions per cost, functions per clocks, and functions per power-unit.

**Figure 7-4.** The Parallel Instruction Fetches and Instruction Executions

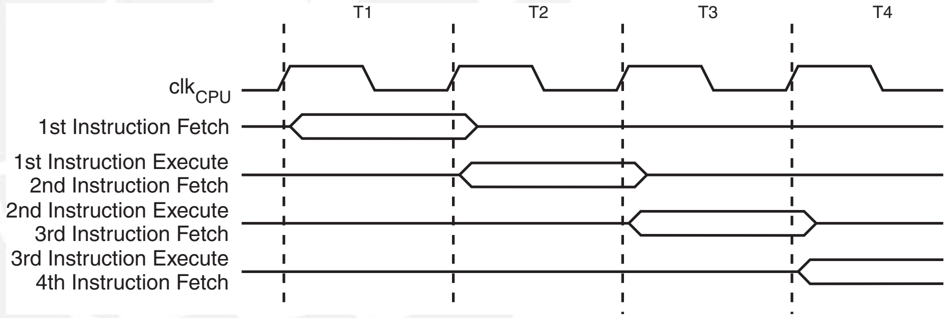
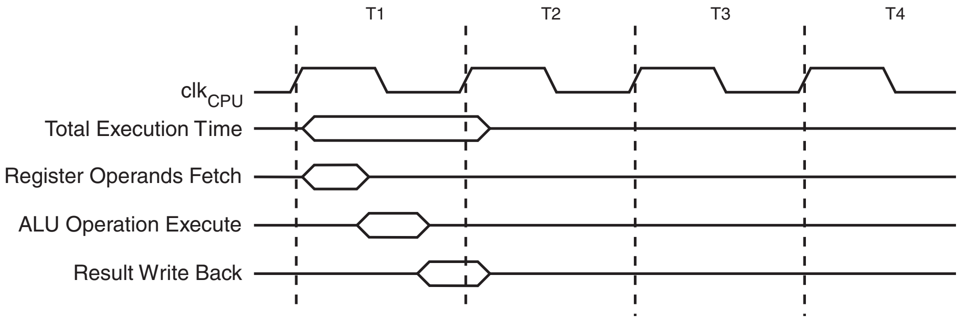


Figure 7-5 shows the internal timing concept for the Register File. In a single clock cycle an ALU operation using two register operands is executed, and the result is stored back to the destination register.

**Figure 7-5.** Single Cycle ALU Operation



## 7.7 Reset and Interrupt Handling

The AVR provides several different interrupt sources. These interrupts and the separate Reset Vector each have a separate program vector in the program memory space. All interrupts are assigned individual enable bits which must be written logic one together with the Global Interrupt Enable bit in the Status Register in order to enable the interrupt. Depending on the Program Counter value, interrupts may be automatically disabled when Boot Lock bits BLB02 or BLB12 are programmed. This feature improves software security. See the section "Memory Programming" on page 294 for details.

The lowest addresses in the program memory space are by default defined as the Reset and Interrupt Vectors. The complete list of vectors is shown in "Interrupts" on page 58. The list also determines the priority levels of the different interrupts. The lower the address the higher is the priority level. RESET has the highest priority, and next is INT0 – the External Interrupt Request 0. The Interrupt Vectors can be moved to the start of the Boot Flash section by setting the IVSEL bit in the MCU Control Register (MCUCR). Refer to "Interrupts" on page 58 for more information. The Reset Vector can also be moved to the start of the Boot Flash section by programming the BOOTRST Fuse, see "Boot Loader Support – Read-While-Write Self-Programming" on page 277.

When an interrupt occurs, the Global Interrupt Enable I-bit is cleared and all interrupts are disabled. The user software can write logic one to the I-bit to enable nested interrupts. All enabled interrupts can then interrupt the current interrupt routine. The I-bit is automatically set when a Return from Interrupt instruction – RETI – is executed.

There are basically two types of interrupts. The first type is triggered by an event that sets the Interrupt Flag. For these interrupts, the Program Counter is vectored to the actual Interrupt Vector in order to execute the interrupt handling routine, and hardware clears the corresponding Interrupt Flag. Interrupt Flags can also be cleared by writing a logic one to the flag bit position(s) to be cleared. If an interrupt condition occurs while the corresponding interrupt enable bit is cleared, the Interrupt Flag will be set and remembered until the interrupt is enabled, or the flag is cleared by software. Similarly, if one or more interrupt conditions occur while the Global Interrupt Enable bit is cleared, the corresponding Interrupt Flag(s) will be set and remembered until the Global Interrupt Enable bit is set, and will then be executed by order of priority.

The second type of interrupts will trigger as long as the interrupt condition is present. These interrupts do not necessarily have Interrupt Flags. If the interrupt condition disappears before the interrupt is enabled, the interrupt will not be triggered.

When the AVR exits from an interrupt, it will always return to the main program and execute one more instruction before any pending interrupt is served.

Note that the Status Register is not automatically stored when entering an interrupt routine, nor restored when returning from an interrupt routine. This must be handled by software.

When using the CLI instruction to disable interrupts, the interrupts will be immediately disabled. No interrupt will be executed after the CLI instruction, even if it occurs simultaneously with the CLI instruction. The following example shows how this can be used to avoid interrupts during the timed EEPROM write sequence.

Assembly Code Example

```
in  r16, SREG     ; store SREG value
cli    ; disable interrupts during timed sequence
sbi EECR, EEMPE   ; start EEPROM write
sbi EECR, EEPE
out SREG, r16     ; restore SREG value (I-bit)
```

C Code Example

```
char cSREG;
cSREG = SREG; /* store SREG value */
/* disable interrupts during timed sequence */
_CLI();
EECR |= (1<<EEMPE); /* start EEPROM write */
EECR |= (1<<EEPE);
SREG = cSREG; /* restore SREG value (I-bit) */
```

When using the SEI instruction to enable interrupts, the instruction following SEI will be executed before any pending interrupts, as shown in this example.

Assembly Code Example

```
sei   ; set Global Interrupt Enable
sleep; enter sleep, waiting for interrupt
; note: will enter sleep before any pending interrupt(s)
```

C Code Example

```
__enable_interrupt(); /* set Global Interrupt Enable */
__sleep(); /* enter sleep, waiting for interrupt */
/* note: will enter sleep before any pending interrupt(s) */
```

### 7.7.1 Interrupt Response Time

The interrupt execution response for all the enabled AVR interrupts is four clock cycles minimum. After four clock cycles the program vector address for the actual interrupt handling routine is executed. During this four clock cycle period, the Program Counter is pushed onto the Stack. The vector is normally a jump to the interrupt routine, and this jump takes three clock cycles. If an interrupt occurs during execution of a multi-cycle instruction, this instruction is completed before the interrupt is served. If an interrupt occurs when the MCU is in sleep mode, the interrupt execution response time is increased by four clock cycles. This increase comes in addition to the start-up time from the selected sleep mode.

A return from an interrupt handling routine takes four clock cycles. During these four clock cycles, the Program Counter (two bytes) is popped back from the Stack, the Stack Pointer is incremented by two, and the I-bit in SREG is set.

# 8. AVR Memories

## 8.1 Overview

This section describes the different memories in the Atmel® ATmega48PA/88PA/168PA. The AVR architecture has two main memory spaces, the Data Memory and the Program Memory space. In addition, the Atmel ATmega48PA/88PA/168PA features an EEPROM Memory for data storage. All three memory spaces are linear and regular.

## 8.2 In-System Reprogrammable Flash Program Memory

The Atmel ATmega48PA/88PA/168PA contains 4/8/16K bytes On-chip In-System Reprogrammable Flash memory for program storage. Since all AVR instructions are 16 or 32 bits wide, the Flash is organized as 2/4/8/16K x 16. For software security, the Flash Program memory space is divided into two sections, Boot Loader Section and Application Program Section in the Atmel ATmega88PA and the Atmel ATmega168PA. See SELFPRGEN description in section "SPMCSR – Store Program Memory Control and Status Register" on page 292 for more details.

The Flash memory has an endurance of at least 10,000 write/erase cycles. The Atmel ATmega48PA/88PA/168PA Program Counter (PC) is 11/12/13/14 bits wide, thus addressing the 2/4/8/16K program memory locations. The operation of Boot Program section and associated Boot Lock bits for software protection are described in detail in "Self-Programming the Flash, Atmel ATmega48PA" on page 269 and "Boot Loader Support – Read-While-Write Self-Programming" on page 277. "Memory Programming" on page 294 contains a detailed description on Flash Programming in SPI- or Parallel Programming mode.

Constant tables can be allocated within the entire program memory address space (see the LPM – Load Program Memory instruction description).

Timing diagrams for instruction fetch and execution are presented in "Instruction Execution Timing" on page 13.

Figure 8-1. Program Memory Map Atmel ATmega48PA

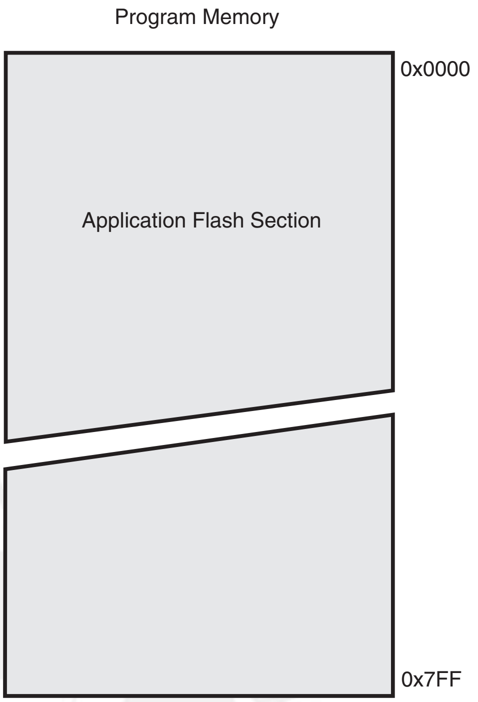
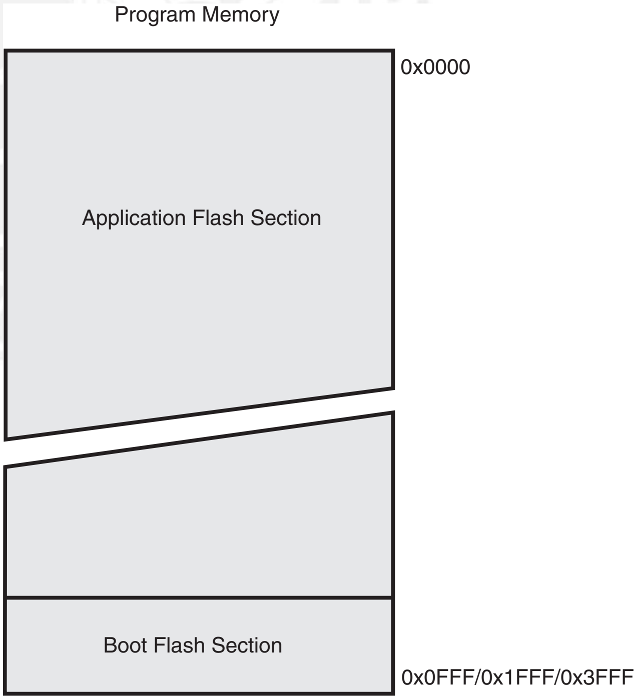


Figure 8-2. Program Memory Map Atmel ATmega88PA, Atmel ATmega168PA



## 8.3 SRAM Data Memory

Figure 8-3 shows how the Atmel® ATmega48PA/88PA/168PA SRAM Memory is organized.

The Atmel ATmega48PA/88PA/168PA is a complex microcontroller with more peripheral units than can be supported within the 64 locations reserved in the Opcode for the IN and OUT instructions. For the Extended I/O space from 0x60 - 0xFF in SRAM, only the ST/STS/STD and LD/LDS/LDD instructions can be used.

The lower 768/1280/1280/2303 data memory locations address both the Register File, the I/O memory, Extended I/O memory, and the internal data SRAM. The first 32 locations address the Register File, the next 64 location the standard I/O memory, then 160 locations of Extended I/O memory, and the next 512/1024/1024/2048 locations address the internal data SRAM.

The five different addressing modes for the data memory cover: Direct, Indirect with Displacement, Indirect, Indirect with Pre-decrement, and Indirect with Post-increment. In the Register File, registers R26 to R31 feature the indirect addressing pointer registers.

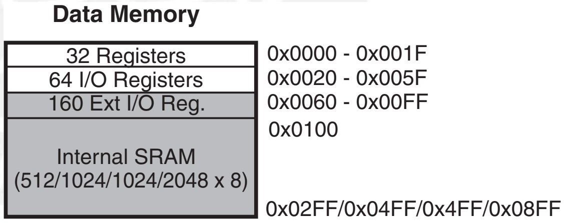
The direct addressing reaches the entire data space.

The Indirect with Displacement mode reaches 63 address locations from the base address given by the Y- or Z-register.

When using register indirect addressing modes with automatic pre-decrement and post-increment, the address registers X, Y, and Z are decremented or incremented.

The 32 general purpose working registers, 64 I/O Registers, 160 Extended I/O Registers, and the 512/1024/1024/2048 bytes of internal data SRAM in the Atmel ATmega48PA/88PA/168PA are all accessible through all these addressing modes. The Register File is described in "General Purpose Register File" on page 11.

**Figure 8-3.** Data Memory Map



### 8.3.1 Data Memory Access Times

This section describes the general access timing concepts for internal memory access. The internal data SRAM access is performed in two $clk_{CPU}$ cycles as described in Figure 8-4.

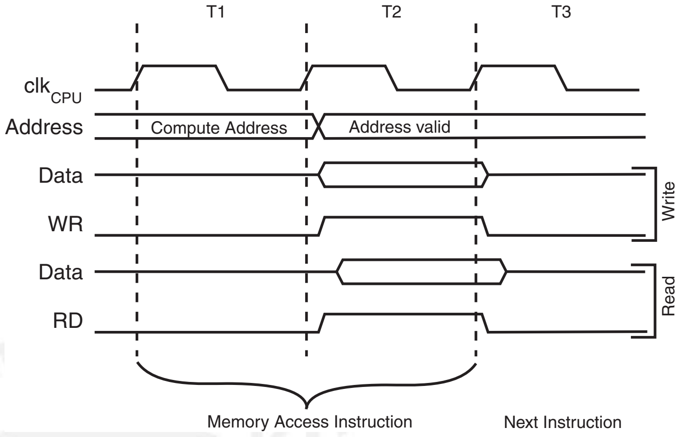


**Figure 8-4.** On-chip Data SRAM Access Cycles

## 8.4 EEPROM Data Memory

The Atmel® ATmega48PA/88PA/168PA contains 256/512/512 bytes of data EEPROM memory. It is organized as a separate data space, in which single bytes can be read and written. The EEPROM has an endurance of at least 100,000 write/erase cycles. The access between the EEPROM and the CPU is described in the following, specifying the EEPROM Address Registers, the EEPROM Data Register, and the EEPROM Control Register.

"Memory Programming" on page 294 contains a detailed description on EEPROM Programming in SPI or Parallel Programming mode.

### 8.4.1 EEPROM Read/Write Access

The EEPROM Access Registers are accessible in the I/O space.

The write access time for the EEPROM is given in Table 8-2. A self-timing function, however, lets the user software detect when the next byte can be written. If the user code contains instructions that write the EEPROM, some precautions must be taken. In heavily filtered power supplies, $V_{CC}$ is likely to rise or fall slowly on power-up/down. This causes the device for some period of time to run at a voltage lower than specified as minimum for the clock frequency used. See "Preventing EEPROM Corruption" on page 20 for details on how to avoid problems in these situations.

In order to prevent unintentional EEPROM writes, a specific write procedure must be followed. Refer to the description of the EEPROM Control Register for details on this.

When the EEPROM is read, the CPU is halted for four clock cycles before the next instruction is executed. When the EEPROM is written, the CPU is halted for two clock cycles before the next instruction is executed.

### 8.4.2 Preventing EEPROM Corruption

During periods of low $V_{CC,}$ the EEPROM data can be corrupted because the supply voltage is too low for the CPU and the EEPROM to operate properly. These issues are the same as for board level systems using EEPROM, and the same design solutions should be applied.

An EEPROM data corruption can be caused by two situations when the voltage is too low. First, a regular write sequence to the EEPROM requires a minimum voltage to operate correctly. Secondly, the CPU itself can execute instructions incorrectly, if the supply voltage is too low.

EEPROM data corruption can easily be avoided by following this design recommendation:

Keep the AVR RESET active (low) during periods of insufficient power supply voltage. This can be done by enabling the internal Brown-out Detector (BOD). If the detection level of the internal BOD does not match the needed detection level, an external low $V_{CC}$ reset Protection circuit can be used. If a reset occurs while a write operation is in progress, the write operation will be completed provided that the power supply voltage is sufficient.

## 8.5 I/O Memory

The I/O space definition of the Atmel® ATmega48PA/88PA/168PA is shown in "Register Summary" on page 356.

All Atmel ATmega48PA/88PA/168PA I/Os and peripherals are placed in the I/O space. All I/O locations may be accessed by the LD/LDS/LDD and ST/STS/STD instructions, transferring data between the 32 general purpose working registers and the I/O space. I/O Registers within the address range 0x00 - 0x1F are directly bit-accessible using the SBI and CBI instructions. In these registers, the value of single bits can be checked by using the SBIS and SBIC instructions. Refer to the instruction set section for more details. When using the I/O specific commands IN and OUT, the I/O addresses 0x00 - 0x3F must be used. When addressing I/O Registers as data space using LD and ST instructions, 0x20 must be added to these addresses. The Atmel ATmega48PA/88PA/168PA is a complex microcontroller with more peripheral units than can be supported within the 64 location reserved in Opcode for the IN and OUT instructions. For the Extended I/O space from 0x60 - 0xFF in SRAM, only the ST/STS/STD and LD/LDS/LDD instructions can be used.

For compatibility with future devices, reserved bits should be written to zero if accessed. Reserved I/O memory addresses should never be written.

Some of the Status Flags are cleared by writing a logical one to them. Note that, unlike most other AVRs, the CBI and SBI instructions will only operate on the specified bit, and can therefore be used on registers containing such Status Flags. The CBI and SBI instructions work with registers 0x00 to 0x1F only.

The I/O and peripherals control registers are explained in later sections.

### 8.5.1 General Purpose I/O Registers

The Atmel ATmega48PA/88PA/168PA contains three General Purpose I/O Registers. These registers can be used for storing any information, and they are particularly useful for storing global variables and Status Flags. General Purpose I/O Registers within the address range 0x00 - 0x1F are directly bit-accessible using the SBI, CBI, SBIS, and SBIC instructions.

## 8.6 Register Description

### 8.6.1 EEARH and EEARL – The EEPROM Address Register

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Bit | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | |
| 0x22 (0x42) | – | – | – | – | – | – | – | EEAR8 | EEARH |
| 0x21 (0x41) | EEAR7 | EEAR6 | EEAR5 | EEAR4 | EEAR3 | EEAR2 | EEAR1 | EEAR0 | EEARL |
| | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
| Read/Write | R | R | R | R | R | R | R | R/W | |
| | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | X | |
| | X | X | X | X | X | X | X | X | |

- **Bits 15:9] – Reserved**

These bits are reserved bits in the Atmel® ATmega48PA/88PA/168PA and will always read as zero.

- **Bits 8:0 – EEAR[8:0]: EEPROM Address**

The EEPROM Address Registers – EEARH and EEARL specify the EEPROM address in the 256/512/512/1K bytes EEPROM space. The EEPROM data bytes are addressed linearly between 0 and 255/511/511/1023. The initial value of EEAR is undefined. A proper value must be written before the EEPROM may be accessed.

EEAR8 is an unused bit in the Atmel ATmega48PA and must always be written to zero.

### 8.6.2 EEDR – The EEPROM Data Register

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
| 0x20 (0x40) | MSB | | | | | | | LSB | EEDR |
| Read/Write | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

- **Bits 7:0 – EEDR[7:0]: EEPROM Data**

For the EEPROM write operation, the EEDR Register contains the data to be written to the EEPROM in the address given by the EEAR Register. For the EEPROM read operation, the EEDR contains the data read out from the EEPROM at the address given by EEAR.

### 8.6.3 EECR – The EEPROM Control Register

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
| 0x1F (0x3F) | – | – | EEPM1 | EEPM0 | EERIE | EEMPE | EEPE | EERE | EECR |
| Read/Write | R | R | R/W | R/W | R/W | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | X | X | 0 | 0 | X | 0 | |

- **Bits 7:6 – Reserved**

These bits are reserved bits in the Atmel ATmega48PA/88PA/168PA and will always read as zero.

- **Bits 5, 4 – EEPM1 and EEPM0: EEPROM Programming Mode Bits**

The EEPROM Programming mode bit setting defines which programming action that will be triggered when writing EEPE. It is possible to program data in one atomic operation (erase the old value and program the new value) or to split the Erase and Write operations in two different operations. The Programming times for the different modes are shown in Table 8-1.

While EEPE is set, any write to EEPMn will be ignored. During reset, the EEPMn bits will be reset to 0b00 unless the EEPROM is busy programming.

**Table 8-1.** EEPROM Mode Bits

| EEPM1 | EEPM0 | Programming Time | Operation |
|---|---|---|---|
| 0 | 0 | 3.4ms | Erase and Write in one operation (Atomic Operation) |
| 0 | 1 | 1.8ms | Erase Only |
| 1 | 0 | 1.8ms | Write Only |
| 1 | 1 | – | Reserved for future use |

- **Bit 3 – EERIE: EEPROM Ready Interrupt Enable**

Writing EERIE to one enables the EEPROM Ready Interrupt if the I bit in SREG is set. Writing EERIE to zero disables the interrupt. The EEPROM Ready interrupt generates a constant interrupt when EEPE is cleared. The interrupt will not be generated during EEPROM write or SPM.

- **Bit 2 – EEMPE: EEPROM Master Write Enable**

The EEMPE bit determines whether setting EEPE to one causes the EEPROM to be written. When EEMPE is set, setting EEPE within four clock cycles will write data to the EEPROM at the selected address If EEMPE is zero, setting EEPE will have no effect. When EEMPE has been written to one by software, hardware clears the bit to zero after four clock cycles. See the description of the EEPE bit for an EEPROM write procedure.

- **Bit 1 – EEPE: EEPROM Write Enable**

The EEPROM Write Enable Signal EEPE is the write strobe to the EEPROM. When address and data are correctly set up, the EEPE bit must be written to one to write the value into the EEPROM. The EEMPE bit must be written to one before a logical one is written to EEPE, otherwise no EEPROM write takes place. The following procedure should be followed when writing the EEPROM (the order of steps 3 and 4 is not essential):

1. Wait until EEPE becomes zero.
2. Wait until SELFPRGEN in SPMCSR becomes zero.
3. Write new EEPROM address to EEAR (optional).
4. Write new EEPROM data to EEDR (optional).
5. Write a logical one to the EEMPE bit while writing a zero to EEPE in EECR.
6. Within four clock cycles after setting EEMPE, write a logical one to EEPE.

The EEPROM can not be programmed during a CPU write to the Flash memory. The software must check that the Flash programming is completed before initiating a new EEPROM write. Step 2 is only relevant if the software contains a Boot Loader allowing the CPU to program the Flash. If the Flash is never being updated by the CPU, step 2 can be omitted. See "Boot Loader Support – Read-While-Write Self-Programming" on page 277 for details about Boot programming.

**Caution:** An interrupt between step 5 and step 6 will make the write cycle fail, since the EEPROM Master Write Enable will time-out. If an interrupt routine accessing the EEPROM is interrupting another EEPROM access, the EEAR or EEDR Register will be modified, causing the interrupted EEPROM access to fail. It is recommended to have the Global Interrupt Flag cleared during all the steps to avoid these problems.

When the write access time has elapsed, the EEPE bit is cleared by hardware. The user software can poll this bit and wait for a zero before writing the next byte. When EEPE has been set, the CPU is halted for two cycles before the next instruction is executed.

- **Bit 0 – EERE: EEPROM Read Enable**

The EEPROM Read Enable Signal EERE is the read strobe to the EEPROM. When the correct address is set up in the EEAR Register, the EERE bit must be written to a logic one to trigger the EEPROM read. The EEPROM read access takes one instruction, and the requested data is available immediately. When the EEPROM is read, the CPU is halted for four cycles before the next instruction is executed.

The user should poll the EEPE bit before starting the read operation. If a write operation is in progress, it is neither possible to read the EEPROM, nor to change the EEAR Register.

The calibrated Oscillator is used to time the EEPROM accesses. Table 8-2 lists the typical programming time for EEPROM access from the CPU.

**Table 8-2.** EEPROM Programming Time

| Symbol | Number of Calibrated RC Oscillator Cycles | Typ Programming Time |
|---|---|---|
| EEPROM write<br>(from CPU) | 26,368 | 3.3ms |

The following code examples show one assembly and one C function for writing to the EEPROM. The examples assume that interrupts are controlled (e.g. by disabling interrupts globally) so that no interrupts will occur during execution of these functions. The examples also assume that no Flash Boot Loader is present in the software. If such code is present, the EEPROM write function must also wait for any ongoing SPM command to finish.

Assembly Code Example

```
EEPROM_write:
  ; Wait for completion of previous write
  sbic EECR,EEPE
  rjmp EEPROM_write
  ; Set up address (r18:r17) in address register
  out EEARH, r18
  out EEARL, r17
  ; Write data (r16) to Data Register
  out EEDR,r16
  ; Write logical one to EEMPE
  sbi EECR,EEMPE
  ; Start eeprom write by setting EEPE
  sbi EECR,EEPE
  ret
```

C Code Example

```
void EEPROM_write(unsigned int uiAddress, unsigned char ucData)
{
  /* Wait for completion of previous write */
  while(EECR & (1<<EEPE))
    ;
  /* Set up address and Data Registers */
  EEAR = uiAddress;
  EEDR = ucData;
  /* Write logical one to EEMPE */
  EECR |= (1<<EEMPE);
  /* Start eeprom write by setting EEPE */
  EECR |= (1<<EEPE);
}
```

The next code examples show assembly and C functions for reading the EEPROM. The examples assume that interrupts are controlled so that no interrupts will occur during execution of these functions.

Assembly Code Example

```
EEPROM_read:
  ; Wait for completion of previous write
  sbic EECR,EEPE
  rjmp EEPROM_read
  ; Set up address (r18:r17) in address register
  out  EEARH, r18
  out  EEARL, r17
  ; Start eeprom read by writing EERE
  sbi  EECR,EERE
  ; Read data from Data Register
  in   r16,EEDR
  ret
```

C Code Example

```
unsigned char EEPROM_read(unsigned int uiAddress)
{
  /* Wait for completion of previous write */
  while(EECR & (1<<EEPE))
    ;
  /* Set up address register */
  EEAR = uiAddress;
  /* Start eeprom read by writing EERE */
  EECR |= (1<<EERE);
  /* Return data from Data Register */
  return EEDR;
}
```

### 8.6.4 GPIOR2 – General Purpose I/O Register 2

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
| 0x2B (0x4B) | MSB | | | | | | | LSB | GPIOR2 |
| Read/Write | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

### 8.6.5 GPIOR1 – General Purpose I/O Register 1

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
| 0x2A (0x4A) | MSB | | | | | | | LSB | GPIOR1 |
| Read/Write | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

### 8.6.6 GPIOR0 – General Purpose I/O Register 0

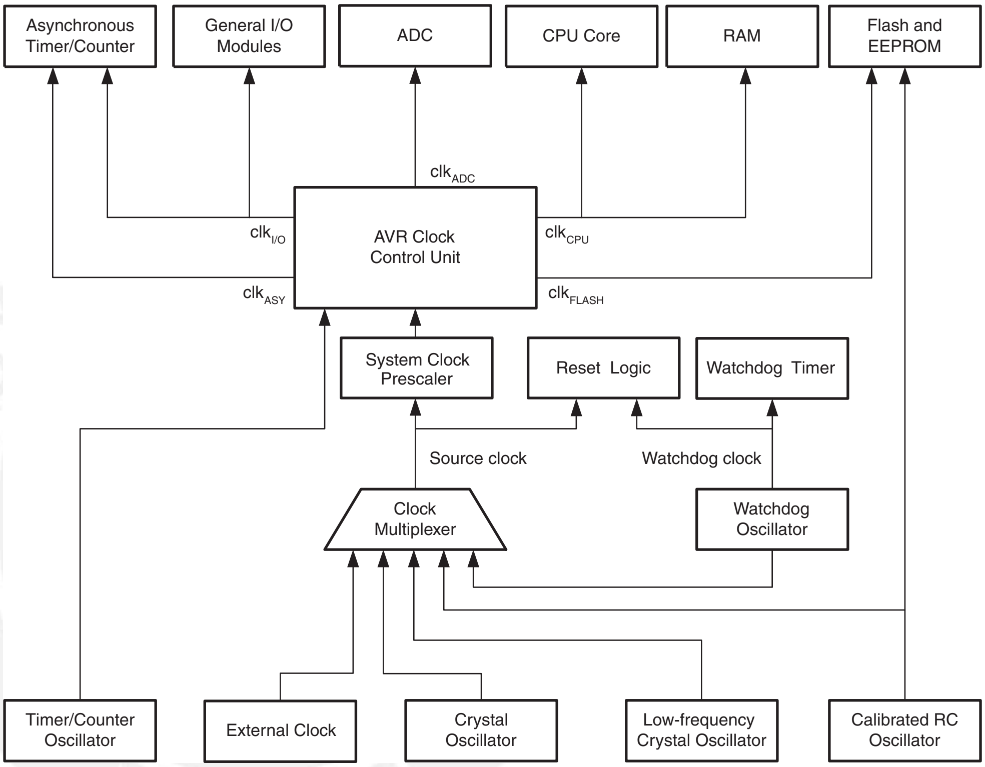
| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
| 0x1E (0x3E) | MSB | | | | | | | LSB | GPIOR0 |
| Read/Write | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

# 9. System Clock and Clock Options

## 9.1 Clock Systems and their Distribution

Figure 9-1 presents the principal clock systems in the AVR and their distribution. All of the clocks need not be active at a given time. In order to reduce power consumption, the clocks to modules not being used can be halted by using different sleep modes, as described in "Power Management and Sleep Modes" on page 39. The clock systems are detailed below.

**Figure 9-1.** Clock Distribution



### 9.1.1 CPU Clock – $clk_{CPU}$

The CPU clock is routed to parts of the system concerned with operation of the AVR core. Examples of such modules are the General Purpose Register File, the Status Register and the data memory holding the Stack Pointer. Halting the CPU clock inhibits the core from performing general operations and calculations.

### 9.1.2 I/O Clock – $clk_{I/O}$

The I/O clock is used by the majority of the I/O modules, like Timer/Counters, SPI, and USART. The I/O clock is also used by the External Interrupt module, but note that start condition detection in the USI module is carried out asynchronously when $clk_{I/O}$ is halted, TWI address recognition in all sleep modes.

Note: Note that if a level triggered interrupt is used for wake-up from Power-down, the required level must be held long enough for the MCU to complete the wake-up to trigger the level interrupt. If the level disappears before the end of the Start-up Time, the MCU will still wake up, but no interrupt will be generated. The start-up time is defined by the SUT and CKSEL Fuses as described in "System Clock and Clock Options" on page 26.

### 9.1.3 Flash Clock – $clk_{FLASH}$

The Flash clock controls operation of the Flash interface. The Flash clock is usually active simultaneously with the CPU clock.

### 9.1.4 Asynchronous Timer Clock – $clk_{ASY}$

The Asynchronous Timer clock allows the Asynchronous Timer/Counter to be clocked directly from an external clock or an external 32kHz clock crystal. The dedicated clock domain allows using this Timer/Counter as a real-time counter even when the device is in sleep mode.

### 9.1.5 ADC Clock – $clk_{ADC}$

The ADC is provided with a dedicated clock domain. This allows halting the CPU and I/O clocks in order to reduce noise generated by digital circuitry. This gives more accurate ADC conversion results.

## 9.2 Clock Sources

The device has the following clock source options, selectable by Flash Fuse bits as shown below. The clock from the selected source is input to the AVR clock generator, and routed to the appropriate modules.

**Table 9-1.** Device Clocking Options Select[(1)]

| Device Clocking Option | CKSEL3...0 |
|---|---|
| Low Power Crystal Oscillator | 1111 - 1000 |
| Full Swing Crystal Oscillator | 0111 - 0110 |
| Low Frequency Crystal Oscillator | 0101 - 0100 |
| Internal 128kHz RC Oscillator | 0011 |
| Calibrated Internal RC Oscillator | 0010 |
| External Clock | 0000 |
| Reserved | 0001 |

Note: 1. For all fuses “1” means unprogrammed while “0” means programmed.

### 9.2.1 Default Clock Source

The device is shipped with internal RC oscillator at 8.0MHz and with the fuse CKDIV8 programmed, resulting in 1.0MHz system clock. The startup time is set to maximum and time-out period enabled. (CKSEL = "0010", SUT = "10", CKDIV8 = "0"). The default setting ensures that all users can make their desired clock source setting using any available programming interface.

### 9.2.2 Clock Startup Sequence

Any clock source needs a sufficient $V_{CC}$ to start oscillating and a minimum number of oscillating cycles before it can be considered stable.

To ensure sufficient $V_{CC}$, the device issues an internal reset with a time-out delay ($t_{TOUT}$) after the device reset is released by all other reset sources. “System Control and Reset” on page 47 describes the start conditions for the internal reset. The delay ($t_{TOUT}$) is timed from the Watchdog Oscillator and the number of cycles in the delay is set by the SUTx and CKSELx fuse bits. The selectable delays are shown in Table 9-2. The frequency of the Watchdog Oscillator is voltage dependent as shown in “Typical Characteristics” on page 324.

**Table 9-2.** Number of Watchdog Oscillator Cycles

| Typ Time-out ($V_{CC}$ = 5.0V) | Typ Time-out ($V_{CC}$ = 3.0V) | Number of Cycles |
|---|---|---|
| 0ms | 0ms | 0 |
| 4.1ms | 4.3ms | 512 |
| 65ms | 69ms | 8K (8,192) |

Main purpose of the delay is to keep the AVR in reset until it is supplied with minimum $V_{CC}$. The delay will not monitor the actual voltage and it will be required to select a delay longer than the $V_{CC}$ rise time. If this is not possible, an internal or external Brown-Out Detection circuit should be used. A BOD circuit will ensure sufficient $V_{CC}$ before it releases the reset, and the time-out delay can be disabled. Disabling the time-out delay without utilizing a Brown-Out Detection circuit is not recommended.

The oscillator is required to oscillate for a minimum number of cycles before the clock is considered stable. An internal ripple counter monitors the oscillator output clock, and keeps the internal reset active for a given number of clock cycles. The reset is then released and the device will start to execute. The recommended oscillator start-up time is dependent on the clock type, and varies from 6 cycles for an externally applied clock to 32K cycles for a low frequency crystal.

The start-up sequence for the clock includes both the time-out delay and the start-up time when the device starts up from reset. When starting up from Power-save or Power-down mode, $V_{CC}$ is assumed to be at a sufficient level and only the start-up time is included.

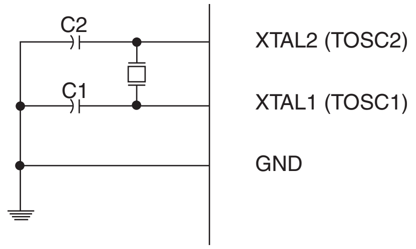
## 9.3 Low Power Crystal Oscillator

Pins XTAL1 and XTAL2 are input and output, respectively, of an inverting amplifier which can be configured for use as an On-chip Oscillator, as shown in Figure 9-2 on page 28. Either a quartz crystal or a ceramic resonator may be used.

This Crystal Oscillator is a low power oscillator, with reduced voltage swing on the XTAL2 output. It gives the lowest power consumption, but is not capable of driving other clock inputs, and may be more susceptible to noise in noisy environments. In these cases, refer to the "Full Swing Crystal Oscillator" on page 30.

C1 and C2 should always be equal for both crystals and resonators. The optimal value of the capacitors depends on the crystal or resonator in use, the amount of stray capacitance, and the electromagnetic noise of the environment. Some initial guidelines for choosing capacitors for use with crystals are given in Table 9-3 on page 29. For ceramic resonators, the capacitor values given by the manufacturer should be used.

**Figure 9-2.** Crystal Oscillator Connections



The Low Power Oscillator can operate in three different modes, each optimized for a specific frequency range. The operating mode is selected by the fuses CKSEL3...1 as shown in Table 9-3 on page 29.

**Table 9-3.** Low Power Crystal Oscillator Operating Modes[3]

| Frequency Range (MHz) | Recommended Range for Capacitors C1 and C2 (pF) | CKSEL3...1[1] |
|---|---|---|
| 0.4 - 0.9 | – | 100[2] |
| 0.9 - 3.0 | 12 - 22 | 101 |
| 3.0 - 8.0 | 12 - 22 | 110 |
| 8.0 - 16.0 | 12 - 22 | 111 |

Notes: 1. This is the recommended CKSEL settings for the difference frequency ranges.

2. This option should not be used with crystals, only with ceramic resonators.

3. If the crystal frequency exceeds the specification of the device (depends on $V_{CC}$), the CKDIV8 Fuse can be programmed in order to divide the internal frequency by 8. It must be ensured that the resulting divided clock meets the frequency specification of the device.

The CKSEL0 Fuse together with the SUT1...0 Fuses select the start-up times as shown in Table 9-4.

**Table 9-4.** Start-up Times for the Low Power Crystal Oscillator Clock Selection

| Oscillator Source / Power Conditions | Start-up Time from Power-down and Power-save | Additional Delay from Reset ($V_{CC}$ = 5.0V) | CKSEL0 | SUT1...0 |
|---|---|---|---|---|
| Ceramic resonator, fast rising power | 258 CK | 14CK + 4.1ms[1] | 0 | 00 |
| Ceramic resonator, slowly rising power | 258 CK | 14CK + 65ms[1] | 0 | 01 |
| Ceramic resonator, BOD enabled | 1K CK | 14CK[2] | 0 | 10 |
| Ceramic resonator, fast rising power | 1K CK | 14CK + 4.1ms[2] | 0 | 11 |
| Ceramic resonator, slowly rising power | 1K CK | 14CK + 65ms[2] | 1 | 00 |
| Crystal Oscillator, BOD enabled | 16K CK | 14CK | 1 | 01 |
| Crystal Oscillator, fast rising power | 16K CK | 14CK + 4.1ms | 1 | 10 |
| Crystal Oscillator, slowly rising power | 16K CK | 14CK + 65ms | 1 | 11 |

Notes: 1. These options should only be used when not operating close to the maximum frequency of the device, and only if frequency stability at start-up is not important for the application. These options are not suitable for crystals.

2. These options are intended for use with ceramic resonators and will ensure frequency stability at start-up. They can also be used with crystals when not operating close to the maximum frequency of the device, and if frequency stability at start-up is not important for the application.

## 9.4 Full Swing Crystal Oscillator

Pins XTAL1 and XTAL2 are input and output, respectively, of an inverting amplifier which can be configured for use as an On-chip Oscillator, as shown in Figure 9-2 on page 28. Either a quartz crystal or a ceramic resonator may be used.

This Crystal Oscillator is a full swing oscillator, with rail-to-rail swing on the XTAL2 output. This is useful for driving other clock inputs and in noisy environments. The current consumption is higher than the "Low Power Crystal Oscillator" on page 28. Note that the Full Swing Crystal Oscillator will only operate for $V_{CC}$ = 2.7 - 5.5V.

C1 and C2 should always be equal for both crystals and resonators. The optimal value of the capacitors depends on the crystal or resonator in use, the amount of stray capacitance, and the electromagnetic noise of the environment. Some initial guidelines for choosing capacitors for use with crystals are given in Table 9-6 on page 31. For ceramic resonators, the capacitor values given by the manufacturer should be used.

The operating mode is selected by the fuses CKSEL3...1 as shown in Table 9-5.

**Table 9-5.** Full Swing Crystal Oscillator operating modes

| Frequency Range[1] (MHz) | Recommended Range for Capacitors C1 and C2 (pF) | CKSEL3...1 |
|---|---|---|
| 0.4 - 20 | 12 - 22 | 011 |

Notes: 1. If the crystal frequency exceeds the specification of the device (depends on $V_{CC}$), the CKDIV8 Fuse can be programmed in order to divide the internal frequency by 8. It must be ensured that the resulting divided clock meets the frequency specification of the device.

**Figure 9-3.** Crystal Oscillator Connections

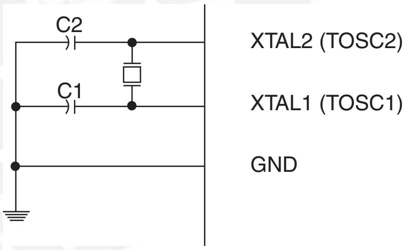


**Table 9-6.** Start-up Times for the Full Swing Crystal Oscillator Clock Selection

| Oscillator Source / Power Conditions | Start-up Time from Power-down and Power-save | Additional Delay from Reset ($V_{CC}$ = 5.0V) | CKSEL0 | SUT1...0 |
|---|---|---|---|---|
| Ceramic resonator, fast rising power | 258 CK | 14CK + 4.1ms[1] | 0 | 00 |
| Ceramic resonator, slowly rising power | 258 CK | 14CK + 65ms[1] | 0 | 01 |
| Ceramic resonator, BOD enabled | 1K CK | 14CK[2] | 0 | 10 |
| Ceramic resonator, fast rising power | 1K CK | 14CK + 4.1ms[2] | 0 | 11 |
| Ceramic resonator, slowly rising power | 1K CK | 14CK + 65ms[2] | 1 | 00 |
| Crystal Oscillator, BOD enabled | 16K CK | 14CK | 1 | 01 |
| Crystal Oscillator, fast rising power | 16K CK | 14CK + 4.1ms | 1 | 10 |
| Crystal Oscillator, slowly rising power | 16K CK | 14CK + 65ms | 1 | 11 |

Notes: 1. These options should only be used when not operating close to the maximum frequency of the device, and only if frequency stability at start-up is not important for the application. These options are not suitable for crystals.

2. These options are intended for use with ceramic resonators and will ensure frequency stability at start-up. They can also be used with crystals when not operating close to the maximum frequency of the device, and if frequency stability at start-up is not important for the application.

## 9.5 Low Frequency Crystal Oscillator

The Low-frequency Crystal Oscillator is optimized for use with a 32.768kHz watch crystal. When selecting crystals, load capacitance and crystal's Equivalent Series Resistance, ESR must be taken into consideration. Both values are specified by the crystal vendor. Atmel® ATmega48PA/88PA/168PA oscillator is optimized for very low power consumption, and thus when selecting crystals, see Table 9-7 for maximum ESR recommendations on 6.5pF, 9.0pF and 12.5pF crystals

**Table 9-7.** Maximum ESR Recommendation for 32.768kHz Crystal

| Crystal CL (pF) | Max ESR [kΩ][1] |
|---|---|
| 6.5 | 75 |
| 9.0 | 65 |
| 12.5 | 30 |

Note: 1. Maximum ESR is typical value based on characterization

The Low-frequency Crystal Oscillator provides an internal load capacitance, see Table 9-8 at each TOSC pin.

**Table 9-8.** Capacitance for Low-frequency Oscillator

| Device | 32kHz Osc. Type | Cap(Xtal1/Tosc1) | Cap(Xtal2/Tosc2) |
|---|---|---|---|
| Atmel ATmega48PA/88PA/168PA | System Osc. | 18pF | 8pF |
|  | Timer Osc. | 18pF | 8pF |

The capacitance (Ce + Ci) needed at each TOSC pin can be calculated by using:

$$C = 2 \times CL - C_S$$

where:

– Ce - is optional external capacitors as described in Figure 9-2 on page 28

– Ci - is the pin capacitance in Table 9-8

– CL - is the load capacitance for a 32.768kHz crystal specified by the crystal vendor

– CS - is the total stray capacitance for one TOSC pin.

Crystals specifying load capacitance (CL) higher than 6pF, require external capacitors applied as described in Figure 9-2 on page 28.

The Low-frequency Crystal Oscillator must be selected by setting the CKSEL Fuses to “0110” or “0111”, as shown in Table 9-10. Start-up times are determined by the SUT Fuses as shown in Table 9-9.

**Table 9-9.** Start-up Times for the Low-frequency Crystal Oscillator Clock Selection

| SUT1...0 | Additional Delay from Reset ($V_{CC}$ = 5.0V) | Recommended Usage |
|---|---|---|
| 00 | 4 CK | Fast rising power or BOD enabled |
| 01 | 4 CK + 4.1ms | Slowly rising power |
| 10 | 4 CK + 65ms | Stable frequency at start-up |
| 11 | Reserved |  |

**Table 9-10.** Start-up Times for the Low-frequency Crystal Oscillator Clock Selection

| CKSEL3...0 | Start-up Time from Power-down and Power-save | Recommended Usage |
|---|---|---|
| 0100(1) | 1K CK |  |
| 0101 | 32K CK | Stable frequency at start-up |

Note: 1. This option should only be used if frequency stability at start-up is not important for the application

## 9.6 Calibrated Internal RC Oscillator

By default, the Internal RC Oscillator provides an approximate 8.0MHz clock. Though voltage and temperature dependent, this clock can be very accurately calibrated by the user. See Table 29-3 on page 316 for more details. The device is shipped with the CKDIV8 Fuse programmed. See "System Clock Prescaler" on page 36 for more details.

This clock may be selected as the system clock by programming the CKSEL Fuses as shown in Table 9-1. If selected, it will operate with no external components. During reset, hardware loads the pre-programmed default 3V calibration value into the OSCCAL Register and thereby automatically calibrates the RC Oscillator for 3V operation. If the device is to be used at 5V then the alternate RC Oscillator 5V Calibration Byte (Table 27-5 on page 286) can be read from Signature Row and stored into the OSCCAL register by the user application program for better 5V frequency accuracy. The accuracy of this calibration is shown as Factory calibration in Table 29-3 on page 316.

By changing the OSCCAL register from SW, see "OSCCAL – Oscillator Calibration Register" on page 37, it is possible to get a higher calibration accuracy than by using the factory calibration. The accuracy of this calibration is shown as User calibration in Table 29-3 on page 316.

When this Oscillator is used as the chip clock, the Watchdog Oscillator will still be used for the Watchdog Timer and for the Reset Time-out. For more information on the pre-programmed calibration value, see the section "Calibration Byte" on page 297.

**Table 9-11.** Internal Calibrated RC Oscillator Operating Modes

| Frequency Range[2] (MHz) | CKSEL3...0 |
|---|---|
| 7.3 - 8.1 | 0010[1] |

Notes: 1. The device is shipped with this option selected.

2. If 8MHz frequency exceeds the specification of the device (depends on $V_{CC}$), the CKDIV8 Fuse can be programmed in order to divide the internal frequency by 8.

When this Oscillator is selected, start-up times are determined by the SUT Fuses as shown in Table 9-12.

**Table 9-12.** Start-up times for the internal calibrated RC Oscillator clock selection

| Power Conditions | Start-up Time from Power-down and Power-save | Additional Delay from Reset ($V_{CC}$ = 5.0V) | SUT1...0 |
|---|---|---|---|
| BOD enabled | 6 CK | 14CK[1] | 00 |
| Fast rising power | 6 CK | 14CK + 4.1ms | 01 |
| Slowly rising power | 6 CK | 14CK + 65ms[2] | 10 |
| Reserved | | | 11 |

Note: 1. If the RSTDISBL fuse is programmed, this start-up time will be increased to 14CK + 4.1ms to ensure programming mode can be entered.

2. The device is shipped with this option selected.

## 9.7 128kHz Internal Oscillator

The 128kHz internal Oscillator is a low power Oscillator providing a clock of 128kHz. The frequency is nominal at 3V and 25°C. This clock may be select as the system clock by programming the CKSEL Fuses to "11" as shown in Table 9-13.

**Table 9-13.** 128kHz Internal Oscillator Operating Modes

| Nominal Frequency[1] | CKSEL3...0 |
|---|---|
| 128kHz | 0011 |

Note: 1. Note that the 128kHz oscillator is a very low power clock source, and is not designed for high accuracy.

When this clock source is selected, start-up times are determined by the SUT Fuses as shown in Table 9-14.

**Table 9-14.** Start-up Times for the 128kHz Internal Oscillator

| Power Conditions | Start-up Time from Power-down and Power-save | Additional Delay from Reset | SUT1...0 |
|---|---|---|---|
| BOD enabled | 6 CK | 14CK[1] | 00 |
| Fast rising power | 6 CK | 14CK + 4ms | 01 |
| Slowly rising power | 6 CK | 14CK + 64ms | 10 |
| | Reserved | | 11 |

Note: 1. If the RSTDISBL fuse is programmed, this start-up time will be increased to 14CK + 4.1ms to ensure programming mode can be entered.

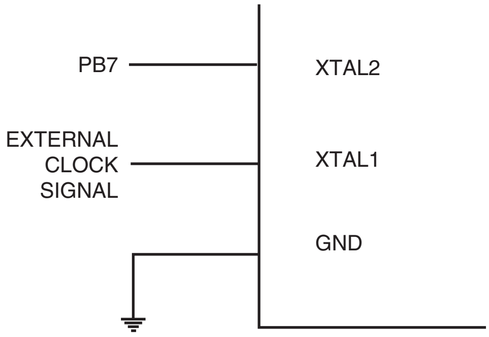
## 9.8 External Clock

To drive the device from an external clock source, XTAL1 should be driven as shown in Figure 9-4. To run the device on an external clock, the CKSEL Fuses must be programmed to "0000" (see Table 9-15).

**Table 9-15.** Crystal Oscillator Clock Frequency

| Frequency | CKSEL3...0 |
|---|---|
| 0 - 16MHz | 0000 |

**Figure 9-4.** External Clock Drive Configuration



When this clock source is selected, start-up times are determined by the SUT Fuses as shown in Table 9-16.

**Table 9-16.** Start-up Times for the External Clock Selection

| Power Conditions | Start-up Time from Power-down and Power-save | Additional Delay from Reset ($V_{CC}$ = 5.0V) | SUT1...0 |
|---|---|---|---|
| BOD enabled | 6 CK | 14CK | 00 |
| Fast rising power | 6 CK | 14CK + 4.1ms | 01 |
| Slowly rising power | 6 CK | 14CK + 65ms | 10 |
| Reserved | | | 11 |

When applying an external clock, it is required to avoid sudden changes in the applied clock frequency to ensure stable operation of the MCU. A variation in frequency of more than 2% from one clock cycle to the next can lead to unpredictable behavior. If changes of more than 2% is required, ensure that the MCU is kept in Reset during the changes.

Note that the System Clock Prescaler can be used to implement run-time changes of the internal clock frequency while still ensuring stable operation. Refer to "System Clock Prescaler" on page 36 for details.

## 9.9 Clock Output Buffer

The device can output the system clock on the CLKO pin. To enable the output, the CKOUT Fuse has to be programmed. This mode is suitable when the chip clock is used to drive other circuits on the system. The clock also will be output during reset, and the normal operation of I/O pin will be overridden when the fuse is programmed. Any clock source, including the internal RC Oscillator, can be selected when the clock is output on CLKO. If the System Clock Prescaler is used, it is the divided system clock that is output.

## 9.10 Timer/Counter Oscillator

The Atmel® ATmega48PA/88PA/168PA uses the same crystal oscillator for Low-frequency Oscillator and Timer/Counter Oscillator. See "Low Frequency Crystal Oscillator" on page 31 for details on the oscillator and crystal requirements.

Atmel ATmega48PA/88PA/168PA share the Timer/Counter Oscillator Pins (TOSC1 and TOSC2) with XTAL1 and XTAL2. When using the Timer/Counter Oscillator, the system clock needs to be four times the oscillator frequency. Due to this and the pin sharing, the Timer/Counter Oscillator can only be used when the Calibrated Internal RC Oscillator is selected as system clock source.

Applying an external clock source to TOSC1 can be done if EXTCLK in the ASSR Register is written to logic one. See "Asynchronous Operation of Timer/Counter2" on page 155 for further description on selecting external clock as input instead of a 32.768kHz watch crystal.

## 9.11 System Clock Prescaler

The Atmel ATmega48PA/88PA/168PA has a system clock prescaler, and the system clock can be divided by setting the "CLKPR – Clock Prescale Register" on page 377. This feature can be used to decrease the system clock frequency and the power consumption when the requirement for processing power is low. This can be used with all clock source options, and it will affect the clock frequency of the CPU and all synchronous peripherals. $\text{clk}_{I/O}$, $\text{clk}_{ADC}$, $\text{clk}_{CPU}$, and $\text{clk}_{FLASH}$ are divided by a factor as shown in Table 29-5 on page 317.

When switching between prescaler settings, the System Clock Prescaler ensures that no glitches occurs in the clock system. It also ensures that no intermediate frequency is higher than neither the clock frequency corresponding to the previous setting, nor the clock frequency corresponding to the new setting. The ripple counter that implements the prescaler runs at the frequency of the undivided clock, which may be faster than the CPU's clock frequency. Hence, it is not possible to determine the state of the prescaler - even if it were readable, and the exact time it takes to switch from one clock division to the other cannot be exactly predicted. From the time the CLKPS values are written, it takes between T1 + T2 and T1 + 2 * T2 before the new clock frequency is active. In this interval, 2 active clock edges are produced. Here, T1 is the previous clock period, and T2 is the period corresponding to the new prescaler setting.

To avoid unintentional changes of clock frequency, a special write procedure must be followed to change the CLKPS bits:

1. Write the Clock Prescaler Change Enable (CLKPCE) bit to one and all other bits in CLKPR to zero.
2. Within four cycles, write the desired value to CLKPS while writing a zero to CLKPCE.

Interrupts must be disabled when changing prescaler setting to make sure the write procedure is not interrupted.

## 9.12 Register Description

### 9.12.1 OSCCAL – Oscillator Calibration Register

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| (0x66) | CAL7 | CAL6 | CAL5 | CAL4 | CAL3 | CAL2 | CAL1 | CAL0 | OSCCAL |
| Read/Write | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | |
| Initial Value | Device Specific Calibration Value | | | | | | | | |

- **Bits 7:0 – CAL[7:0]: Oscillator Calibration Value**

The Oscillator Calibration Register is used to trim the Calibrated Internal RC Oscillator to remove process variations from the oscillator frequency. A pre-programmed calibration value is automatically written to this register during chip reset, giving the Factory calibrated frequency as specified in Table 29-3 on page 316. The application software can write this register to change the oscillator frequency. The oscillator can be calibrated to frequencies as specified in Table 29-3 on page 316. Calibration outside that range is not guaranteed.

Note that this oscillator is used to time EEPROM and Flash write accesses, and these write times will be affected accordingly. If the EEPROM or Flash are written, do not calibrate to more than 8.8MHz. Otherwise, the EEPROM or Flash write may fail.

The CAL7 bit determines the range of operation for the oscillator. Setting this bit to 0 gives the lowest frequency range, setting this bit to 1 gives the highest frequency range. The two frequency ranges are overlapping, in other words a setting of OSCCAL = 0x7F gives a higher frequency than OSCCAL = 0x80.

The CAL6...0 bits are used to tune the frequency within the selected range. A setting of 0x00 gives the lowest frequency in that range, and a setting of 0x7F gives the highest frequency in the range.

### 9.12.2 CLKPR – Clock Prescale Register

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| (0x61) | CLKPCE | – | – | – | CLKPS3 | CLKPS2 | CLKPS1 | CLKPS0 | CLKPR |
| Read/Write | R/W | R | R | R | R/W | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | See Bit Description | | | | |

- **Bit 7 – CLKPCE: Clock Prescaler Change Enable**

The CLKPCE bit must be written to logic one to enable change of the CLKPS bits. The CLKPCE bit is only updated when the other bits in CLKPR are simultaneously written to zero. CLKPCE is cleared by hardware four cycles after it is written or when CLKPS bits are written. Rewriting the CLKPCE bit within this time-out period does neither extend the time-out period, nor clear the CLKPCE bit.

- **Bits 3:0 – CLKPS[3:0]: Clock Prescaler Select Bits 3 - 0**

These bits define the division factor between the selected clock source and the internal system clock. These bits can be written run-time to vary the clock frequency to suit the application requirements. As the divider divides the master clock input to the MCU, the speed of all synchronous peripherals is reduced when a division factor is used. The division factors are given in Table 9-17 on page 38.

The CKDIV8 Fuse determines the initial value of the CLKPS bits. If CKDIV8 is unprogrammed, the CLKPS bits will be reset to “0000”. If CKDIV8 is programmed, CLKPS bits are reset to “0011”, giving a division factor of 8 at start up. This feature should be used if the selected clock source has a higher frequency than the maximum frequency of the device at the present operating conditions. Note that any value can be written to the CLKPS bits regardless of the CKDIV8 Fuse setting. The Application software must ensure that a sufficient division factor is chosen if the selected clock source has a higher frequency than the maximum frequency of the device at the present operating conditions. The device is shipped with the CKDIV8 Fuse programmed.

**Table 9-17.** Clock Prescaler Select

| CLKPS3 | CLKPS2 | CLKPS1 | CLKPS0 | Clock Division Factor |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 0 | 1 | 2 |
| 0 | 0 | 1 | 0 | 4 |
| 0 | 0 | 1 | 1 | 8 |
| 0 | 1 | 0 | 0 | 16 |
| 0 | 1 | 0 | 1 | 32 |
| 0 | 1 | 1 | 0 | 64 |
| 0 | 1 | 1 | 1 | 128 |
| 1 | 0 | 0 | 0 | 256 |
| 1 | 0 | 0 | 1 | Reserved |
| 1 | 0 | 1 | 0 | Reserved |
| 1 | 0 | 1 | 1 | Reserved |
| 1 | 1 | 0 | 0 | Reserved |
| 1 | 1 | 0 | 1 | Reserved |
| 1 | 1 | 1 | 0 | Reserved |
| 1 | 1 | 1 | 1 | Reserved |

# 10. Power Management and Sleep Modes

Sleep modes enable the application to shut down unused modules in the MCU, thereby saving power. The AVR provides various sleep modes allowing the user to tailor the power consumption to the application's requirements.

When enabled, the Brown-out Detector (BOD) actively monitors the power supply voltage during the sleep periods. To further save power, it is possible to disable the BOD in some sleep modes. See "BOD Disable()" on page 40 for more details.

## 10.1 Sleep Modes

Figure 9-1 on page 26 presents the different clock systems in the Atmel® ATmega48PA/88PA/168PA, and their distribution. The figure is helpful in selecting an appropriate sleep mode. Table 10-1 shows the different sleep modes, their wake up sources BOD disable ability[1].

Note: 1. BOD disable is only available for ATmega48PA/88PA/168PA.

**Table 10-1.** Active Clock Domains and Wake-up Sources in the Different Sleep Modes.

| | Active Clock Domains | | | | | Oscillators | | Wake-up Sources | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Sleep Mode** | $clk_{CPU}$ | $clk_{FLASH}$ | $clk_{IO}$ | $clk_{ADC}$ | $clk_{ASY}$ | Main Clock Source Enabled | Timer Oscillator Enabled | INT1, INT0 and Pin Change | TWI Address Match | Timer2 | SPM/EEPROM Ready | ADC | WDT | Other I/O | Software BOD Disable |
| Idle | | | X | X | X | X | X[2] | X | X | X | X | X | X | X | |
| ADC Noise Reduction | | | | X | X | X | X[2] | X[3] | X | X[2] | X | X | X | | |
| Power-down | | | | | | | | X[3] | X | | | | X | | X |
| Power-save | | | | | X | | X[2] | X[3] | X | X | | | X | | X |
| Standby[1] | | | | | | X | | X[3] | X | | | | X | | X |
| Extended Standby | | | | | X[2] | X | X[2] | X[3] | X | X | | | X | | X |

Notes: 1. Only recommended with external crystal or resonator selected as clock source.
2. If Timer/Counter2 is running in asynchronous mode.
3. For INT1 and INT0, only level interrupt.

To enter any of the six sleep modes, the SE bit in SMCR must be written to logic one and a SLEEP instruction must be executed. The SM2, SM1, and SM0 bits in the SMCR Register select which sleep mode (Idle, ADC Noise Reduction, Power-down, Power-save, Standby, or Extended Standby) will be activated by the SLEEP instruction. See Table 10-2 on page 44 for a summary.

If an enabled interrupt occurs while the MCU is in a sleep mode, the MCU wakes up. The MCU is then halted for four cycles in addition to the start-up time, executes the interrupt routine, and resumes execution from the instruction following SLEEP. The contents of the Register File and SRAM are unaltered when the device wakes up from sleep. If a reset occurs during sleep mode, the MCU wakes up and executes from the Reset Vector.

## 10.2 BOD Disable[]

When the Brown-out Detector (BOD) is enabled by BODLEVEL fuses - see Table 28-6 on page 296 and onwards, the BOD is actively monitoring the power supply voltage during a sleep period. To save power, it is possible to disable the BOD by software for some of the sleep modes, see Table 10-1 on page 39. The sleep mode power consumption will then be at the same level as when BOD is globally disabled by fuses. If BOD is disabled in software, the BOD function is turned off immediately after entering the sleep mode. Upon wake-up from sleep, BOD is automatically enabled again. This ensures safe operation in case the $V_{CC}$ level has dropped during the sleep period.

When the BOD has been disabled, the wake-up time from sleep mode will be approximately 60 µs to ensure that the BOD is working correctly before the MCU continues executing code.

BOD disable is controlled by bit 6, BODS (BOD Sleep) in the control register MCUCR, see "MCUCR – MCU Control Register" on page 45. Writing this bit to one turns off the BOD in relevant sleep modes, while a zero in this bit keeps BOD active. Default setting keeps BOD active, i.e. BODS set to zero.

Writing to the BODS bit is controlled by a timed sequence and an enable bit, see "MCUCR – MCU Control Register" on page 45.

Note: 1. BOD disable only available in picoPower devices ATmega48PA/88PA/168PA

## 10.3 Idle Mode

When the SM2...0 bits are written to 000, the SLEEP instruction makes the MCU enter Idle mode, stopping the CPU but allowing the SPI, USART, Analog Comparator, ADC, 2-wire Serial Interface, Timer/Counters, Watchdog, and the interrupt system to continue operating. This sleep mode basically halts $clk_{CPU}$ and $clk_{FLASH}$, while allowing the other clocks to run.

Idle mode enables the MCU to wake up from external triggered interrupts as well as internal ones like the Timer Overflow and USART Transmit Complete interrupts. If wake-up from the Analog Comparator interrupt is not required, the Analog Comparator can be powered down by setting the ACD bit in the Analog Comparator Control and Status Register – ACSR. This will reduce power consumption in Idle mode. If the ADC is enabled, a conversion starts automatically when this mode is entered.

## 10.4 ADC Noise Reduction Mode

When the SM2...0 bits are written to 001, the SLEEP instruction makes the MCU enter ADC Noise Reduction mode, stopping the CPU but allowing the ADC, the external interrupts, the 2-wire Serial Interface address watch, Timer/Counter2[1], and the Watchdog to continue operating (if enabled). This sleep mode basically halts $clk_{I/O}$, $clk_{CPU}$, and $clk_{FLASH}$, while allowing the other clocks to run.

This improves the noise environment for the ADC, enabling higher resolution measurements. If the ADC is enabled, a conversion starts automatically when this mode is entered. Apart from the ADC Conversion Complete interrupt, only an External Reset, a Watchdog System Reset, a Watchdog Interrupt, a Brown-out Reset, a 2-wire Serial Interface address match, a Timer/Counter2 interrupt, an SPM/EEPROM ready interrupt, an external level interrupt on INT0 or INT1 or a pin change interrupt can wake up the MCU from ADC Noise Reduction mode.

Note: 1. Timer/Counter2 will only keep running in asynchronous mode, see "8-bit Timer/Counter2 with PWM and Asynchronous Operation" on page 143 for details.

## 10.5 Power-down Mode

When the SM2...0 bits are written to 010, the SLEEP instruction makes the MCU enter Power-down mode. In this mode, the external Oscillator is stopped, while the external interrupts, the 2-wire Serial Interface address watch, and the Watchdog continue operating (if enabled). Only an External Reset, a Watchdog System Reset, a Watchdog Interrupt, a Brown-out Reset, a 2-wire Serial Interface address match, an external level interrupt on INT0 or INT1, or a pin change interrupt can wake up the MCU. This sleep mode basically halts all generated clocks, allowing operation of asynchronous modules only.

Note: If a level triggered interrupt is used for wake-up from Power-down, the required level must be held long enough for the MCU to complete the wake-up to trigger the level interrupt. If the level disappears before the end of the Start-up Time, the MCU will still wake up, but no interrupt will be generated. "External Interrupts" on page 68. The start-up time is defined by the SUT and CKSEL Fuses as described in "System Clock and Clock Options" on page 26.

When waking up from Power-down mode, there is a delay from the wake-up condition occurs until the wake-up becomes effective. This allows the clock to restart and become stable after having been stopped. The wake-up period is defined by the same CKSEL Fuses that define the Reset Time-out period, as described in "Clock Sources" on page 27.

## 10.6 Power-save Mode

When the SM2...0 bits are written to 011, the SLEEP instruction makes the MCU enter Power-save mode. This mode is identical to Power-down, with one exception:

If Timer/Counter2 is enabled, it will keep running during sleep. The device can wake up from either Timer Overflow or Output Compare event from Timer/Counter2 if the corresponding Timer/Counter2 interrupt enable bits are set in TIMSK2, and the Global Interrupt Enable bit in SREG is set.

If Timer/Counter2 is not running, Power-down mode is recommended instead of Power-save mode.

The Timer/Counter2 can be clocked both synchronously and asynchronously in Power-save mode. If Timer/Counter2 is not using the asynchronous clock, the Timer/Counter Oscillator is stopped during sleep. If Timer/Counter2 is not using the synchronous clock, the clock source is stopped during sleep. Note that even if the synchronous clock is running in Power-save, this clock is only available for Timer/Counter2.

## 10.7 Standby Mode

When the SM2...0 bits are 110 and an external crystal/resonator clock option is selected, the SLEEP instruction makes the MCU enter Standby mode. This mode is identical to Power-down with the exception that the Oscillator is kept running. From Standby mode, the device wakes up in six clock cycles.

## 10.8 Extended Standby Mode

When the SM2...0 bits are 111 and an external crystal/resonator clock option is selected, the SLEEP instruction makes the MCU enter Extended Standby mode. This mode is identical to Power-save with the exception that the Oscillator is kept running. From Extended Standby mode, the device wakes up in six clock cycles.

## 10.9 Power Reduction Register

The Power Reduction Register (PRR), see "PRR – Power Reduction Register" on page 45, provides a method to stop the clock to individual peripherals to reduce power consumption. The current state of the peripheral is frozen and the I/O registers can not be read or written. Resources used by the peripheral when stopping the clock will remain occupied, hence the peripheral should in most cases be disabled before stopping the clock. Waking up a module, which is done by clearing the bit in PRR, puts the module in the same state as before shutdown.

Module shutdown can be used in Idle mode and Active mode to significantly reduce the overall power consumption. In all other sleep modes, the clock is already stopped.

## 10.10 Minimizing Power Consumption

There are several possibilities to consider when trying to minimize the power consumption in an AVR controlled system. In general, sleep modes should be used as much as possible, and the sleep mode should be selected so that as few as possible of the device's functions are operating. All functions not needed should be disabled. In particular, the following modules may need special consideration when trying to achieve the lowest possible power consumption.

### 10.10.1 Analog to Digital Converter

If enabled, the ADC will be enabled in all sleep modes. To save power, the ADC should be disabled before entering any sleep mode. When the ADC is turned off and on again, the next conversion will be an extended conversion. Refer to "Analog-to-Digital Converter" on page 248 for details on ADC operation.

### 10.10.2 Analog Comparator

When entering Idle mode, the Analog Comparator should be disabled if not used. When entering ADC Noise Reduction mode, the Analog Comparator should be disabled. In other sleep modes, the Analog Comparator is automatically disabled. However, if the Analog Comparator is set up to use the Internal Voltage Reference as input, the Analog Comparator should be disabled in all sleep modes. Otherwise, the Internal Voltage Reference will be enabled, independent of sleep mode. Refer to "Analog Comparator" on page 244 for details on how to configure the Analog Comparator.

### 10.10.3 Brown-out Detector

If the Brown-out Detector is not needed by the application, this module should be turned off. If the Brown-out Detector is enabled by the BODLEVEL Fuses, it will be enabled in all sleep modes, and hence, always consume power. In the deeper sleep modes, this will contribute significantly to the total current consumption. Refer to "Brown-out Detection" on page 50 for details on how to configure the Brown-out Detector.

### 10.10.4 Internal Voltage Reference

The Internal Voltage Reference will be enabled when needed by the Brown-out Detection, the Analog Comparator or the ADC. If these modules are disabled as described in the sections above, the internal voltage reference will be disabled and it will not be consuming power. When turned on again, the user must allow the reference to start up before the output is used. If the reference is kept on in sleep mode, the output can be used immediately. Refer to "Internal Voltage Reference" on page 51 for details on the start-up time.

### 10.10.5 Watchdog Timer

If the Watchdog Timer is not needed in the application, the module should be turned off. If the Watchdog Timer is enabled, it will be enabled in all sleep modes and hence always consume power. In the deeper sleep modes, this will contribute significantly to the total current consumption. Refer to "Watchdog Timer" on page 51 for details on how to configure the Watchdog Timer.

### 10.10.6 Port Pins

When entering a sleep mode, all port pins should be configured to use minimum power. The most important is then to ensure that no pins drive resistive loads. In sleep modes where both the I/O clock ($clk_{I/O}$) and the ADC clock ($clk_{ADC}$) are stopped, the input buffers of the device will be disabled. This ensures that no power is consumed by the input logic when not needed. In some cases, the input logic is needed for detecting wake-up conditions, and it will then be enabled. Refer to the section "Digital Input Enable and Sleep Modes" on page 77 for details on which pins are enabled. If the input buffer is enabled and the input signal is left floating or have an analog signal level close to $V_{CC}/2$, the input buffer will use excessive power.

For analog input pins, the digital input buffer should be disabled at all times. An analog signal level close to $V_{CC}/2$ on an input pin can cause significant current even in active mode. Digital input buffers can be disabled by writing to the Digital Input Disable Registers (DIDR1 and DIDR0). Refer to "DIDR1 – Digital Input Disable Register 1" on page 247 and "DIDR0 – Digital Input Disable Register 0" on page 266 for details.

### 10.10.7 On-chip Debug System

If the On-chip debug system is enabled by the DWEN Fuse and the chip enters sleep mode, the main clock source is enabled and hence always consumes power. In the deeper sleep modes, this will contribute significantly to the total current consumption.

## 10.11 Register Description

### 10.11.1 SMCR – Sleep Mode Control Register

The Sleep Mode Control Register contains control bits for power management.

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| 0x33 (0x53) | – | – | – | – | **SM2** | **SM1** | **SM0** | **SE** | **SMCR** |
| Read/Write | R | R | R | R | R/W | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

- **Bits [7:4]: Reserved**

These bits are unused in the Atmel® ATmega48PA/88PA/168PA, and will always be read as zero.

- **Bits 3:1 – SM[2:0]: Sleep Mode Select Bits 2, 1, and 0**

These bits select between the five available sleep modes as shown in Table 10-2.

**Table 10-2.** Sleep Mode Select

| SM2 | SM1 | SM0 | Sleep Mode |
|---|---|---|---|
| 0 | 0 | 0 | Idle |
| 0 | 0 | 1 | ADC Noise Reduction |
| 0 | 1 | 0 | Power-down |
| 0 | 1 | 1 | Power-save |
| 1 | 0 | 0 | Reserved |
| 1 | 0 | 1 | Reserved |
| 1 | 1 | 0 | Standby[1] |
| 1 | 1 | 1 | External Standby[1] |

Note: [1] Standby mode is only recommended for use with external crystals or resonators.

- **Bit 0 – SE: Sleep Enable**

The SE bit must be written to logic one to make the MCU enter the sleep mode when the SLEEP instruction is executed. To avoid the MCU entering the sleep mode unless it is the programmer’s purpose, it is recommended to write the Sleep Enable (SE) bit to one just before the execution of the SLEEP instruction and to clear it immediately after waking up.

### 10.11.2 MCUCR – MCU Control Register

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| 0x35 (0x55) | – | BODS() | BODSE() | PUD | – | – | IVSEL | IVCE | MCUCR |
| Read/Write | R | R/W | R/W | R/W | R | R | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

- **Bit 6 – BODS: BOD Sleep()**

The BODS bit must be written to logic one in order to turn off BOD during sleep, see Table 10-1 on page 39. Writing to the BODS bit is controlled by a timed sequence and an enable bit, BODSE in MCUCR. To disable BOD in relevant sleep modes, both BODS and BODSE must first be set to one. Then, to set the BODS bit, BODS must be set to one and BODSE must be set to zero within four clock cycles.

The BODS bit is active three clock cycles after it is set. A sleep instruction must be executed while BODS is active in order to turn off the BOD for the actual sleep mode. The BODS bit is automatically cleared after three clock cycles.

- **Bit 5 – BODSE: BOD Sleep Enable()**

BODSE enables setting of BODS control bit, as explained in BODS bit description. BOD disable is controlled by a timed sequence.

Note: 1. BODS and BODSE only available for picoPower devices ATmega48PA/88PA/168PA

### 10.11.3 PRR – Power Reduction Register

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| (0x64) | PRTWI | PRTIM2 | PRTIM0 | – | PRTIM1 | PRSPI | PRUSART0 | PRADC | PRR |
| Read/Write | R/W | R/W | R/W | R | R/W | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

- **Bit 7 – PRTWI: Power Reduction TWI**

Writing a logic one to this bit shuts down the TWI by stopping the clock to the module. When waking up the TWI again, the TWI should be re initialized to ensure proper operation.

- **Bit 6 – PRTIM2: Power Reduction Timer/Counter2**

Writing a logic one to this bit shuts down the Timer/Counter2 module in synchronous mode (AS2 is 0). When the Timer/Counter2 is enabled, operation will continue like before the shutdown.

- **Bit 5 – PRTIM0: Power Reduction Timer/Counter0**

Writing a logic one to this bit shuts down the Timer/Counter0 module. When the Timer/Counter0 is enabled, operation will continue like before the shutdown.

- **Bit 4 – Reserved**

This bit is reserved in Atmel® ATmega48PA/88PA/168PA and will always read as zero.

- **Bit 3 – PRTIM1: Power Reduction Timer/Counter1**

Writing a logic one to this bit shuts down the Timer/Counter1 module. When the Timer/Counter1 is enabled, operation will continue like before the shutdown.

- **Bit 2 – PRSPI: Power Reduction Serial Peripheral Interface**

If using debugWIRE On-chip Debug System, this bit should not be written to one.

Writing a logic one to this bit shuts down the Serial Peripheral Interface by stopping the clock to the module. When waking up the SPI again, the SPI should be re initialized to ensure proper operation.

- **Bit 1 – PRUSART0: Power Reduction USART0**

Writing a logic one to this bit shuts down the USART by stopping the clock to the module. When waking up the USART again, the USART should be re initialized to ensure proper operation.

- **Bit 0 – PRADC: Power Reduction ADC**

Writing a logic one to this bit shuts down the ADC. The ADC must be disabled before shut down. The analog comparator cannot use the ADC input MUX when the ADC is shut down.

# 11. System Control and Reset

## 11.1 Resetting the AVR

During reset, all I/O Registers are set to their initial values, and the program starts execution from the Reset Vector. For Atmel® ATmega168PA the instruction placed at the Reset Vector must be a JMP – Absolute Jump – instruction to the reset handling routine. For the Atmel ATmega48PA and Atmel ATmega88PA, the instruction placed at the Reset Vector must be an RJMP – Relative Jump – instruction to the reset handling routine. If the program never enables an interrupt source, the Interrupt Vectors are not used, and regular program code can be placed at these locations. This is also the case if the Reset Vector is in the Application section while the Interrupt Vectors are in the Boot section or vice versa (Atmel ATmega88PA/168PA only). The circuit diagram in Figure 11-1 on page 48 shows the reset logic. Table 29-5 on page 317 defines the electrical parameters of the reset circuitry.

The I/O ports of the AVR are immediately reset to their initial state when a reset source goes active. This does not require any clock source to be running.

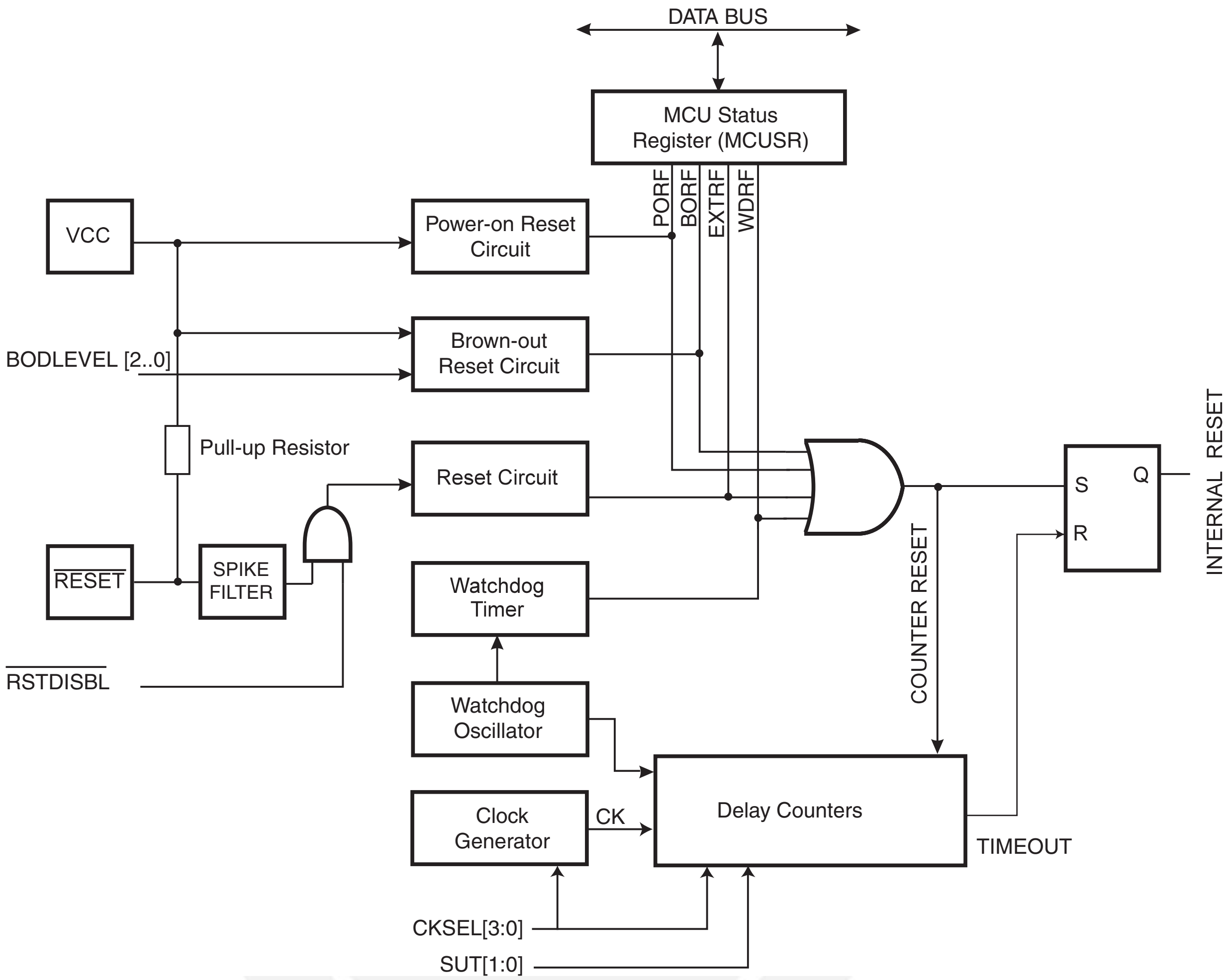
After all reset sources have gone inactive, a delay counter is invoked, stretching the internal reset. This allows the power to reach a stable level before normal operation starts. The time-out period of the delay counter is defined by the user through the SUT and CKSEL Fuses. The different selections for the delay period are presented in "Clock Sources" on page 27.

## 11.2 Reset Sources

The Atmel® ATmega48PA/88PA/168PA has four sources of reset:

- Power-on Reset. The MCU is reset when the supply voltage is below the Power-on Reset threshold ($V_{POT}$).
- External Reset. The MCU is reset when a low level is present on the $\overline{\text{RESET}}$ pin for longer than the minimum pulse length.
- Watchdog System Reset. The MCU is reset when the Watchdog Timer period expires and the Watchdog System Reset mode is enabled.
- Brown-out Reset. The MCU is reset when the supply voltage $V_{CC}$ is below the Brown-out Reset threshold ($V_{BOT}$) and the Brown-out Detector is enabled.

**Figure 11-1.** Reset Logic



## 11.3 Power-on Reset

A Power-on Reset (POR) pulse is generated by an On-chip detection circuit. The detection level is defined in "System and Reset Characteristics" on page 317. The POR is activated whenever $V_{CC}$ is below the detection level. The POR circuit can be used to trigger the start-up Reset, as well as to detect a failure in supply voltage.

A Power-on Reset (POR) circuit ensures that the device is reset from Power-on. Reaching the Power-on Reset threshold voltage invokes the delay counter, which determines how long the device is kept in RESET after $V_{CC}$ rise. The RESET signal is activated again, without any delay, when $V_{CC}$ decreases below the detection level.

**Figure 11-2.** MCU Start-up, $\overline{\text{RESET}}$ Tied to $V_{CC}$

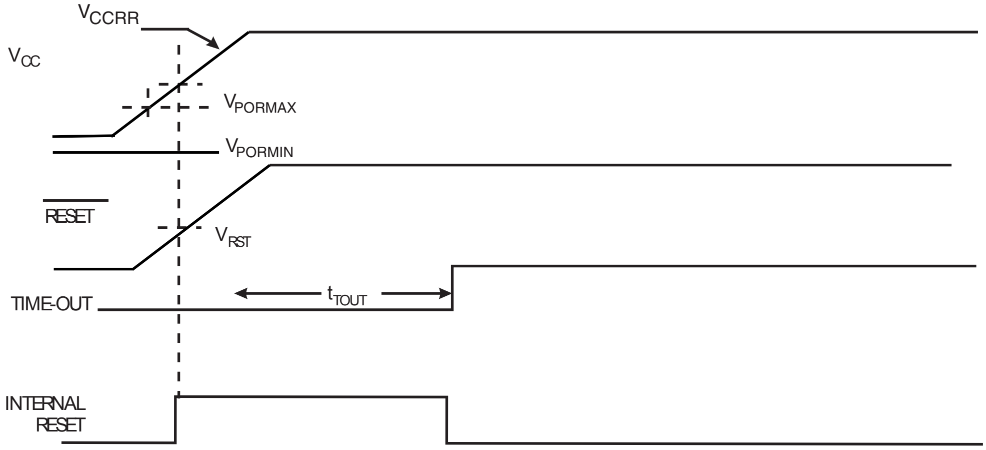
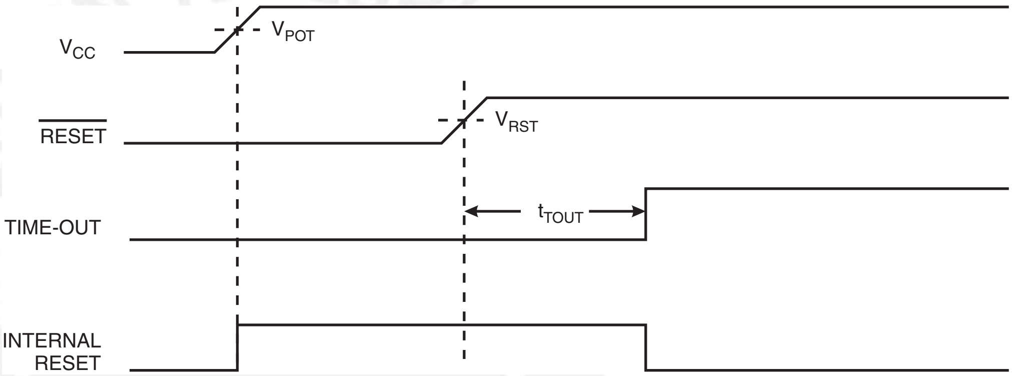


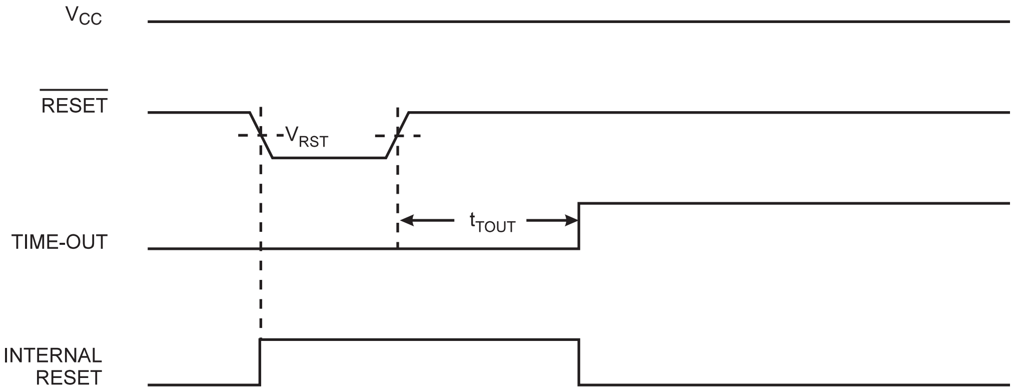
**Figure 11-3.** MCU Start-up, $\overline{\text{RESET}}$ Extended Externally



## 11.4 External Reset

An External Reset is generated by a low level on the $\overline{\text{RESET}}$ pin. Reset pulses longer than the minimum pulse width (see "System and Reset Characteristics" on page 317) will generate a reset, even if the clock is not running. Shorter pulses are not guaranteed to generate a reset. When the applied signal reaches the Reset Threshold Voltage – $V_{RST}$ – on its positive edge, the delay counter starts the MCU after the Time-out period – $t_{TOUT}$ – has expired. The External Reset can be disabled by the RSTDISBL fuse, see Table 28-6 on page 296.

**Figure 11-4.** External Reset During Operation

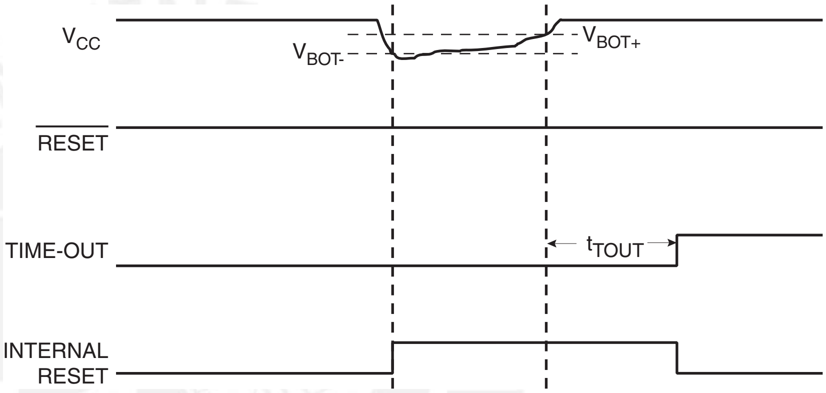


## 11.5 Brown-out Detection

The Atmel® ATmega48PA/88PA/168PA has an On-chip Brown-out Detection (BOD) circuit for monitoring the $V_{CC}$ level during operation by comparing it to a fixed trigger level. The trigger level for the BOD can be selected by the BODLEVEL Fuses. The trigger level has a hysteresis to ensure spike free Brown-out Detection. The hysteresis on the detection level should be interpreted as $V_{BOT+} = V_{BOT} + V_{HYST}/2$ and $V_{BOT-} = V_{BOT} - V_{HYST}/2$.When the BOD is enabled, and $V_{CC}$ decreases to a value below the trigger level ($V_{BOT-}$ in Figure 11-5 on page 50), the Brown-out Reset is immediately activated. When $V_{CC}$ increases above the trigger level ($V_{BOT+}$ in Figure 11-5 on page 50), the delay counter starts the MCU after the Time-out period $t_{TOUT}$ has expired.

The BOD circuit will only detect a drop in $V_{CC}$ if the voltage stays below the trigger level for longer than $t_{BOD}$ given in "System and Reset Characteristics" on page 317.

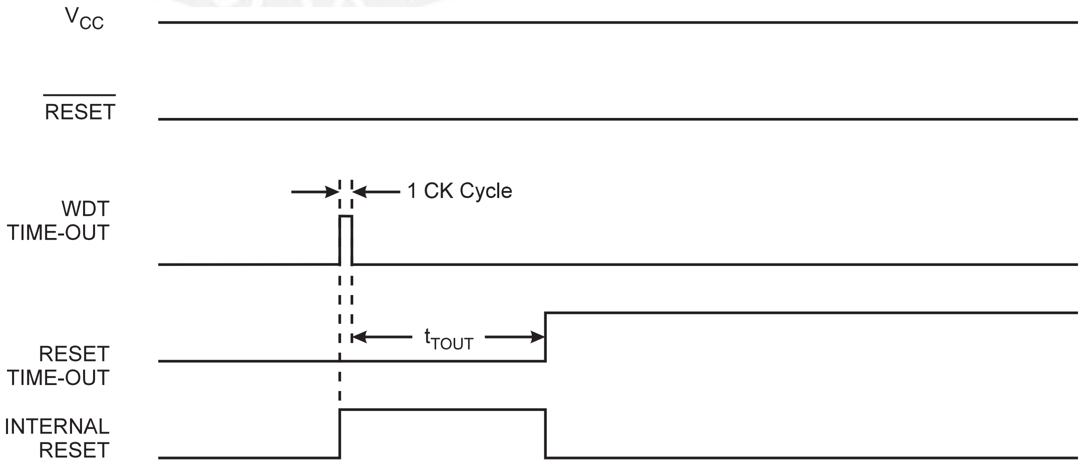
**Figure 11-5.** Brown-out Reset During Operation



## 11.6 Watchdog System Reset

When the Watchdog times out, it will generate a short reset pulse of one CK cycle duration. On the falling edge of this pulse, the delay timer starts counting the Time-out period $t_{TOUT}$. Refer to page 51 for details on operation of the Watchdog Timer.

**Figure 11-6.** Watchdog System Reset During Operation



## 11.7 Internal Voltage Reference

The Atmel® ATmega48PA/88PA/168PA features an internal bandgap reference. This reference is used for Brown-out Detection, and it can be used as an input to the Analog Comparator or the ADC.

### 11.7.1 Voltage Reference Enable Signals and Start-up Time

The voltage reference has a start-up time that may influence the way it should be used. The start-up time is given in "System and Reset Characteristics" on page 317. To save power, the reference is not always turned on. The reference is on during the following situations:

1. When the BOD is enabled (by programming the BODLEVEL [2:0] Fuses).
2. When the bandgap reference is connected to the Analog Comparator (by setting the ACBG bit in ACSR).
3. When the ADC is enabled.

Thus, when the BOD is not enabled, after setting the ACBG bit or enabling the ADC, the user must always allow the reference to start up before the output from the Analog Comparator or ADC is used. To reduce power consumption in Power-down mode, the user can avoid the three conditions above to ensure that the reference is turned off before entering Power-down mode.

## 11.8 Watchdog Timer

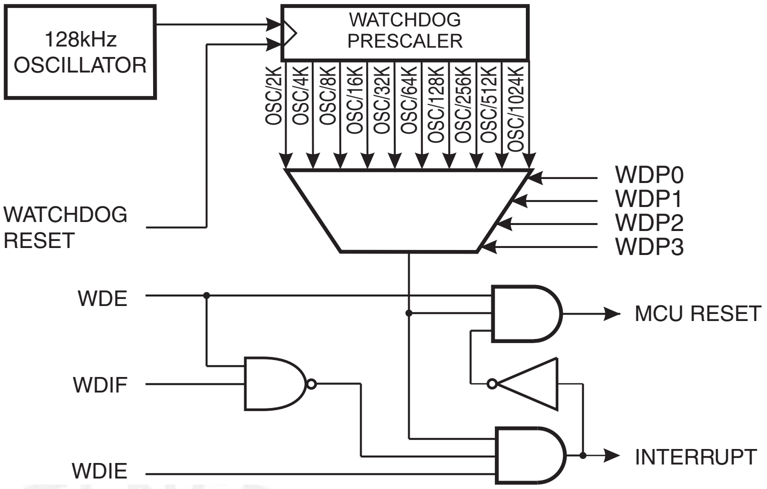
### 11.8.1 Features

- **Clocked from separate On-chip Oscillator**
- **3 Operating modes**
  - **Interrupt**
  - **System Reset**
  - **Interrupt and System Reset**
- **Selectable Time-out period from 16ms to 8s**
- **Possible Hardware fuse Watchdog always on (WDTON) for fail-safe mode**

### 11.8.2 Overview

The Atmel ATmega48PA/88PA/168PA has an Enhanced Watchdog Timer (WDT). The WDT is a timer counting cycles of a separate on-chip 128kHz oscillator. The WDT gives an interrupt or a system reset when the counter reaches a given time-out value. In normal operation mode, it is required that the system uses the WDR - Watchdog Timer Reset - instruction to restart the counter before the time-out value is reached. If the system doesn't restart the counter, an interrupt or system reset will be issued.

Figure 11-7. Watchdog Timer



In Interrupt mode, the WDT gives an interrupt when the timer expires. This interrupt can be used to wake the device from sleep-modes, and also as a general system timer. One example is to limit the maximum time allowed for certain operations, giving an interrupt when the operation has run longer than expected. In System Reset mode, the WDT gives a reset when the timer expires. This is typically used to prevent system hang-up in case of runaway code. The third mode, Interrupt and System Reset mode, combines the other two modes by first giving an interrupt and then switch to System Reset mode. This mode will for instance allow a safe shutdown by saving critical parameters before a system reset.

The Watchdog always on (WDTON) fuse, if programmed, will force the Watchdog Timer to System Reset mode. With the fuse programmed the System Reset mode bit (WDE) and Interrupt mode bit (WDIE) are locked to 1 and 0 respectively. To further ensure program security, alterations to the Watchdog set-up must follow timed sequences. The sequence for clearing WDE and changing time-out configuration is as follows:

1. In the same operation, write a logic one to the Watchdog change enable bit (WDCE) and WDE. A logic one must be written to WDE regardless of the previous value of the WDE bit.
2. Within the next four clock cycles, write the WDE and Watchdog prescaler bits (WDP) as desired, but with the WDCE bit cleared. This must be done in one operation.

The following code example shows one assembly and one C function for turning off the Watchdog Timer. The example assumes that interrupts are controlled (e.g. by disabling interrupts globally) so that no interrupts will occur during the execution of these functions.

Assembly Code Example[1]

```
WDT_off:
  ; Turn off global interrupt
  cli
  ; Reset Watchdog Timer
  wdr
  ; Clear WDRF in MCUSR
  in    r16, MCUSR
  andi  r16, (0xff & (0<<WDRF))
  out   MCUSR, r16
  ; Write logical one to WDCE and WDE
  ; Keep old prescaler setting to prevent unintentional time-out
  lds r16, WDTCSR
  ori   r16, (1<<WDCE) | (1<<WDE)
  sts WDTCSR, r16
  ; Turn off WDT
  ldi   r16, (0<<WDE)
  sts WDTCSR, r16
  ; Turn on global interrupt
  sei
  ret
```

C Code Example[1]

```
void WDT_off(void)
{
  __disable_interrupt();
  __watchdog_reset();
  /* Clear WDRF in MCUSR */
  MCUSR &= ~(1<<WDRF);
  /* Write logical one to WDCE and WDE */
  /* Keep old prescaler setting to prevent unintentional time-out */
  WDTCSR |= (1<<WDCE) | (1<<WDE);
  /* Turn off WDT */
  WDTCSR = 0x00;
  __enable_interrupt();
}
```

Notes:
1. See Section 6. “About Code Examples” on page 7
2. If the Watchdog is accidentally enabled, for example by a runaway pointer or brown-out condition, the device will be reset and the Watchdog Timer will stay enabled. If the code is not set up to handle the Watchdog, this might lead to an eternal loop of time-out resets. To avoid this situation, the application software should always clear the Watchdog System Reset Flag (WDRF) and the WDE control bit in the initialization routine, even if the Watchdog is not in use.

The following code example shows one assembly and one C function for changing the time-out value of the Watchdog Timer.

Assembly Code Example[1]

```
WDT_Prescaler_Change:
  ; Turn off global interrupt
  cli
  ; Reset Watchdog Timer
  wdr
  ; Start timed sequence
  lds r16, WDTCSR
  ori   r16, (1<<WDCE) | (1<<WDE)
  sts WDTCSR, r16
  ; --  Got four cycles to set the new values from here -
  ; Set new prescaler(time-out) value = 64K cycles (~0.5 s)
  ldi   r16, (1<<WDE) | (1<<WDP2) | (1<<WDP0)
  sts WDTCSR, r16
  ; --  Finished setting new values, used 2 cycles -
  ; Turn on global interrupt
  sei
  ret
```

C Code Example[1]

```
void WDT_Prescaler_Change(void)
{
  __disable_interrupt();
  __watchdog_reset();
  /* Start timed  sequence */
  WDTCSR |= (1<<WDCE) | (1<<WDE);
  /* Set new prescaler(time-out) value = 64K cycles (~0.5 s) */
  WDTCSR  = (1<<WDE) | (1<<WDP2) | (1<<WDP0);
  __enable_interrupt();
}
```

Notes:

1. See Section 6. “About Code Examples” on page 7
2. The Watchdog Timer should be reset before any change of the WDP bits, since a change in the WDP bits can result in a time-out when switching to a shorter time-out period.

## 11.9 Register Description

### 11.9.1 MCUSR – MCU Status Register

The MCU Status Register provides information on which reset source caused an MCU reset.

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| 0x34 (0x54) | – | – | – | – | WDRF | BORF | EXTRF | PORF | MCUSR |
| Read/Write | R | R | R | R | R/W | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | See Bit Description | | | | |

- **Bit 7:4: Reserved**

These bits are unused bits in the Atmel® ATmega48PA/88PA/168PA, and will always read as zero.

- **Bit 3 – WDRF: Watchdog System Reset Flag**

This bit is set if a Watchdog System Reset occurs. The bit is reset by a Power-on Reset, or by writing a logic zero to the flag.

- **Bit 2 – BORF: Brown-out Reset Flag**

This bit is set if a Brown-out Reset occurs. The bit is reset by a Power-on Reset, or by writing a logic zero to the flag.

- **Bit 1 – EXTRF: External Reset Flag**

This bit is set if an External Reset occurs. The bit is reset by a Power-on Reset, or by writing a logic zero to the flag.

- **Bit 0 – PORF: Power-on Reset Flag**

This bit is set if a Power-on Reset occurs. The bit is reset only by writing a logic zero to the flag.

To make use of the Reset Flags to identify a reset condition, the user should read and then Reset the MCUSR as early as possible in the program. If the register is cleared before another reset occurs, the source of the reset can be found by examining the Reset Flags.

### 11.9.2 WDTCSR – Watchdog Timer Control Register

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| (0x60) | WDIF | WDIE | WDP3 | WDCE | WDE | WDP2 | WDP1 | WDP0 | WDTCSR |
| Read/Write | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | X | 0 | 0 | 0 | |

- **Bit 7 – WDIF: Watchdog Interrupt Flag**

This bit is set when a time-out occurs in the Watchdog Timer and the Watchdog Timer is configured for interrupt. WDIF is cleared by hardware when executing the corresponding interrupt handling vector. Alternatively, WDIF is cleared by writing a logic one to the flag. When the I-bit in SREG and WDIE are set, the Watchdog Time-out Interrupt is executed.

- **Bit 6 – WDIE: Watchdog Interrupt Enable**

When this bit is written to one and the I-bit in the Status Register is set, the Watchdog Interrupt is enabled. If WDE is cleared in combination with this setting, the Watchdog Timer is in Interrupt Mode, and the corresponding interrupt is executed if time-out in the Watchdog Timer occurs. If WDE is set, the Watchdog Timer is in Interrupt and System Reset Mode.

The first time-out in the Watchdog Timer will set WDIF. Executing the corresponding interrupt vector will clear WDIE and WDIF automatically by hardware (the Watchdog goes to System Reset Mode). This is useful for keeping the Watchdog Timer security while using the interrupt. To stay in Interrupt and System Reset Mode, WDIE must be set after each interrupt. This should however not be done within the interrupt service routine itself, as this might compromise the safety-function of the Watchdog System Reset mode. If the interrupt is not executed before the next time-out, a System Reset will be applied.

**Table 11-1.** Watchdog Timer Configuration

| WDTON[1] | WDE | WDIE | Mode | Action on Time-out |
|---|---|---|---|---|
| 1 | 0 | 0 | Stopped | None |
| 1 | 0 | 1 | Interrupt Mode | Interrupt |
| 1 | 1 | 0 | System Reset Mode | Reset |
| 1 | 1 | 1 | Interrupt and System Reset Mode | Interrupt, then go to System Reset Mode |
| 0 | x | x | System Reset Mode | Reset |

Note: 1. WDTON Fuse set to "0" means programmed and "1" means unprogrammed.

- **Bit 4 – WDCE: Watchdog Change Enable**

This bit is used in timed sequences for changing WDE and prescaler bits. To clear the WDE bit, and/or change the prescaler bits, WDCE must be set.

Once written to one, hardware will clear WDCE after four clock cycles.

- **Bit 3 – WDE: Watchdog System Reset Enable**

WDE is overridden by WDRF in MCUSR. This means that WDE is always set when WDRF is set. To clear WDE, WDRF must be cleared first. This feature ensures multiple resets during conditions causing failure, and a safe start-up after the failure.

- **Bit 5, 2:0 - WDP[3:0]: Watchdog Timer Prescaler 3, 2, 1 and 0**

The WDP[3:0] bits determine the Watchdog Timer prescaling when the Watchdog Timer is running. The different prescaling values and their corresponding time-out periods are shown in Table 11-2 on page 57.

**Table 11-2.** Watchdog Timer Prescale Select

| WDP3 | WDP2 | WDP1 | WDP0 | Number of WDT Oscillator Cycles | Typical Time-out at $V_{CC}$ = 5.0V |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 2K (2048) cycles | 16ms |
| 0 | 0 | 0 | 1 | 4K (4096) cycles | 32ms |
| 0 | 0 | 1 | 0 | 8K (8192) cycles | 64ms |
| 0 | 0 | 1 | 1 | 16K (16384) cycles | 0.125s |
| 0 | 1 | 0 | 0 | 32K (32768) cycles | 0.25s |
| 0 | 1 | 0 | 1 | 64K (65536) cycles | 0.5s |
| 0 | 1 | 1 | 0 | 128K (131072) cycles | 1.0s |
| 0 | 1 | 1 | 1 | 256K (262144) cycles | 2.0s |
| 1 | 0 | 0 | 0 | 512K (524288) cycles | 4.0s |
| 1 | 0 | 0 | 1 | 1024K (1048576) cycles | 8.0s |
| 1 | 0 | 1 | 0 | Reserved | |
| 1 | 0 | 1 | 1 | | |
| 1 | 1 | 0 | 0 | | |
| 1 | 1 | 0 | 1 | | |
| 1 | 1 | 1 | 0 | | |
| 1 | 1 | 1 | 1 | | |

# 12. Interrupts

This section describes the specifics of the interrupt handling as performed in the Atmel® ATmega48PA/88PA/168PA. For a general explanation of the AVR interrupt handling, refer to "Reset and Interrupt Handling" on page 14.

The interrupt vectors in the Atmel ATmega48PA, Atmel ATmega88PA, and ATmega168PA are generally the same, with the following differences:

- Each Interrupt Vector occupies two instruction words in Atmel ATmega168PA and one instruction word in the Atmel ATmega48PA and Atmel ATmega88PA.
- Atmel ATmega48PA does not have a separate Boot Loader Section. In the Atmel ATmega88PA, and Atmel ATmega168PA, the Reset Vector is affected by the BOOTRST fuse, and the Interrupt Vector start address is affected by the IVSEL bit in MCUCR.

## 12.1 Interrupt Vectors in Atmel ATmega48PA

**Table 12-1.** Reset and Interrupt Vectors in ATmega48PA

| Vector No. | Program Address | Source | Interrupt Definition |
|---|---|---|---|
| 1 | 0x000 | RESET | External Pin, Power-on Reset, Brown-out Reset and Watchdog System Reset |
| 2 | 0x001 | INT0 | External Interrupt Request 0 |
| 3 | 0x002 | INT1 | External Interrupt Request 1 |
| 4 | 0x003 | PCINT0 | Pin Change Interrupt Request 0 |
| 5 | 0x004 | PCINT1 | Pin Change Interrupt Request 1 |
| 6 | 0x005 | PCINT2 | Pin Change Interrupt Request 2 |
| 7 | 0x006 | WDT | Watchdog Time-out Interrupt |
| 8 | 0x007 | TIMER2 COMPA | Timer/Counter2 Compare Match A |
| 9 | 0x008 | TIMER2 COMPB | Timer/Counter2 Compare Match B |
| 10 | 0x009 | TIMER2 OVF | Timer/Counter2 Overflow |
| 11 | 0x00A | TIMER1 CAPT | Timer/Counter1 Capture Event |
| 12 | 0x00B | TIMER1 COMPA | Timer/Counter1 Compare Match A |
| 13 | 0x00C | TIMER1 COMPB | Timer/Coutner1 Compare Match B |
| 14 | 0x00D | TIMER1 OVF | Timer/Counter1 Overflow |
| 15 | 0x00E | TIMER0 COMPA | Timer/Counter0 Compare Match A |
| 16 | 0x00F | TIMER0 COMPB | Timer/Counter0 Compare Match B |
| 17 | 0x010 | TIMER0 OVF | Timer/Counter0 Overflow |
| 18 | 0x011 | SPI, STC | SPI Serial Transfer Complete |
| 19 | 0x012 | USART, RX | USART Rx Complete |
| 20 | 0x013 | USART, UDRE | USART, Data Register Empty |
| 21 | 0x014 | USART, TX | USART, Tx Complete |
| 22 | 0x015 | ADC | ADC Conversion Complete |
| 23 | 0x016 | EE READY | EEPROM Ready |
| 24 | 0x017 | ANALOG COMP | Analog Comparator |
| 25 | 0x018 | TWI | 2-wire Serial Interface |
| 26 | 0x019 | SPM READY | Store Program Memory Ready |

The most typical and general program setup for the Reset and Interrupt Vector Addresses in Atmel® ATmega48PA is:

```
Address  Labels Code                 Comments
0x000           rjmp   RESET         ; Reset Handler
0x001           rjmp   EXT_INT0      ; IRQ0 Handler
0x002           rjmp   EXT_INT1      ; IRQ1 Handler
0x003           rjmp   PCINT0        ; PCINT0 Handler
0x004           rjmp   PCINT1        ; PCINT1 Handler
0x005           rjmp   PCINT2        ; PCINT2 Handler
0x006           rjmp   WDT           ; Watchdog Timer Handler
0x007           rjmp   TIM2_COMPA    ; Timer2 Compare A Handler
0x008           rjmp   TIM2_COMPB    ; Timer2 Compare B Handler
0x009           rjmp   TIM2_OVF      ; Timer2 Overflow Handler
0x00A           rjmp   TIM1_CAPT     ; Timer1 Capture Handler
0x00B           rjmp   TIM1_COMPA    ; Timer1 Compare A Handler
0x00C           rjmp   TIM1_COMPB    ; Timer1 Compare B Handler
0x00D           rjmp   TIM1_OVF      ; Timer1 Overflow Handler
0x00E           rjmp   TIM0_COMPA    ; Timer0 Compare A Handler
0x00F           rjmp   TIM0_COMPB    ; Timer0 Compare B Handler
0x010           rjmp   TIM0_OVF      ; Timer0 Overflow Handler
0x011           rjmp   SPI_STC       ; SPI Transfer Complete Handler
0x012           rjmp   USART_RXC     ; USART, RX Complete Handler
0x013           rjmp   USART_UDRE    ; USART, UDR Empty Handler
0x014           rjmp   USART_TXC     ; USART, TX Complete Handler
0x015           rjmp   ADC           ; ADC Conversion Complete Handler
0x016           rjmp   EE_RDY        ; EEPROM Ready Handler
0x017           rjmp   ANA_COMP      ; Analog Comparator Handler
0x018           rjmp   TWI           ; 2-wire Serial Interface Handler
0x019           rjmp   SPM_RDY       ; Store Program Memory Ready Handler
;
0x01ARESET:     ldi    r16, high(RAMEND); Main program start
0x01B           out    SPH,r16       ; Set Stack Pointer to top of RAM
0x01C           ldi    r16, low(RAMEND)
0x01D           out    SPL,r16
0x01E           sei                  ; Enable interrupts
0x01F           <instr>  xxx
  ...     ...     ...  ...
```

## 12.2 Interrupt Vectors in the Atmel ATmega88PA

**Table 12-2.** Reset and Interrupt Vectors in the Atmel ATmega88PA

| Vector No. | Program Address[2] | Source | Interrupt Definition |
|---|---|---|---|
| 1 | 0x000[1] | RESET | External Pin, Power-on Reset, Brown-out Reset and Watchdog System Reset |
| 2 | 0x001 | INT0 | External Interrupt Request 0 |
| 3 | 0x002 | INT1 | External Interrupt Request 1 |
| 4 | 0x003 | PCINT0 | Pin Change Interrupt Request 0 |
| 5 | 0x004 | PCINT1 | Pin Change Interrupt Request 1 |
| 6 | 0x005 | PCINT2 | Pin Change Interrupt Request 2 |
| 7 | 0x006 | WDT | Watchdog Time-out Interrupt |
| 8 | 0x007 | TIMER2 COMPA | Timer/Counter2 Compare Match A |
| 9 | 0x008 | TIMER2 COMPB | Timer/Counter2 Compare Match B |
| 10 | 0x009 | TIMER2 OVF | Timer/Counter2 Overflow |
| 11 | 0x00A | TIMER1 CAPT | Timer/Counter1 Capture Event |
| 12 | 0x00B | TIMER1 COMPA | Timer/Counter1 Compare Match A |
| 13 | 0x00C | TIMER1 COMPB | Timer/Coutner1 Compare Match B |
| 14 | 0x00D | TIMER1 OVF | Timer/Counter1 Overflow |
| 15 | 0x00E | TIMER0 COMPA | Timer/Counter0 Compare Match A |
| 16 | 0x00F | TIMER0 COMPB | Timer/Counter0 Compare Match B |
| 17 | 0x010 | TIMER0 OVF | Timer/Counter0 Overflow |
| 18 | 0x011 | SPI, STC | SPI Serial Transfer Complete |
| 19 | 0x012 | USART, RX | USART Rx Complete |
| 20 | 0x013 | USART, UDRE | USART, Data Register Empty |
| 21 | 0x014 | USART, TX | USART, Tx Complete |
| 22 | 0x015 | ADC | ADC Conversion Complete |
| 23 | 0x016 | EE READY | EEPROM Ready |
| 24 | 0x017 | ANALOG COMP | Analog Comparator |
| 25 | 0x018 | TWI | 2-wire Serial Interface |
| 26 | 0x019 | SPM READY | Store Program Memory Ready |

Notes: 1. When the BOOTRST Fuse is programmed, the device will jump to the Boot Loader address at reset, see "Boot Loader Support – Read-While-Write Self-Programming" on page 277.

2. When the IVSEL bit in MCUCR is set, Interrupt Vectors will be moved to the start of the Boot Flash Section. The address of each Interrupt Vector will then be the address in this table added to the start address of the Boot Flash Section.

Table 12-3 on page 61 shows reset and Interrupt Vectors placement for the various combinations of BOOTRST and IVSEL settings. If the program never enables an interrupt source, the Interrupt Vectors are not used, and regular program code can be placed at these locations. This is also the case if the Reset Vector is in the Application section while the Interrupt Vectors are in the Boot section or vice versa.

**Table 12-3.** Reset and Interrupt Vectors Placement in the Atmel ATmega88PA[1]

| BOOTRST | IVSEL | Reset Address | Interrupt Vectors Start Address |
|---|---|---|---|
| 1 | 0 | 0x000 | 0x001 |
| 1 | 1 | 0x000 | Boot Reset Address + 0x001 |
| 0 | 0 | Boot Reset Address | 0x001 |
| 0 | 1 | Boot Reset Address | Boot Reset Address + 0x001 |

Note: 1. The Boot Reset Address is shown in Table 27-7 on page 290. For the BOOTRST Fuse “1” means unprogrammed while “0” means programmed.

The most typical and general program setup for the Reset and Interrupt Vector Addresses in the Atmel® ATmega88PA is:

```
Address  Labels Code                 Comments
0x000          rjmp   RESET          ; Reset Handler
0x001          rjmp   EXT_INT0       ; IRQ0 Handler
0x002          rjmp   EXT_INT1       ; IRQ1 Handler
0x003          rjmp   PCINT0         ; PCINT0 Handler
0x004          rjmp   PCINT1         ; PCINT1 Handler
0x005          rjmp   PCINT2         ; PCINT2 Handler
0x006          rjmp   WDT            ; Watchdog Timer Handler
0x007          rjmp   TIM2_COMPA     ; Timer2 Compare A Handler
0X008          rjmp   TIM2_COMPB     ; Timer2 Compare B Handler
0x009          rjmp   TIM2_OVF       ; Timer2 Overflow Handler
0x00A          rjmp   TIM1_CAPT      ; Timer1 Capture Handler
0x00B          rjmp   TIM1_COMPA     ; Timer1 Compare A Handler
0x00C          rjmp   TIM1_COMPB     ; Timer1 Compare B Handler
0x00D          rjmp   TIM1_OVF       ; Timer1 Overflow Handler
0x00E          rjmp   TIM0_COMPA     ; Timer0 Compare A Handler
0x00F          rjmp   TIM0_COMPB     ; Timer0 Compare B Handler
0x010          rjmp   TIM0_OVF       ; Timer0 Overflow Handler
0x011          rjmp   SPI_STC        ; SPI Transfer Complete Handler
0x012          rjmp   USART_RXC      ; USART, RX Complete Handler
0x013          rjmp   USART_UDRE     ; USART, UDR Empty Handler
0x014          rjmp   USART_TXC      ; USART, TX Complete Handler
0x015          rjmp   ADC            ; ADC Conversion Complete Handler
0x016          rjmp   EE_RDY         ; EEPROM Ready Handler
0x017          rjmp   ANA_COMP       ; Analog Comparator Handler
0x018          rjmp   TWI            ; 2-wire Serial Interface Handler
0x019          rjmp   SPM_RDY        ; Store Program Memory Ready Handler
;
0x01ARESET:    ldi    r16, high(RAMEND); Main program start
0x01B          out    SPH,r16        ; Set Stack Pointer to top of RAM
0x01C          ldi    r16, low(RAMEND)
0x01D          out    SPL,r16
0x01E          sei                   ; Enable interrupts
0x01F          <instr>  xxx
```

When the BOOTRST Fuse is unprogrammed, the Boot section size set to 2K bytes and the IVSEL bit in the MCUCR Register is set before any interrupts are enabled, the most typical and general program setup for the Reset and Interrupt Vector Addresses in the Atmel® ATmega88PA is:

```
Address  Labels Code                  Comments
0x000    RESET: ldi    r16,high(RAMEND); Main program start
0x001           out    SPH,r16        ; Set Stack Pointer to top of RAM
0x002           ldi    r16,low(RAMEND)
0x003           out    SPL,r16
0x004           sei                   ; Enable interrupts
0x005           <instr>  xxx
;
.org 0xC01
0xC01           rjmp   EXT_INT0       ; IRQ0 Handler
0xC02           rjmp   EXT_INT1       ; IRQ1 Handler
...             ...    ...            ;
0xC19           rjmp   SPM_RDY        ; Store Program Memory Ready Handler
```

When the BOOTRST Fuse is programmed and the Boot section size set to 2K bytes, the most typical and general program setup for the Reset and Interrupt Vector Addresses in the Atmel ATmega88PA is:

```
Address  Labels Code             Comments
.org 0x001
0x001           rjmp   EXT_INT0       ; IRQ0 Handler
0x002           rjmp   EXT_INT1       ; IRQ1 Handler
...             ...    ...            ;
0x019           rjmp   SPM_RDY        ; Store Program Memory Ready Handler
;
.org 0xC00
0xC00    RESET: ldi    r16,high(RAMEND); Main program start
0xC01           out    SPH,r16        ; Set Stack Pointer to top of RAM
0xC02           ldi    r16,low(RAMEND)
0xC03           out    SPL,r16
0xC04           sei                   ; Enable interrupts
0xC05           <instr>  xxx
```

When the BOOTRST Fuse is programmed, the Boot section size set to 2K bytes and the IVSEL bit in the MCUCR Register is set before any interrupts are enabled, the most typical and general program setup for the Reset and Interrupt Vector Addresses in the Atmel ATmega88PA is:

```
Address  Labels Code                  Comments
;
.org 0xC00
0xC00           rjmp   RESET          ; Reset handler
0xC01           rjmp   EXT_INT0       ; IRQ0 Handler
0xC02           rjmp   EXT_INT1       ; IRQ1 Handler
...             ...    ...            ;
```

```
0xC19          rjmp   SPM_RDY        ; Store Program Memory Ready Handler
;
0xC1A    RESET: ldi    r16,high(RAMEND); Main program start
0xC1B          out    SPH,r16        ; Set Stack Pointer to top of RAM
0xC1C          ldi    r16,low(RAMEND)
0xC1D          out    SPL,r16
0xC1E          sei                   ; Enable interrupts
0xC1F          <instr>  xxx
```

## 12.3 Interrupt Vectors in the Atmel ATmega168PA

**Table 12-4.** Reset and Interrupt Vectors in the Atmel ATmega168PA

| VectorNo. | Program Address[2] | Source | Interrupt Definition |
|---|---|---|---|
| 1 | 0x0000[1] | RESET | External Pin, Power-on Reset, Brown-out Reset and Watchdog System Reset |
| 2 | 0x0002 | INT0 | External Interrupt Request 0 |
| 3 | 0x0004 | INT1 | External Interrupt Request 1 |
| 4 | 0x0006 | PCINT0 | Pin Change Interrupt Request 0 |
| 5 | 0x0008 | PCINT1 | Pin Change Interrupt Request 1 |
| 6 | 0x000A | PCINT2 | Pin Change Interrupt Request 2 |
| 7 | 0x000C | WDT | Watchdog Time-out Interrupt |
| 8 | 0x000E | TIMER2 COMPA | Timer/Counter2 Compare Match A |
| 9 | 0x0010 | TIMER2 COMPB | Timer/Counter2 Compare Match B |
| 10 | 0x0012 | TIMER2 OVF | Timer/Counter2 Overflow |
| 11 | 0x0014 | TIMER1 CAPT | Timer/Counter1 Capture Event |
| 12 | 0x0016 | TIMER1 COMPA | Timer/Counter1 Compare Match A |
| 13 | 0x0018 | TIMER1 COMPB | Timer/Coutner1 Compare Match B |
| 14 | 0x001A | TIMER1 OVF | Timer/Counter1 Overflow |
| 15 | 0x001C | TIMER0 COMPA | Timer/Counter0 Compare Match A |
| 16 | 0x001E | TIMER0 COMPB | Timer/Counter0 Compare Match B |
| 17 | 0x0020 | TIMER0 OVF | Timer/Counter0 Overflow |
| 18 | 0x0022 | SPI, STC | SPI Serial Transfer Complete |
| 19 | 0x0024 | USART, RX | USART Rx Complete |
| 20 | 0x0026 | USART, UDRE | USART, Data Register Empty |
| 21 | 0x0028 | USART, TX | USART, Tx Complete |
| 22 | 0x002A | ADC | ADC Conversion Complete |
| 23 | 0x002C | EE READY | EEPROM Ready |
| 24 | 0x002E | ANALOG COMP | Analog Comparator |
| 25 | 0x0030 | TWI | 2-wire Serial Interface |
| 26 | 0x0032 | SPM READY | Store Program Memory Ready |

Notes: 1. When the BOOTRST Fuse is programmed, the device will jump to the Boot Loader address at reset, see "Boot Loader Support – Read-While-Write Self-Programming" on page 277.

2. When the IVSEL bit in MCUCR is set, Interrupt Vectors will be moved to the start of the Boot Flash Section. The address of each Interrupt Vector will then be the address in this table added to the start address of the Boot Flash Section.

Table 12-5 on page 64 shows reset and Interrupt Vectors placement for the various combinations of BOOTRST and IVSEL settings. If the program never enables an interrupt source, the Interrupt Vectors are not used, and regular program code can be placed at these locations. This is also the case if the Reset Vector is in the Application section while the Interrupt Vectors are in the Boot section or vice versa.

**Table 12-5.** Reset and Interrupt Vectors Placement in the Atmel ATmega168PA[1]

| BOOTRST | IVSEL | Reset Address | Interrupt Vectors Start Address |
|---|---|---|---|
| 1 | 0 | 0x000 | 0x002 |
| 1 | 1 | 0x000 | Boot Reset Address + 0x0002 |
| 0 | 0 | Boot Reset Address | 0x002 |
| 0 | 1 | Boot Reset Address | Boot Reset Address + 0x0002 |

Note: 1. The Boot Reset Address is shown in Table 27-7 on page 290. For the BOOTRST Fuse “1” means unprogrammed while “0” means programmed.

The most typical and general program setup for the Reset and Interrupt Vector Addresses in the Atmel® ATmega168PA is:

```
Address  Labels Code                Comments
0x0000          jmp    RESET        ; Reset Handler
0x0002          jmp    EXT_INT0     ; IRQ0 Handler
0x0004          jmp    EXT_INT1     ; IRQ1 Handler
0x0006          jmp    PCINT0       ; PCINT0 Handler
0x0008          jmp    PCINT1       ; PCINT1 Handler
0x000A          jmp    PCINT2       ; PCINT2 Handler
0x000C          jmp    WDT          ; Watchdog Timer Handler
0x000E          jmp    TIM2_COMPA   ; Timer2 Compare A Handler
0x0010          jmp    TIM2_COMPB   ; Timer2 Compare B Handler
0x0012          jmp    TIM2_OVF     ; Timer2 Overflow Handler
0x0014          jmp    TIM1_CAPT    ; Timer1 Capture Handler
0x0016          jmp    TIM1_COMPA   ; Timer1 Compare A Handler
0x0018          jmp    TIM1_COMPB   ; Timer1 Compare B Handler
0x001A          jmp    TIM1_OVF     ; Timer1 Overflow Handler
0x001C          jmp    TIM0_COMPA   ; Timer0 Compare A Handler
0x001E          jmp    TIM0_COMPB   ; Timer0 Compare B Handler
0x0020          jmp    TIM0_OVF     ; Timer0 Overflow Handler
0x0022          jmp    SPI_STC      ; SPI Transfer Complete Handler
0x0024          jmp    USART_RXC    ; USART, RX Complete Handler
0x0026          jmp    USART_UDRE   ; USART, UDR Empty Handler
0x0028          jmp    USART_TXC    ; USART, TX Complete Handler
0x002A          jmp    ADC          ; ADC Conversion Complete Handler
0x002C          jmp    EE_RDY       ; EEPROM Ready Handler
0x002E          jmp    ANA_COMP     ; Analog Comparator Handler
0x0030          jmp    TWI          ; 2-wire Serial Interface Handler
0x0032          jmp    SPM_RDY      ; Store Program Memory Ready Handler
;
```

```
0x0033RESET:   ldi    r16, high(RAMEND); Main program start
0x0034         out    SPH,r16          ; Set Stack Pointer to top of RAM
0x0035         ldi    r16, low(RAMEND)
0x0036         out    SPL,r16
0x0037         sei                     ; Enable interrupts
0x0038         <instr>  xxx
  ...     ...     ...  ...
```

When the BOOTRST Fuse is unprogrammed, the Boot section size set to 2K bytes and the IVSEL bit in the MCUCR Register is set before any interrupts are enabled, the most typical and general program setup for the Reset and Interrupt Vector Addresses in the Atmel® ATmega168PA is:

```
Address  Labels Code                  Comments
0x0000   RESET: ldi    r16,high(RAMEND); Main program start
0x0001          out    SPH,r16          ; Set Stack Pointer to top of RAM
0x0002          ldi    r16,low(RAMEND)
0x0003          out    SPL,r16
0x0004          sei                     ; Enable interrupts
0x0005          <instr>  xxx
;
.org 0x1C02
0x1C02          jmp    EXT_INT0         ; IRQ0 Handler
0x1C04          jmp    EXT_INT1         ; IRQ1 Handler
...             ...    ...              ;
0x1C32          jmp    SPM_RDY          ; Store Program Memory Ready Handler
```

When the BOOTRST Fuse is programmed and the Boot section size set to 2K bytes, the most typical and general program setup for the Reset and Interrupt Vector Addresses in the Atmel ATmega168PA is:

```
Address  Labels Code             Comments
.org 0x0002
0x0002          jmp    EXT_INT0         ; IRQ0 Handler
0x0004          jmp    EXT_INT1         ; IRQ1 Handler
...             ...    ...              ;
0x0032          jmp    SPM_RDY          ; Store Program Memory Ready Handler
;
.org 0x1C00
0x1C00   RESET: ldi    r16,high(RAMEND); Main program start
0x1C01          out    SPH,r16          ; Set Stack Pointer to top of RAM
0x1C02          ldi    r16,low(RAMEND)
0x1C03          out    SPL,r16
0x1C04          sei                     ; Enable interrupts
0x1C05          <instr>  xxx
```

When the BOOTRST Fuse is programmed, the Boot section size set to 2K bytes and the IVSEL bit in the MCUCR Register is set before any interrupts are enabled, the most typical and general program setup for the Reset and Interrupt Vector Addresses in the Atmel® ATmega168PA is:

```
Address  Labels Code                 Comments
;
.org 0x1C00
0x1C00          jmp    RESET           ; Reset handler
0x1C02          jmp    EXT_INT0        ; IRQ0 Handler
0x1C04          jmp    EXT_INT1        ; IRQ1 Handler
...             ...    ...             ;
0x1C32          jmp    SPM_RDY         ; Store Program Memory Ready Handler
;
0x1C33   RESET: ldi    r16,high(RAMEND); Main program start
0x1C34          out    SPH,r16         ; Set Stack Pointer to top of RAM
0x1C35          ldi    r16,low(RAMEND)
0x1C36          out    SPL,r16
0x1C37          sei                    ; Enable interrupts
0x1C38          <instr>  xxx
```

## 12.4 Register Description

### 12.4.1 Moving Interrupts Between Application and Boot Space, Atmel ATmega88PA, ATmega168PA

The MCU Control Register controls the placement of the Interrupt Vector table.

MCUCR – MCU Control Register

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| 0x35 (0x55) | – | BODS[(1)] | BODSE[(1)] | PUD | – | – | IVSEL | IVCE | MCUCR |
| Read/Write | R | R/W | R/W | R/W | R | R | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

Note: 1. BODS and BODSE only available for picoPower devices ATmega48PA/88PA/168PA

- **Bit 1 – IVSEL: Interrupt Vector Select**

When the IVSEL bit is cleared (zero), the Interrupt Vectors are placed at the start of the Flash memory. When this bit is set (one), the Interrupt Vectors are moved to the beginning of the Boot Loader section of the Flash. The actual address of the start of the Boot Flash Section is determined by the BOOTSZ Fuses. Refer to the section "Boot Loader Support – Read-While-Write Self-Programming" on page 277 for details. To avoid unintentional changes of Interrupt Vector tables, a special write procedure must be followed to change the IVSEL bit:

a. Write the Interrupt Vector Change Enable (IVCE) bit to one.

b. Within four cycles, write the desired value to IVSEL while writing a zero to IVCE.

Interrupts will automatically be disabled while this sequence is executed. Interrupts are disabled in the cycle IVCE is set, and they remain disabled until after the instruction following the write to IVSEL. If IVSEL is not written, interrupts remain disabled for four cycles. The I-bit in the Status Register is unaffected by the automatic disabling.

Note: If Interrupt Vectors are placed in the Boot Loader section and Boot Lock bit BLB02 is programmed, interrupts are disabled while executing from the Application section. If Interrupt Vectors are placed in the Application section and Boot Lock bit BLB12 is programed, interrupts are disabled while executing from the Boot Loader section. Refer to the section "Boot Loader Support – Read-While-Write Self-Programming" on page 277 for details on Boot Lock bits.

- **Bit 0 – IVCE: Interrupt Vector Change Enable**

The IVCE bit must be written to logic one to enable change of the IVSEL bit. IVCE is cleared by hardware four cycles after it is written or when IVSEL is written. Setting the IVCE bit will disable interrupts, as explained in the IVSEL description above. See Code Example below.

Assembly Code Example

```
Move_interrupts:
  ; Enable change of Interrupt Vectors
  ldi  r16, (1<<IVCE)
  out  MCUCR, r16
  ; Move interrupts to Boot Flash section
  ldi  r16, (1<<IVSEL)
  out  MCUCR, r16
  ret
```

C Code Example

```
void Move_interrupts(void)
{
  /* Enable change of Interrupt Vectors */
  MCUCR = (1<<IVCE);
  /* Move interrupts to Boot Flash section */
  MCUCR = (1<<IVSEL);
}
```

# 13. External Interrupts

The External Interrupts are triggered by the INT0 and INT1 pins or any of the PCINT23...0 pins. Observe that, if enabled, the interrupts will trigger even if the INT0 and INT1 or PCINT23...0 pins are configured as outputs. This feature provides a way of generating a software interrupt. The pin change interrupt PCI2 will trigger if any enabled PCINT[23:16] pin toggles. The pin change interrupt PCI1 will trigger if any enabled PCINT[14:8] pin toggles. The pin change interrupt PCI0 will trigger if any enabled PCINT[7:0] pin toggles. The PCMSK2, PCMSK1 and PCMSK0 Registers control which pins contribute to the pin change interrupts. Pin change interrupts on PCINT23...0 are detected asynchronously. This implies that these interrupts can be used for waking the part also from sleep modes other than Idle mode.

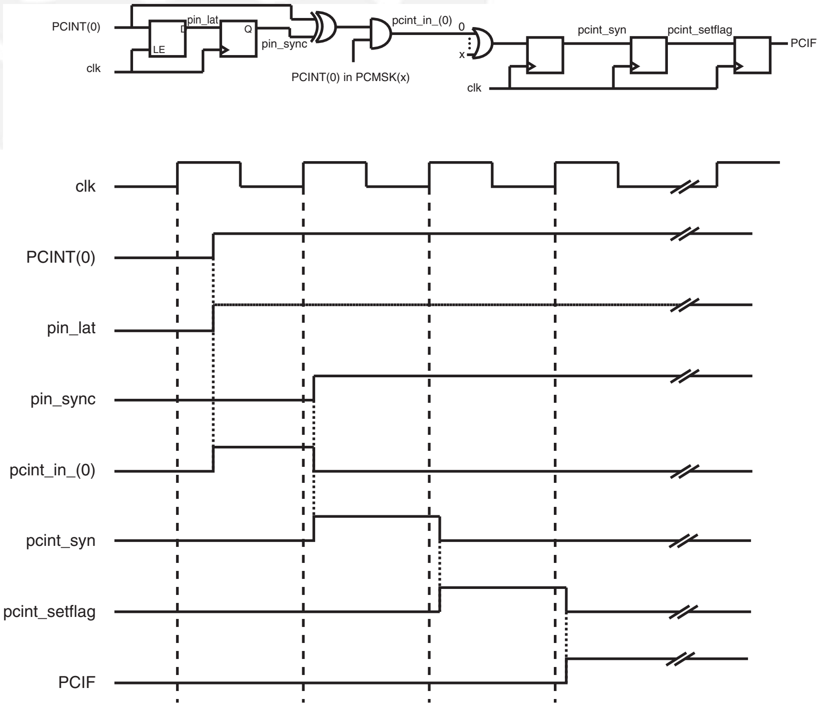
The INT0 and INT1 interrupts can be triggered by a falling or rising edge or a low level. This is set up as indicated in the specification for the External Interrupt Control Register A – EICRA. When the INT0 or INT1 interrupts are enabled and are configured as level triggered, the interrupts will trigger as long as the pin is held low. Note that recognition of falling or rising edge interrupts on INT0 or INT1 requires the presence of an I/O clock, described in "Clock Systems and their Distribution" on page 26. Low level interrupt on INT0 and INT1 is detected asynchronously. This implies that this interrupt can be used for waking the part also from sleep modes other than Idle mode. The I/O clock is halted in all sleep modes except Idle mode.

Note: Note that if a level triggered interrupt is used for wake-up from Power-down, the required level must be held long enough for the MCU to complete the wake-up to trigger the level interrupt. If the level disappears before the end of the Start-up Time, the MCU will still wake up, but no interrupt will be generated. The start-up time is defined by the SUT and CKSEL Fuses as described in "System Clock and Clock Options" on page 26.

## 13.1 Pin Change Interrupt Timing

An example of timing of a pin change interrupt is shown in Figure 13-1.

**Figure 13-1.** Timing of pin change interrupts



## 13.2 Register Description

### 13.2.1 EICRA – External Interrupt Control Register A

The External Interrupt Control Register A contains control bits for interrupt sense control.

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| (0x69) | – | – | – | – | ISC11 | ISC10 | ISC01 | ISC00 | EICRA |
| Read/Write | R | R | R | R | R/W | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

- **Bit 7:4 – Reserved**

These bits are unused bits in the Atmel® ATmega48PA/88PA/168PA, and will always read as zero.

- **Bit 3, 2 – ISC11, ISC10: Interrupt Sense Control 1 Bit 1 and Bit 0**

The External Interrupt 1 is activated by the external pin INT1 if the SREG I-flag and the corresponding interrupt mask are set. The level and edges on the external INT1 pin that activate the interrupt are defined in Table 13-1. The value on the INT1 pin is sampled before detecting edges. If edge or toggle interrupt is selected, pulses that last longer than one clock period will generate an interrupt. Shorter pulses are not guaranteed to generate an interrupt. If low level interrupt is selected, the low level must be held until the completion of the currently executing instruction to generate an interrupt.

**Table 13-1.** Interrupt 1 Sense Control

| ISC11 | ISC10 | Description |
|---|---|---|
| 0 | 0 | The low level of INT1 generates an interrupt request. |
| 0 | 1 | Any logical change on INT1 generates an interrupt request. |
| 1 | 0 | The falling edge of INT1 generates an interrupt request. |
| 1 | 1 | The rising edge of INT1 generates an interrupt request. |

- **Bit 1, 0 – ISC01, ISC00: Interrupt Sense Control 0 Bit 1 and Bit 0**

The External Interrupt 0 is activated by the external pin INT0 if the SREG I-flag and the corresponding interrupt mask are set. The level and edges on the external INT0 pin that activate the interrupt are defined in Table 13-2. The value on the INT0 pin is sampled before detecting edges. If edge or toggle interrupt is selected, pulses that last longer than one clock period will generate an interrupt. Shorter pulses are not guaranteed to generate an interrupt. If low level interrupt is selected, the low level must be held until the completion of the currently executing instruction to generate an interrupt.

**Table 13-2.** Interrupt 0 Sense Control

| ISC01 | ISC00 | Description |
|---|---|---|
| 0 | 0 | The low level of INT0 generates an interrupt request. |
| 0 | 1 | Any logical change on INT0 generates an interrupt request. |
| 1 | 0 | The falling edge of INT0 generates an interrupt request. |
| 1 | 1 | The rising edge of INT0 generates an interrupt request. |

### 13.2.2 EIMSK – External Interrupt Mask Register

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| 0x1D (0x3D) | – | – | – | – | – | – | INT1 | INT0 | EIMSK |
| Read/Write | R | R | R | R | R | R | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

- **Bit 7:2 – Reserved**

These bits are unused bits in the Atmel® ATmega48PA/88PA/168PA, and will always read as zero.

- **Bit 1 – INT1: External Interrupt Request 1 Enable**

When the INT1 bit is set (one) and the I-bit in the Status Register (SREG) is set (one), the external pin interrupt is enabled. The Interrupt Sense Control1 bits 1/0 (ISC11 and ISC10) in the External Interrupt Control Register A (EICRA) define whether the external interrupt is activated on rising and/or falling edge of the INT1 pin or level sensed. Activity on the pin will cause an interrupt request even if INT1 is configured as an output. The corresponding interrupt of External Interrupt Request 1 is executed from the INT1 Interrupt Vector.

- **Bit 0 – INT0: External Interrupt Request 0 Enable**

When the INT0 bit is set (one) and the I-bit in the Status Register (SREG) is set (one), the external pin interrupt is enabled. The Interrupt Sense Control0 bits 1/0 (ISC01 and ISC00) in the External Interrupt Control Register A (EICRA) define whether the external interrupt is activated on rising and/or falling edge of the INT0 pin or level sensed. Activity on the pin will cause an interrupt request even if INT0 is configured as an output. The corresponding interrupt of External Interrupt Request 0 is executed from the INT0 Interrupt Vector.

### 13.2.3 EIFR – External Interrupt Flag Register

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| 0x1C (0x3C) | – | – | – | – | – | – | INTF1 | INTF0 | EIFR |
| Read/Write | R | R | R | R | R | R | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

- **Bit 7:2 – Reserved**

These bits are unused bits in the Atmel ATmega48PA/88PA/168PA, and will always read as zero.

- **Bit 1 – INTF1: External Interrupt Flag 1**

When an edge or logic change on the INT1 pin triggers an interrupt request, INTF1 becomes set (one). If the I-bit in SREG and the INT1 bit in EIMSK are set (one), the MCU will jump to the corresponding Interrupt Vector. The flag is cleared when the interrupt routine is executed. Alternatively, the flag can be cleared by writing a logical one to it. This flag is always cleared when INT1 is configured as a level interrupt.

- **Bit 0 – INTF0: External Interrupt Flag 0**

When an edge or logic change on the INT0 pin triggers an interrupt request, INTF0 becomes set (one). If the I-bit in SREG and the INT0 bit in EIMSK are set (one), the MCU will jump to the corresponding Interrupt Vector. The flag is cleared when the interrupt routine is executed. Alternatively, the flag can be cleared by writing a logical one to it. This flag is always cleared when INT0 is configured as a level interrupt.

### 13.2.4 PCICR – Pin Change Interrupt Control Register

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| (0x68) | – | – | – | – | – | PCIE2 | PCIE1 | PCIE0 | PCICR |
| Read/Write | R | R | R | R | R | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

- **Bit 7:3 – Reserved**

These bits are unused bits in the Atmel® ATmega48PA/88PA/168PA, and will always read as zero.

- **Bit 2 – PCIE2: Pin Change Interrupt Enable 2**

When the PCIE2 bit is set (one) and the I-bit in the Status Register (SREG) is set (one), pin change interrupt 2 is enabled. Any change on any enabled PCINT[23:16] pin will cause an interrupt. The corresponding interrupt of Pin Change Interrupt Request is executed from the PCI2 Interrupt Vector. PCINT[23:16] pins are enabled individually by the PCMSK2 Register.

- **Bit 1 – PCIE1: Pin Change Interrupt Enable 1**

When the PCIE1 bit is set (one) and the I-bit in the Status Register (SREG) is set (one), pin change interrupt 1 is enabled. Any change on any enabled PCINT[14:8] pin will cause an interrupt. The corresponding interrupt of Pin Change Interrupt Request is executed from the PCI1 Interrupt Vector. PCINT[14:8] pins are enabled individually by the PCMSK1 Register.

- **Bit 0 – PCIE0: Pin Change Interrupt Enable 0**

When the PCIE0 bit is set (one) and the I-bit in the Status Register (SREG) is set (one), pin change interrupt 0 is enabled. Any change on any enabled PCINT[7:0] pin will cause an interrupt. The corresponding interrupt of Pin Change Interrupt Request is executed from the PCI0 Interrupt Vector. PCINT[7:0] pins are enabled individually by the PCMSK0 Register.

### 13.2.5 PCIFR – Pin Change Interrupt Flag Register

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| 0x1B (0x3B) | – | – | – | – | – | PCIF2 | PCIF1 | PCIF0 | PCIFR |
| Read/Write | R | R | R | R | R | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

- **Bit 7:3 – Reserved**

These bits are unused bits in the Atmel ATmega48PA/88PA/168PA, and will always read as zero.

- **Bit 2 – PCIF2: Pin Change Interrupt Flag 2**

When a logic change on any PCINT[23:16] pin triggers an interrupt request, PCIF2 becomes set (one). If the I-bit in SREG and the PCIE2 bit in PCICR are set (one), the MCU will jump to the corresponding Interrupt Vector. The flag is cleared when the interrupt routine is executed. Alternatively, the flag can be cleared by writing a logical one to it.

- **Bit 1 – PCIF1: Pin Change Interrupt Flag 1**

When a logic change on any PCINT[14:8] pin triggers an interrupt request, PCIF1 becomes set (one). If the I-bit in SREG and the PCIE1 bit in PCICR are set (one), the MCU will jump to the corresponding Interrupt Vector. The flag is cleared when the interrupt routine is executed. Alternatively, the flag can be cleared by writing a logical one to it.

- **Bit 0 – PCIF0: Pin Change Interrupt Flag 0**

When a logic change on any PCINT[7:0] pin triggers an interrupt request, PCIF0 becomes set (one). If the I-bit in SREG and the PCIE0 bit in PCICR are set (one), the MCU will jump to the corresponding Interrupt Vector. The flag is cleared when the interrupt routine is executed. Alternatively, the flag can be cleared by writing a logical one to it.

### 13.2.6 PCMSK2 – Pin Change Mask Register 2

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| (0x6D) | PCINT23 | PCINT22 | PCINT21 | PCINT20 | PCINT19 | PCINT18 | PCINT17 | PCINT16 | PCMSK2 |
| Read/Write | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

- **Bit 7:0 – PCINT[23:16]: Pin Change Enable Mask 23...16**

Each PCINT[23:16]-bit selects whether pin change interrupt is enabled on the corresponding I/O pin. If PCINT[23:16] is set and the PCIE2 bit in PCICR is set, pin change interrupt is enabled on the corresponding I/O pin. If PCINT[23:16] is cleared, pin change interrupt on the corresponding I/O pin is disabled.

### 13.2.7 PCMSK1 – Pin Change Mask Register 1

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| (0x6C) | – | PCINT14 | PCINT13 | PCINT12 | PCINT11 | PCINT10 | PCINT9 | PCINT8 | PCMSK1 |
| Read/Write | R | R/W | R/W | R/W | R/W | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

- **Bit 7 – Reserved**

This bit is an unused bit in the Atmel® ATmega48PA/88PA/168PA, and will always read as zero.

- **Bit 6:0 – PCINT[14:8]: Pin Change Enable Mask 14...8**

Each PCINT[14:8]-bit selects whether pin change interrupt is enabled on the corresponding I/O pin. If PCINT[14:8] is set and the PCIE1 bit in PCICR is set, pin change interrupt is enabled on the corresponding I/O pin. If PCINT[14:8] is cleared, pin change interrupt on the corresponding I/O pin is disabled.

### 13.2.8 PCMSK0 – Pin Change Mask Register 0

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| (0x6B) | PCINT7 | PCINT6 | PCINT5 | PCINT4 | PCINT3 | PCINT2 | PCINT1 | PCINT0 | PCMSK0 |
| Read/Write | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

- **Bit 7:0 – PCINT[7:0]: Pin Change Enable Mask 7...0**

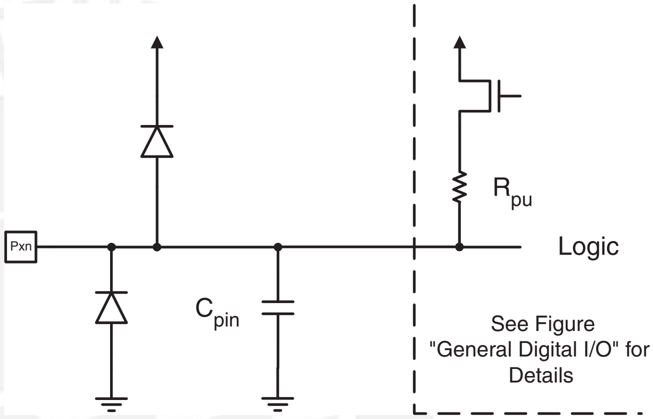
Each PCINT[7:0] bit selects whether pin change interrupt is enabled on the corresponding I/O pin. If PCINT[7:0] is set and the PCIE0 bit in PCICR is set, pin change interrupt is enabled on the corresponding I/O pin. If PCINT[7:0] is cleared, pin change interrupt on the corresponding I/O pin is disabled.

# 14. I/O-Ports

## 14.1 Overview

All AVR ports have true Read-Modify-Write functionality when used as general digital I/O ports. This means that the direction of one port pin can be changed without unintentionally changing the direction of any other pin with the SBI and CBI instructions. The same applies when changing drive value (if configured as output) or enabling/disabling of pull-up resistors (if configured as input). Each output buffer has symmetrical drive characteristics with both high sink and source capability. The pin driver is strong enough to drive LED displays directly. All port pins have individually selectable pull-up resistors with a supply-voltage invariant resistance. All I/O pins have protection diodes to both $V_{CC}$ and Ground as indicated in Figure 14-1. Refer to "Electrical Characteristics" on page 313 for a complete list of parameters.

**Figure 14-1.** I/O Pin Equivalent Schematic



All registers and bit references in this section are written in general form. A lower case "x" represents the numbering letter for the port, and a lower case "n" represents the bit number. However, when using the register or bit defines in a program, the precise form must be used. For example, PORTB3 for bit no. 3 in Port B, here documented generally as PORTxn. The physical I/O Registers and bit locations are listed in "Register Description" on page 91.

Three I/O memory address locations are allocated for each port, one each for the Data Register – PORTx, Data Direction Register – DDRx, and the Port Input Pins – PINx. The Port Input Pins I/O location is read only, while the Data Register and the Data Direction Register are read/write. However, writing a logic one to a bit in the PINx Register, will result in a toggle in the corresponding bit in the Data Register. In addition, the Pull-up Disable – PUD bit in MCUCR disables the pull-up function for all pins in all ports when set.

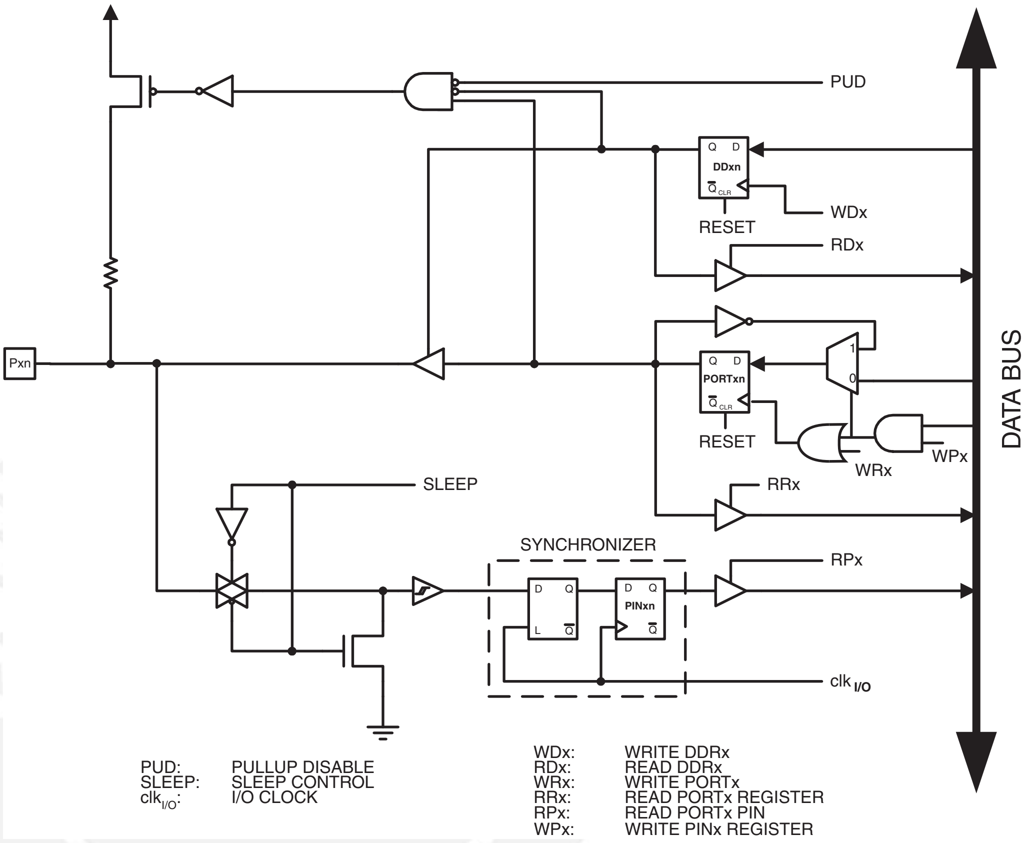
Using the I/O port as General Digital I/O is described in "Ports as General Digital I/O" on page 74. Most port pins are multiplexed with alternate functions for the peripheral features on the device. How each alternate function interferes with the port pin is described in "Alternate Port Functions" on page 79. Refer to the individual module sections for a full description of the alternate functions.

Note that enabling the alternate function of some of the port pins does not affect the use of the other pins in the port as general digital I/O.

## 14.2 Ports as General Digital I/O

The ports are bi-directional I/O ports with optional internal pull-ups. Figure 14-2 shows a functional description of one I/O-port pin, here generically called Pxn.

**Figure 14-2.** General Digital I/O(1)



Note: 1. WRx, WPx, WDx, RRx, RPx, and RDx are common to all pins within the same port. $\text{clk}_{I/O}$, SLEEP, and PUD are common to all ports.

### 14.2.1 Configuring the Pin

Each port pin consists of three register bits: DDxn, PORTxn, and PINxn. As shown in "Register Description" on page 91, the DDxn bits are accessed at the DDRx I/O address, the PORTxn bits at the PORTx I/O address, and the PINxn bits at the PINx I/O address.

The DDxn bit in the DDRx Register selects the direction of this pin. If DDxn is written logic one, Pxn is configured as an output pin. If DDxn is written logic zero, Pxn is configured as an input pin.

If PORTxn is written logic one when the pin is configured as an input pin, the pull-up resistor is activated. To switch the pull-up resistor off, PORTxn has to be written logic zero or the pin has to be configured as an output pin. The port pins are tri-stated when reset condition becomes active, even if no clocks are running.

If PORTxn is written logic one when the pin is configured as an output pin, the port pin is driven high (one). If PORTxn is written logic zero when the pin is configured as an output pin, the port pin is driven low (zero).

### 14.2.2 Toggling the Pin

Writing a logic one to PINxn toggles the value of PORTxn, independent on the value of DDRxn. Note that the SBI instruction can be used to toggle one single bit in a port.

### 14.2.3 Switching Between Input and Output

When switching between tri-state ({DDxn, PORTxn} = 0b00) and output high ({DDxn, PORTxn} = 0b11), an intermediate state with either pull-up enabled {DDxn, PORTxn} = 0b01) or output low ({DDxn, PORTxn} = 0b10) must occur. Normally, the pull-up enabled state is fully acceptable, as a high-impedance environment will not notice the difference between a strong high driver and a pull-up. If this is not the case, the PUD bit in the MCUCR Register can be set to disable all pull-ups in all ports.

Switching between input with pull-up and output low generates the same problem. The user must use either the tri-state ({DDxn, PORTxn} = 0b00) or the output high state ({DDxn, PORTxn} = 0b11) as an intermediate step.

Table 14-1 summarizes the control signals for the pin value.

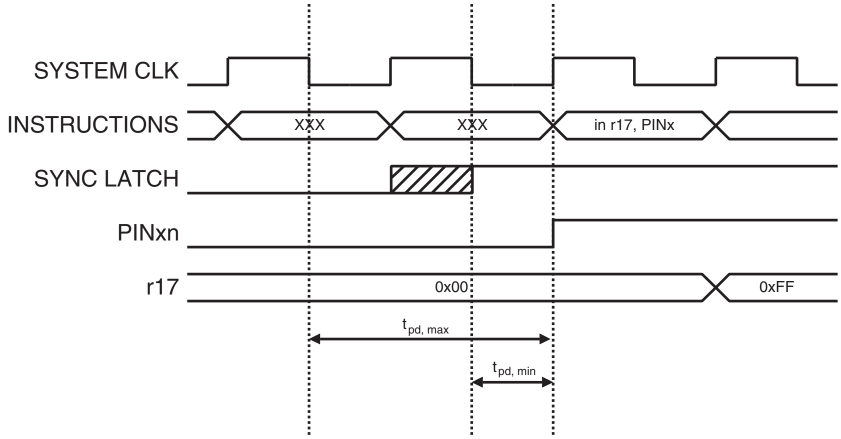
**Table 14-1.** Port Pin Configurations

| DDxn | PORTxn | PUD (in MCUCR) | I/O | Pull-up | Comment |
|---|---|---|---|---|---|
| 0 | 0 | X | Input | No | Tri-state (Hi-Z) |
| 0 | 1 | 0 | Input | Yes | Pxn will source current if ext. pulled low. |
| 0 | 1 | 1 | Input | No | Tri-state (Hi-Z) |
| 1 | 0 | X | Output | No | Output Low (Sink) |
| 1 | 1 | X | Output | No | Output High (Source) |

### 14.2.4 Reading the Pin Value

Independent of the setting of Data Direction bit DDxn, the port pin can be read through the PINxn Register bit. As shown in Figure 14-2, the PINxn Register bit and the preceding latch constitute a synchronizer. This is needed to avoid metastability if the physical pin changes value near the edge of the internal clock, but it also introduces a delay. Figure 14-3 shows a timing diagram of the synchronization when reading an externally applied pin value. The maximum and minimum propagation delays are denoted $t_{pd,max}$ and $t_{pd,min}$ respectively.

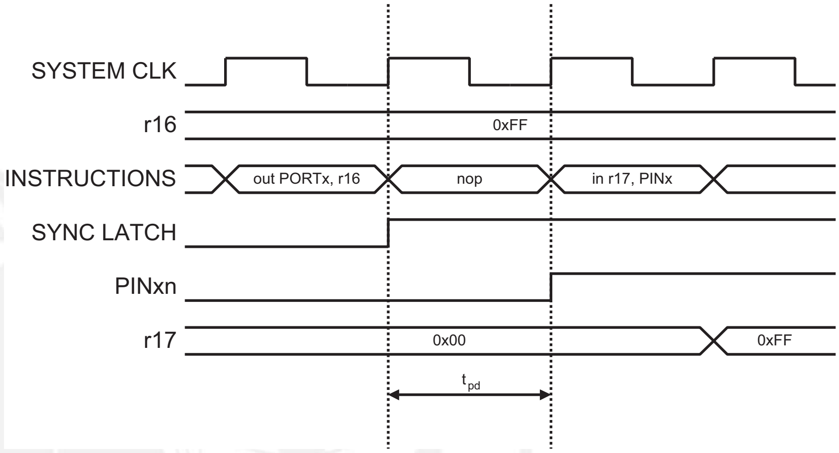
**Figure 14-3.** Synchronization when Reading an Externally Applied Pin value



Consider the clock period starting shortly after the first falling edge of the system clock. The latch is closed when the clock is low, and goes transparent when the clock is high, as indicated by the shaded region of the "SYNC LATCH" signal. The signal value is latched when the system clock goes low. It is clocked into the PINxn Register at the succeeding positive clock edge. As indicated by the two arrows tpd,max and tpd,min, a single signal transition on the pin will be delayed between ½ and 1½ system clock period depending upon the time of assertion.

When reading back a software assigned pin value, a nop instruction must be inserted as indicated in Figure 14-4. The out instruction sets the "SYNC LATCH" signal at the positive edge of the clock. In this case, the delay tpd through the synchronizer is 1 system clock period.

**Figure 14-4.** Synchronization when Reading a Software Assigned Pin Value



The following code example shows how to set port B pins 0 and 1 high, 2 and 3 low, and define the port pins from 4 to 7 as input with pull-ups assigned to port pins 6 and 7. The resulting pin values are read back again, but as previously discussed, a nop instruction is included to be able to read back the value recently assigned to some of the pins.

Assembly Code Example[1]

```
    ...
    ; Define pull-ups and set outputs high
    ; Define directions for port pins
    ldi  r16,(1<<PB7)|(1<<PB6)|(1<<PB1)|(1<<PB0)
    ldi  r17,(1<<DDB3)|(1<<DDB2)|(1<<DDB1)|(1<<DDB0)
    out  PORTB,r16
    out  DDRB,r17
    ; Insert nop for synchronization
    nop
    ; Read port pins
    in   r16,PINB
    ...
```

C Code Example

```
  unsigned char i;
    ...
    /* Define pull-ups and set outputs high */
    /* Define directions for port pins */
    PORTB = (1<<PB7)|(1<<PB6)|(1<<PB1)|(1<<PB0);
    DDRB = (1<<DDB3)|(1<<DDB2)|(1<<DDB1)|(1<<DDB0);
    /* Insert nop for synchronization*/
    __no_operation();
    /* Read port pins */
    i = PINB;
    ...
```

Note: 1. For the assembly program, two temporary registers are used to minimize the time from pull-ups are set on pins 0, 1, 6, and 7, until the direction bits are correctly set, defining bit 2 and 3 as low and redefining bits 0 and 1 as strong high drivers.

### 14.2.5 Digital Input Enable and Sleep Modes

As shown in Figure 14-2, the digital input signal can be clamped to ground at the input of the Schmitt Trigger. The signal denoted SLEEP in the figure, is set by the MCU Sleep Controller in Power-down mode, Power-save mode, and Standby mode to avoid high power consumption if some input signals are left floating, or have an analog signal level close to $V_{CC}/2$.

SLEEP is overridden for port pins enabled as external interrupt pins. If the external interrupt request is not enabled, SLEEP is active also for these pins. SLEEP is also overridden by various other alternate functions as described in "Alternate Port Functions" on page 79.

If a logic high level ("one") is present on an asynchronous external interrupt pin configured as "Interrupt on Rising Edge, Falling Edge, or Any Logic Change on Pin" while the external interrupt is *not* enabled, the corresponding External Interrupt Flag will be set when resuming from the above mentioned Sleep mode, as the clamping in these sleep mode produces the requested logic change.

### 14.2.6 Unconnected Pins

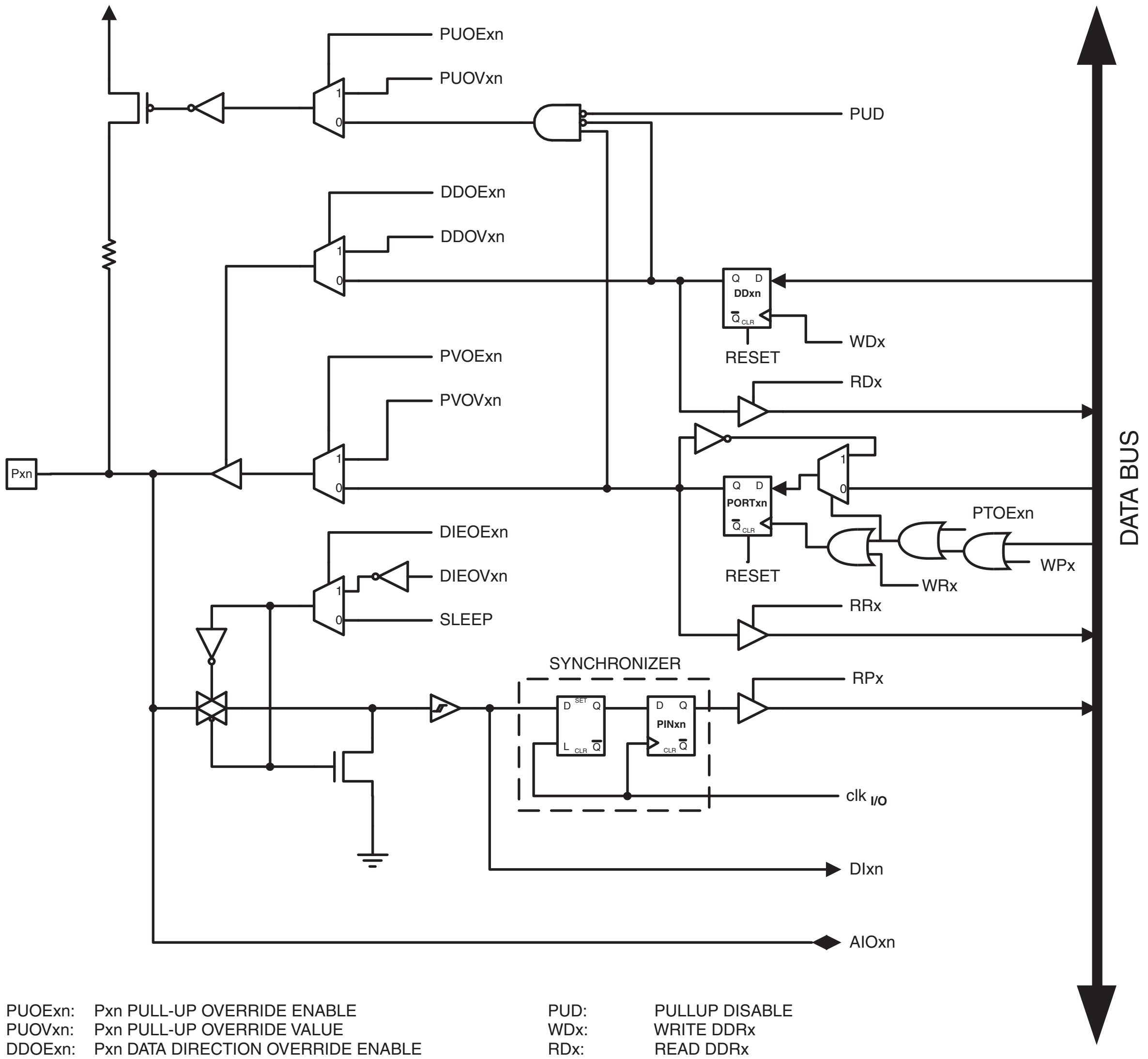
If some pins are unused, it is recommended to ensure that these pins have a defined level. Even though most of the digital inputs are disabled in the deep sleep modes as described above, floating inputs should be avoided to reduce current consumption in all other modes where the digital inputs are enabled (Reset, Active mode and Idle mode).

The simplest method to ensure a defined level of an unused pin, is to enable the internal pull-up. In this case, the pull-up will be disabled during reset. If low power consumption during reset is important, it is recommended to use an external pull-up or pull-down. Connecting unused pins directly to $V_{CC}$ or GND is not recommended, since this may cause excessive currents if the pin is accidentally configured as an output.

## 14.3 Alternate Port Functions

Most port pins have alternate functions in addition to being general digital I/Os. Figure 14-5 shows how the port pin control signals from the simplified Figure 14-2 on page 74 can be overridden by alternate functions. The overriding signals may not be present in all port pins, but the figure serves as a generic description applicable to all port pins in the AVR microcontroller family.

**Figure 14-5.** Alternate Port Functions(1)



| | | | |
|---|---|---|---|
| PUOExn: | Pxn PULL-UP OVERRIDE ENABLE | PUD: | PULLUP DISABLE |
| PUOVxn: | Pxn PULL-UP OVERRIDE VALUE | WDx: | WRITE DDRx |
| DDOExn: | Pxn DATA DIRECTION OVERRIDE ENABLE | RDx: | READ DDRx |
| DDOVxn: | Pxn DATA DIRECTION OVERRIDE VALUE | RRx: | READ PORTx REGISTER |
| PVOExn: | Pxn PORT VALUE OVERRIDE ENABLE | WRx: | WRITE PORTx |
| PVOVxn: | Pxn PORT VALUE OVERRIDE VALUE | RPx: | READ PORTx PIN |
| DIEOExn: | Pxn DIGITAL INPUT-ENABLE OVERRIDE ENABLE | WPx: | WRITE PINx |
| DIEOVxn: | Pxn DIGITAL INPUT-ENABLE OVERRIDE VALUE | $clk_{I/O}$: | I/O CLOCK |
| SLEEP: | SLEEP CONTROL | DIxn: | DIGITAL INPUT PIN n ON PORTx |
| PTOExn: | Pxn, PORT TOGGLE OVERRIDE ENABLE | AIOxn: | ANALOG INPUT/OUTPUT PIN n ON PORTx |

Note: 1. WRx, WPx, WDx, RRx, RPx, and RDx are common to all pins within the same port. $clk_{I/O}$, SLEEP, and PUD are common to all ports. All other signals are unique for each pin.

Table 14-2 summarizes the function of the overriding signals. The pin and port indexes from Figure 14-5 on page 79 are not shown in the succeeding tables. The overriding signals are generated internally in the modules having the alternate function.

**Table 14-2.** Generic Description of Overriding Signals for Alternate Functions

| Signal Name | Full Name | Description |
|---|---|---|
| PUOE | Pull-up Override Enable | If this signal is set, the pull-up enable is controlled by the PUOV signal. If this signal is cleared, the pull-up is enabled when {DDxn, PORTxn, PUD} = 0b010. |
| PUOV | Pull-up Override Value | If PUOE is set, the pull-up is enabled/disabled when PUOV is set/cleared, regardless of the setting of the DDxn, PORTxn, and PUD Register bits. |
| DDOE | Data Direction Override Enable | If this signal is set, the Output Driver Enable is controlled by the DDOV signal. If this signal is cleared, the Output driver is enabled by the DDxn Register bit. |
| DDOV | Data Direction Override Value | If DDOE is set, the Output Driver is enabled/disabled when DDOV is set/cleared, regardless of the setting of the DDxn Register bit. |
| PVOE | Port Value Override Enable | If this signal is set and the Output Driver is enabled, the port value is controlled by the PVOV signal. If PVOE is cleared, and the Output Driver is enabled, the port Value is controlled by the PORTxn Register bit. |
| PVOV | Port Value Override Value | If PVOE is set, the port value is set to PVOV, regardless of the setting of the PORTxn Register bit. |
| PTOE | Port Toggle Override Enable | If PTOE is set, the PORTxn Register bit is inverted. |
| DIEOE | Digital Input Enable Override Enable | If this bit is set, the Digital Input Enable is controlled by the DIEOV signal. If this signal is cleared, the Digital Input Enable is determined by MCU state (Normal mode, sleep mode). |
| DIEOV | Digital Input Enable Override Value | If DIEOE is set, the Digital Input is enabled/disabled when DIEOV is set/cleared, regardless of the MCU state (Normal mode, sleep mode). |
| DI | Digital Input | This is the Digital Input to alternate functions. In the figure, the signal is connected to the output of the Schmitt Trigger but before the synchronizer. Unless the Digital Input is used as a clock source, the module with the alternate function will use its own synchronizer. |
| AIO | Analog Input/Output | This is the Analog Input/output to/from alternate functions. The signal is connected directly to the pad, and can be used bi-directionally. |

The following subsections shortly describe the alternate functions for each port, and relate the overriding signals to the alternate function. Refer to the alternate function description for further details.

### 14.3.1 Alternate Functions of Port B

The Port B pins with alternate functions are shown in Table 14-3.

**Table 14-3.** Port B Pins Alternate Functions

| Port Pin | Alternate Functions |
|---|---|
| PB7 | XTAL2 (Chip Clock Oscillator pin 2)<br>TOSC2 (Timer Oscillator pin 2)<br>PCINT7 (Pin Change Interrupt 7) |
| PB6 | XTAL1 (Chip Clock Oscillator pin 1 or External clock input)<br>TOSC1 (Timer Oscillator pin 1)<br>PCINT6 (Pin Change Interrupt 6) |
| PB5 | SCK (SPI Bus Master clock Input)<br>PCINT5 (Pin Change Interrupt 5) |
| PB4 | MISO (SPI Bus Master Input/Slave Output)<br>PCINT4 (Pin Change Interrupt 4) |
| PB3 | MOSI (SPI Bus Master Output/Slave Input)<br>OC2A (Timer/Counter2 Output Compare Match A Output)<br>PCINT3 (Pin Change Interrupt 3) |
| PB2 | $\overline{SS}$ (SPI Bus Master Slave select)<br>OC1B (Timer/Counter1 Output Compare Match B Output)<br>PCINT2 (Pin Change Interrupt 2) |
| PB1 | OC1A (Timer/Counter1 Output Compare Match A Output)<br>PCINT1 (Pin Change Interrupt 1) |
| PB0 | ICP1 (Timer/Counter1 Input Capture Input)<br>CLKO (Divided System Clock Output)<br>PCINT0 (Pin Change Interrupt 0) |

The alternate pin configuration is as follows:

- **XTAL2/TOSC2/PCINT7 – Port B, Bit 7**

XTAL2: Chip clock Oscillator pin 2. Used as clock pin for crystal Oscillator or Low-frequency crystal Oscillator. When used as a clock pin, the pin can not be used as an I/O pin.

TOSC2: Timer Oscillator pin 2. Used only if internal calibrated RC Oscillator is selected as chip clock source, and the asynchronous timer is enabled by the correct setting in ASSR. When the AS2 bit in ASSR is set (one) and the EXCLK bit is cleared (zero) to enable asynchronous clocking of Timer/Counter2 using the Crystal Oscillator, pin PB7 is disconnected from the port, and becomes the inverting output of the Oscillator amplifier. In this mode, a crystal Oscillator is connected to this pin, and the pin cannot be used as an I/O pin.

PCINT7: Pin Change Interrupt source 7. The PB7 pin can serve as an external interrupt source.

If PB7 is used as a clock pin, DDB7, PORTB7 and PINB7 will all read 0.

- **XTAL1/TOSC1/PCINT6 – Port B, Bit 6**

XTAL1: Chip clock Oscillator pin 1. Used for all chip clock sources except internal calibrated RC Oscillator. When used as a clock pin, the pin can not be used as an I/O pin.

TOSC1: Timer Oscillator pin 1. Used only if internal calibrated RC Oscillator is selected as chip clock source, and the asynchronous timer is enabled by the correct setting in ASSR. When the AS2 bit in ASSR is set (one) to enable asynchronous clocking of Timer/Counter2, pin PB6 is disconnected from the port, and becomes the input of the inverting Oscillator amplifier. In this mode, a crystal Oscillator is connected to this pin, and the pin can not be used as an I/O pin.

PCINT6: Pin Change Interrupt source 6. The PB6 pin can serve as an external interrupt source.

If PB6 is used as a clock pin, DDB6, PORTB6 and PINB6 will all read 0.

- **SCK/PCINT5 – Port B, Bit 5**

SCK: Master Clock output, Slave Clock input pin for SPI channel. When the SPI is enabled as a Slave, this pin is configured as an input regardless of the setting of DDB5. When the SPI is enabled as a Master, the data direction of this pin is controlled by DDB5. When the pin is forced by the SPI to be an input, the pull-up can still be controlled by the PORTB5 bit.

PCINT5: Pin Change Interrupt source 5. The PB5 pin can serve as an external interrupt source.

- **MISO/PCINT4 – Port B, Bit 4**

MISO: Master Data input, Slave Data output pin for SPI channel. When the SPI is enabled as a Master, this pin is configured as an input regardless of the setting of DDB4. When the SPI is enabled as a Slave, the data direction of this pin is controlled by DDB4. When the pin is forced by the SPI to be an input, the pull-up can still be controlled by the PORTB4 bit.

PCINT4: Pin Change Interrupt source 4. The PB4 pin can serve as an external interrupt source.

- **MOSI/OC2/PCINT3 – Port B, Bit 3**

MOSI: SPI Master Data output, Slave Data input for SPI channel. When the SPI is enabled as a Slave, this pin is configured as an input regardless of the setting of DDB3. When the SPI is enabled as a Master, the data direction of this pin is controlled by DDB3. When the pin is forced by the SPI to be an input, the pull-up can still be controlled by the PORTB3 bit.

OC2, Output Compare Match Output: The PB3 pin can serve as an external output for the Timer/Counter2 Compare Match. The PB3 pin has to be configured as an output (DDB3 set (one)) to serve this function. The OC2 pin is also the output pin for the PWM mode timer function.

PCINT3: Pin Change Interrupt source 3. The PB3 pin can serve as an external interrupt source.

- **$\overline{\text{SS}}$/OC1B/PCINT2 – Port B, Bit 2**

$\overline{\text{SS}}$: Slave Select input. When the SPI is enabled as a Slave, this pin is configured as an input regardless of the setting of DDB2. As a Slave, the SPI is activated when this pin is driven low. When the SPI is enabled as a Master, the data direction of this pin is controlled by DDB2. When the pin is forced by the SPI to be an input, the pull-up can still be controlled by the PORTB2 bit.

OC1B, Output Compare Match output: The PB2 pin can serve as an external output for the Timer/Counter1 Compare Match B. The PB2 pin has to be configured as an output (DDB2 set (one)) to serve this function. The OC1B pin is also the output pin for the PWM mode timer function.

PCINT2: Pin Change Interrupt source 2. The PB2 pin can serve as an external interrupt source.

- **OC1A/PCINT1 – Port B, Bit 1**

OC1A, Output Compare Match output: The PB1 pin can serve as an external output for the Timer/Counter1 Compare Match A. The PB1 pin has to be configured as an output (DDB1 set (one)) to serve this function. The OC1A pin is also the output pin for the PWM mode timer function.

PCINT1: Pin Change Interrupt source 1. The PB1 pin can serve as an external interrupt source.

- **ICP1/CLKO/PCINT0 – Port B, Bit 0**

ICP1, Input Capture Pin: The PB0 pin can act as an Input Capture Pin for Timer/Counter1.

CLKO, Divided System Clock: The divided system clock can be output on the PB0 pin. The divided system clock will be output if the CKOUT Fuse is programmed, regardless of the PORTB0 and DDB0 settings. It will also be output during reset.

PCINT0: Pin Change Interrupt source 0. The PB0 pin can serve as an external interrupt source.

Table 14-4 and Table 14-5 on page 84 relate the alternate functions of Port B to the overriding signals shown in Figure 14-5 on page 79. SPI MSTR INPUT and SPI SLAVE OUTPUT constitute the MISO signal, while MOSI is divided into SPI MSTR OUTPUT and SPI SLAVE INPUT.

**Table 14-4.** Overriding Signals for Alternate Functions in PB7...PB4

| Signal Name | PB7/XTAL2/ TOSC2/PCINT7[1] | PB6/XTAL1/ TOSC1/PCINT6[1] | PB5/SCK/ PCINT5 | PB4/MISO/ PCINT4 |
|---|---|---|---|---|
| PUOE | $\overline{\text{INTRC}} \times \overline{\text{EXTCK}}$+ AS2 | $\overline{\text{INTRC}}$ + AS2 | SPE × $\overline{\text{MSTR}}$ | SPE  MSTR |
| PUOV | 0 | 0 | PORTB5  $\overline{\text{PUD}}$ | PORTB4  $\overline{\text{PUD}}$ |
| DDOE | $\overline{\text{INTRC}}$  $\overline{\text{EXTCK}}$+ AS2 | $\overline{\text{INTRC}}$ + AS2 | SPE  $\overline{\text{MSTR}}$ | SPE  MSTR |
| DDOV | 0 | 0 | 0 | 0 |
| PVOE | 0 | 0 | SPE  MSTR | SPE  $\overline{\text{MSTR}}$ |
| PVOV | 0 | 0 | SCK OUTPUT | SPI SLAVE OUTPUT |
| DIEOE | $\overline{\text{INTRC}}$  $\overline{\text{EXTCK}}$ + AS2 + PCINT7 PCIE0 | $\overline{\text{INTRC}}$ + AS2 + PCINT6  PCIE0 | PCINT5  PCIE0 | PCINT4  PCIE0 |
| DIEOV | (INTRC + EXTCK) $\overline{\text{AS2}}$ | INTRC  $\overline{\text{AS2}}$ | 1 | 1 |
| DI | PCINT7 INPUT | PCINT6 INPUT | PCINT5 INPUT<br>SCK INPUT | PCINT4 INPUT<br>SPI MSTR INPUT |
| AIO | Oscillator Output | Oscillator/Clock Input | – | – |

Notes: 1. INTRC means that one of the internal RC Oscillators are selected (by the CKSEL fuses), EXTCK means that external clock is selected (by the CKSEL fuses)

**Table 14-5.** Overriding Signals for Alternate Functions in PB3...PB0

| Signal Name | PB3/MOSI/ OC2/PCINT3 | PB2/$\overline{\text{SS}}$/ OC1B/PCINT2 | PB1/OC1A/ PCINT1 | PB0/ICP1/ PCINT0 |
|---|---|---|---|---|
| PUOE | SPE  $\overline{\text{MSTR}}$ | SPE  $\overline{\text{MSTR}}$ | 0 | 0 |
| PUOV | PORTB3  $\overline{\text{PUD}}$ | PORTB2  $\overline{\text{PUD}}$ | 0 | 0 |
| DDOE | SPE  $\overline{\text{MSTR}}$ | SPE  $\overline{\text{MSTR}}$ | 0 | 0 |
| DDOV | 0 | 0 | 0 | 0 |
| PVOE | SPE  MSTR + OC2A ENABLE | OC1B ENABLE | OC1A ENABLE | 0 |
| PVOV | SPI MSTR OUTPUT + OC2A | OC1B | OC1A | 0 |
| DIEOE | PCINT3  PCIE0 | PCINT2  PCIE0 | PCINT1  PCIE0 | PCINT0  PCIE0 |
| DIEOV | 1 | 1 | 1 | 1 |
| DI | PCINT3 INPUT<br>SPI SLAVE INPUT | PCINT2 INPUT<br>SPI $\overline{\text{SS}}$ | PCINT1 INPUT | PCINT0 INPUT<br>ICP1 INPUT |
| AIO | – | – | – | – |

### 14.3.2 Alternate Functions of Port C

The Port C pins with alternate functions are shown in Table 14-6.

**Table 14-6.** Port C Pins Alternate Functions

| Port Pin | Alternate Function |
|---|---|
| PC6 | $\overline{\text{RESET}}$ (Reset pin)<br>PCINT14 (Pin Change Interrupt 14) |
| PC5 | ADC5 (ADC Input Channel 5)<br>SCL (2-wire Serial Bus Clock Line)<br>PCINT13 (Pin Change Interrupt 13) |
| PC4 | ADC4 (ADC Input Channel 4)<br>SDA (2-wire Serial Bus Data Input/Output Line)<br>PCINT12 (Pin Change Interrupt 12) |
| PC3 | ADC3 (ADC Input Channel 3)<br>PCINT11 (Pin Change Interrupt 11) |
| PC2 | ADC2 (ADC Input Channel 2)<br>PCINT10 (Pin Change Interrupt 10) |
| PC1 | ADC1 (ADC Input Channel 1)<br>PCINT9 (Pin Change Interrupt 9) |
| PC0 | ADC0 (ADC Input Channel 0)<br>PCINT8 (Pin Change Interrupt 8) |

The alternate pin configuration is as follows:

- **$\overline{\text{RESET}}$/PCINT14 – Port C, Bit 6**

$\overline{\text{RESET}}$, Reset pin: When the RSTDISBL Fuse is programmed, this pin functions as a normal I/O pin, and the part will have to rely on Power-on Reset and Brown-out Reset as its reset sources. When the RSTDISBL Fuse is unprogrammed, the reset circuitry is connected to the pin, and the pin can not be used as an I/O pin.

If PC6 is used as a reset pin, DDC6, PORTC6 and PINC6 will all read 0.

PCINT14: Pin Change Interrupt source 14. The PC6 pin can serve as an external interrupt source.

- **SCL/ADC5/PCINT13 – Port C, Bit 5**

SCL, 2-wire Serial Interface Clock: When the TWEN bit in TWCR is set (one) to enable the 2-wire Serial Interface, pin PC5 is disconnected from the port and becomes the Serial Clock I/O pin for the 2-wire Serial Interface. In this mode, there is a spike filter on the pin to suppress spikes shorter than 50 ns on the input signal, and the pin is driven by an open drain driver with slew-rate limitation.

PC5 can also be used as ADC input Channel 5. Note that ADC input channel 5 uses digital power.

PCINT13: Pin Change Interrupt source 13. The PC5 pin can serve as an external interrupt source.

- **SDA/ADC4/PCINT12 – Port C, Bit 4**

SDA, 2-wire Serial Interface Data: When the TWEN bit in TWCR is set (one) to enable the 2-wire Serial Interface, pin PC4 is disconnected from the port and becomes the Serial Data I/O pin for the 2-wire Serial Interface. In this mode, there is a spike filter on the pin to suppress spikes shorter than 50 ns on the input signal, and the pin is driven by an open drain driver with slew-rate limitation.

PC4 can also be used as ADC input Channel 4. Note that ADC input channel 4 uses digital power.

PCINT12: Pin Change Interrupt source 12. The PC4 pin can serve as an external interrupt source.

- **ADC3/PCINT11 – Port C, Bit 3**

PC3 can also be used as ADC input Channel 3. Note that ADC input channel 3 uses analog power.

PCINT11: Pin Change Interrupt source 11. The PC3 pin can serve as an external interrupt source.

- **ADC2/PCINT10 – Port C, Bit 2**

PC2 can also be used as ADC input Channel 2. Note that ADC input channel 2 uses analog power.

PCINT10: Pin Change Interrupt source 10. The PC2 pin can serve as an external interrupt source.

- **ADC1/PCINT9 – Port C, Bit 1**

PC1 can also be used as ADC input Channel 1. Note that ADC input channel 1 uses analog power.

PCINT9: Pin Change Interrupt source 9. The PC1 pin can serve as an external interrupt source.

- **ADC0/PCINT8 – Port C, Bit 0**

PC0 can also be used as ADC input Channel 0. Note that ADC input channel 0 uses analog power.

PCINT8: Pin Change Interrupt source 8. The PC0 pin can serve as an external interrupt source.

Table 14-7 and Table 14-8 relate the alternate functions of Port C to the overriding signals shown in Figure 14-5 on page 79.

**Table 14-7.** Overriding Signals for Alternate Functions in PC6...PC4[1]

| Signal Name | PC6/$\overline{\text{RESET}}$/PCINT14 | PC5/SCL/ADC5/PCINT13 | PC4/SDA/ADC4/PCINT12 |
|---|---|---|---|
| PUOE | RSTDISBL | TWEN | TWEN |
| PUOV | 1 | PORTC5 $\overline{\text{PUD}}$ | PORTC4 $\overline{\text{PUD}}$ |
| DDOE | RSTDISBL | TWEN | TWEN |
| DDOV | 0 | SCL_OUT | SDA_OUT |
| PVOE | 0 | TWEN | TWEN |
| PVOV | 0 | 0 | 0 |
| DIEOE | RSTDISBL + PCINT14 × PCIE1 | PCINT13 × PCIE1 + ADC5D | PCINT12 × PCIE1 + ADC4D |
| DIEOV | RSTDISBL | PCINT13 × PCIE1 | PCINT12 × PCIE1 |
| DI | PCINT14 INPUT | PCINT13 INPUT | PCINT12 INPUT |
| AIO | RESET INPUT | ADC5 INPUT / SCL INPUT | ADC4 INPUT / SDA INPUT |

Note: 1. When enabled, the 2-wire Serial Interface enables slew-rate controls on the output pins PC4 and PC5. This is not shown in the figure. In addition, spike filters are connected between the AIO outputs shown in the port figure and the digital logic of the TWI module.

**Table 14-8.** Overriding Signals for Alternate Functions in PC3...PC0

| Signal Name | PC3/ADC3/ PCINT11 | PC2/ADC2/ PCINT10 | PC1/ADC1/ PCINT9 | PC0/ADC0/ PCINT8 |
|---|---|---|---|---|
| PUOE | 0 | 0 | 0 | 0 |
| PUOV | 0 | 0 | 0 | 0 |
| DDOE | 0 | 0 | 0 | 0 |
| DDOV | 0 | 0 | 0 | 0 |
| PVOE | 0 | 0 | 0 | 0 |
| PVOV | 0 | 0 | 0 | 0 |
| DIEOE | PCINT11 × PCIE1 + ADC3D | PCINT10 × PCIE1 + ADC2D | PCINT9 × PCIE1 + ADC1D | PCINT8 × PCIE1 + ADC0D |
| DIEOV | PCINT11 × PCIE1 | PCINT10 × PCIE1 | PCINT9 × PCIE1 | PCINT8 × PCIE1 |
| DI | PCINT11 INPUT | PCINT10 INPUT | PCINT9 INPUT | PCINT8 INPUT |
| AIO | ADC3 INPUT | ADC2 INPUT | ADC1 INPUT | ADC0 INPUT |

### 14.3.3 Alternate Functions of Port D

The Port D pins with alternate functions are shown in Table 14-9.

**Table 14-9.** Port D Pins Alternate Functions

| Port Pin | Alternate Function |
|---|---|
| PD7 | AIN1 (Analog Comparator Negative Input)<br>PCINT23 (Pin Change Interrupt 23) |
| PD6 | AIN0 (Analog Comparator Positive Input)<br>OC0A (Timer/Counter0 Output Compare Match A Output)<br>PCINT22 (Pin Change Interrupt 22) |
| PD5 | T1 (Timer/Counter 1 External Counter Input)<br>OC0B (Timer/Counter0 Output Compare Match B Output)<br>PCINT21 (Pin Change Interrupt 21) |
| PD4 | XCK (USART External Clock Input/Output)<br>T0 (Timer/Counter 0 External Counter Input)<br>PCINT20 (Pin Change Interrupt 20) |
| PD3 | INT1 (External Interrupt 1 Input)<br>OC2B (Timer/Counter2 Output Compare Match B Output)<br>PCINT19 (Pin Change Interrupt 19) |
| PD2 | INT0 (External Interrupt 0 Input)<br>PCINT18 (Pin Change Interrupt 18) |
| PD1 | TXD (USART Output Pin)<br>PCINT17 (Pin Change Interrupt 17) |
| PD0 | RXD (USART Input Pin)<br>PCINT16 (Pin Change Interrupt 16) |

The alternate pin configuration is as follows:

- **AIN1/OC2B/PCINT23 – Port D, Bit 7**

AIN1, Analog Comparator Negative Input. Configure the port pin as input with the internal pull-up switched off to avoid the digital port function from interfering with the function of the Analog Comparator.

PCINT23: Pin Change Interrupt source 23. The PD7 pin can serve as an external interrupt source.

- **AIN0/OC0A/PCINT22 – Port D, Bit 6**

AIN0, Analog Comparator Positive Input. Configure the port pin as input with the internal pull-up switched off to avoid the digital port function from interfering with the function of the Analog Comparator.

OC0A, Output Compare Match output: The PD6 pin can serve as an external output for the Timer/Counter0 Compare Match A. The PD6 pin has to be configured as an output (DDD6 set (one)) to serve this function. The OC0A pin is also the output pin for the PWM mode timer function.

PCINT22: Pin Change Interrupt source 22. The PD6 pin can serve as an external interrupt source.

- **T1/OC0B/PCINT21 – Port D, Bit 5**

T1, Timer/Counter1 counter source.

OC0B, Output Compare Match output: The PD5 pin can serve as an external output for the Timer/Counter0 Compare Match B. The PD5 pin has to be configured as an output (DDD5 set (one)) to serve this function. The OC0B pin is also the output pin for the PWM mode timer function.

PCINT21: Pin Change Interrupt source 21. The PD5 pin can serve as an external interrupt source.

- **XCK/T0/PCINT20 – Port D, Bit 4**

XCK, USART external clock.

T0, Timer/Counter0 counter source.

PCINT20: Pin Change Interrupt source 20. The PD4 pin can serve as an external interrupt source.

- **INT1/OC2B/PCINT19 – Port D, Bit 3**

INT1, External Interrupt source 1: The PD3 pin can serve as an external interrupt source.

OC2B, Output Compare Match output: The PD3 pin can serve as an external output for the Timer/Counter0 Compare Match B. The PD3 pin has to be configured as an output (DDD3 set (one)) to serve this function. The OC2B pin is also the output pin for the PWM mode timer function.

PCINT19: Pin Change Interrupt source 19. The PD3 pin can serve as an external interrupt source.

- **INT0/PCINT18 – Port D, Bit 2**

INT0, External Interrupt source 0: The PD2 pin can serve as an external interrupt source.

PCINT18: Pin Change Interrupt source 18. The PD2 pin can serve as an external interrupt source.

- **TXD/PCINT17 – Port D, Bit 1**

TXD, Transmit Data (Data output pin for the USART). When the USART Transmitter is enabled, this pin is configured as an output regardless of the value of DDD1.

PCINT17: Pin Change Interrupt source 17. The PD1 pin can serve as an external interrupt source.

- **RXD/PCINT16 – Port D, Bit 0**

RXD, Receive Data (Data input pin for the USART). When the USART Receiver is enabled this pin is configured as an input regardless of the value of DDD0. When the USART forces this pin to be an input, the pull-up can still be controlled by the PORTD0 bit.

PCINT16: Pin Change Interrupt source 16. The PD0 pin can serve as an external interrupt source.

Table 14-10 and Table 14-11 relate the alternate functions of Port D to the overriding signals shown in Figure 14-5 on page 79.

**Table 14-10.** Overriding Signals for Alternate Functions PD7...PD4

| Signal Name | PD7/AIN1 /PCINT23 | PD6/AIN0/ OC0A/PCINT22 | PD5/T1/OC0B/ PCINT21 | PD4/XCK/ T0/PCINT20 |
|---|---|---|---|---|
| PUOE | 0 | 0 | 0 | 0 |
| PUO | 0 | 0 | 0 | 0 |
| DDOE | 0 | 0 | 0 | 0 |
| DDOV | 0 | 0 | 0 | 0 |
| PVOE | 0 | OC0A ENABLE | OC0B ENABLE | UMSEL |
| PVOV | 0 | OC0A | OC0B | XCK OUTPUT |
| DIEOE | PCINT23 × PCIE2 | PCINT22 × PCIE2 | PCINT21 × PCIE2 | PCINT20 × PCIE2 |
| DIEOV | 1 | 1 | 1 | 1 |
| DI | PCINT23 INPUT | PCINT22 INPUT | PCINT21 INPUT<br>T1 INPUT | PCINT20 INPUT<br>XCK INPUT<br>T0 INPUT |
| AIO | AIN1 INPUT | AIN0 INPUT | – | – |

**Table 14-11.** Overriding Signals for Alternate Functions in PD3...PD0

| Signal Name | PD3/OC2B/INT1/ PCINT19 | PD2/INT0/ PCINT18 | PD1/TXD/ PCINT17 | PD0/RXD/ PCINT16 |
|---|---|---|---|---|
| PUOE | 0 | 0 | TXEN | RXEN |
| PUO | 0 | 0 | 0 | PORTD0 × $\overline{\text{PUD}}$ |
| DDOE | 0 | 0 | TXEN | RXEN |
| DDOV | 0 | 0 | 1 | 0 |
| PVOE | OC2B ENABLE | 0 | TXEN | 0 |
| PVOV | OC2B | 0 | TXD | 0 |
| DIEOE | INT1 ENABLE + PCINT19 × PCIE2 | INT0 ENABLE + PCINT18 × PCIE1 | PCINT17 × PCIE2 | PCINT16 × PCIE2 |
| DIEOV | 1 | 1 | 1 | 1 |
| DI | PCINT19 INPUT<br>INT1 INPUT | PCINT18 INPUT<br>INT0 INPUT | PCINT17 INPUT | PCINT16 INPUT<br>RXD |
| AIO | – | – | – | – |

## 14.4 Register Description

### 14.4.1 MCUCR – MCU Control Register

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| 0x35 (0x55) | – | BODS[1] | BODSE[1] | PUD | – | – | IVSEL | IVCE | MCUCR |
| Read/Write | R | R/W | R/W | R/W | R | R | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

Note: 1. BODS and BODSE only available for picoPower devices ATmega48PA/88PA/168PA

- **Bit 4 – PUD: Pull-up Disable**

When this bit is written to one, the pull-ups in the I/O ports are disabled even if the DDxn and PORTxn Registers are configured to enable the pull-ups ({DDxn, PORTxn} = 0b01). See "Configuring the Pin" on page 74 for more details about this feature.

### 14.4.2 PORTB – The Port B Data Register

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| 0x05 (0x25) | PORTB7 | PORTB6 | PORTB5 | PORTB4 | PORTB3 | PORTB2 | PORTB1 | PORTB0 | PORTB |
| Read/Write | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

### 14.4.3 DDRB – The Port B Data Direction Register

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| 0x04 (0x24) | DDB7 | DDB6 | DDB5 | DDB4 | DDB3 | DDB2 | DDB1 | DDB0 | DDRB |
| Read/Write | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

### 14.4.4 PINB – The Port B Input Pins Address

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| 0x03 (0x23) | PINB7 | PINB6 | PINB5 | PINB4 | PINB3 | PINB2 | PINB1 | PINB0 | PINB |
| Read/Write | R | R | R | R | R | R | R | R | |
| Initial Value | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | |

### 14.4.5 PORTC – The Port C Data Register

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| 0x08 (0x28) | – | PORTC6 | PORTC5 | PORTC4 | PORTC3 | PORTC2 | PORTC1 | PORTC0 | PORTC |
| Read/Write | R | R/W | R/W | R/W | R/W | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

### 14.4.6 DDRC – The Port C Data Direction Register

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| 0x07 (0x27) | – | DDC6 | DDC5 | DDC4 | DDC3 | DDC2 | DDC1 | DDC0 | DDRC |
| Read/Write | R | R/W | R/W | R/W | R/W | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

### 14.4.7 PINC – The Port C Input Pins Address

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| 0x06 (0x26) | – | PINC6 | PINC5 | PINC4 | PINC3 | PINC2 | PINC1 | PINC0 | PINC |
| Read/Write | R | R | R | R | R | R | R | R | |
| Initial Value | 0 | N/A | N/A | N/A | N/A | N/A | N/A | N/A | |

### 14.4.8 PORTD – The Port D Data Register

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| 0x0B (0x2B) | **PORTD7** | **PORTD6** | **PORTD5** | **PORTD4** | **PORTD3** | **PORTD2** | **PORTD1** | **PORTD0** | **PORTD** |
| Read/Write | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

### 14.4.9 DDRD – The Port D Data Direction Register

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| 0x0A (0x2A) | **DDD7** | **DDD6** | **DDD5** | **DDD4** | **DDD3** | **DDD2** | **DDD1** | **DDD0** | **DDRD** |
| Read/Write | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

### 14.4.10 PIND – The Port D Input Pins Address

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| 0x09 (0x29) | **PIND7** | **PIND6** | **PIND5** | **PIND4** | **PIND3** | **PIND2** | **PIND1** | **PIND0** | **PIND** |
| Read/Write | R | R | R | R | R | R | R | R | |
| Initial Value | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | |

# 15. 8-bit Timer/Counter0 with PWM

## 15.1 Features

- **Two Independent Output Compare Units**
- **Double Buffered Output Compare Registers**
- **Clear Timer on Compare Match (Auto Reload)**
- **Glitch Free, Phase Correct Pulse Width Modulator (PWM)**
- **Variable PWM Period**
- **Frequency Generator**
- **Three Independent Interrupt Sources (TOV0, OCF0A, and OCF0B)**

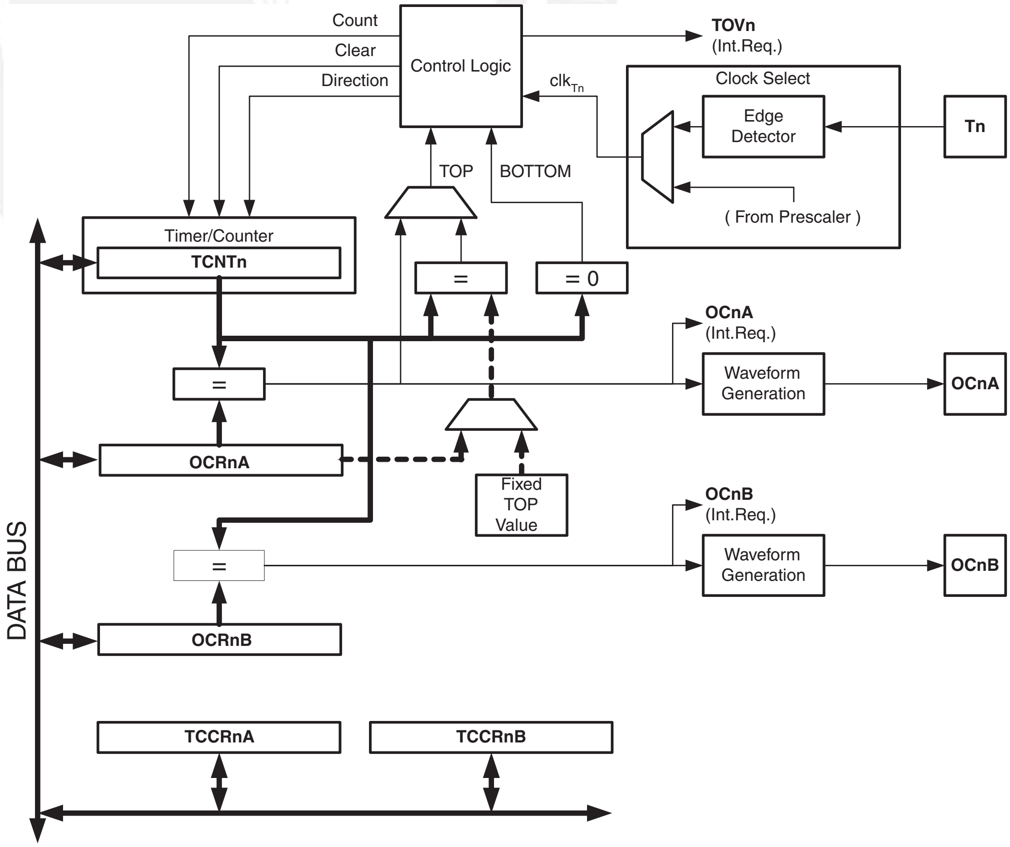
## 15.2 Overview

Timer/Counter0 is a general purpose 8-bit Timer/Counter module, with two independent Output Compare Units, and with PWM support. It allows accurate program execution timing (event management) and wave generation.

A simplified block diagram of the 8-bit Timer/Counter is shown in Figure 15-1. For the actual placement of I/O pins, refer to "Pinout Atmel® ATmega48PA/88PA/168PA" on page 2. CPU accessible I/O Registers, including I/O bits and I/O pins, are shown in bold. The device-specific I/O Register and bit locations are listed in the "Register Description" on page 104.

The PRTIM0 bit in "Minimizing Power Consumption" on page 42 must be written to zero to enable Timer/Counter0 module.

**Figure 15-1.** 8-bit Timer/Counter Block Diagram



### 15.2.1 Definitions

Many register and bit references in this section are written in general form. A lower case “n” replaces the Timer/Counter number, in this case 0. A lower case “x” replaces the Output Compare Unit, in this case Compare Unit A or Compare Unit B. However, when using the register or bit defines in a program, the precise form must be used, i.e., TCNT0 for accessing Timer/Counter0 counter value and so on.

The definitions in Table 15-1 are also used extensively throughout the document.

**Table 15-1.** Definitions

| | |
|---|---|
| **BOTTOM** | The counter reaches the BOTTOM when it becomes 0x00. |
| **MAX** | The counter reaches its MAXimum when it becomes 0xFF (decimal 255). |
| **TOP** | The counter reaches the TOP when it becomes equal to the highest value in the count sequence. The TOP value can be assigned to be the fixed value 0xFF (MAX) or the value stored in the OCR0A Register. The assignment is dependent on the mode of operation. |

### 15.2.2 Registers

The Timer/Counter (TCNT0) and Output Compare Registers (OCR0A and OCR0B) are 8-bit registers. Interrupt request (abbreviated to Int.Req. in the figure) signals are all visible in the Timer Interrupt Flag Register (TIFR0). All interrupts are individually masked with the Timer Interrupt Mask Register (TIMSK0). TIFR0 and TIMSK0 are not shown in the figure.

The Timer/Counter can be clocked internally, via the prescaler, or by an external clock source on the T0 pin. The Clock Select logic block controls which clock source and edge the Timer/Counter uses to increment (or decrement) its value. The Timer/Counter is inactive when no clock source is selected. The output from the Clock Select logic is referred to as the timer clock ($clk_{T0}$).

The double buffered Output Compare Registers (OCR0A and OCR0B) are compared with the Timer/Counter value at all times. The result of the compare can be used by the Waveform Generator to generate a PWM or variable frequency output on the Output Compare pins (OC0A and OC0B). See “Using the Output Compare Unit” on page 121. for details. The compare match event will also set the Compare Flag (OCF0A or OCF0B) which can be used to generate an Output Compare interrupt request.

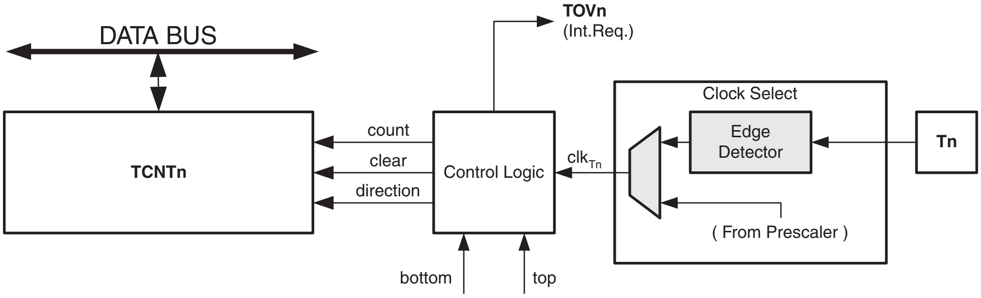
## 15.3 Timer/Counter Clock Sources

The Timer/Counter can be clocked by an internal or an external clock source. The clock source is selected by the Clock Select logic which is controlled by the Clock Select (CS02:0) bits located in the Timer/Counter Control Register (TCCR0B). For details on clock sources and prescaler, see “Timer/Counter0 and Timer/Counter1 Prescalers” on page 140.

## 15.4 Counter Unit

The main part of the 8-bit Timer/Counter is the programmable bi-directional counter unit. Figure 15-2 shows a block diagram of the counter and its surroundings.

**Figure 15-2.** Counter Unit Block Diagram



Signal description (internal signals):

**count** Increment or decrement TCNT0 by 1.

**direction** Select between increment and decrement.

**clear** Clear TCNT0 (set all bits to zero).

**$clk_{Tn}$** Timer/Counter clock, referred to as $clk_{T0}$ in the following.

**top** Signalize that TCNT0 has reached maximum value.

**bottom** Signalize that TCNT0 has reached minimum value (zero).

Depending of the mode of operation used, the counter is cleared, incremented, or decremented at each timer clock ($clk_{T0}$). $clk_{T0}$ can be generated from an external or internal clock source, selected by the Clock Select bits (CS02:0). When no clock source is selected (CS02:0 = 0) the timer is stopped. However, the TCNT0 value can be accessed by the CPU, regardless of whether $clk_{T0}$ is present or not. A CPU write overrides (has priority over) all counter clear or count operations.

The counting sequence is determined by the setting of the WGM01 and WGM00 bits located in the Timer/Counter Control Register (TCCR0A) and the WGM02 bit located in the Timer/Counter Control Register B (TCCR0B). There are close connections between how the counter behaves (counts) and how waveforms are generated on the Output Compare outputs OC0A and OC0B. For more details about advanced counting sequences and waveform generation, see "Modes of Operation" on page 98.

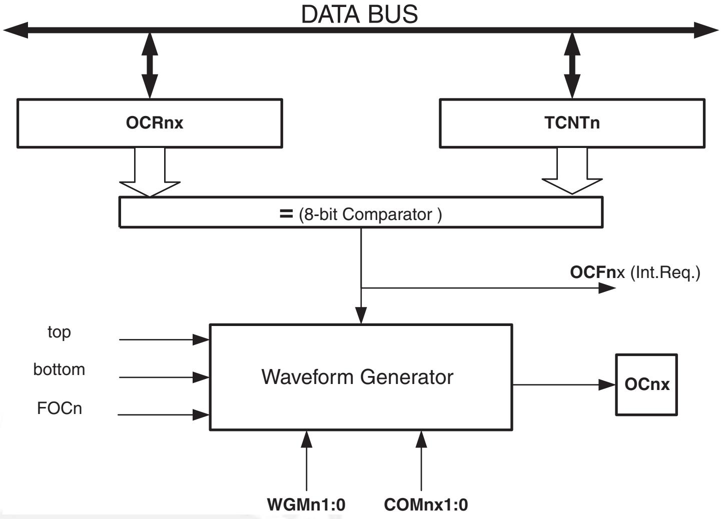
The Timer/Counter Overflow Flag (TOV0) is set according to the mode of operation selected by the WGM02:0 bits. TOV0 can be used for generating a CPU interrupt.

## 15.5 Output Compare Unit

The 8-bit comparator continuously compares TCNT0 with the Output Compare Registers (OCR0A and OCR0B). Whenever TCNT0 equals OCR0A or OCR0B, the comparator signals a match. A match will set the Output Compare Flag (OCF0A or OCF0B) at the next timer clock cycle. If the corresponding interrupt is enabled, the Output Compare Flag generates an Output Compare interrupt. The Output Compare Flag is automatically cleared when the interrupt is executed. Alternatively, the flag can be cleared by software by writing a logical one to its I/O bit location. The Waveform Generator uses the match signal to generate an output according to operating mode set by the WGM02:0 bits and Compare Output mode (COM0x1:0) bits. The max and bottom signals are used by the Waveform Generator for handling the special cases of the extreme values in some modes of operation ("Modes of Operation" on page 98).

Figure 15-3 shows a block diagram of the Output Compare unit.

**Figure 15-3.** Output Compare Unit, Block Diagram



The OCR0x Registers are double buffered when using any of the Pulse Width Modulation (PWM) modes. For the normal and Clear Timer on Compare (CTC) modes of operation, the double buffering is disabled. The double buffering synchronizes the update of the OCR0x Compare Registers to either top or bottom of the counting sequence. The synchronization prevents the occurrence of odd-length, non-symmetrical PWM pulses, thereby making the output glitch-free.

The OCR0x Register access may seem complex, but this is not case. When the double buffering is enabled, the CPU has access to the OCR0x Buffer Register, and if double buffering is disabled the CPU will access the OCR0x directly.

### 15.5.1 Force Output Compare

In non-PWM waveform generation modes, the match output of the comparator can be forced by writing a one to the Force Output Compare (FOC0x) bit. Forcing compare match will not set the OCF0x Flag or reload/clear the timer, but the OC0x pin will be updated as if a real compare match had occurred (the COM0x1:0 bits settings define whether the OC0x pin is set, cleared or toggled).

### 15.5.2 Compare Match Blocking by TCNT0 Write

All CPU write operations to the TCNT0 Register will block any compare match that occur in the next timer clock cycle, even when the timer is stopped. This feature allows OCR0x to be initialized to the same value as TCNT0 without triggering an interrupt when the Timer/Counter clock is enabled.

### 15.5.3 Using the Output Compare Unit

Since writing TCNT0 in any mode of operation will block all compare matches for one timer clock cycle, there are risks involved when changing TCNT0 when using the Output Compare Unit, independently of whether the Timer/Counter is running or not. If the value written to TCNT0 equals the OCR0x value, the compare match will be missed, resulting in incorrect waveform generation. Similarly, do not write the TCNT0 value equal to BOTTOM when the counter is downcounting.

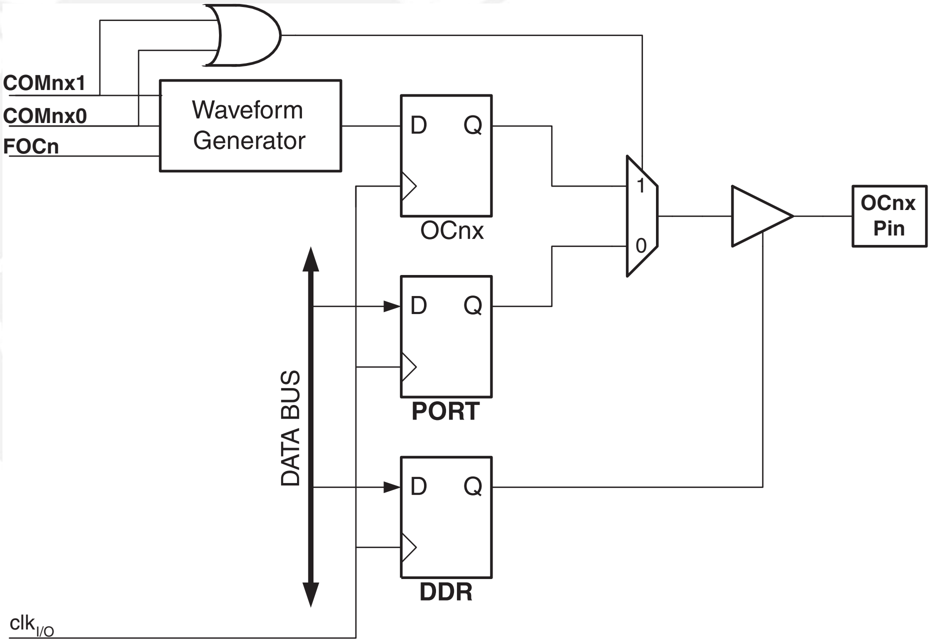
The setup of the OC0x should be performed before setting the Data Direction Register for the port pin to output. The easiest way of setting the OC0x value is to use the Force Output Compare (FOC0x) strobe bits in Normal mode. The OC0x Registers keep their values even when changing between Waveform Generation modes.

Be aware that the COM0x1:0 bits are not double buffered together with the compare value. Changing the COM0x1:0 bits will take effect immediately.

## 15.6 Compare Match Output Unit

The Compare Output mode (COM0x1:0) bits have two functions. The Waveform Generator uses the COM0x1:0 bits for defining the Output Compare (OC0x) state at the next compare match. Also, the COM0x1:0 bits control the OC0x pin output source. Figure 15-4 shows a simplified schematic of the logic affected by the COM0x1:0 bit setting. The I/O Registers, I/O bits, and I/O pins in the figure are shown in bold. Only the parts of the general I/O port control registers (DDR and PORT) that are affected by the COM0x1:0 bits are shown. When referring to the OC0x state, the reference is for the internal OC0x Register, not the OC0x pin. If a system reset occur, the OC0x Register is reset to “0”.

**Figure 15-4.** Compare Match Output Unit, Schematic



The general I/O port function is overridden by the Output Compare (OC0x) from the Waveform Generator if either of the COM0x1:0 bits are set. However, the OC0x pin direction (input or output) is still controlled by the Data Direction Register (DDR) for the port pin. The Data Direction Register bit for the OC0x pin (DDR_OC0x) must be set as output before the OC0x value is visible on the pin. The port override function is independent of the Waveform Generation mode.

The design of the Output Compare pin logic allows initialization of the OC0x state before the output is enabled. Note that some COM0x1:0 bit settings are reserved for certain modes of operation. See “Register Description” on page 104.

### 15.6.1 Compare Output Mode and Waveform Generation

The Waveform Generator uses the COM0x1:0 bits differently in Normal, CTC, and PWM modes. For all modes, setting the COM0x1:0 = 0 tells the Waveform Generator that no action on the OC0x Register is to be performed on the next compare match. For compare output actions in the non-PWM modes refer to Table 15-2 on page 104. For fast PWM mode, refer to Table 15-3 on page 104, and for phase correct PWM refer to Table 15-4 on page 105.

A change of the COM0x1:0 bits state will have effect at the first compare match after the bits are written. For non-PWM modes, the action can be forced to have immediate effect by using the FOC0x strobe bits.

## 15.7 Modes of Operation

The mode of operation, i.e., the behavior of the Timer/Counter and the Output Compare pins, is defined by the combination of the Waveform Generation mode (WGM02:0) and Compare Output mode (COM0x1:0) bits. The Compare Output mode bits do not affect the counting sequence, while the Waveform Generation mode bits do. The COM0x1:0 bits control whether the PWM output generated should be inverted or not (inverted or non-inverted PWM). For non-PWM modes the COM0x1:0 bits control whether the output should be set, cleared, or toggled at a compare match (See "Compare Match Output Unit" on page 97.).

For detailed timing information refer to "Timer/Counter Timing Diagrams" on page 102.

### 15.7.1 Normal Mode

The simplest mode of operation is the Normal mode (WGM02:0 = 0). In this mode the counting direction is always up (incrementing), and no counter clear is performed. The counter simply overruns when it passes its maximum 8-bit value (TOP = 0xFF) and then restarts from the bottom (0x00). In normal operation the Timer/Counter Overflow Flag (TOV0) will be set in the same timer clock cycle as the TCNT0 becomes zero. The TOV0 Flag in this case behaves like a ninth bit, except that it is only set, not cleared. However, combined with the timer overflow interrupt that automatically clears the TOV0 Flag, the timer resolution can be increased by software. There are no special cases to consider in the Normal mode, a new counter value can be written anytime.

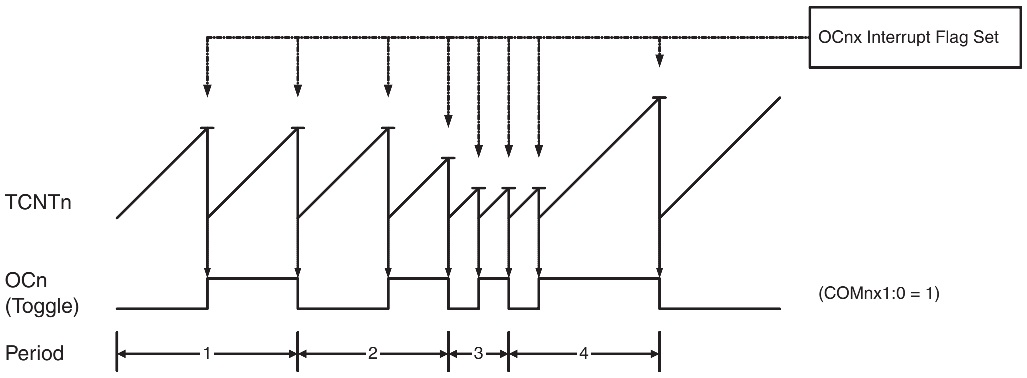
The Output Compare unit can be used to generate interrupts at some given time. Using the Output Compare to generate waveforms in Normal mode is not recommended, since this will occupy too much of the CPU time.

### 15.7.2 Clear Timer on Compare Match (CTC) Mode

In Clear Timer on Compare or CTC mode (WGM02:0 = 2), the OCR0A Register is used to manipulate the counter resolution. In CTC mode the counter is cleared to zero when the counter value (TCNT0) matches the OCR0A. The OCR0A defines the top value for the counter, hence also its resolution. This mode allows greater control of the compare match output frequency. It also simplifies the operation of counting external events.

The timing diagram for the CTC mode is shown in Figure 15-5. The counter value (TCNT0) increases until a compare match occurs between TCNT0 and OCR0A, and then counter (TCNT0) is cleared.

**Figure 15-5.** CTC Mode, Timing Diagram



An interrupt can be generated each time the counter value reaches the TOP value by using the OCF0A Flag. If the interrupt is enabled, the interrupt handler routine can be used for updating the TOP value. However, changing TOP to a value close to BOTTOM when the counter is running with none or a low prescaler value must be done with care since the CTC mode does not have the double buffering feature. If the new value written to OCR0A is lower than the current value of TCNT0, the counter will miss the compare match. The counter will then have to count to its maximum value (0xFF) and wrap around starting at 0x00 before the compare match can occur.

For generating a waveform output in CTC mode, the OC0A output can be set to toggle its logical level on each compare match by setting the Compare Output mode bits to toggle mode (COM0A1:0 = 1). The OC0A value will not be visible on the port pin unless the data direction for the pin is set to output. The waveform generated will have a maximum frequency of $f_{OC0} = f_{clk\_I/O}/2$ when OCR0A is set to zero (0x00). The waveform frequency is defined by the following equation:

$$f_{OCnx} = \frac{f_{clk\_I/O}}{2 \times N \times (1 + OCRnx)}$$

The *N* variable represents the prescale factor (1, 8, 64, 256, or 1024).

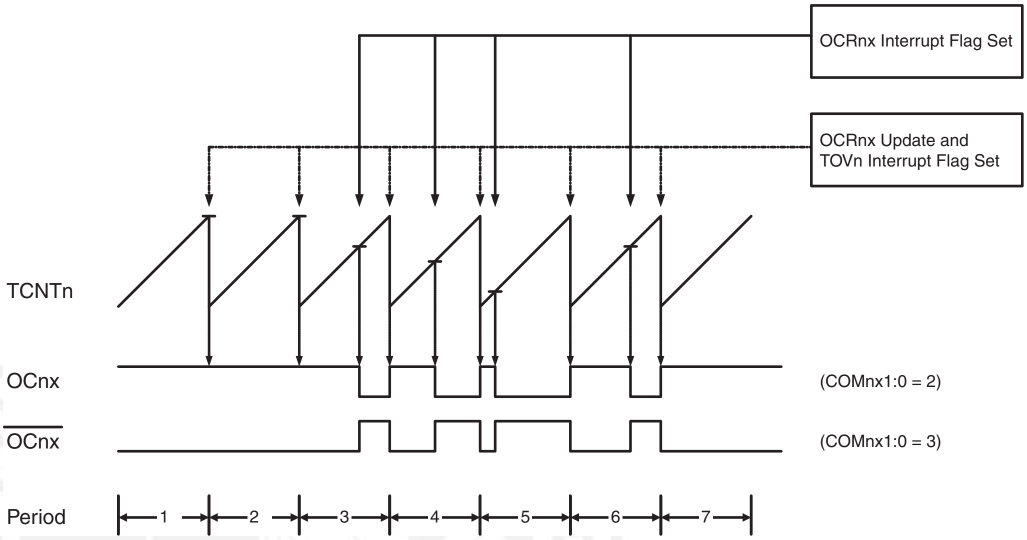
As for the Normal mode of operation, the TOV0 Flag is set in the same timer clock cycle that the counter counts from MAX to 0x00.

### 15.7.3 Fast PWM Mode

The fast Pulse Width Modulation or fast PWM mode (WGM02:0 = 3 or 7) provides a high frequency PWM waveform generation option. The fast PWM differs from the other PWM option by its single-slope operation. The counter counts from BOTTOM to TOP then restarts from BOTTOM. TOP is defined as 0xFF when WGM2:0 = 3, and OCR0A when WGM2:0 = 7. In non-inverting Compare Output mode, the Output Compare (OC0x) is cleared on the compare match between TCNT0 and OCR0x, and set at BOTTOM. In inverting Compare Output mode, the output is set on compare match and cleared at BOTTOM. Due to the single-slope operation, the operating frequency of the fast PWM mode can be twice as high as the phase correct PWM mode that use dual-slope operation. This high frequency makes the fast PWM mode well suited for power regulation, rectification, and DAC applications. High frequency allows physically small sized external components (coils, capacitors), and therefore reduces total system cost.

In fast PWM mode, the counter is incremented until the counter value matches the TOP value. The counter is then cleared at the following timer clock cycle. The timing diagram for the fast PWM mode is shown in Figure 15-6. The TCNT0 value is in the timing diagram shown as a histogram for illustrating the single-slope operation. The diagram includes non-inverted and inverted PWM outputs. The small horizontal line marks on the TCNT0 slopes represent compare matches between OCR0x and TCNT0.

**Figure 15-6.** Fast PWM Mode, Timing Diagram



The Timer/Counter Overflow Flag (TOV0) is set each time the counter reaches TOP. If the interrupt is enabled, the interrupt handler routine can be used for updating the compare value.

In fast PWM mode, the compare unit allows generation of PWM waveforms on the OC0x pins. Setting the COM0x1:0 bits to two will produce a non-inverted PWM and an inverted PWM output can be generated by setting the COM0x1:0 to three: Setting the COM0A1:0 bits to one allows the OC0A pin to toggle on Compare Matches if the WGM02 bit is set. This option is not available for the OC0B pin (see Table 15-6 on page 105). The actual OC0x value will only be visible on the port pin if the data direction for the port pin is set as output. The PWM waveform is generated by setting (or clearing) the OC0x Register at the compare match between OCR0x and TCNT0, and clearing (or setting) the OC0x Register at the timer clock cycle the counter is cleared (changes from TOP to BOTTOM).

The PWM frequency for the output can be calculated by the following equation:

$$f_{OCnxPWM} = \frac{f_{clk\_I/O}}{N \times 256}$$

The *N* variable represents the prescale factor (1, 8, 64, 256, or 1024).

The extreme values for the OCR0A Register represents special cases when generating a PWM waveform output in the fast PWM mode. If the OCR0A is set equal to BOTTOM, the output will be a narrow spike for each MAX+1 timer clock cycle. Setting the OCR0A equal to MAX will result in a constantly high or low output (depending on the polarity of the output set by the COM0A1:0 bits.)

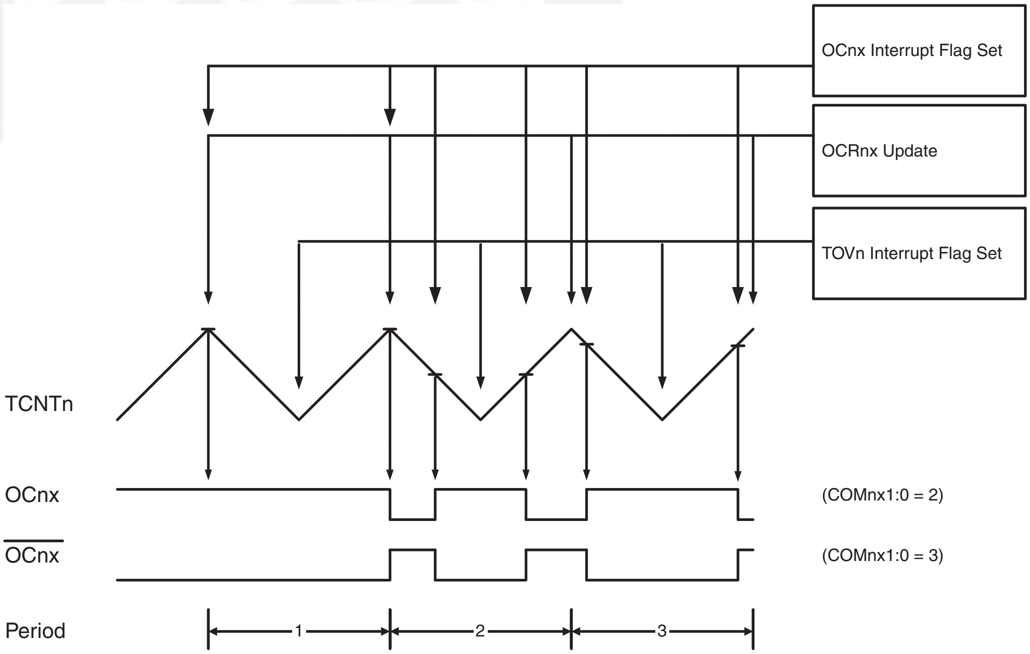
A frequency (with 50% duty cycle) waveform output in fast PWM mode can be achieved by setting OC0x to toggle its logical level on each compare match (COM0x1:0 = 1). The waveform generated will have a maximum frequency of $f_{OC0} = f_{clk\_I/O}/2$ when OCR0A is set to zero. This feature is similar to the OC0A toggle in CTC mode, except the double buffer feature of the Output Compare unit is enabled in the fast PWM mode.

### 15.7.4 Phase Correct PWM Mode

The phase correct PWM mode (WGM02:0 = 1 or 5) provides a high resolution phase correct PWM waveform generation option. The phase correct PWM mode is based on a dual-slope operation. The counter counts repeatedly from BOTTOM to TOP and then from TOP to BOTTOM. TOP is defined as 0xFF when WGM2:0 = 1, and OCR0A when WGM2:0 = 5. In non-inverting Compare Output mode, the Output Compare (OC0x) is cleared on the compare match between TCNT0 and OCR0x while upcounting, and set on the compare match while downcounting. In inverting Output Compare mode, the operation is inverted. The dual-slope operation has lower maximum operation frequency than single slope operation. However, due to the symmetric feature of the dual-slope PWM modes, these modes are preferred for motor control applications.

In phase correct PWM mode the counter is incremented until the counter value matches TOP. When the counter reaches TOP, it changes the count direction. The TCNT0 value will be equal to TOP for one timer clock cycle. The timing diagram for the phase correct PWM mode is shown on Figure 15-7. The TCNT0 value is in the timing diagram shown as a histogram for illustrating the dual-slope operation. The diagram includes non-inverted and inverted PWM outputs. The small horizontal line marks on the TCNT0 slopes represent compare matches between OCR0x and TCNT0.

**Figure 15-7.** Phase Correct PWM Mode, Timing Diagram



The Timer/Counter Overflow Flag (TOV0) is set each time the counter reaches BOTTOM. The Interrupt Flag can be used to generate an interrupt each time the counter reaches the BOTTOM value.

In phase correct PWM mode, the compare unit allows generation of PWM waveforms on the OC0x pins. Setting the COM0x1:0 bits to two will produce a non-inverted PWM. An inverted PWM output can be generated by setting the COM0x1:0 to three: Setting the COM0A0 bits to one allows the OC0A pin to toggle on Compare Matches if the WGM02 bit is set. This option is not available for the OC0B pin (see Table 15-7 on page 106). The actual OC0x value will only be visible on the port pin if the data direction for the port pin is set as output. The PWM waveform is generated by clearing (or setting) the OC0x Register at the compare match between OCR0x and TCNT0 when the counter increments, and setting (or clearing) the OC0x Register at compare match between OCR0x and TCNT0 when the counter decrements. The PWM frequency for the output when using phase correct PWM can be calculated by the following equation:

$$f_{OCnxPCPWM} = \frac{f_{clk\_I/O}}{N \times 510}$$

The N variable represents the prescale factor (1, 8, 64, 256, or 1024).

The extreme values for the OCR0A Register represent special cases when generating a PWM waveform output in the phase correct PWM mode. If the OCR0A is set equal to BOTTOM, the output will be continuously low and if set equal to MAX the output will be continuously high for non-inverted PWM mode. For inverted PWM the output will have the opposite logic values.

At the very start of period 2 in Figure 15-7 OCnx has a transition from high to low even though there is no Compare Match. The point of this transition is to guarantee symmetry around BOTTOM. There are two cases that give a transition without Compare Match.

- OCRnx changes its value from MAX, like in Figure 15-7. When the OCR0A value is MAX the OCn pin value is the same as the result of a down-counting Compare Match. To ensure symmetry around BOTTOM the OCnx value at MAX must correspond to the result of an up-counting Compare Match.
- The timer starts counting from a value higher than the one in OCRnx, and for that reason misses the Compare Match and hence the OCnx change that would have happened on the way up.

## 15.8 Timer/Counter Timing Diagrams

The Timer/Counter is a synchronous design and the timer clock ($clk_{T0}$) is therefore shown as a clock enable signal in the following figures. The figures include information on when interrupt flags are set. Figure 15-8 contains timing data for basic Timer/Counter operation. The figure shows the count sequence close to the MAX value in all modes other than phase correct PWM mode.

**Figure 15-8.** Timer/Counter Timing Diagram, no Prescaling

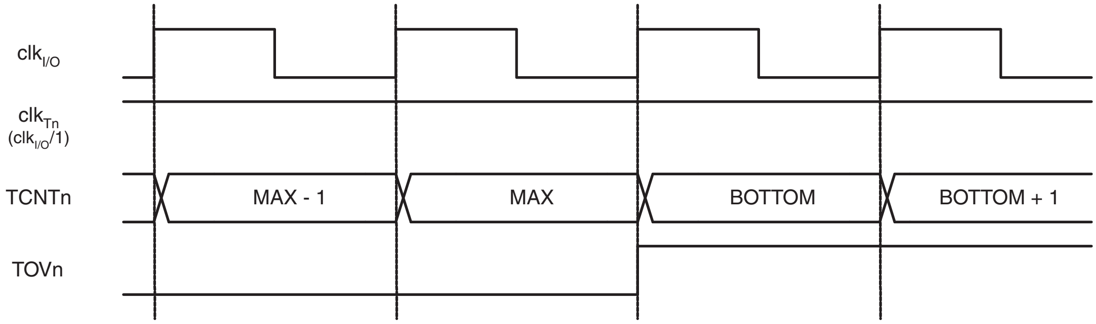


Figure 15-9 shows the same timing data, but with the prescaler enabled.

**Figure 15-9.** Timer/Counter Timing Diagram, with Prescaler ($f_{clk\_I/O}/8$)

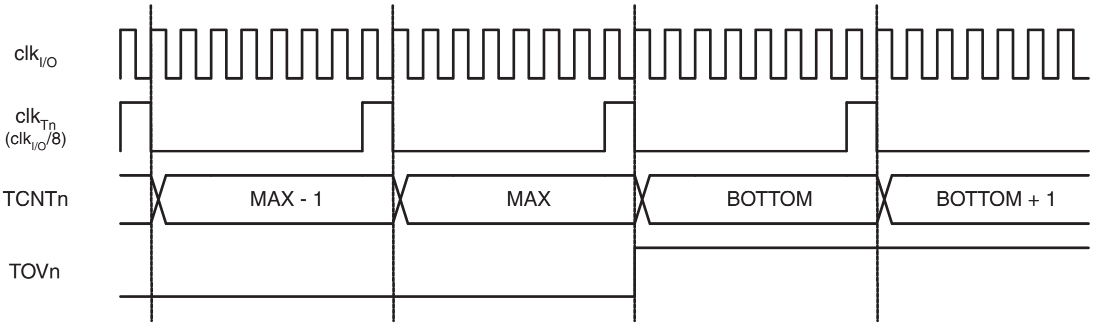


Figure 15-10 shows the setting of OCF0B in all modes and OCF0A in all modes except CTC mode and PWM mode, where OCR0A is TOP.

**Figure 15-10.** Timer/Counter Timing Diagram, Setting of OCF0x, with Prescaler ($f_{clk\_I/O}/8$)

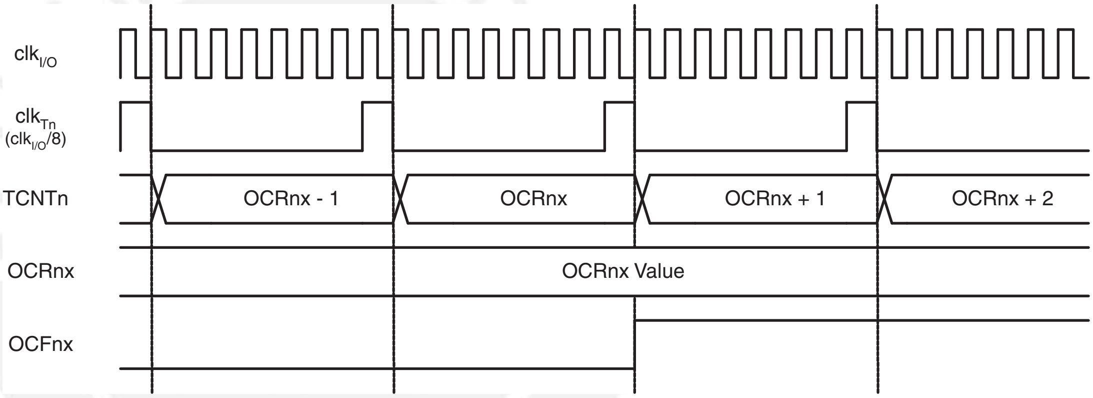
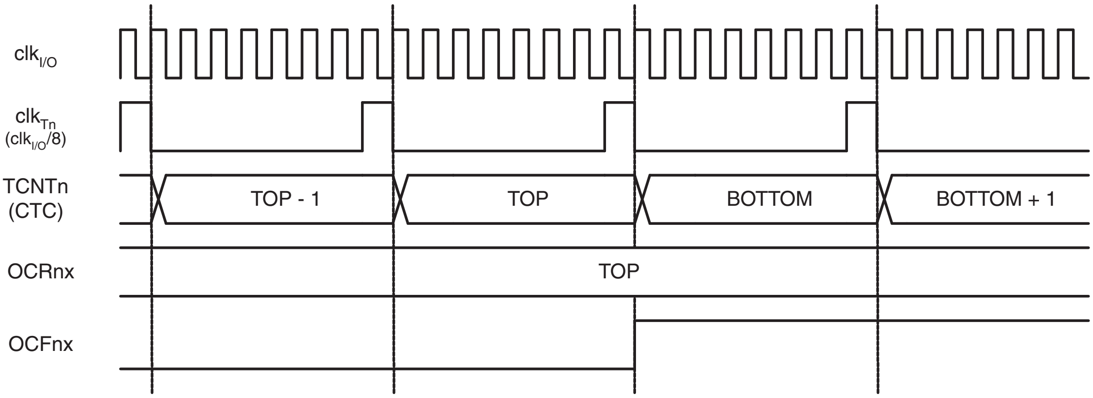


Figure 15-11 shows the setting of OCF0A and the clearing of TCNT0 in CTC mode and fast PWM mode where OCR0A is TOP.

**Figure 15-11.** Timer/Counter Timing Diagram, Clear Timer on Compare Match mode, with Prescaler ($f_{clk\_I/O}/8$)



## 15.9 Register Description

### 15.9.1 TCCR0A – Timer/Counter Control Register A

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| 0x24 (0x44) | COM0A1 | COM0A0 | COM0B1 | COM0B0 | – | – | WGM01 | WGM00 | TCCR0A |
| Read/Write | R/W | R/W | R/W | R/W | R | R | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

- **Bits 7:6 – COM0A1:0: Compare Match Output A Mode**

These bits control the Output Compare pin (OC0A) behavior. If one or both of the COM0A1:0 bits are set, the OC0A output overrides the normal port functionality of the I/O pin it is connected to. However, note that the Data Direction Register (DDR) bit corresponding to the OC0A pin must be set in order to enable the output driver.

When OC0A is connected to the pin, the function of the COM0A1:0 bits depends on the WGM02:0 bit setting. Table 15-2 shows the COM0A1:0 bit functionality when the WGM02:0 bits are set to a normal or CTC mode (non-PWM).

**Table 15-2.** Compare Output Mode, non-PWM Mode

| COM0A1 | COM0A0 | Description |
|---|---|---|
| 0 | 0 | Normal port operation, OC0A disconnected. |
| 0 | 1 | Toggle OC0A on Compare Match |
| 1 | 0 | Clear OC0A on Compare Match |
| 1 | 1 | Set OC0A on Compare Match |

Table 15-3 shows the COM0A1:0 bit functionality when the WGM01:0 bits are set to fast PWM mode.

**Table 15-3.** Compare Output Mode, Fast PWM Mode[1]

| COM0A1 | COM0A0 | Description |
|---|---|---|
| 0 | 0 | Normal port operation, OC0A disconnected. |
| 0 | 1 | WGM02 = 0: Normal Port Operation, OC0A Disconnected.<br>WGM02 = 1: Toggle OC0A on Compare Match. |
| 1 | 0 | Clear OC0A on Compare Match, set OC0A at BOTTOM, (non-inverting mode). |
| 1 | 1 | Set OC0A on Compare Match, clear OC0A at BOTTOM, (inverting mode). |

Note: 1. A special case occurs when OCR0A equals TOP and COM0A1 is set. In this case, the Compare Match is ignored, but the set or clear is done at BOTTOM. See "Fast PWM Mode" on page 99 for more details.

Table 15-4 shows the COM0A1:0 bit functionality when the WGM02:0 bits are set to phase correct PWM mode.

**Table 15-4.** Compare Output Mode, Phase Correct PWM Mode[1]

| COM0A1 | COM0A0 | Description |
|---|---|---|
| 0 | 0 | Normal port operation, OC0A disconnected. |
| 0 | 1 | WGM02 = 0: Normal Port Operation, OC0A Disconnected.<br>WGM02 = 1: Toggle OC0A on Compare Match. |
| 1 | 0 | Clear OC0A on Compare Match when up-counting. Set OC0A on Compare Match when down-counting. |
| 1 | 1 | Set OC0A on Compare Match when up-counting. Clear OC0A on Compare Match when down-counting. |

Note: 1. A special case occurs when OCR0A equals TOP and COM0A1 is set. In this case, the Compare Match is ignored, but the set or clear is done at TOP. See "Phase Correct PWM Mode" on page 127 for more details.

- **Bits 5:4 – COM0B1:0: Compare Match Output B Mode**

These bits control the Output Compare pin (OC0B) behavior. If one or both of the COM0B1:0 bits are set, the OC0B output overrides the normal port functionality of the I/O pin it is connected to. However, note that the Data Direction Register (DDR) bit corresponding to the OC0B pin must be set in order to enable the output driver.

When OC0B is connected to the pin, the function of the COM0B1:0 bits depends on the WGM02:0 bit setting. Table 15-5 on page 105 shows the COM0B1:0 bit functionality when the WGM02:0 bits are set to a normal or CTC mode (non-PWM).

**Table 15-5.** Compare Output Mode, non-PWM Mode

| COM0B1 | COM0B0 | Description |
|---|---|---|
| 0 | 0 | Normal port operation, OC0B disconnected. |
| 0 | 1 | Toggle OC0B on Compare Match |
| 1 | 0 | Clear OC0B on Compare Match |
| 1 | 1 | Set OC0B on Compare Match |

Table 15-6 shows the COM0B1:0 bit functionality when the WGM02:0 bits are set to fast PWM mode.

**Table 15-6.** Compare Output Mode, Fast PWM Mode[1]

| COM0B1 | COM0B0 | Description |
|---|---|---|
| 0 | 0 | Normal port operation, OC0B disconnected. |
| 0 | 1 | Reserved |
| 1 | 0 | Clear OC0B on Compare Match, set OC0B at BOTTOM, (non-inverting mode) |
| 1 | 1 | Set OC0B on Compare Match, clear OC0B at BOTTOM, (inverting mode). |

Note: 1. A special case occurs when OCR0B equals TOP and COM0B1 is set. In this case, the Compare Match is ignored, but the set or clear is done at TOP. See "Fast PWM Mode" on page 99 for more details.

Table 15-7 shows the COM0B1:0 bit functionality when the WGM02:0 bits are set to phase correct PWM mode.

**Table 15-7.** Compare Output Mode, Phase Correct PWM Mode[1]

| COM0B1 | COM0B0 | Description |
|---|---|---|
| 0 | 0 | Normal port operation, OC0B disconnected. |
| 0 | 1 | Reserved |
| 1 | 0 | Clear OC0B on Compare Match when up-counting. Set OC0B on Compare Match when down-counting. |
| 1 | 1 | Set OC0B on Compare Match when up-counting. Clear OC0B on Compare Match when down-counting. |

Note: 1. A special case occurs when OCR0B equals TOP and COM0B1 is set. In this case, the Compare Match is ignored, but the set or clear is done at TOP. See "Phase Correct PWM Mode" on page 101 for more details.

- **Bits 3, 2 – Reserved**

These bits are reserved bits in the Atmel® ATmega48PA/88PA/168PA and will always read as zero.

- **Bits 1:0 – WGM01:0: Waveform Generation Mode**

Combined with the WGM02 bit found in the TCCR0B Register, these bits control the counting sequence of the counter, the source for maximum (TOP) counter value, and what type of waveform generation to be used, see Table 15-8. Modes of operation supported by the Timer/Counter unit are: Normal mode (counter), Clear Timer on Compare Match (CTC) mode, and two types of Pulse Width Modulation (PWM) modes (see "Modes of Operation" on page 98).

**Table 15-8.** Waveform Generation Mode Bit Description

| Mode | WGM02 | WGM01 | WGM00 | Timer/Counter Mode of Operation | TOP | Update of OCRx at | TOV Flag Set on[1][2] |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | Normal | 0xFF | Immediate | MAX |
| 1 | 0 | 0 | 1 | PWM, Phase Correct | 0xFF | TOP | BOTTOM |
| 2 | 0 | 1 | 0 | CTC | OCRA | Immediate | MAX |
| 3 | 0 | 1 | 1 | Fast PWM | 0xFF | BOTTOM | MAX |
| 4 | 1 | 0 | 0 | Reserved | – | – | – |
| 5 | 1 | 0 | 1 | PWM, Phase Correct | OCRA | TOP | BOTTOM |
| 6 | 1 | 1 | 0 | Reserved | – | – | – |
| 7 | 1 | 1 | 1 | Fast PWM | OCRA | BOTTOM | TOP |

Notes: 1. MAX = 0xFF

2. BOTTOM = 0x00

### 15.9.2 TCCR0B – Timer/Counter Control Register B

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| 0x25 (0x45) | FOC0A | FOC0B | – | – | WGM02 | CS02 | CS01 | CS00 | TCCR0B |
| Read/Write | W | W | R | R | R/W | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

- **Bit 7 – FOC0A: Force Output Compare A**

The FOC0A bit is only active when the WGM bits specify a non-PWM mode.

However, for ensuring compatibility with future devices, this bit must be set to zero when TCCR0B is written when operating in PWM mode. When writing a logical one to the FOC0A bit, an immediate Compare Match is forced on the Waveform Generation unit. The OC0A output is changed according to its COM0A1:0 bits setting. Note that the FOC0A bit is implemented as a strobe. Therefore it is the value present in the COM0A1:0 bits that determines the effect of the forced compare.

A FOC0A strobe will not generate any interrupt, nor will it clear the timer in CTC mode using OCR0A as TOP.

The FOC0A bit is always read as zero.

- **Bit 6 – FOC0B: Force Output Compare B**

The FOC0B bit is only active when the WGM bits specify a non-PWM mode.

However, for ensuring compatibility with future devices, this bit must be set to zero when TCCR0B is written when operating in PWM mode. When writing a logical one to the FOC0B bit, an immediate Compare Match is forced on the Waveform Generation unit. The OC0B output is changed according to its COM0B1:0 bits setting. Note that the FOC0B bit is implemented as a strobe. Therefore it is the value present in the COM0B1:0 bits that determines the effect of the forced compare.

A FOC0B strobe will not generate any interrupt, nor will it clear the timer in CTC mode using OCR0B as TOP.

The FOC0B bit is always read as zero.

- **Bits 5:4 – Reserved**

These bits are reserved bits in the Atmel® ATmega48PA/88PA/168PA and will always read as zero.

- **Bit 3 – WGM02: Waveform Generation Mode**

See the description in the "TCCR0A – Timer/Counter Control Register A" on page 104.

- **Bits 2:0 – CS02:0: Clock Select**

The three Clock Select bits select the clock source to be used by the Timer/Counter.

**Table 15-9.** Clock Select Bit Description

| CS02 | CS01 | CS00 | Description |
|---|---|---|---|
| 0 | 0 | 0 | No clock source (Timer/Counter stopped) |
| 0 | 0 | 1 | $clk_{I/O}$/(No prescaling) |
| 0 | 1 | 0 | $clk_{I/O}$/8 (From prescaler) |
| 0 | 1 | 1 | $clk_{I/O}$/64 (From prescaler) |
| 1 | 0 | 0 | $clk_{I/O}$/256 (From prescaler) |
| 1 | 0 | 1 | $clk_{I/O}$/1024 (From prescaler) |
| 1 | 1 | 0 | External clock source on T0 pin. Clock on falling edge. |
| 1 | 1 | 1 | External clock source on T0 pin. Clock on rising edge. |

If external pin modes are used for the Timer/Counter0, transitions on the T0 pin will clock the counter even if the pin is configured as an output. This feature allows software control of the counting.

### 15.9.3 TCNT0 – Timer/Counter Register

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| 0x26 (0x46) | TCNT0[7:0] | | | | | | | | TCNT0 |
| Read/Write | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

The Timer/Counter Register gives direct access, both for read and write operations, to the Timer/Counter unit 8-bit counter. Writing to the TCNT0 Register blocks (removes) the Compare Match on the following timer clock. Modifying the counter (TCNT0) while the counter is running, introduces a risk of missing a Compare Match between TCNT0 and the OCR0x Registers.

### 15.9.4 OCR0A – Output Compare Register A

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| 0x27 (0x47) | OCR0A[7:0] | | | | | | | | OCR0A |
| Read/Write | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

The Output Compare Register A contains an 8-bit value that is continuously compared with the counter value (TCNT0). A match can be used to generate an Output Compare interrupt, or to generate a waveform output on the OC0A pin.

### 15.9.5 OCR0B – Output Compare Register B

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| 0x28 (0x48) | OCR0B[7:0] | | | | | | | | OCR0B |
| Read/Write | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

The Output Compare Register B contains an 8-bit value that is continuously compared with the counter value (TCNT0). A match can be used to generate an Output Compare interrupt, or to generate a waveform output on the OC0B pin.

### 15.9.6 TIMSK0 – Timer/Counter Interrupt Mask Register

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| (0x6E) | – | – | – | – | – | OCIE0B | OCIE0A | TOIE0 | TIMSK0 |
| Read/Write | R | R | R | R | R | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

- **Bits 7:3 – Reserved**

These bits are reserved bits in the Atmel® ATmega48PA/88PA/168PA and will always read as zero.

- **Bit 2 – OCIE0B: Timer/Counter Output Compare Match B Interrupt Enable**

When the OCIE0B bit is written to one, and the I-bit in the Status Register is set, the Timer/Counter Compare Match B interrupt is enabled. The corresponding interrupt is executed if a Compare Match in Timer/Counter occurs, i.e., when the OCF0B bit is set in the Timer/Counter Interrupt Flag Register – TIFR0.

- **Bit 1 – OCIE0A: Timer/Counter0 Output Compare Match A Interrupt Enable**

When the OCIE0A bit is written to one, and the I-bit in the Status Register is set, the Timer/Counter0 Compare Match A interrupt is enabled. The corresponding interrupt is executed if a Compare Match in Timer/Counter0 occurs, i.e., when the OCF0A bit is set in the Timer/Counter 0 Interrupt Flag Register – TIFR0.

- **Bit 0 – TOIE0: Timer/Counter0 Overflow Interrupt Enable**

When the TOIE0 bit is written to one, and the I-bit in the Status Register is set, the Timer/Counter0 Overflow interrupt is enabled. The corresponding interrupt is executed if an overflow in Timer/Counter0 occurs, i.e., when the TOV0 bit is set in the Timer/Counter 0 Interrupt Flag Register – TIFR0.

### 15.9.7 TIFR0 – Timer/Counter 0 Interrupt Flag Register

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| 0x15 (0x35) | – | – | – | – | – | OCF0B | OCF0A | TOV0 | TIFR0 |
| Read/Write | R | R | R | R | R | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

- **Bits 7:3 – Reserved**

These bits are reserved bits in the Atmel® ATmega48PA/88PA/168PA and will always read as zero.

- **Bit 2 – OCF0B: Timer/Counter 0 Output Compare B Match Flag**

The OCF0B bit is set when a Compare Match occurs between the Timer/Counter and the data in OCR0B – Output Compare Register0 B. OCF0B is cleared by hardware when executing the corresponding interrupt handling vector. Alternatively, OCF0B is cleared by writing a logic one to the flag. When the I-bit in SREG, OCIE0B (Timer/Counter Compare B Match Interrupt Enable), and OCF0B are set, the Timer/Counter Compare Match Interrupt is executed.

- **Bit 1 – OCF0A: Timer/Counter 0 Output Compare A Match Flag**

The OCF0A bit is set when a Compare Match occurs between the Timer/Counter0 and the data in OCR0A – Output Compare Register0. OCF0A is cleared by hardware when executing the corresponding interrupt handling vector. Alternatively, OCF0A is cleared by writing a logic one to the flag. When the I-bit in SREG, OCIE0A (Timer/Counter0 Compare Match Interrupt Enable), and OCF0A are set, the Timer/Counter0 Compare Match Interrupt is executed.

- **Bit 0 – TOV0: Timer/Counter0 Overflow Flag**

The bit TOV0 is set when an overflow occurs in Timer/Counter0. TOV0 is cleared by hardware when executing the corresponding interrupt handling vector. Alternatively, TOV0 is cleared by writing a logic one to the flag. When the SREG I-bit, TOIE0 (Timer/Counter0 Overflow Interrupt Enable), and TOV0 are set, the Timer/Counter0 Overflow interrupt is executed.

The setting of this flag is dependent of the WGM02:0 bit setting. Refer to Table 15-8, "Waveform Generation Mode Bit Description" on page 106.

# 16. 16-bit Timer/Counter1 with PWM

## 16.1 Features

- **True 16-bit Design (i.e., Allows 16-bit PWM)**
- **Two independent Output Compare Units**
- **Double Buffered Output Compare Registers**
- **One Input Capture Unit**
- **Input Capture Noise Canceler**
- **Clear Timer on Compare Match (Auto Reload)**
- **Glitch-free, Phase Correct Pulse Width Modulator (PWM)**
- **Variable PWM Period**
- **Frequency Generator**
- **External Event Counter**
- **Four independent interrupt Sources (TOV1, OCF1A, OCF1B, and ICF1)**

## 16.2 Overview

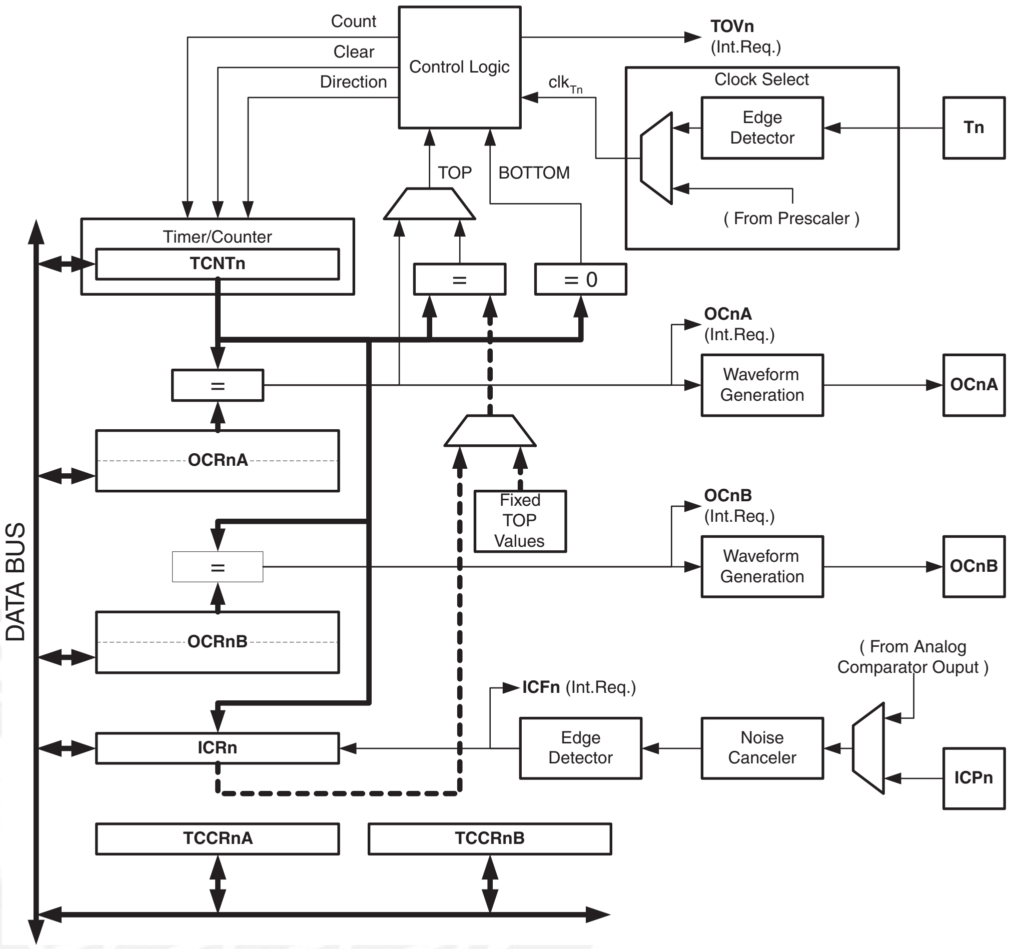
The 16-bit Timer/Counter unit allows accurate program execution timing (event management), wave generation, and signal timing measurement.

Most register and bit references in this section are written in general form. A lower case “n” replaces the Timer/Counter number, and a lower case “x” replaces the Output Compare unit channel. However, when using the register or bit defines in a program, the precise form must be used, i.e., TCNT1 for accessing Timer/Counter1 counter value and so on.

A simplified block diagram of the 16-bit Timer/Counter is shown in Figure 16-1. For the actual placement of I/O pins, refer to “Pinout Atmel® ATmega48PA/88PA/168PA” on page 2. CPU accessible I/O Registers, including I/O bits and I/O pins, are shown in bold. The device-specific I/O Register and bit locations are listed in the “Register Description” on page 133.

The PRTIM1 bit in “PRR – Power Reduction Register” on page 45 must be written to zero to enable Timer/Counter1 module.

**Figure 16-1.** 16-bit Timer/Counter Block Diagram[1]



Note: 1. Refer to Figure 1-1 on page 2, Table 14-3 on page 81 and Table 14-9 on page 88 for Timer/Counter1 pin placement and description.

### 16.2.1 Registers

The *Timer/Counter* (TCNT1), *Output Compare Registers* (OCR1A/B), and *Input Capture Register* (ICR1) are all 16-bit registers. Special procedures must be followed when accessing the 16-bit registers. These procedures are described in the section "Accessing 16-bit Registers" on page 113. The *Timer/Counter Control Registers* (TCCR1A/B) are 8-bit registers and have no CPU access restrictions. Interrupt requests (abbreviated to Int.Req. in the figure) signals are all visible in the *Timer Interrupt Flag Register* (TIFR1). All interrupts are individually masked with the *Timer Interrupt Mask Register* (TIMSK1). TIFR1 and TIMSK1 are not shown in the figure.

The Timer/Counter can be clocked internally, via the prescaler, or by an external clock source on the T1 pin. The Clock Select logic block controls which clock source and edge the Timer/Counter uses to increment (or decrement) its value. The Timer/Counter is inactive when no clock source is selected. The output from the Clock Select logic is referred to as the timer clock (clk$_{T1}$).

The double buffered Output Compare Registers (OCR1A/B) are compared with the Timer/Counter value at all time. The result of the compare can be used by the Waveform Generator to generate a PWM or variable frequency output on the Output Compare pin (OC1A/B). See "Output Compare Units" on page 120. The compare match event will also set the Compare Match Flag (OCF1A/B) which can be used to generate an Output Compare interrupt request.

The Input Capture Register can capture the Timer/Counter value at a given external (edge triggered) event on either the Input Capture pin (ICP1) or on the Analog Comparator pins (See "Analog Comparator" on page 244) The Input Capture unit includes a digital filtering unit (Noise Canceler) for reducing the chance of capturing noise spikes.

The TOP value, or maximum Timer/Counter value, can in some modes of operation be defined by either the OCR1A Register, the ICR1 Register, or by a set of fixed values. When using OCR1A as TOP value in a PWM mode, the OCR1A Register can not be used for generating a PWM output. However, the TOP value will in this case be double buffered allowing the TOP value to be changed in run time. If a fixed TOP value is required, the ICR1 Register can be used as an alternative, freeing the OCR1A to be used as PWM output.

### 16.2.2 Definitions

The following definitions are used extensively throughout the section:

| | |
|---|---|
| **BOTTOM** | The counter reaches the *BOTTOM* when it becomes 0x0000. |
| **MAX** | The counter reaches its *MAX*imum when it becomes 0xFFFF (decimal 65535). |
| **TOP** | The counter reaches the *TOP* when it becomes equal to the highest value in the count sequence. The TOP value can be assigned to be one of the fixed values: 0x00FF, 0x01FF, or 0x03FF, or to the value stored in the OCR1A or ICR1 Register. The assignment is dependent of the mode of operation. |

## 16.3 Accessing 16-bit Registers

The TCNT1, OCR1A/B, and ICR1 are 16-bit registers that can be accessed by the AVR CPU via the 8-bit data bus. The 16-bit register must be byte accessed using two read or write operations. Each 16-bit timer has a single 8-bit register for temporary storing of the high byte of the 16-bit access. The same temporary register is shared between all 16-bit registers within each 16-bit timer. Accessing the low byte triggers the 16-bit read or write operation. When the low byte of a 16-bit register is written by the CPU, the high byte stored in the temporary register, and the low byte written are both copied into the 16-bit register in the same clock cycle. When the low byte of a 16-bit register is read by the CPU, the high byte of the 16-bit register is copied into the temporary register in the same clock cycle as the low byte is read.

Not all 16-bit accesses uses the temporary register for the high byte. Reading the OCR1A/B 16-bit registers does not involve using the temporary register.

To do a 16-bit write, the high byte must be written before the low byte. For a 16-bit read, the low byte must be read before the high byte.

The following code examples show how to access the 16-bit Timer Registers assuming that no interrupts updates the temporary register. The same principle can be used directly for accessing the OCR1A/B and ICR1 Registers. Note that when using "C", the compiler handles the 16-bit access.

Assembly Code Examples[1]

```
...
; Set TCNT1 to 0x01FF
ldi r17,0x01
ldi r16,0xFF
out TCNT1H,r17
out TCNT1L,r16
; Read TCNT1 into r17:r16
in  r16,TCNT1L
in  r17,TCNT1H
...
```

C Code Examples[1]

```
unsigned int i;
...
/* Set TCNT1 to 0x01FF */
TCNT1 = 0x1FF;
/* Read TCNT1 into i */
i = TCNT1;
...
```

Note: 1. See ”About Code Examples” on page 7.
For I/O Registers located in extended I/O map, “IN”, “OUT”, “SBIS”, “SBIC”, “CBI”, and “SBI” instructions must be replaced with instructions that allow access to extended I/O. Typically “LDS” and “STS” combined with “SBRS”, “SBRC”, “SBR”, and “CBR”.

It is important to notice that accessing 16-bit registers are atomic operations. If an interrupt occurs between the two instructions accessing the 16-bit register, and the interrupt code updates the temporary register by accessing the same or any other of the 16-bit Timer Registers, then the result of the access outside the interrupt will be corrupted. Therefore, when both the main code and the interrupt code update the temporary register, the main code must disable the interrupts during the 16-bit access.

The following code examples show how to do an atomic read of the TCNT1 Register contents. Reading any of the OCR1A/B or ICR1 Registers can be done by using the same principle.

Assembly Code Example(1)

```
TIM16_ReadTCNT1:
  ; Save global interrupt flag
  in  r18,SREG
  ; Disable interrupts
  cli
  ; Read TCNT1 into r17:r16
  in  r16,TCNT1L
  in  r17,TCNT1H
  ; Restore global interrupt flag
  out SREG,r18
  ret
```

C Code Example(1)

```
unsigned int TIM16_ReadTCNT1( void )
{
  unsigned char sreg;
  unsigned int i;
  /* Save global interrupt flag */
  sreg = SREG;
  /* Disable interrupts */
  _CLI();
  /* Read TCNT1 into i */
  i = TCNT1;
  /* Restore global interrupt flag */
  SREG = sreg;
  return i;
}
```

Note: 1. See ”About Code Examples” on page 7.
For I/O Registers located in extended I/O map, “IN”, “OUT”, “SBIS”, “SBIC”, “CBI”, and “SBI” instructions must be replaced with instructions that allow access to extended I/O. Typically “LDS” and “STS” combined with “SBRS”, “SBRC”, “SBR”, and “CBR”.

The following code examples show how to do an atomic write of the TCNT1 Register contents. Writing any of the OCR1A/B or ICR1 Registers can be done by using the same principle.

Assembly Code Example[1]

```
TIM16_WriteTCNT1:
  ; Save global interrupt flag
  in  r18,SREG
  ; Disable interrupts
  cli
  ; Set TCNT1 to r17:r16
  out TCNT1H,r17
  out TCNT1L,r16
  ; Restore global interrupt flag
  out SREG,r18
  ret
```

C Code Example[1]

```
void TIM16_WriteTCNT1( unsigned int i )
{
  unsigned char sreg;
  unsigned int i;
  /* Save global interrupt flag */
  sreg = SREG;
  /* Disable interrupts */
  _CLI();
  /* Set TCNT1 to i */
  TCNT1 = i;
  /* Restore global interrupt flag */
  SREG = sreg;
}
```

Note: 1. See "About Code Examples" on page 7.
For I/O Registers located in extended I/O map, "IN", "OUT", "SBIS", "SBIC", "CBI", and "SBI" instructions must be replaced with instructions that allow access to extended I/O. Typically "LDS" and "STS" combined with "SBRS", "SBRC", "SBR", and "CBR".

The assembly code example requires that the r17:r16 register pair contains the value to be written to TCNT1.

### 16.3.1 Reusing the Temporary High Byte Register

If writing to more than one 16-bit register where the high byte is the same for all registers written, then the high byte only needs to be written once. However, note that the same rule of atomic operation described previously also applies in this case.

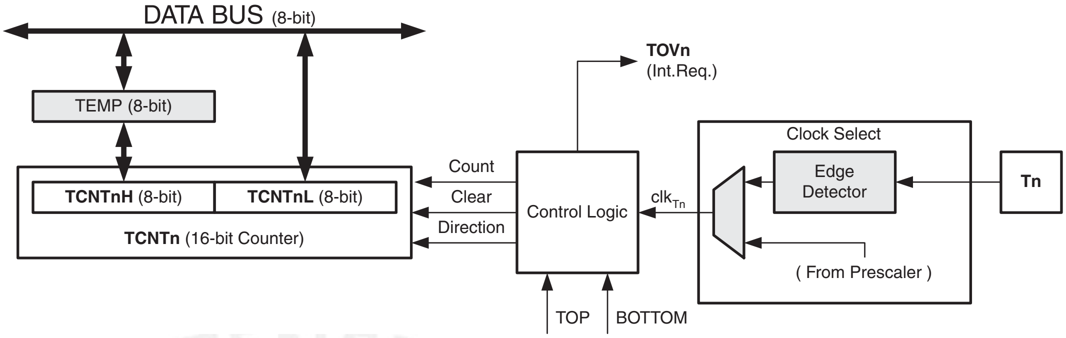
## 16.4 Timer/Counter Clock Sources

The Timer/Counter can be clocked by an internal or an external clock source. The clock source is selected by the Clock Select logic which is controlled by the *Clock Select* (CS12:0) bits located in the *Timer/Counter control Register B* (TCCR1B). For details on clock sources and prescaler, see "Timer/Counter0 and Timer/Counter1 Prescalers" on page 140.

## 16.5 Counter Unit

The main part of the 16-bit Timer/Counter is the programmable 16-bit bi-directional counter unit. Figure 16-2 shows a block diagram of the counter and its surroundings.

**Figure 16-2.** Counter Unit Block Diagram



Signal description (internal signals):

**Count** Increment or decrement TCNT1 by 1.

**Direction** Select between increment and decrement.

**Clear** Clear TCNT1 (set all bits to zero).

**$clk_{T1}$** Timer/Counter clock.

**TOP** Signalize that TCNT1 has reached maximum value.

**BOTTOM** Signalize that TCNT1 has reached minimum value (zero).

The 16-bit counter is mapped into two 8-bit I/O memory locations: *Counter High* (TCNT1H) containing the upper eight bits of the counter, and *Counter Low* (TCNT1L) containing the lower eight bits. The TCNT1H Register can only be indirectly accessed by the CPU. When the CPU does an access to the TCNT1H I/O location, the CPU accesses the high byte temporary register (TEMP). The temporary register is updated with the TCNT1H value when the TCNT1L is read, and TCNT1H is updated with the temporary register value when TCNT1L is written. This allows the CPU to read or write the entire 16-bit counter value within one clock cycle via the 8-bit data bus. It is important to notice that there are special cases of writing to the TCNT1 Register when the counter is counting that will give unpredictable results. The special cases are described in the sections where they are of importance.

Depending on the mode of operation used, the counter is cleared, incremented, or decremented at each *timer clock* ($clk_{T1}$). The $clk_{T1}$ can be generated from an external or internal clock source, selected by the *Clock Select* bits (CS12:0). When no clock source is selected (CS12:0 = 0) the timer is stopped. However, the TCNT1 value can be accessed by the CPU, independent of whether $clk_{T1}$ is present or not. A CPU write overrides (has priority over) all counter clear or count operations.

The counting sequence is determined by the setting of the *Waveform Generation mode* bits (WGM13:0) located in the *Timer/Counter Control Registers* A and B (TCCR1A and TCCR1B). There are close connections between how the counter behaves (counts) and how waveforms are generated on the Output Compare outputs OC1x. For more details about advanced counting sequences and waveform generation, see "Modes of Operation" on page 123.

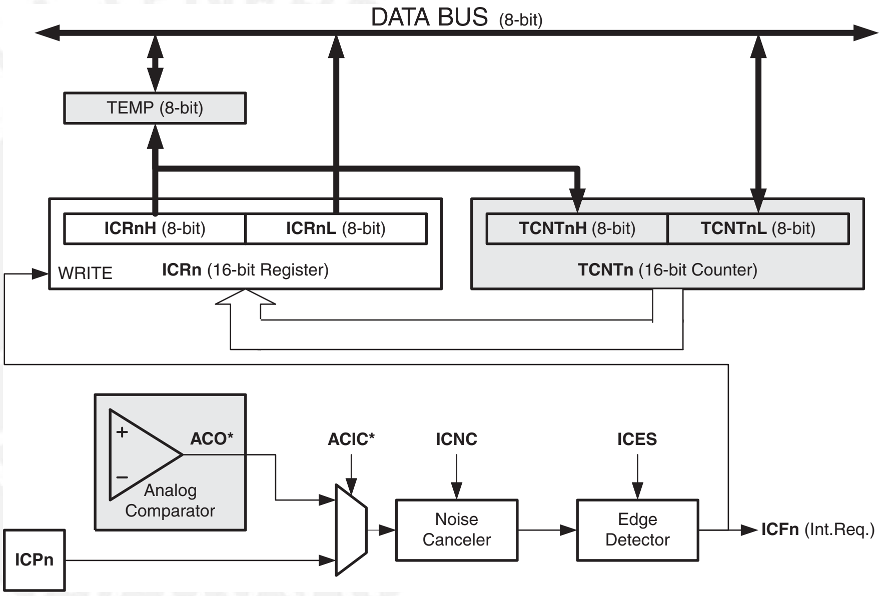
The Timer/Counter Overflow Flag (TOV1) is set according to the mode of operation selected by the WGM13:0 bits. TOV1 can be used for generating a CPU interrupt.

## 16.6 Input Capture Unit

The Timer/Counter incorporates an Input Capture unit that can capture external events and give them a time-stamp indicating time of occurrence. The external signal indicating an event, or multiple events, can be applied via the ICP1 pin or alternatively, via the analog-comparator unit. The time-stamps can then be used to calculate frequency, duty-cycle, and other features of the signal applied. Alternatively the time-stamps can be used for creating a log of the events.

The Input Capture unit is illustrated by the block diagram shown in Figure 16-3. The elements of the block diagram that are not directly a part of the Input Capture unit are gray shaded. The small "n" in register and bit names indicates the Timer/Counter number.

**Figure 16-3.** Input Capture Unit Block Diagram



When a change of the logic level (an event) occurs on the *Input Capture pin* (ICP1), alternatively on the *Analog Comparator output* (ACO), and this change confirms to the setting of the edge detector, a capture will be triggered. When a capture is triggered, the 16-bit value of the counter (TCNT1) is written to the *Input Capture Register* (ICR1). The *Input Capture Flag* (ICF1) is set at the same system clock as the TCNT1 value is copied into ICR1 Register. If enabled (ICIE1 = 1), the Input Capture Flag generates an Input Capture interrupt. The ICF1 Flag is automatically cleared when the interrupt is executed. Alternatively the ICF1 Flag can be cleared by software by writing a logical one to its I/O bit location.

Reading the 16-bit value in the *Input Capture Register* (ICR1) is done by first reading the low byte (ICR1L) and then the high byte (ICR1H). When the low byte is read the high byte is copied into the high byte temporary register (TEMP). When the CPU reads the ICR1H I/O location it will access the TEMP Register.

The ICR1 Register can only be written when using a Waveform Generation mode that utilizes the ICR1 Register for defining the counter's TOP value. In these cases the *Waveform Generation mode* (WGM13:0) bits must be set before the TOP value can be written to the ICR1 Register. When writing the ICR1 Register the high byte must be written to the ICR1H I/O location before the low byte is written to ICR1L.

For more information on how to access the 16-bit registers refer to "Accessing 16-bit Registers" on page 113.

### 16.6.1 Input Capture Trigger Source

The main trigger source for the Input Capture unit is the *Input Capture pin* (ICP1). Timer/Counter1 can alternatively use the Analog Comparator output as trigger source for the Input Capture unit. The Analog Comparator is selected as trigger source by setting the *Analog Comparator Input Capture* (ACIC) bit in the *Analog Comparator Control and Status Register* (ACSR). Be aware that changing trigger source can trigger a capture. The Input Capture Flag must therefore be cleared after the change.

Both the *Input Capture pin* (ICP1) and the *Analog Comparator output* (ACO) inputs are sampled using the same technique as for the T1 pin (Figure 17-1 on page 140). The edge detector is also identical. However, when the noise canceler is enabled, additional logic is inserted before the edge detector, which increases the delay by four system clock cycles. Note that the input of the noise canceler and edge detector is always enabled unless the Timer/Counter is set in a Waveform Generation mode that uses ICR1 to define TOP.

An Input Capture can be triggered by software by controlling the port of the ICP1 pin.

### 16.6.2 Noise Canceler

The noise canceler improves noise immunity by using a simple digital filtering scheme. The noise canceler input is monitored over four samples, and all four must be equal for changing the output that in turn is used by the edge detector.

The noise canceler is enabled by setting the *Input Capture Noise Canceler* (ICNC1) bit in *Timer/Counter Control Register B* (TCCR1B). When enabled the noise canceler introduces additional four system clock cycles of delay from a change applied to the input, to the update of the ICR1 Register. The noise canceler uses the system clock and is therefore not affected by the prescaler.

### 16.6.3 Using the Input Capture Unit

The main challenge when using the Input Capture unit is to assign enough processor capacity for handling the incoming events. The time between two events is critical. If the processor has not read the captured value in the ICR1 Register before the next event occurs, the ICR1 will be overwritten with a new value. In this case the result of the capture will be incorrect.

When using the Input Capture interrupt, the ICR1 Register should be read as early in the interrupt handler routine as possible. Even though the Input Capture interrupt has relatively high priority, the maximum interrupt response time is dependent on the maximum number of clock cycles it takes to handle any of the other interrupt requests.

Using the Input Capture unit in any mode of operation when the TOP value (resolution) is actively changed during operation, is not recommended.

Measurement of an external signal's duty cycle requires that the trigger edge is changed after each capture. Changing the edge sensing must be done as early as possible after the ICR1 Register has been read. After a change of the edge, the Input Capture Flag (ICF1) must be cleared by software (writing a logical one to the I/O bit location). For measuring frequency only, the clearing of the ICF1 Flag is not required (if an interrupt handler is used).

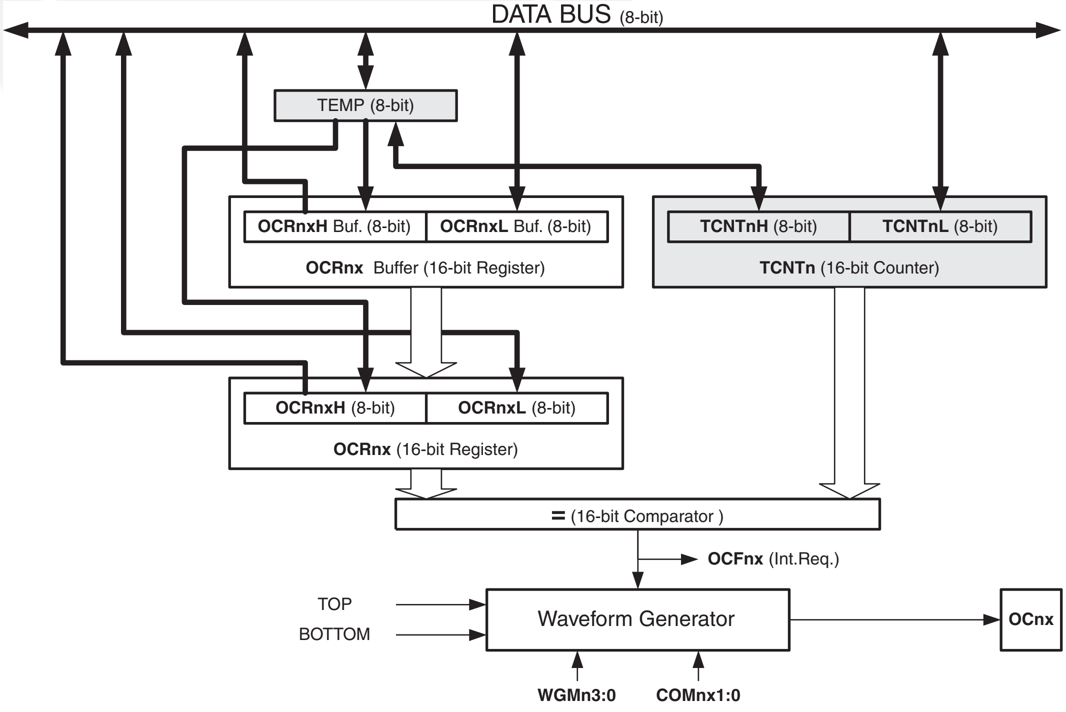
## 16.7 Output Compare Units

The 16-bit comparator continuously compares TCNT1 with the *Output Compare Register* (OCR1x). If TCNT equals OCR1x the comparator signals a match. A match will set the *Output Compare Flag* (OCF1x) at the next timer clock cycle. If enabled (OCIE1x = 1), the Output Compare Flag generates an Output Compare interrupt. The OCF1x Flag is automatically cleared when the interrupt is executed. Alternatively the OCF1x Flag can be cleared by software by writing a logical one to its I/O bit location. The Waveform Generator uses the match signal to generate an output according to operating mode set by the *Waveform Generation mode* (WGM13:0) bits and *Compare Output mode* (COM1x1:0) bits. The TOP and BOTTOM signals are used by the Waveform Generator for handling the special cases of the extreme values in some modes of operation (See "Modes of Operation" on page 123.)

A special feature of Output Compare unit A allows it to define the Timer/Counter TOP value (i.e., counter resolution). In addition to the counter resolution, the TOP value defines the period time for waveforms generated by the Waveform Generator.

Figure 16-4 shows a block diagram of the Output Compare unit. The small "n" in the register and bit names indicates the device number (n = 1 for Timer/Counter 1), and the "x" indicates Output Compare unit (A/B). The elements of the block diagram that are not directly a part of the Output Compare unit are gray shaded.

**Figure 16-4.** Output Compare Unit, Block Diagram



The OCR1x Register is double buffered when using any of the twelve *Pulse Width Modulation* (PWM) modes. For the Normal and *Clear Timer on Compare* (CTC) modes of operation, the double buffering is disabled. The double buffering synchronizes the update of the OCR1x Compare Register to either TOP or BOTTOM of the counting sequence. The synchronization prevents the occurrence of odd-length, non-symmetrical PWM pulses, thereby making the output glitch-free.

The OCR1x Register access may seem complex, but this is not case. When the double buffering is enabled, the CPU has access to the OCR1x Buffer Register, and if double buffering is disabled the CPU will access the OCR1x directly. The content of the OCR1x (Buffer or Compare) Register is only changed by a write operation (the Timer/Counter does not update this register automatically as the TCNT1 and ICR1 Register). Therefore OCR1x is not read via the high byte temporary register (TEMP). However, it is a good practice to read the low byte first as when accessing other 16-bit registers. Writing the OCR1x Registers must be done via the TEMP Register since the compare of all 16 bits is done continuously. The high byte (OCR1xH) has to be written first. When the high byte I/O location is written by the CPU, the TEMP Register will be updated by the value written. Then when the low byte (OCR1xL) is written to the lower eight bits, the high byte will be copied into the upper 8-bits of either the OCR1x buffer or OCR1x Compare Register in the same system clock cycle.

For more information of how to access the 16-bit registers refer to "Accessing 16-bit Registers" on page 113.

### 16.7.1 Force Output Compare

In non-PWM Waveform Generation modes, the match output of the comparator can be forced by writing a one to the *Force Output Compare* (FOC1x) bit. Forcing compare match will not set the OCF1x Flag or reload/clear the timer, but the OC1x pin will be updated as if a real compare match had occurred (the COM11:0 bits settings define whether the OC1x pin is set, cleared or toggled).

### 16.7.2 Compare Match Blocking by TCNT1 Write

All CPU writes to the TCNT1 Register will block any compare match that occurs in the next timer clock cycle, even when the timer is stopped. This feature allows OCR1x to be initialized to the same value as TCNT1 without triggering an interrupt when the Timer/Counter clock is enabled.

### 16.7.3 Using the Output Compare Unit

Since writing TCNT1 in any mode of operation will block all compare matches for one timer clock cycle, there are risks involved when changing TCNT1 when using any of the Output Compare channels, independent of whether the Timer/Counter is running or not. If the value written to TCNT1 equals the OCR1x value, the compare match will be missed, resulting in incorrect waveform generation. Do not write the TCNT1 equal to TOP in PWM modes with variable TOP values. The compare match for the TOP will be ignored and the counter will continue to 0xFFFF. Similarly, do not write the TCNT1 value equal to BOTTOM when the counter is downcounting.

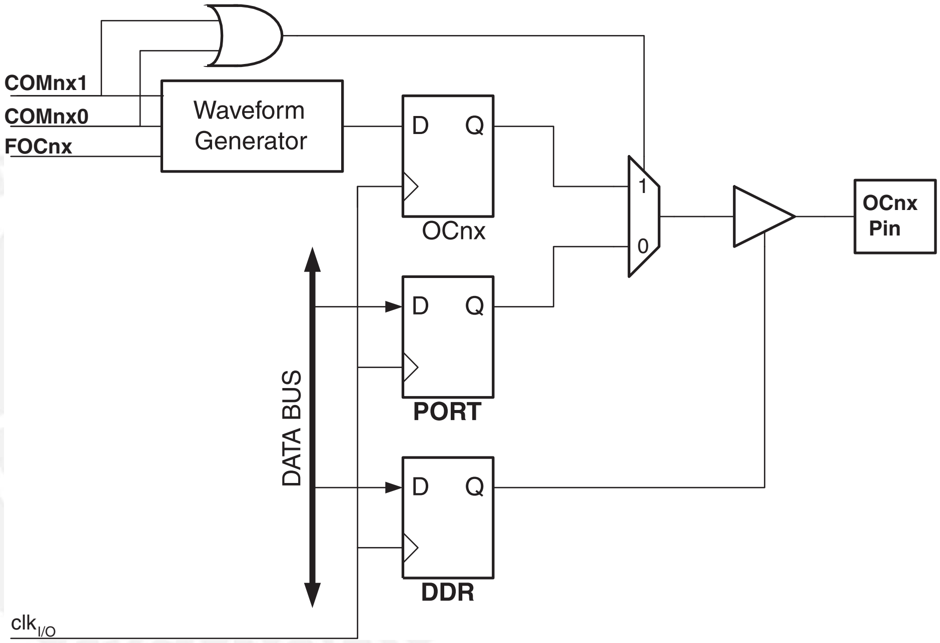
The setup of the OC1x should be performed before setting the Data Direction Register for the port pin to output. The easiest way of setting the OC1x value is to use the Force Output Compare (FOC1x) strobe bits in Normal mode. The OC1x Register keeps its value even when changing between Waveform Generation modes.

Be aware that the COM1x1:0 bits are not double buffered together with the compare value. Changing the COM1x1:0 bits will take effect immediately.

## 16.8 Compare Match Output Unit

The *Compare Output mode* (COM1x1:0) bits have two functions. The Waveform Generator uses the COM1x1:0 bits for defining the Output Compare (OC1x) state at the next compare match. Secondly the COM1x1:0 bits control the OC1x pin output source. Figure 16-5 shows a simplified schematic of the logic affected by the COM1x1:0 bit setting. The I/O Registers, I/O bits, and I/O pins in the figure are shown in bold. Only the parts of the general I/O Port Control Registers (DDR and PORT) that are affected by the COM1x1:0 bits are shown. When referring to the OC1x state, the reference is for the internal OC1x Register, not the OC1x pin. If a system reset occur, the OC1x Register is reset to “0”.

**Figure 16-5.** Compare Match Output Unit, Schematic



The general I/O port function is overridden by the Output Compare (OC1x) from the Waveform Generator if either of the COM1x1:0 bits are set. However, the OC1x pin direction (input or output) is still controlled by the *Data Direction Register* (DDR) for the port pin. The Data Direction Register bit for the OC1x pin (DDR_OC1x) must be set as output before the OC1x value is visible on the pin. The port override function is generally independent of the Waveform Generation mode, but there are some exceptions. Refer to Table 16-1, Table 16-2 and Table 16-3 for details.

The design of the Output Compare pin logic allows initialization of the OC1x state before the output is enabled. Note that some COM1x1:0 bit settings are reserved for certain modes of operation. See “Register Description” on page 133.

The COM1x1:0 bits have no effect on the Input Capture unit.

### 16.8.1 Compare Output Mode and Waveform Generation

The Waveform Generator uses the COM1x1:0 bits differently in normal, CTC, and PWM modes. For all modes, setting the COM1x1:0 = 0 tells the Waveform Generator that no action on the OC1x Register is to be performed on the next compare match. For compare output actions in the non-PWM modes refer to Table 16-1 on page 133. For fast PWM mode refer to Table 16-2 on page 133, and for phase correct and phase and frequency correct PWM refer to Table 16-3 on page 134.

A change of the COM1x1:0 bits state will have effect at the first compare match after the bits are written. For non-PWM modes, the action can be forced to have immediate effect by using the FOC1x strobe bits.

## 16.9 Modes of Operation

The mode of operation, i.e., the behavior of the Timer/Counter and the Output Compare pins, is defined by the combination of the *Waveform Generation mode* (WGM13:0) and *Compare Output mode* (COM1x1:0) bits. The Compare Output mode bits do not affect the counting sequence, while the Waveform Generation mode bits do. The COM1x1:0 bits control whether the PWM output generated should be inverted or not (inverted or non-inverted PWM). For non-PWM modes the COM1x1:0 bits control whether the output should be set, cleared or toggle at a compare match (See “Compare Match Output Unit” on page 122.)

For detailed timing information refer to “Timer/Counter Timing Diagrams” on page 131.

### 16.9.1 Normal Mode

The simplest mode of operation is the *Normal mode* (WGM13:0 = 0). In this mode the counting direction is always up (incrementing), and no counter clear is performed. The counter simply overruns when it passes its maximum 16-bit value (MAX = 0xFFFF) and then restarts from the BOTTOM (0x0000). In normal operation the *Timer/Counter Overflow Flag* (TOV1) will be set in the same timer clock cycle as the TCNT1 becomes zero. The TOV1 Flag in this case behaves like a 17th bit, except that it is only set, not cleared. However, combined with the timer overflow interrupt that automatically clears the TOV1 Flag, the timer resolution can be increased by software. There are no special cases to consider in the Normal mode, a new counter value can be written anytime.

The Input Capture unit is easy to use in Normal mode. However, observe that the maximum interval between the external events must not exceed the resolution of the counter. If the interval between events are too long, the timer overflow interrupt or the prescaler must be used to extend the resolution for the capture unit.

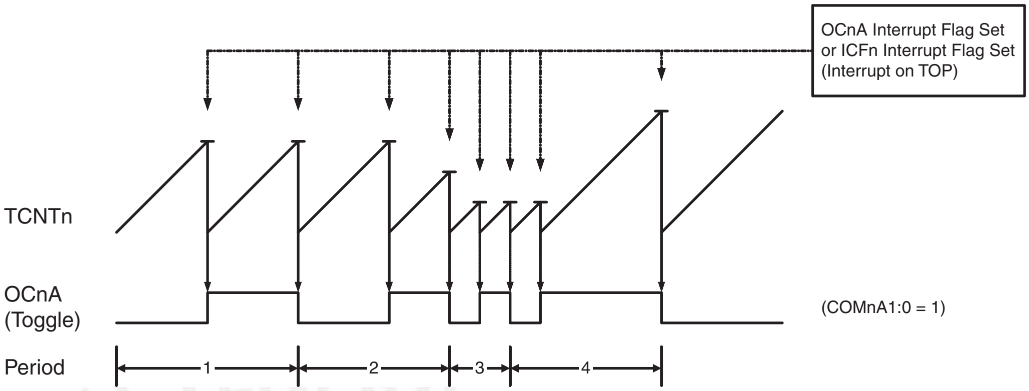
The Output Compare units can be used to generate interrupts at some given time. Using the Output Compare to generate waveforms in Normal mode is not recommended, since this will occupy too much of the CPU time.

### 16.9.2 Clear Timer on Compare Match (CTC) Mode

In *Clear Timer on Compare* or CTC mode (WGM13:0 = 4 or 12), the OCR1A or ICR1 Register are used to manipulate the counter resolution. In CTC mode the counter is cleared to zero when the counter value (TCNT1) matches either the OCR1A (WGM13:0 = 4) or the ICR1 (WGM13:0 = 12). The OCR1A or ICR1 define the top value for the counter, hence also its resolution. This mode allows greater control of the compare match output frequency. It also simplifies the operation of counting external events.

The timing diagram for the CTC mode is shown in Figure 16-6. The counter value (TCNT1) increases until a compare match occurs with either OCR1A or ICR1, and then counter (TCNT1) is cleared.

**Figure 16-6.** CTC Mode, Timing Diagram



An interrupt can be generated at each time the counter value reaches the TOP value by either using the OCF1A or ICF1 Flag according to the register used to define the TOP value. If the interrupt is enabled, the interrupt handler routine can be used for updating the TOP value. However, changing the TOP to a value close to BOTTOM when the counter is running with none or a low prescaler value must be done with care since the CTC mode does not have the double buffering feature. If the new value written to OCR1A or ICR1 is lower than the current value of TCNT1, the counter will miss the compare match. The counter will then have to count to its maximum value (0xFFFF) and wrap around starting at 0x0000 before the compare match can occur. In many cases this feature is not desirable. An alternative will then be to use the fast PWM mode using OCR1A for defining TOP (WGM13:0 = 15) since the OCR1A then will be double buffered.

For generating a waveform output in CTC mode, the OC1A output can be set to toggle its logical level on each compare match by setting the Compare Output mode bits to toggle mode (COM1A1:0 = 1). The OC1A value will not be visible on the port pin unless the data direction for the pin is set to output (DDR_OC1A = 1). The waveform generated will have a maximum frequency of $f_{OC1A} = f_{clk\_I/O}/2$ when OCR1A is set to zero (0x0000). The waveform frequency is defined by the following equation:

$$f_{OCnA} = \frac{f_{clk\_I/O}}{2 \times N \times (1 + OCRnA)}$$

The *N* variable represents the prescaler factor (1, 8, 64, 256, or 1024).

As for the Normal mode of operation, the TOV1 Flag is set in the same timer clock cycle that the counter counts from MAX to 0x0000.

### 16.9.3 Fast PWM Mode

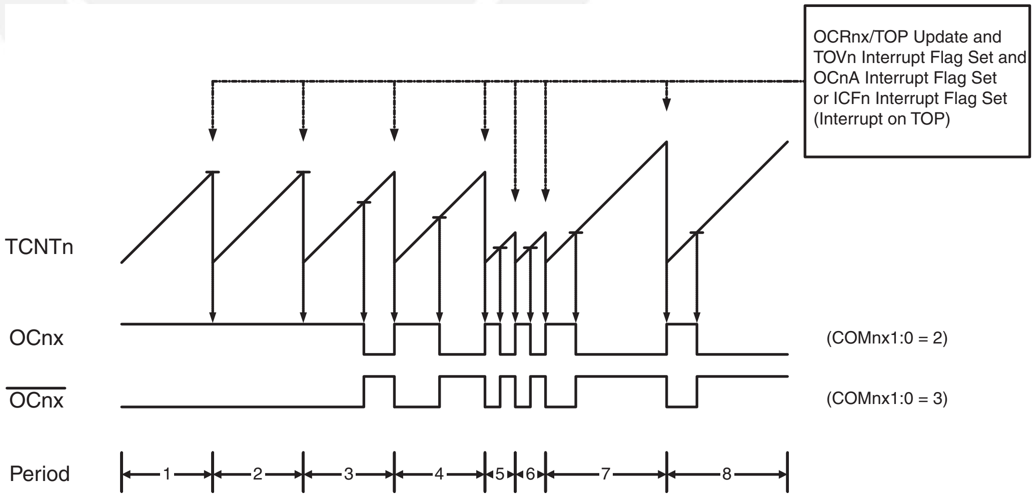
The *fast Pulse Width Modulation* or fast PWM mode (WGM13:0 = 5, 6, 7, 14, or 15) provides a high frequency PWM waveform generation option. The fast PWM differs from the other PWM options by its single-slope operation. The counter counts from BOTTOM to TOP then restarts from BOTTOM. In non-inverting Compare Output mode, the Output Compare (OC1x) is cleared on the compare match between TCNT1 and OCR1x, and set at BOTTOM. In inverting Compare Output mode output is set on compare match and cleared at BOTTOM. Due to the single-slope operation, the operating frequency of the fast PWM mode can be twice as high as the phase correct and phase and frequency correct PWM modes that use dual-slope operation. This high frequency makes the fast PWM mode well suited for power regulation, rectification, and DAC applications. High frequency allows physically small sized external components (coils, capacitors), hence reduces total system cost.

The PWM resolution for fast PWM can be fixed to 8-, 9-, or 10-bit, or defined by either ICR1 or OCR1A. The minimum resolution allowed is 2-bit (ICR1 or OCR1A set to 0x0003), and the maximum resolution is 16-bit (ICR1 or OCR1A set to MAX). The PWM resolution in bits can be calculated by using the following equation:

$$R_{FPWM} = \frac{\log(TOP + 1)}{\log(2)}$$

In fast PWM mode the counter is incremented until the counter value matches either one of the fixed values 0x00FF, 0x01FF, or 0x03FF (WGM13:0 = 5, 6, or 7), the value in ICR1 (WGM13:0 = 14), or the value in OCR1A (WGM13:0 = 15). The counter is then cleared at the following timer clock cycle. The timing diagram for the fast PWM mode is shown in Figure 16-7. The figure shows fast PWM mode when OCR1A or ICR1 is used to define TOP. The TCNT1 value is in the timing diagram shown as a histogram for illustrating the single-slope operation. The diagram includes non-inverted and inverted PWM outputs. The small horizontal line marks on the TCNT1 slopes represent compare matches between OCR1x and TCNT1. The OC1x Interrupt Flag will be set when a compare match occurs.

**Figure 16-7.** Fast PWM Mode, Timing Diagram



The Timer/Counter Overflow Flag (TOV1) is set each time the counter reaches TOP. In addition the OC1A or ICF1 Flag is set at the same timer clock cycle as TOV1 is set when either OCR1A or ICR1 is used for defining the TOP value. If one of the interrupts are enabled, the interrupt handler routine can be used for updating the TOP and compare values.

When changing the TOP value the program must ensure that the new TOP value is higher or equal to the value of all of the Compare Registers. If the TOP value is lower than any of the Compare Registers, a compare match will never occur between the TCNT1 and the OCR1x. Note that when using fixed TOP values the unused bits are masked to zero when any of the OCR1x Registers are written.

The procedure for updating ICR1 differs from updating OCR1A when used for defining the TOP value. The ICR1 Register is not double buffered. This means that if ICR1 is changed to a low value when the counter is running with none or a low prescaler value, there is a risk that the new ICR1 value written is lower than the current value of TCNT1. The result will then be that the counter will miss the compare match at the TOP value. The counter will then have to count to the MAX value (0xFFFF) and wrap around starting at 0x0000 before the compare match can occur. The OCR1A Register however, is double buffered. This feature allows the OCR1A I/O location to be written anytime. When the OCR1A I/O location is written the value written will be put into the OCR1A Buffer Register. The OCR1A Compare Register will then be updated with the value in the Buffer Register at the next timer clock cycle the TCNT1 matches TOP. The update is done at the same timer clock cycle as the TCNT1 is cleared and the TOV1 Flag is set.

Using the ICR1 Register for defining TOP works well when using fixed TOP values. By using ICR1, the OCR1A Register is free to be used for generating a PWM output on OC1A. However, if the base PWM frequency is actively changed (by changing the TOP value), using the OCR1A as TOP is clearly a better choice due to its double buffer feature.

In fast PWM mode, the compare units allow generation of PWM waveforms on the OC1x pins. Setting the COM1x1:0 bits to two will produce a inverted PWM and an non-inverted PWM output can be generated by setting the COM1x1:0 to three (see Table  on page 133). The actual OC1x value will only be visible on the port pin if the data direction for the port pin is set as output (DDR_OC1x). The PWM waveform is generated by setting (or clearing) the OC1x Register at the compare match between OCR1x and TCNT1, and clearing (or setting) the OC1x Register at the timer clock cycle the counter is cleared (changes from TOP to BOTTOM).

The PWM frequency for the output can be calculated by the following equation:

$$f_{OCnxPWM} = \frac{f_{clk\_I/O}}{N \times (1 + TOP)}$$

The N variable represents the prescaler divider (1, 8, 64, 256, or 1024).

The extreme values for the OCR1x Register represents special cases when generating a PWM waveform output in the fast PWM mode. If the OCR1x is set equal to BOTTOM (0x0000) the output will be a narrow spike for each TOP+1 timer clock cycle. Setting the OCR1x equal to TOP will result in a constant high or low output (depending on the polarity of the output set by the COM1x1:0 bits.)

A frequency (with 50% duty cycle) waveform output in fast PWM mode can be achieved by setting OC1A to toggle its logical level on each compare match (COM1A1:0 = 1). This applies only if OCR1A is used to define the TOP value (WGM13:0 = 15). The waveform generated will have a maximum frequency of $f_{OC1A} = f_{clk\_I/O}/2$ when OCR1A is set to zero (0x0000). This feature is similar to the OC1A toggle in CTC mode, except the double buffer feature of the Output Compare unit is enabled in the fast PWM mode.

### 16.9.4 Phase Correct PWM Mode

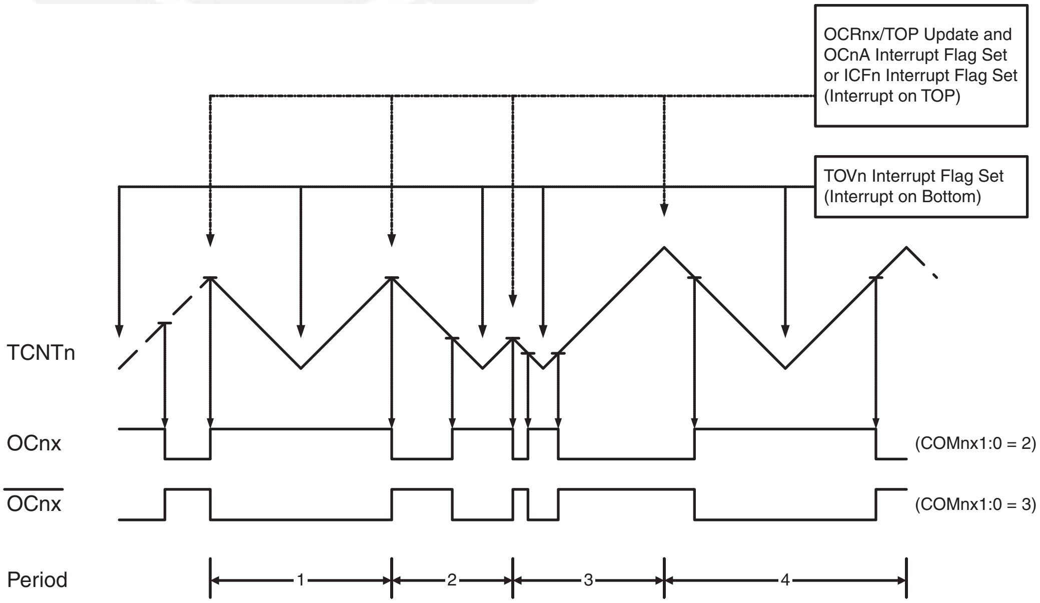
The *phase correct Pulse Width Modulation* or phase correct PWM mode (WGM13:0 = 1, 2, 3, 10, or 11) provides a high resolution phase correct PWM waveform generation option. The phase correct PWM mode is, like the phase and frequency correct PWM mode, based on a dual-slope operation. The counter counts repeatedly from BOTTOM (0x0000) to TOP and then from TOP to BOTTOM. In non-inverting Compare Output mode, the Output Compare (OC1x) is cleared on the compare match between TCNT1 and OCR1x while upcounting, and set on the compare match while downcounting. In inverting Output Compare mode, the operation is inverted. The dual-slope operation has lower maximum operation frequency than single slope operation. However, due to the symmetric feature of the dual-slope PWM modes, these modes are preferred for motor control applications.

The PWM resolution for the phase correct PWM mode can be fixed to 8-, 9-, or 10-bit, or defined by either ICR1 or OCR1A. The minimum resolution allowed is 2-bit (ICR1 or OCR1A set to 0x0003), and the maximum resolution is 16-bit (ICR1 or OCR1A set to MAX). The PWM resolution in bits can be calculated by using the following equation:

$$R_{PCPWM} = \frac{\log(TOP + 1)}{\log(2)}$$

In phase correct PWM mode the counter is incremented until the counter value matches either one of the fixed values 0x00FF, 0x01FF, or 0x03FF (WGM13:0 = 1, 2, or 3), the value in ICR1 (WGM13:0 = 10), or the value in OCR1A (WGM13:0 = 11). The counter has then reached the TOP and changes the count direction. The TCNT1 value will be equal to TOP for one timer clock cycle. The timing diagram for the phase correct PWM mode is shown on Figure 16-8. The figure shows phase correct PWM mode when OCR1A or ICR1 is used to define TOP. The TCNT1 value is in the timing diagram shown as a histogram for illustrating the dual-slope operation. The diagram includes non-inverted and inverted PWM outputs. The small horizontal line marks on the TCNT1 slopes represent compare matches between OCR1x and TCNT1. The OC1x Interrupt Flag will be set when a compare match occurs.

**Figure 16-8.** Phase Correct PWM Mode, Timing Diagram



The Timer/Counter Overflow Flag (TOV1) is set each time the counter reaches BOTTOM. When either OCR1A or ICR1 is used for defining the TOP value, the OC1A or ICF1 Flag is set accordingly at the same timer clock cycle as the OCR1x Registers are updated with the double buffer value (at TOP). The Interrupt Flags can be used to generate an interrupt each time the counter reaches the TOP or BOTTOM value.

When changing the TOP value the program must ensure that the new TOP value is higher or equal to the value of all of the Compare Registers. If the TOP value is lower than any of the Compare Registers, a compare match will never occur between the TCNT1 and the OCR1x. Note that when using fixed TOP values, the unused bits are masked to zero when any of the OCR1x Registers are written. As the third period shown in Figure 16-8 illustrates, changing the TOP actively while the Timer/Counter is running in the phase correct mode can result in an unsymmetrical output. The reason for this can be found in the time of update of the OCR1x Register. Since the OCR1x update occurs at TOP, the PWM period starts and ends at TOP. This implies that the length of the falling slope is determined by the previous TOP value, while the length of the rising slope is determined by the new TOP value. When these two values differ the two slopes of the period will differ in length. The difference in length gives the unsymmetrical result on the output.

It is recommended to use the phase and frequency correct mode instead of the phase correct mode when changing the TOP value while the Timer/Counter is running. When using a static TOP value there are practically no differences between the two modes of operation.

In phase correct PWM mode, the compare units allow generation of PWM waveforms on the OC1x pins. Setting the COM1x1:0 bits to two will produce a non-inverted PWM and an inverted PWM output can be generated by setting the COM1x1:0 to three (See Table  on page 134). The actual OC1x value will only be visible on the port pin if the data direction for the port pin is set as output (DDR_OC1x). The PWM waveform is generated by setting (or clearing) the OC1x Register at the compare match between OCR1x and TCNT1 when the counter increments, and clearing (or setting) the OC1x Register at compare match between OCR1x and TCNT1 when the counter decrements. The PWM frequency for the output when using phase correct PWM can be calculated by the following equation:

$$f_{OCnxPCPWM} = \frac{f_{clk\_I/O}}{2 \times N \times TOP}$$

The N variable represents the prescaler divider (1, 8, 64, 256, or 1024).

The extreme values for the OCR1x Register represent special cases when generating a PWM waveform output in the phase correct PWM mode. If the OCR1x is set equal to BOTTOM the output will be continuously low and if set equal to TOP the output will be continuously high for non-inverted PWM mode. For inverted PWM the output will have the opposite logic values. If OCR1A is used to define the TOP value (WGM13:0 = 11) and COM1A1:0 = 1, the OC1A output will toggle with a 50% duty cycle.

### 16.9.5 Phase and Frequency Correct PWM Mode

The *phase and frequency correct Pulse Width Modulation,* or phase and frequency correct PWM mode (WGM13:0 = 8 or 9) provides a high resolution phase and frequency correct PWM waveform generation option. The phase and frequency correct PWM mode is, like the phase correct PWM mode, based on a dual-slope operation. The counter counts repeatedly from BOTTOM (0x0000) to TOP and then from TOP to BOTTOM. In non-inverting Compare Output mode, the Output Compare (OC1x) is cleared on the compare match between TCNT1 and OCR1x while upcounting, and set on the compare match while downcounting. In inverting Compare Output mode, the operation is inverted. The dual-slope operation gives a lower maximum operation frequency compared to the single-slope operation. However, due to the symmetric feature of the dual-slope PWM modes, these modes are preferred for motor control applications.

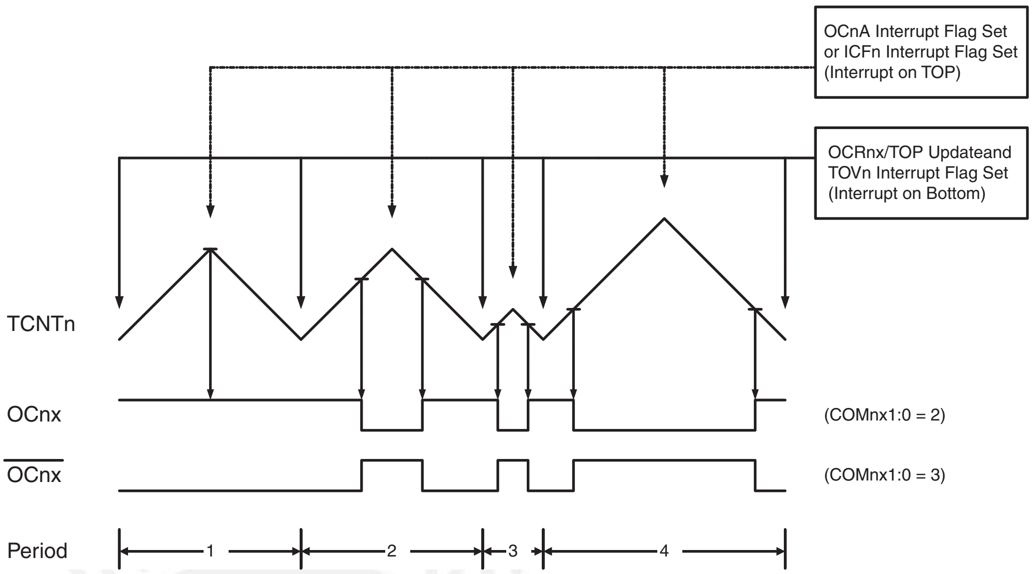
The main difference between the phase correct, and the phase and frequency correct PWM mode is the time the OCR1x Register is updated by the OCR1x Buffer Register, (see Figure 16-8 and Figure 16-9).

The PWM resolution for the phase and frequency correct PWM mode can be defined by either ICR1 or OCR1A. The minimum resolution allowed is 2-bit (ICR1 or OCR1A set to 0x0003), and the maximum resolution is 16-bit (ICR1 or OCR1A set to MAX). The PWM resolution in bits can be calculated using the following equation:

$$R_{PFCPWM} = \frac{\log(TOP + 1)}{\log(2)}$$

In phase and frequency correct PWM mode the counter is incremented until the counter value matches either the value in ICR1 (WGM13:0 = 8), or the value in OCR1A (WGM13:0 = 9). The counter has then reached the TOP and changes the count direction. The TCNT1 value will be equal to TOP for one timer clock cycle. The timing diagram for the phase correct and frequency correct PWM mode is shown on Figure 16-9. The figure shows phase and frequency correct PWM mode when OCR1A or ICR1 is used to define TOP. The TCNT1 value is in the timing diagram shown as a histogram for illustrating the dual-slope operation. The diagram includes non-inverted and inverted PWM outputs. The small horizontal line marks on the TCNT1 slopes represent compare matches between OCR1x and TCNT1. The OC1x Interrupt Flag will be set when a compare match occurs.

**Figure 16-9.** Phase and Frequency Correct PWM Mode, Timing Diagram



The Timer/Counter Overflow Flag (TOV1) is set at the same timer clock cycle as the OCR1x Registers are updated with the double buffer value (at BOTTOM). When either OCR1A or ICR1 is used for defining the TOP value, the OC1A or ICF1 Flag set when TCNT1 has reached TOP. The Interrupt Flags can then be used to generate an interrupt each time the counter reaches the TOP or BOTTOM value.

When changing the TOP value the program must ensure that the new TOP value is higher or equal to the value of all of the Compare Registers. If the TOP value is lower than any of the Compare Registers, a compare match will never occur between the TCNT1 and the OCR1x.

As Figure 16-9 shows the output generated is, in contrast to the phase correct mode, symmetrical in all periods. Since the OCR1x Registers are updated at BOTTOM, the length of the rising and the falling slopes will always be equal. This gives symmetrical output pulses and is therefore frequency correct.

Using the ICR1 Register for defining TOP works well when using fixed TOP values. By using ICR1, the OCR1A Register is free to be used for generating a PWM output on OC1A. However, if the base PWM frequency is actively changed by changing the TOP value, using the OCR1A as TOP is clearly a better choice due to its double buffer feature.

In phase and frequency correct PWM mode, the compare units allow generation of PWM waveforms on the OC1x pins. Setting the COM1x1:0 bits to two will produce a non-inverted PWM and an inverted PWM output can be generated by setting the COM1x1:0 to three (See Table  on page 134). The actual OC1x value will only be visible on the port pin if the data direction for the port pin is set as output (DDR_OC1x). The PWM waveform is generated by setting (or clearing) the OC1x Register at the compare match between OCR1x and TCNT1 when the counter increments, and clearing (or setting) the OC1x Register at compare match between OCR1x and TCNT1 when the counter decrements. The PWM frequency for the output when using phase and frequency correct PWM can be calculated by the following equation:

$$f_{OCnxPFCPWM} = \frac{f_{clk\_I/O}}{2 \times N \times TOP}$$

The N variable represents the prescaler divider (1, 8, 64, 256, or 1024).

The extreme values for the OCR1x Register represents special cases when generating a PWM waveform output in the phase correct PWM mode. If the OCR1x is set equal to BOTTOM the output will be continuously low and if set equal to TOP the output will be set to high for non-inverted PWM mode. For inverted PWM the output will have the opposite logic values. If OCR1A is used to define the TOP value (WGM13:0 = 9) and COM1A1:0 = 1, the OC1A output will toggle with a 50% duty cycle.

## 16.10 Timer/Counter Timing Diagrams

The Timer/Counter is a synchronous design and the timer clock ($clk_{T1}$) is therefore shown as a clock enable signal in the following figures. The figures include information on when Interrupt Flags are set, and when the OCR1x Register is updated with the OCR1x buffer value (only for modes utilizing double buffering). Figure 16-10 shows a timing diagram for the setting of OCF1x.

**Figure 16-10.** Timer/Counter Timing Diagram, Setting of OCF1x, no Prescaling

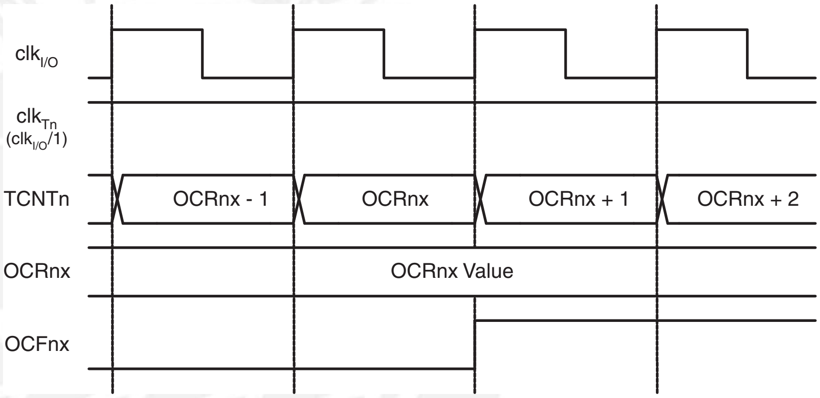


Figure 16-11 shows the same timing data, but with the prescaler enabled.

**Figure 16-11.** Timer/Counter Timing Diagram, Setting of OCF1x, with Prescaler ($f_{clk\_I/O}/8$)

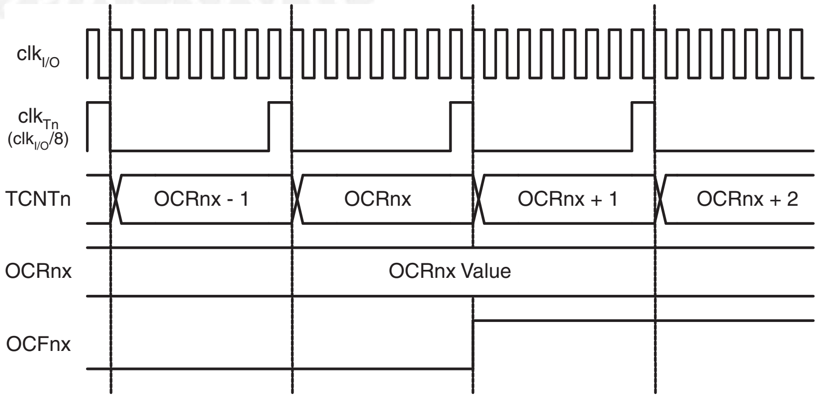


Figure 16-12 shows the count sequence close to TOP in various modes. When using phase and frequency correct PWM mode the OCR1x Register is updated at BOTTOM. The timing diagrams will be the same, but TOP should be replaced by BOTTOM, TOP-1 by BOTTOM+1 and so on. The same renaming applies for modes that set the TOV1 Flag at BOTTOM.

**Figure 16-12.** Timer/Counter Timing Diagram, no Prescaling

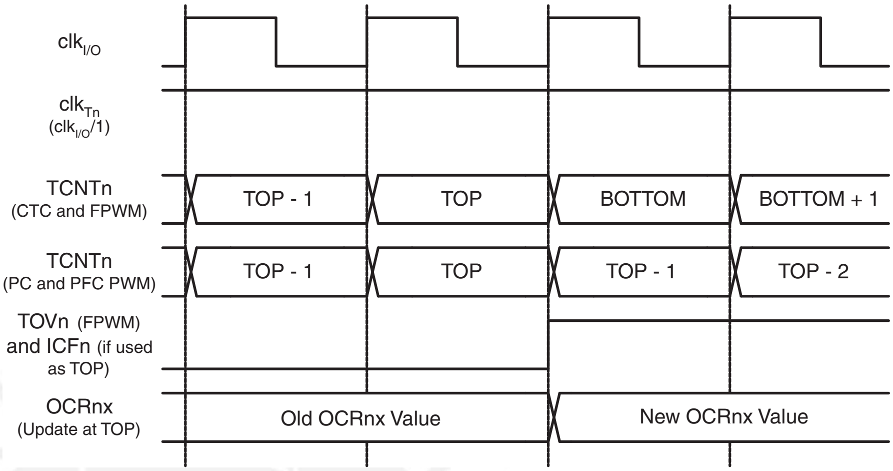
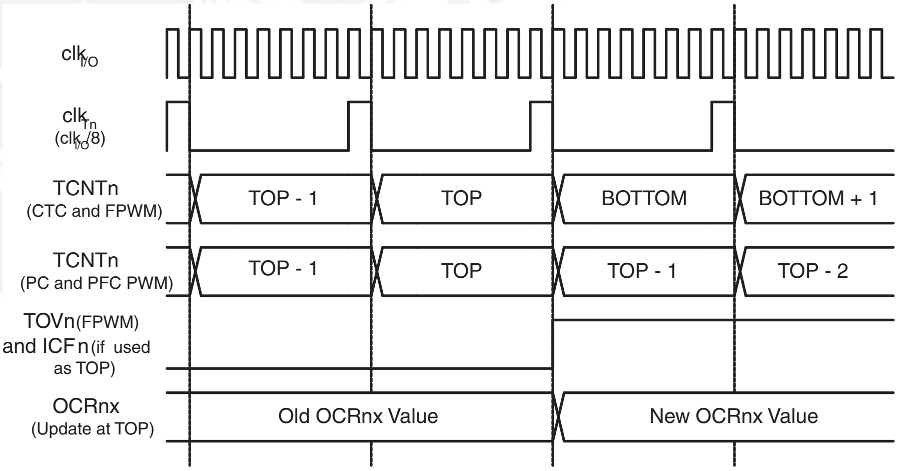


Figure 16-13 shows the same timing data, but with the prescaler enabled.

**Figure 16-13.** Timer/Counter Timing Diagram, with Prescaler ($f_{clk\_I/O}/8$)



## 16.11 Register Description

### 16.11.1 TCCR1A – Timer/Counter1 Control Register A

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| (0x80) | COM1A1 | COM1A0 | COM1B1 | COM1B0 | – | – | WGM11 | WGM10 | TCCR1A |
| Read/Write | R/W | R/W | R/W | R/W | R | R | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

- **Bit 7:6 – COM1A1:0: Compare Output Mode for Channel A**

- **Bit 5:4 – COM1B1:0: Compare Output Mode for Channel B**

The COM1A1:0 and COM1B1:0 control the Output Compare pins (OC1A and OC1B respectively) behavior. If one or both of the COM1A1:0 bits are written to one, the OC1A output overrides the normal port functionality of the I/O pin it is connected to. If one or both of the COM1B1:0 bit are written to one, the OC1B output overrides the normal port functionality of the I/O pin it is connected to. However, note that the *Data Direction Register* (DDR) bit corresponding to the OC1A or OC1B pin must be set in order to enable the output driver.

When the OC1A or OC1B is connected to the pin, the function of the COM1x1:0 bits is dependent of the WGM13:0 bits setting. Table 16-1 shows the COM1x1:0 bit functionality when the WGM13:0 bits are set to a Normal or a CTC mode (non-PWM).

**Table 16-1.** Compare Output Mode, non-PWM

| COM1A1/COM1B1 | COM1A0/COM1B0 | Description |
|---|---|---|
| 0 | 0 | Normal port operation, OC1A/OC1B disconnected. |
| 0 | 1 | Toggle OC1A/OC1B on Compare Match. |
| 1 | 0 | Clear OC1A/OC1B on Compare Match (Set output to low level). |
| 1 | 1 | Set OC1A/OC1B on Compare Match (Set output to high level). |

Table 16-2 shows the COM1x1:0 bit functionality when the WGM13:0 bits are set to the fast PWM mode.

**Table 16-2.** Compare Output Mode, Fast PWM[1]

| COM1A1/COM1B1 | COM1A0/COM1B0 | Description |
|---|---|---|
| 0 | 0 | Normal port operation, OC1A/OC1B disconnected. |
| 0 | 1 | WGM13:0 = 14 or 15: Toggle OC1A on Compare Match, OC1B disconnected (normal port operation). For all other WGM1 settings, normal port operation, OC1A/OC1B disconnected. |
| 1 | 0 | Clear OC1A/OC1B on Compare Match, set OC1A/OC1B at BOTTOM (non-inverting mode) |
| 1 | 1 | Set OC1A/OC1B on Compare Match, clear OC1A/OC1B at BOTTOM (inverting mode) |

Note: 1. A special case occurs when OCR1A/OCR1B equals TOP and COM1A1/COM1B1 is set. In this case the compare match is ignored, but the set or clear is done at BOTTOM. See "Fast PWM Mode" on page 125. for more details.

Table 16-3 shows the COM1x1:0 bit functionality when the WGM13:0 bits are set to the phase correct or the phase and frequency correct, PWM mode.

**Table 16-3.** Compare Output Mode, Phase Correct and Phase and Frequency Correct PWM(1)

| COM1A1/COM1B1 | COM1A0/COM1B0 | Description |
|---|---|---|
| 0 | 0 | Normal port operation, OC1A/OC1B disconnected. |
| 0 | 1 | WGM13:0 = 9 or 11: Toggle OC1A on Compare Match, OC1B disconnected (normal port operation). For all other WGM1 settings, normal port operation, OC1A/OC1B disconnected. |
| 1 | 0 | Clear OC1A/OC1B on Compare Match when up-counting. Set OC1A/OC1B on Compare Match when downcounting. |
| 1 | 1 | Set OC1A/OC1B on Compare Match when up-counting. Clear OC1A/OC1B on Compare Match when downcounting. |

Note: 1. A special case occurs when OCR1A/OCR1B equals TOP and COM1A1/COM1B1 is set. See "Phase Correct PWM Mode" on page 127. for more details.

- **Bit 1:0 – WGM11:0: Waveform Generation Mode**

Combined with the WGM13:2 bits found in the TCCR1B Register, these bits control the counting sequence of the counter, the source for maximum (TOP) counter value, and what type of waveform generation to be used, see Table 16-4. Modes of operation supported by the Timer/Counter unit are: Normal mode (counter), Clear Timer on Compare match (CTC) mode, and three types of Pulse Width Modulation (PWM) modes. (See "Modes of Operation" on page 123.).

**Table 16-4.** Waveform Generation Mode Bit Description[1]

| Mode | WGM13 | WGM12 (CTC1) | WGM11 (PWM11) | WGM10 (PWM10) | Timer/Counter Mode of Operation | TOP | Update of OCR1x at | TOV1 Flag Set on |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | Normal | 0xFFFF | Immediate | MAX |
| 1 | 0 | 0 | 0 | 1 | PWM, Phase Correct, 8-bit | 0x00FF | TOP | BOTTOM |
| 2 | 0 | 0 | 1 | 0 | PWM, Phase Correct, 9-bit | 0x01FF | TOP | BOTTOM |
| 3 | 0 | 0 | 1 | 1 | PWM, Phase Correct, 10-bit | 0x03FF | TOP | BOTTOM |
| 4 | 0 | 1 | 0 | 0 | CTC | OCR1A | Immediate | MAX |
| 5 | 0 | 1 | 0 | 1 | Fast PWM, 8-bit | 0x00FF | BOTTOM | TOP |
| 6 | 0 | 1 | 1 | 0 | Fast PWM, 9-bit | 0x01FF | BOTTOM | TOP |
| 7 | 0 | 1 | 1 | 1 | Fast PWM, 10-bit | 0x03FF | BOTTOM | TOP |
| 8 | 1 | 0 | 0 | 0 | PWM, Phase and Frequency Correct | ICR1 | BOTTOM | BOTTOM |
| 9 | 1 | 0 | 0 | 1 | PWM, Phase and Frequency Correct | OCR1A | BOTTOM | BOTTOM |
| 10 | 1 | 0 | 1 | 0 | PWM, Phase Correct | ICR1 | TOP | BOTTOM |
| 11 | 1 | 0 | 1 | 1 | PWM, Phase Correct | OCR1A | TOP | BOTTOM |
| 12 | 1 | 1 | 0 | 0 | CTC | ICR1 | Immediate | MAX |
| 13 | 1 | 1 | 0 | 1 | (Reserved) | – | – | – |
| 14 | 1 | 1 | 1 | 0 | Fast PWM | ICR1 | BOTTOM | TOP |
| 15 | 1 | 1 | 1 | 1 | Fast PWM | OCR1A | BOTTOM | TOP |

Note: 1. The CTC1 and PWM11:0 bit definition names are obsolete. Use the WGM12:0 definitions. However, the functionality and location of these bits are compatible with previous versions of the timer.

### 16.11.2 TCCR1B – Timer/Counter1 Control Register B

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| (0x81) | ICNC1 | ICES1 | – | WGM13 | WGM12 | CS12 | CS11 | CS10 | TCCR1B |
| Read/Write | R/W | R/W | R | R/W | R/W | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

- **Bit 7 – ICNC1: Input Capture Noise Canceler**

Setting this bit (to one) activates the Input Capture Noise Canceler. When the noise canceler is activated, the input from the Input Capture pin (ICP1) is filtered. The filter function requires four successive equal valued samples of the ICP1 pin for changing its output. The Input Capture is therefore delayed by four Oscillator cycles when the noise canceler is enabled.

- **Bit 6 – ICES1: Input Capture Edge Select**

This bit selects which edge on the Input Capture pin (ICP1) that is used to trigger a capture event. When the ICES1 bit is written to zero, a falling (negative) edge is used as trigger, and when the ICES1 bit is written to one, a rising (positive) edge will trigger the capture.

When a capture is triggered according to the ICES1 setting, the counter value is copied into the Input Capture Register (ICR1). The event will also set the Input Capture Flag (ICF1), and this can be used to cause an Input Capture Interrupt, if this interrupt is enabled.

When the ICR1 is used as TOP value (see description of the WGM13:0 bits located in the TCCR1A and the TCCR1B Register), the ICP1 is disconnected and consequently the Input Capture function is disabled.

- **Bit 5 – Reserved**

This bit is reserved for future use. For ensuring compatibility with future devices, this bit must be written to zero when TCCR1B is written.

- **Bit 4:3 – WGM13:2: Waveform Generation Mode**

See TCCR1A Register description.

- **Bit 2:0 – CS12:0: Clock Select**

The three Clock Select bits select the clock source to be used by the Timer/Counter, see Figure 16-10 and Figure 16-11.

**Table 16-5.** Clock Select Bit Description

| CS12 | CS11 | CS10 | Description |
|---|---|---|---|
| 0 | 0 | 0 | No clock source (Timer/Counter stopped). |
| 0 | 0 | 1 | $clk_{I/O}$/1 (No prescaling) |
| 0 | 1 | 0 | $clk_{I/O}$/8 (From prescaler) |
| 0 | 1 | 1 | $clk_{I/O}$/64 (From prescaler) |
| 1 | 0 | 0 | $clk_{I/O}$/256 (From prescaler) |
| 1 | 0 | 1 | $clk_{I/O}$/1024 (From prescaler) |
| 1 | 1 | 0 | External clock source on T1 pin. Clock on falling edge. |
| 1 | 1 | 1 | External clock source on T1 pin. Clock on rising edge. |

If external pin modes are used for the Timer/Counter1, transitions on the T1 pin will clock the counter even if the pin is configured as an output. This feature allows software control of the counting.

### 16.11.3 TCCR1C – Timer/Counter1 Control Register C

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| (0x82) | FOC1A | FOC1B | – | – | – | – | – | – | TCCR1C |
| Read/Write | R/W | R/W | R | R | R | R | R | R | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

- **Bit 7 – FOC1A: Force Output Compare for Channel A**

- **Bit 6 – FOC1B: Force Output Compare for Channel B**

The FOC1A/FOC1B bits are only active when the WGM13:0 bits specifies a non-PWM mode. When writing a logical one to the FOC1A/FOC1B bit, an immediate compare match is forced on the Waveform Generation unit. The OC1A/OC1B output is changed according to its COM1x1:0 bits setting. Note that the FOC1A/FOC1B bits are implemented as strobes. Therefore it is the value present in the COM1x1:0 bits that determine the effect of the forced compare.

A FOC1A/FOC1B strobe will not generate any interrupt nor will it clear the timer in Clear Timer on Compare match (CTC) mode using OCR1A as TOP. The FOC1A/FOC1B bits are always read as zero.

### 16.11.4 TCNT1H and TCNT1L – Timer/Counter1

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| (0x85) | TCNT1[15:8] | | | | | | | | TCNT1H |
| (0x84) | TCNT1[7:0] | | | | | | | | TCNT1L |
| Read/Write | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

The two *Timer/Counter* I/O locations (TCNT1H and TCNT1L, combined TCNT1) give direct access, both for read and for write operations, to the Timer/Counter unit 16-bit counter. To ensure that both the high and low bytes are read and written simultaneously when the CPU accesses these registers, the access is performed using an 8-bit temporary High Byte Register (TEMP). This temporary register is shared by all the other 16-bit registers. See "Accessing 16-bit Registers" on page 113.

Modifying the counter (TCNT1) while the counter is running introduces a risk of missing a compare match between TCNT1 and one of the OCR1x Registers.

Writing to the TCNT1 Register blocks (removes) the compare match on the following timer clock for all compare units.

### 16.11.5 OCR1AH and OCR1AL – Output Compare Register 1 A

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| (0x89) | OCR1A[15:8] | | | | | | | | OCR1AH |
| (0x88) | OCR1A[7:0] | | | | | | | | OCR1AL |
| Read/Write | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

### 16.11.6 OCR1BH and OCR1BL – Output Compare Register 1 B

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| (0x8B) | OCR1B[15:8] | | | | | | | | OCR1BH |
| (0x8A) | OCR1B[7:0] | | | | | | | | OCR1BL |
| Read/Write | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

The Output Compare Registers contain a 16-bit value that is continuously compared with the counter value (TCNT1). A match can be used to generate an Output Compare interrupt, or to generate a waveform output on the OC1x pin.

The Output Compare Registers are 16-bit in size. To ensure that both the high and low bytes are written simultaneously when the CPU writes to these registers, the access is performed using an 8-bit temporary High Byte Register (TEMP). This temporary register is shared by all the other 16-bit registers. See "Accessing 16-bit Registers" on page 113.

### 16.11.7 ICR1H and ICR1L – Input Capture Register 1

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| (0x87) | ICR1[15:8] | | | | | | | | ICR1H |
| (0x86) | ICR1[7:0] | | | | | | | | ICR1L |
| Read/Write | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

The Input Capture is updated with the counter (TCNT1) value each time an event occurs on the ICP1 pin (or optionally on the Analog Comparator output for Timer/Counter1). The Input Capture can be used for defining the counter TOP value.

The Input Capture Register is 16-bit in size. To ensure that both the high and low bytes are read simultaneously when the CPU accesses these registers, the access is performed using an 8-bit temporary High Byte Register (TEMP). This temporary register is shared by all the other 16-bit registers. See “Accessing 16-bit Registers” on page 113.

### 16.11.8 TIMSK1 – Timer/Counter1 Interrupt Mask Register

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| (0x6F) | – | – | ICIE1 | – | – | OCIE1B | OCIE1A | TOIE1 | TIMSK1 |
| Read/Write | R | R | R/W | R | R | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

- **Bit 7, 6 – Reserved**

These bits are unused bits in the Atmel® ATmega48PA/88PA/168PA, and will always read as zero.

- **Bit 5 – ICIE1: Timer/Counter1, Input Capture Interrupt Enable**

When this bit is written to one, and the I-flag in the Status Register is set (interrupts globally enabled), the Timer/Counter1 Input Capture interrupt is enabled. The corresponding Interrupt Vector (see “Interrupts” on page 58) is executed when the ICF1 Flag, located in TIFR1, is set.

- **Bit 4, 3 – Reserved**

These bits are unused bits in the Atmel ATmega48PA/88PA/168PA, and will always read as zero.

- **Bit 2 – OCIE1B: Timer/Counter1, Output Compare B Match Interrupt Enable**

When this bit is written to one, and the I-flag in the Status Register is set (interrupts globally enabled), the Timer/Counter1 Output Compare B Match interrupt is enabled. The corresponding Interrupt Vector (see “Interrupts” on page 58) is executed when the OCF1B Flag, located in TIFR1, is set.

- **Bit 1 – OCIE1A: Timer/Counter1, Output Compare A Match Interrupt Enable**

When this bit is written to one, and the I-flag in the Status Register is set (interrupts globally enabled), the Timer/Counter1 Output Compare A Match interrupt is enabled. The corresponding Interrupt Vector (see “Interrupts” on page 58) is executed when the OCF1A Flag, located in TIFR1, is set.

- **Bit 0 – TOIE1: Timer/Counter1, Overflow Interrupt Enable**

When this bit is written to one, and the I-flag in the Status Register is set (interrupts globally enabled), the Timer/Counter1 Overflow interrupt is enabled. The corresponding Interrupt Vector (See “Interrupts” on page 58) is executed when the TOV1 Flag, located in TIFR1, is set.

### 16.11.9 TIFR1 – Timer/Counter1 Interrupt Flag Register

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| 0x16 (0x36) | – | – | ICF1 | – | – | OCF1B | OCF1A | TOV1 | TIFR1 |
| Read/Write | R | R | R/W | R | R | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

- **Bit 7, 6 – Reserved**

These bits are unused bits in the Atmel® ATmega48PA/88PA/168PA, and will always read as zero.

- **Bit 5 – ICF1: Timer/Counter1, Input Capture Flag**

This flag is set when a capture event occurs on the ICP1 pin. When the Input Capture Register (ICR1) is set by the WGM13:0 to be used as the TOP value, the ICF1 Flag is set when the counter reaches the TOP value.

ICF1 is automatically cleared when the Input Capture Interrupt Vector is executed. Alternatively, ICF1 can be cleared by writing a logic one to its bit location.

- **Bit 4, 3 – Reserved**

These bits are unused bits in the Atmel ATmega48PA/88PA/168PA, and will always read as zero.

- **Bit 2 – OCF1B: Timer/Counter1, Output Compare B Match Flag**

This flag is set in the timer clock cycle after the counter (TCNT1) value matches the Output Compare Register B (OCR1B).

Note that a Forced Output Compare (FOC1B) strobe will not set the OCF1B Flag.

OCF1B is automatically cleared when the Output Compare Match B Interrupt Vector is executed. Alternatively, OCF1B can be cleared by writing a logic one to its bit location.

- **Bit 1 – OCF1A: Timer/Counter1, Output Compare A Match Flag**

This flag is set in the timer clock cycle after the counter (TCNT1) value matches the Output Compare Register A (OCR1A).

Note that a Forced Output Compare (FOC1A) strobe will not set the OCF1A Flag.

OCF1A is automatically cleared when the Output Compare Match A Interrupt Vector is executed. Alternatively, OCF1A can be cleared by writing a logic one to its bit location.

- **Bit 0 – TOV1: Timer/Counter1, Overflow Flag**

The setting of this flag is dependent of the WGM13:0 bits setting. In Normal and CTC modes, the TOV1 Flag is set when the timer overflows. Refer to Table 16-4 on page 135 for the TOV1 Flag behavior when using another WGM13:0 bit setting.

TOV1 is automatically cleared when the Timer/Counter1 Overflow Interrupt Vector is executed. Alternatively, TOV1 can be cleared by writing a logic one to its bit location.

# 17. Timer/Counter0 and Timer/Counter1 Prescalers

"8-bit Timer/Counter0 with PWM" on page 93 and "16-bit Timer/Counter1 with PWM" on page 111 share the same prescaler module, but the Timer/Counters can have different prescaler settings. The description below applies to both Timer/Counter1 and Timer/Counter0.

## 17.1 Internal Clock Source

The Timer/Counter can be clocked directly by the system clock (by setting the CSn2:0 = 1). This provides the fastest operation, with a maximum Timer/Counter clock frequency equal to system clock frequency ($f_{CLK\_I/O}$). Alternatively, one of four taps from the prescaler can be used as a clock source. The prescaled clock has a frequency of either $f_{CLK\_I/O}/8$, $f_{CLK\_I/O}/64$, $f_{CLK\_I/O}/256$, or $f_{CLK\_I/O}/1024$.

## 17.2 Prescaler Reset

The prescaler is free running, i.e., operates independently of the Clock Select logic of the Timer/Counter, and it is shared by Timer/Counter1 and Timer/Counter0. Since the prescaler is not affected by the Timer/Counter's clock select, the state of the prescaler will have implications for situations where a prescaled clock is used. One example of prescaling artifacts occurs when the timer is enabled and clocked by the prescaler (6 > CSn2:0 > 1). The number of system clock cycles from when the timer is enabled to the first count occurs can be from 1 to N+1 system clock cycles, where N equals the prescaler divisor (8, 64, 256, or 1024).

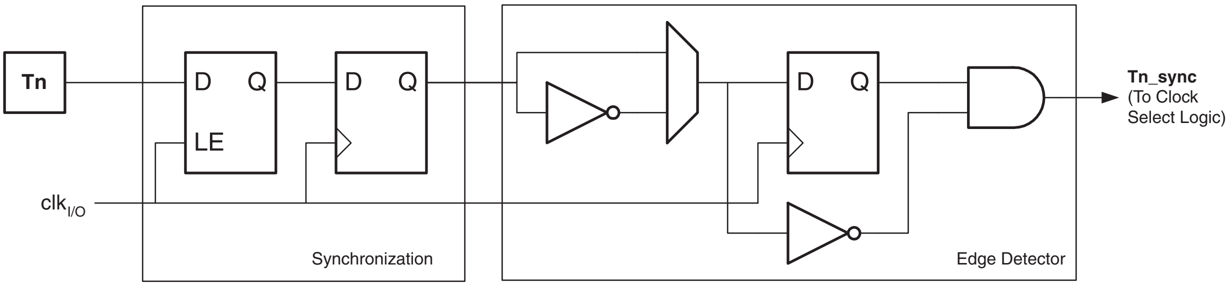
It is possible to use the prescaler reset for synchronizing the Timer/Counter to program execution. However, care must be taken if the other Timer/Counter that shares the same prescaler also uses prescaling. A prescaler reset will affect the prescaler period for all Timer/Counters it is connected to.

## 17.3 External Clock Source

An external clock source applied to the T1/T0 pin can be used as Timer/Counter clock ($clk_{T1}/clk_{T0}$). The T1/T0 pin is sampled once every system clock cycle by the pin synchronization logic. The synchronized (sampled) signal is then passed through the edge detector. Figure 17-1 shows a functional equivalent block diagram of the T1/T0 synchronization and edge detector logic. The registers are clocked at the positive edge of the internal system clock ($clk_{I/O}$). The latch is transparent in the high period of the internal system clock.

The edge detector generates one $clk_{T1}/clk_{T0}$ pulse for each positive (CSn2:0 = 7) or negative (CSn2:0 = 6) edge it detects.

**Figure 17-1.** T1/T0 Pin Sampling



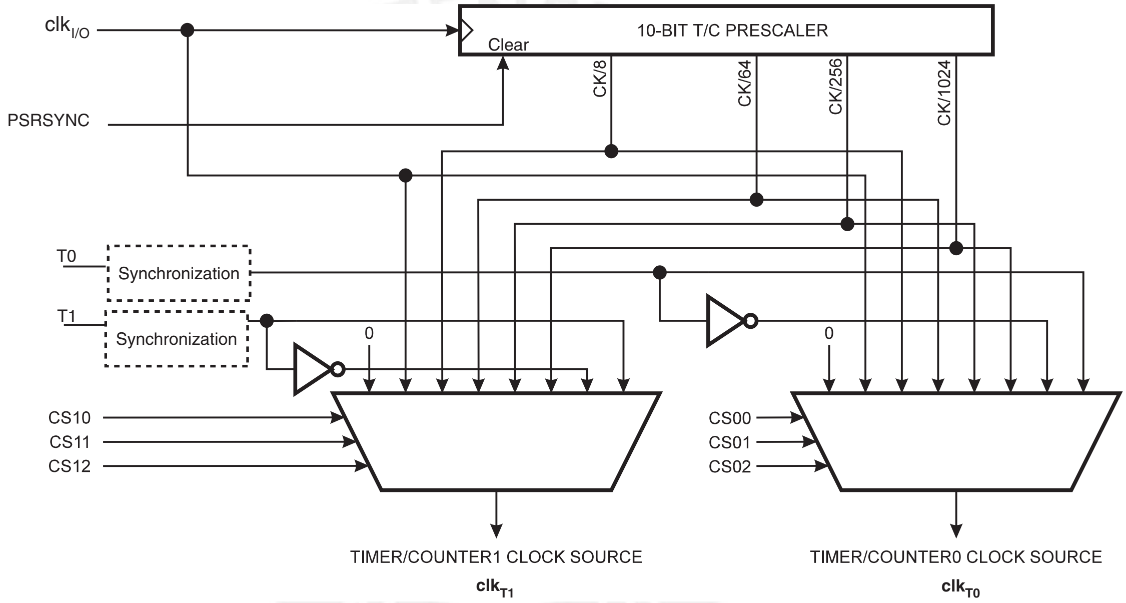
The synchronization and edge detector logic introduces a delay of 2.5 to 3.5 system clock cycles from an edge has been applied to the T1/T0 pin to the counter is updated.

Enabling and disabling of the clock input must be done when T1/T0 has been stable for at least one system clock cycle, otherwise it is a risk that a false Timer/Counter clock pulse is generated.

Each half period of the external clock applied must be longer than one system clock cycle to ensure correct sampling. The external clock must be guaranteed to have less than half the system clock frequency ($f_{ExtClk} < f_{clk\_I/O}/2$) given a 50/50% duty cycle. Since the edge detector uses sampling, the maximum frequency of an external clock it can detect is half the sampling frequency (Nyquist sampling theorem). However, due to variation of the system clock frequency and duty cycle caused by Oscillator source (crystal, resonator, and capacitors) tolerances, it is recommended that maximum frequency of an external clock source is less than $f_{clk\_I/O}/2.5$.

An external clock source can not be prescaled.

**Figure 17-2.** Prescaler for Timer/Counter0 and Timer/Counter1[(1)]



Note: 1. The synchronization logic on the input pins (T1/T0) is shown in Figure 17-1.

## 17.4 Register Description

### 17.4.1 GTCCR – General Timer/Counter Control Register

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| 0x23 (0x43) | TSM | – | – | – | – | – | PSRASY | PSRSYNC | GTCCR |
| Read/Write | R/W | R | R | R | R | R | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

- **Bit 7 – TSM: Timer/Counter Synchronization Mode**

Writing the TSM bit to one activates the Timer/Counter Synchronization mode. In this mode, the value that is written to the PSRASY and PSRSYNC bits is kept, hence keeping the corresponding prescaler reset signals asserted. This ensures that the corresponding Timer/Counters are halted and can be configured to the same value without the risk of one of them advancing during configuration. When the TSM bit is written to zero, the PSRASY and PSRSYNC bits are cleared by hardware, and the Timer/Counters start counting simultaneously.

- **Bit 0 – PSRSYNC: Prescaler Reset**

When this bit is one, Timer/Counter1 and Timer/Counter0 prescaler will be Reset. This bit is normally cleared immediately by hardware, except if the TSM bit is set. Note that Timer/Counter1 and Timer/Counter0 share the same prescaler and a reset of this prescaler will affect both timers.

# 18. 8-bit Timer/Counter2 with PWM and Asynchronous Operation

## 18.1 Features

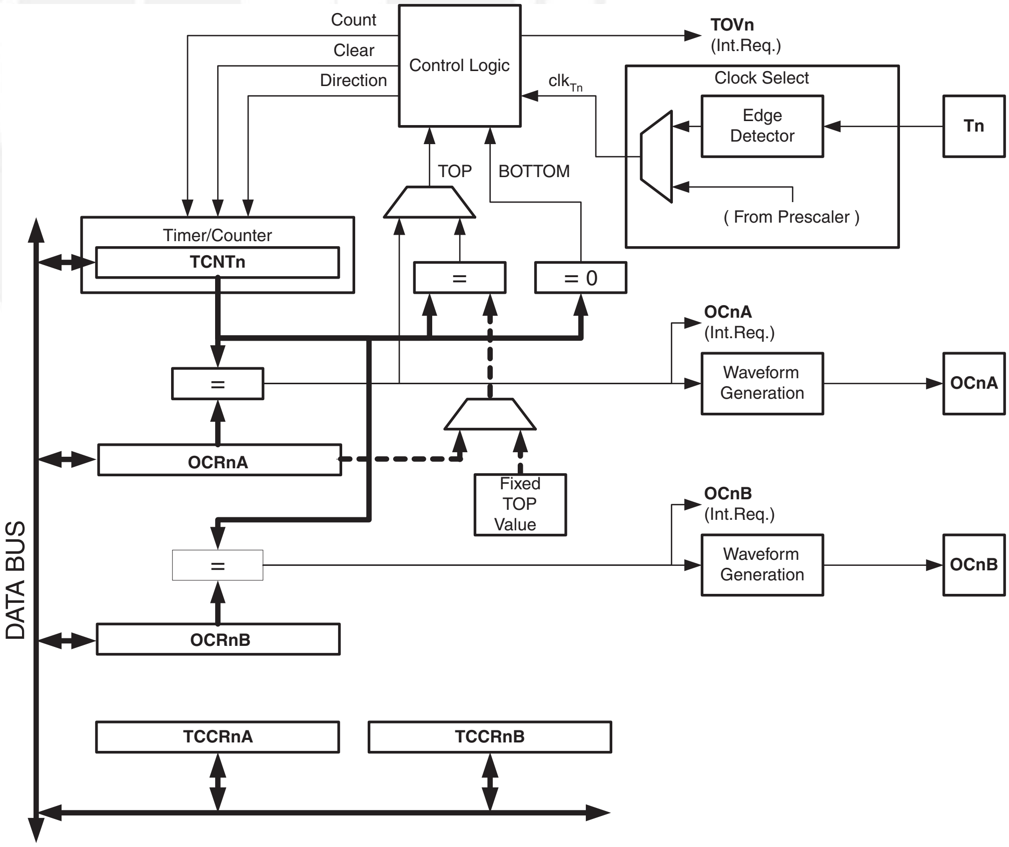
- **Single Channel Counter**
- **Clear Timer on Compare Match (Auto Reload)**
- **Glitch-free, Phase Correct Pulse Width Modulator (PWM)**
- **Frequency Generator**
- **10-bit Clock Prescaler**
- **Overflow and Compare Match Interrupt Sources (TOV2, OCF2A and OCF2B)**
- **Allows Clocking from External 32kHz Watch Crystal Independent of the I/O Clock**

## 18.2 Overview

Timer/Counter2 is a general purpose, single channel, 8-bit Timer/Counter module. A simplified block diagram of the 8-bit Timer/Counter is shown in Figure 18-1. For the actual placement of I/O pins, refer to "Pinout Atmel® ATmega48PA/88PA/168PA" on page 2. CPU accessible I/O Registers, including I/O bits and I/O pins, are shown in bold. The device-specific I/O Register and bit locations are listed in the "Register Description" on page 158.

The PRTIM2 bit in "Minimizing Power Consumption" on page 42 must be written to zero to enable Timer/Counter2 module.

**Figure 18-1.** 8-bit Timer/Counter Block Diagram



### 18.2.1 Registers

The Timer/Counter (TCNT2) and Output Compare Register (OCR2A and OCR2B) are 8-bit registers. Interrupt request (shorten as Int.Req.) signals are all visible in the Timer Interrupt Flag Register (TIFR2). All interrupts are individually masked with the Timer Interrupt Mask Register (TIMSK2). TIFR2 and TIMSK2 are not shown in the figure.

The Timer/Counter can be clocked internally, via the prescaler, or asynchronously clocked from the TOSC1/2 pins, as detailed later in this section. The asynchronous operation is controlled by the Asynchronous Status Register (ASSR). The Clock Select logic block controls which clock source he Timer/Counter uses to increment (or decrement) its value. The Timer/Counter is inactive when no clock source is selected. The output from the Clock Select logic is referred to as the timer clock ($clk_{T2}$).

The double buffered Output Compare Register (OCR2A and OCR2B) are compared with the Timer/Counter value at all times. The result of the compare can be used by the Waveform Generator to generate a PWM or variable frequency output on the Output Compare pins (OC2A and OC2B). See "Output Compare Unit" on page 146 for details. The compare match event will also set the Compare Flag (OCF2A or OCF2B) which can be used to generate an Output Compare interrupt request.

### 18.2.2 Definitions

Many register and bit references in this document are written in general form. A lower case "n" replaces the Timer/Counter number, in this case 2. However, when using the register or bit defines in a program, the precise form must be used, i.e., TCNT2 for accessing Timer/Counter2 counter value and so on.

The definitions in Table 18-1 are also used extensively throughout the section.

**Table 18-1.** Definitions

| | |
|---|---|
| **BOTTOM** | The counter reaches the BOTTOM when it becomes zero (0x00). |
| **MAX** | The counter reaches its MAXimum when it becomes 0xFF (decimal 255). |
| **TOP** | The counter reaches the TOP when it becomes equal to the highest value in the count sequence. The TOP value can be assigned to be the fixed value 0xFF (MAX) or the value stored in the OCR2A Register. The assignment is dependent on the mode of operation. |

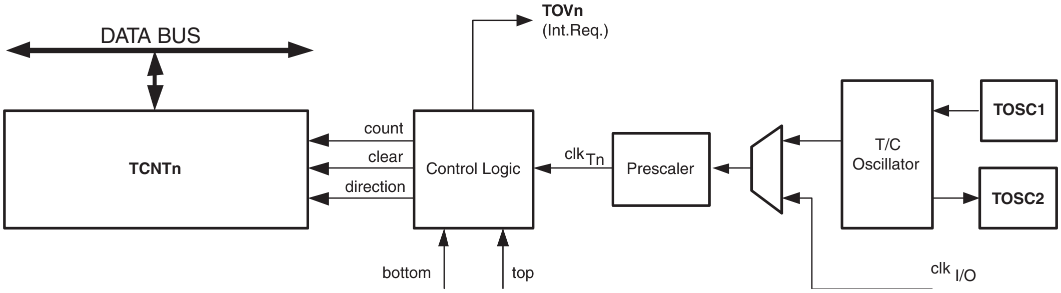
## 18.3 Timer/Counter Clock Sources

The Timer/Counter can be clocked by an internal synchronous or an external asynchronous clock source. The clock source $clk_{T2}$ is by default equal to the MCU clock, $clk_{I/O}$. When the AS2 bit in the ASSR Register is written to logic one, the clock source is taken from the Timer/Counter Oscillator connected to TOSC1 and TOSC2. For details on asynchronous operation, see "ASSR – Asynchronous Status Register" on page 164. For details on clock sources and prescaler, see "Timer/Counter Prescaler" on page 157.

## 18.4 Counter Unit

The main part of the 8-bit Timer/Counter is the programmable bi-directional counter unit. Figure 18-2 on page 145 shows a block diagram of the counter and its surrounding environment.

**Figure 18-2.** Counter Unit Block Diagram



Signal description (internal signals):

| | |
|---|---|
| **count** | Increment or decrement TCNT2 by 1. |
| **direction** | Selects between increment and decrement. |
| **clear** | Clear TCNT2 (set all bits to zero). |
| **$clk_{Tn}$** | Timer/Counter clock, referred to as $clk_{T2}$ in the following. |
| **top** | Signalizes that TCNT2 has reached maximum value. |
| **bottom** | Signalizes that TCNT2 has reached minimum value (zero). |

Depending on the mode of operation used, the counter is cleared, incremented, or decremented at each timer clock ($clk_{T2}$). $clk_{T2}$ can be generated from an external or internal clock source, selected by the Clock Select bits (CS22:0). When no clock source is selected (CS22:0 = 0) the timer is stopped. However, the TCNT2 value can be accessed by the CPU, regardless of whether $clk_{T2}$ is present or not. A CPU write overrides (has priority over) all counter clear or count operations.

The counting sequence is determined by the setting of the WGM21 and WGM20 bits located in the Timer/Counter Control Register (TCCR2A) and the WGM22 located in the Timer/Counter Control Register B (TCCR2B). There are close connections between how the counter behaves (counts) and how waveforms are generated on the Output Compare outputs OC2A and OC2B. For more details about advanced counting sequences and waveform generation, see "Modes of Operation" on page 149.

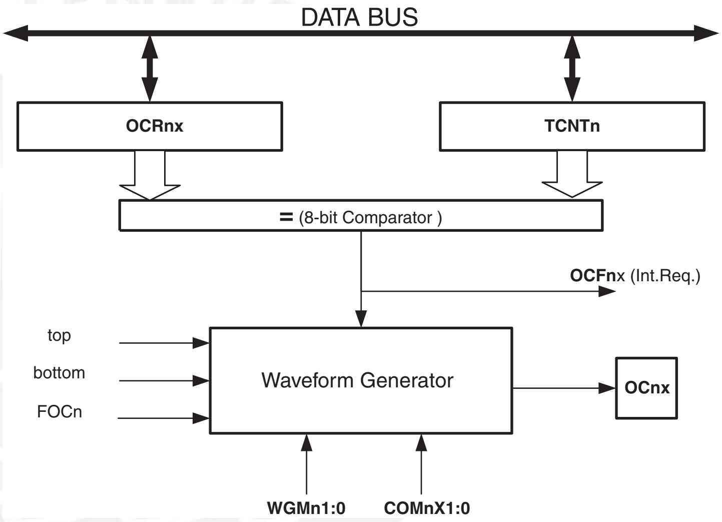
The Timer/Counter Overflow Flag (TOV2) is set according to the mode of operation selected by the WGM22:0 bits. TOV2 can be used for generating a CPU interrupt.

## 18.5 Output Compare Unit

The 8-bit comparator continuously compares TCNT2 with the Output Compare Register (OCR2A and OCR2B). Whenever TCNT2 equals OCR2A or OCR2B, the comparator signals a match. A match will set the Output Compare Flag (OCF2A or OCF2B) at the next timer clock cycle. If the corresponding interrupt is enabled, the Output Compare Flag generates an Output Compare interrupt. The Output Compare Flag is automatically cleared when the interrupt is executed. Alternatively, the Output Compare Flag can be cleared by software by writing a logical one to its I/O bit location. The Waveform Generator uses the match signal to generate an output according to operating mode set by the WGM22:0 bits and Compare Output mode (COM2x1:0) bits. The max and bottom signals are used by the Waveform Generator for handling the special cases of the extreme values in some modes of operation ("Modes of Operation" on page 149).

Figure 18-3 shows a block diagram of the Output Compare unit.

**Figure 18-3.** Output Compare Unit, Block Diagram



The OCR2x Register is double buffered when using any of the Pulse Width Modulation (PWM) modes. For the Normal and Clear Timer on Compare (CTC) modes of operation, the double buffering is disabled. The double buffering synchronizes the update of the OCR2x Compare Register to either top or bottom of the counting sequence. The synchronization prevents the occurrence of odd-length, non-symmetrical PWM pulses, thereby making the output glitch-free.

The OCR2x Register access may seem complex, but this is not case. When the double buffering is enabled, the CPU has access to the OCR2x Buffer Register, and if double buffering is disabled the CPU will access the OCR2x directly.

### 18.5.1 Force Output Compare

In non-PWM waveform generation modes, the match output of the comparator can be forced by writing a one to the Force Output Compare (FOC2x) bit. Forcing compare match will not set the OCF2x Flag or reload/clear the timer, but the OC2x pin will be updated as if a real compare match had occurred (the COM2x1:0 bits settings define whether the OC2x pin is set, cleared or toggled).

### 18.5.2 Compare Match Blocking by TCNT2 Write

All CPU write operations to the TCNT2 Register will block any compare match that occurs in the next timer clock cycle, even when the timer is stopped. This feature allows OCR2x to be initialized to the same value as TCNT2 without triggering an interrupt when the Timer/Counter clock is enabled.

### 18.5.3 Using the Output Compare Unit

Since writing TCNT2 in any mode of operation will block all compare matches for one timer clock cycle, there are risks involved when changing TCNT2 when using the Output Compare channel, independently of whether the Timer/Counter is running or not. If the value written to TCNT2 equals the OCR2x value, the compare match will be missed, resulting in incorrect waveform generation. Similarly, do not write the TCNT2 value equal to BOTTOM when the counter is downcounting.

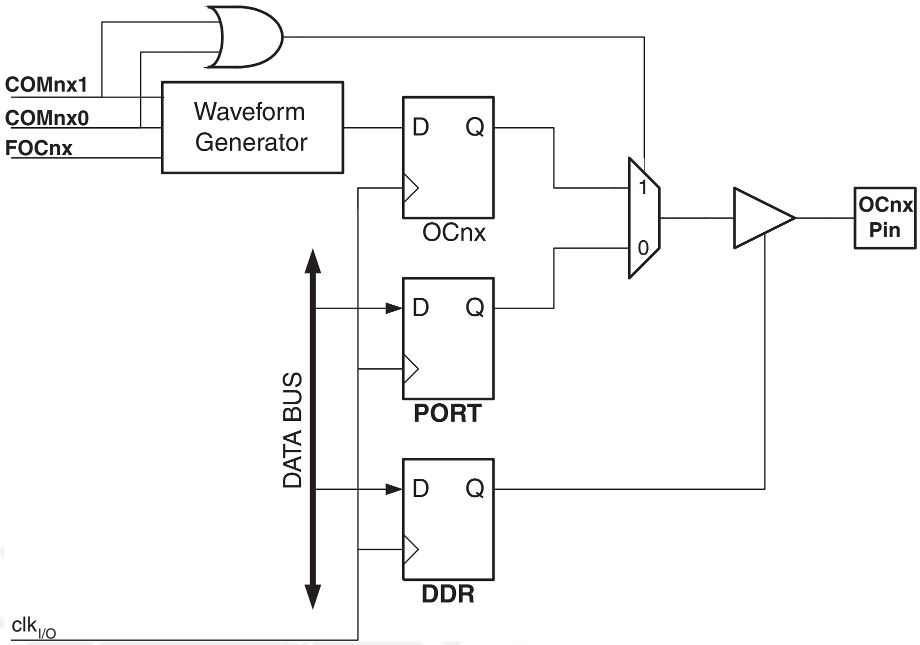
The setup of the OC2x should be performed before setting the Data Direction Register for the port pin to output. The easiest way of setting the OC2x value is to use the Force Output Compare (FOC2x) strobe bit in Normal mode. The OC2x Register keeps its value even when changing between Waveform Generation modes.

Be aware that the COM2x1:0 bits are not double buffered together with the compare value. Changing the COM2x1:0 bits will take effect immediately.

## 18.6 Compare Match Output Unit

The Compare Output mode (COM2x1:0) bits have two functions. The Waveform Generator uses the COM2x1:0 bits for defining the Output Compare (OC2x) state at the next compare match. Also, the COM2x1:0 bits control the OC2x pin output source. Figure 18-4 shows a simplified schematic of the logic affected by the COM2x1:0 bit setting. The I/O Registers, I/O bits, and I/O pins in the figure are shown in bold. Only the parts of the general I/O Port Control Registers (DDR and PORT) that are affected by the COM2x1:0 bits are shown. When referring to the OC2x state, the reference is for the internal OC2x Register, not the OC2x pin.

**Figure 18-4.** Compare Match Output Unit, Schematic



The general I/O port function is overridden by the Output Compare (OC2x) from the Waveform Generator if either of the COM2x1:0 bits are set. However, the OC2x pin direction (input or output) is still controlled by the Data Direction Register (DDR) for the port pin. The Data Direction Register bit for the OC2x pin (DDR_OC2x) must be set as output before the OC2x value is visible on the pin. The port override function is independent of the Waveform Generation mode.

The design of the Output Compare pin logic allows initialization of the OC2x state before the output is enabled. Note that some COM2x1:0 bit settings are reserved for certain modes of operation. See "Register Description" on page 158.

### 18.6.1 Compare Output Mode and Waveform Generation

The Waveform Generator uses the COM2x1:0 bits differently in normal, CTC, and PWM modes. For all modes, setting the COM2x1:0 = 0 tells the Waveform Generator that no action on the OC2x Register is to be performed on the next compare match. For compare output actions in the non-PWM modes refer to Table 18-5 on page 159. For fast PWM mode, refer to Table 18-6 on page 159, and for phase correct PWM refer to Table 18-7 on page 160.

A change of the COM2x1:0 bits state will have effect at the first compare match after the bits are written. For non-PWM modes, the action can be forced to have immediate effect by using the FOC2x strobe bits.

## 18.7 Modes of Operation

The mode of operation, i.e., the behavior of the Timer/Counter and the Output Compare pins, is defined by the combination of the Waveform Generation mode (WGM22:0) and Compare Output mode (COM2x1:0) bits. The Compare Output mode bits do not affect the counting sequence, while the Waveform Generation mode bits do. The COM2x1:0 bits control whether the PWM output generated should be inverted or not (inverted or non-inverted PWM). For non-PWM modes the COM2x1:0 bits control whether the output should be set, cleared, or toggled at a compare match (See "Compare Match Output Unit" on page 147.).

For detailed timing information refer to "Timer/Counter Timing Diagrams" on page 154.

### 18.7.1 Normal Mode

The simplest mode of operation is the Normal mode (WGM22:0 = 0). In this mode the counting direction is always up (incrementing), and no counter clear is performed. The counter simply overruns when it passes its maximum 8-bit value (TOP = 0xFF) and then restarts from the bottom (0x00). In normal operation the Timer/Counter Overflow Flag (TOV2) will be set in the same timer clock cycle as the TCNT2 becomes zero. The TOV2 Flag in this case behaves like a ninth bit, except that it is only set, not cleared. However, combined with the timer overflow interrupt that automatically clears the TOV2 Flag, the timer resolution can be increased by software. There are no special cases to consider in the Normal mode, a new counter value can be written anytime.

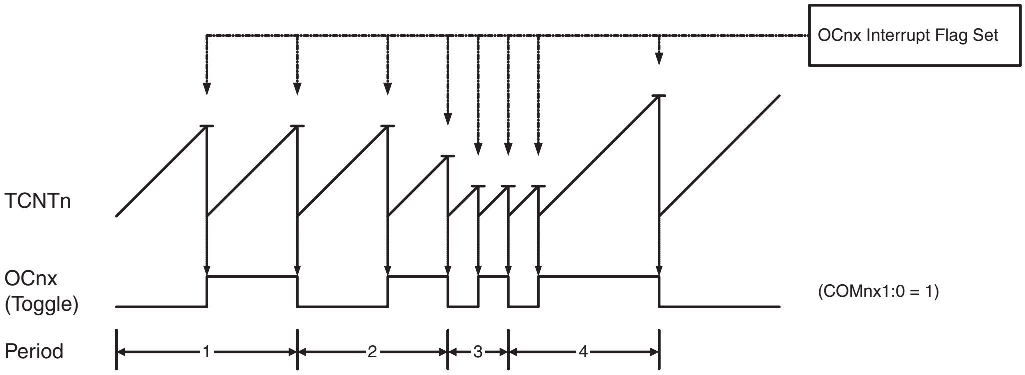
The Output Compare unit can be used to generate interrupts at some given time. Using the Output Compare to generate waveforms in Normal mode is not recommended, since this will occupy too much of the CPU time.

### 18.7.2 Clear Timer on Compare Match (CTC) Mode

In Clear Timer on Compare or CTC mode (WGM22:0 = 2), the OCR2A Register is used to manipulate the counter resolution. In CTC mode the counter is cleared to zero when the counter value (TCNT2) matches the OCR2A. The OCR2A defines the top value for the counter, hence also its resolution. This mode allows greater control of the compare match output frequency. It also simplifies the operation of counting external events.

The timing diagram for the CTC mode is shown in Figure 18-5. The counter value (TCNT2) increases until a compare match occurs between TCNT2 and OCR2A, and then counter (TCNT2) is cleared.

**Figure 18-5.** CTC Mode, Timing Diagram



An interrupt can be generated each time the counter value reaches the TOP value by using the OCF2A Flag. If the interrupt is enabled, the interrupt handler routine can be used for updating the TOP value. However, changing TOP to a value close to BOTTOM when the counter is running with none or a low prescaler value must be done with care since the CTC mode does not have the double buffering feature. If the new value written to OCR2A is lower than the current value of TCNT2, the counter will miss the compare match. The counter will then have to count to its maximum value (0xFF) and wrap around starting at 0x00 before the compare match can occur.

For generating a waveform output in CTC mode, the OC2A output can be set to toggle its logical level on each compare match by setting the Compare Output mode bits to toggle mode (COM2A1:0 = 1). The OC2A value will not be visible on the port pin unless the data direction for the pin is set to output. The waveform generated will have a maximum frequency of $f_{OC2A} = f_{clk\_I/O}/2$ when OCR2A is set to zero (0x00). The waveform frequency is defined by the following equation:

$$f_{OCnx} = \frac{f_{clk\_I/O}}{2 \times N \times (1 + OCRnx)}$$

The *N* variable represents the prescale factor (1, 8, 32, 64, 128, 256, or 1024).

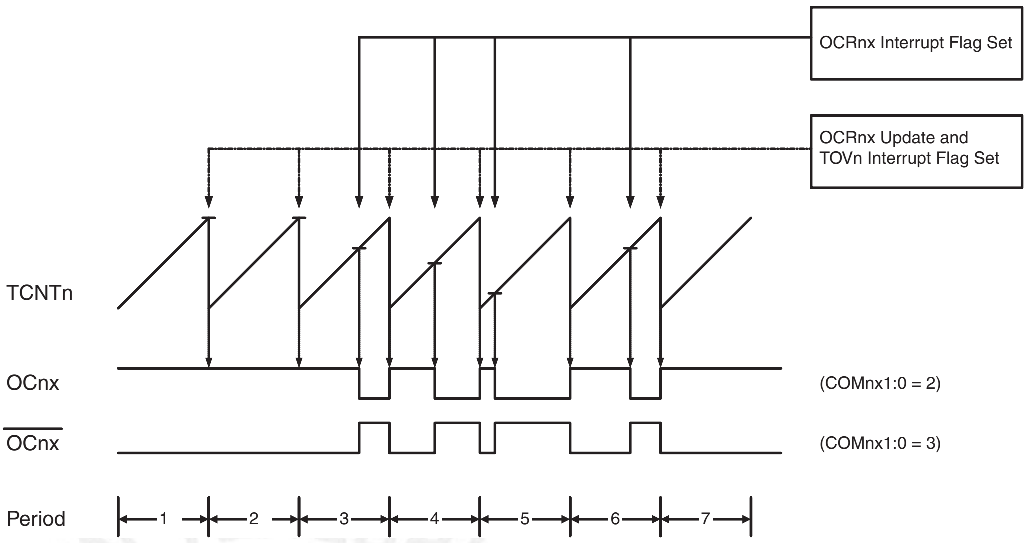
As for the Normal mode of operation, the TOV2 Flag is set in the same timer clock cycle that the counter counts from MAX to 0x00.

### 18.7.3 Fast PWM Mode

The fast Pulse Width Modulation or fast PWM mode (WGM22:0 = 3 or 7) provides a high frequency PWM waveform generation option. The fast PWM differs from the other PWM option by its single-slope operation. The counter counts from BOTTOM to TOP then restarts from BOTTOM. TOP is defined as 0xFF when WGM2:0 = 3, and OCR2A when MGM2:0 = 7. In non-inverting Compare Output mode, the Output Compare (OC2x) is cleared on the compare match between TCNT2 and OCR2x, and set at BOTTOM. In inverting Compare Output mode, the output is set on compare match and cleared at BOTTOM. Due to the single-slope operation, the operating frequency of the fast PWM mode can be twice as high as the phase correct PWM mode that uses dual-slope operation. This high frequency makes the fast PWM mode well suited for power regulation, rectification, and DAC applications. High frequency allows physically small sized external components (coils, capacitors), and therefore reduces total system cost.

In fast PWM mode, the counter is incremented until the counter value matches the TOP value. The counter is then cleared at the following timer clock cycle. The timing diagram for the fast PWM mode is shown in Figure 18-6. The TCNT2 value is in the timing diagram shown as a histogram for illustrating the single-slope operation. The diagram includes non-inverted and inverted PWM outputs. The small horizontal line marks on the TCNT2 slopes represent compare matches between OCR2x and TCNT2.

**Figure 18-6.** Fast PWM Mode, Timing Diagram



The Timer/Counter Overflow Flag (TOV2) is set each time the counter reaches TOP. If the interrupt is enabled, the interrupt handler routine can be used for updating the compare value.

In fast PWM mode, the compare unit allows generation of PWM waveforms on the OC2x pin. Setting the COM2x1:0 bits to two will produce a non-inverted PWM and an inverted PWM output can be generated by setting the COM2x1:0 to three. TOP is defined as 0xFF when WGM2:0 = 3, and OCR2A when MGM2:0 = 7. (See Table 18-3 on page 158). The actual OC2x value will only be visible on the port pin if the data direction for the port pin is set as output. The PWM waveform is generated by setting (or clearing) the OC2x Register at the compare match between OCR2x and TCNT2, and clearing (or setting) the OC2x Register at the timer clock cycle the counter is cleared (changes from TOP to BOTTOM).

The PWM frequency for the output can be calculated by the following equation:

$$f_{OCnxPWM} = \frac{f_{clk\_I/O}}{N \times 256}$$

The *N* variable represents the prescale factor (1, 8, 32, 64, 128, 256, or 1024).

The extreme values for the OCR2A Register represent special cases when generating a PWM waveform output in the fast PWM mode. If the OCR2A is set equal to BOTTOM, the output will be a narrow spike for each MAX+1 timer clock cycle. Setting the OCR2A equal to MAX will result in a constantly high or low output (depending on the polarity of the output set by the COM2A1:0 bits.)

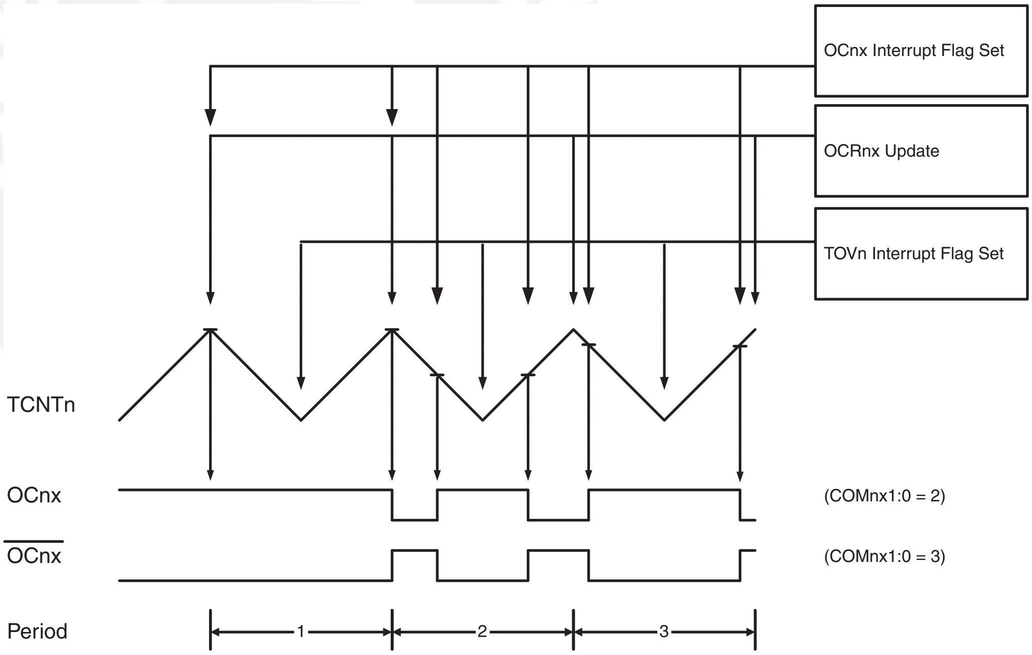
A frequency (with 50% duty cycle) waveform output in fast PWM mode can be achieved by setting OC2x to toggle its logical level on each compare match (COM2x1:0 = 1). The waveform generated will have a maximum frequency of $f_{oc2} = f_{clk\_I/O}/2$ when OCR2A is set to zero. This feature is similar to the OC2A toggle in CTC mode, except the double buffer feature of the Output Compare unit is enabled in the fast PWM mode.

### 18.7.4 Phase Correct PWM Mode

The phase correct PWM mode (WGM22:0 = 1 or 5) provides a high resolution phase correct PWM waveform generation option. The phase correct PWM mode is based on a dual-slope operation. The counter counts repeatedly from BOTTOM to TOP and then from TOP to BOTTOM. TOP is defined as 0xFF when WGM2:0 = 3, and OCR2A when MGM2:0 = 7. In non-inverting Compare Output mode, the Output Compare (OC2x) is cleared on the compare match between TCNT2 and OCR2x while upcounting, and set on the compare match while downcounting. In inverting Output Compare mode, the operation is inverted. The dual-slope operation has lower maximum operation frequency than single slope operation. However, due to the symmetric feature of the dual-slope PWM modes, these modes are preferred for motor control applications.

In phase correct PWM mode the counter is incremented until the counter value matches TOP. When the counter reaches TOP, it changes the count direction. The TCNT2 value will be equal to TOP for one timer clock cycle. The timing diagram for the phase correct PWM mode is shown on Figure 18-7. The TCNT2 value is in the timing diagram shown as a histogram for illustrating the dual-slope operation. The diagram includes non-inverted and inverted PWM outputs. The small horizontal line marks on the TCNT2 slopes represent compare matches between OCR2x and TCNT2.

**Figure 18-7.** Phase Correct PWM Mode, Timing Diagram



The Timer/Counter Overflow Flag (TOV2) is set each time the counter reaches BOTTOM. The Interrupt Flag can be used to generate an interrupt each time the counter reaches the BOTTOM value.

In phase correct PWM mode, the compare unit allows generation of PWM waveforms on the OC2x pin. Setting the COM2x1:0 bits to two will produce a non-inverted PWM. An inverted PWM output can be generated by setting the COM2x1:0 to three. TOP is defined as 0xFF when WGM2:0 = 3, and OCR2A when MGM2:0 = 7 (See Table 18-4 on page 159). The actual OC2x value will only be visible on the port pin if the data direction for the port pin is set as output. The PWM waveform is generated by clearing (or setting) the OC2x Register at the compare match between OCR2x and TCNT2 when the counter increments, and setting (or clearing) the OC2x Register at compare match between OCR2x and TCNT2 when the counter decrements. The PWM frequency for the output when using phase correct PWM can be calculated by the following equation:

$$f_{OCnxPCPWM} = \frac{f_{clk\_I/O}}{N \times 510}$$

The *N* variable represents the prescale factor (1, 8, 32, 64, 128, 256, or 1024).

The extreme values for the OCR2A Register represent special cases when generating a PWM waveform output in the phase correct PWM mode. If the OCR2A is set equal to BOTTOM, the output will be continuously low and if set equal to MAX the output will be continuously high for non-inverted PWM mode. For inverted PWM the output will have the opposite logic values.

At the very start of period 2 in Figure 18-7 OCnx has a transition from high to low even though there is no Compare Match. The point of this transition is to guarantee symmetry around BOTTOM. There are two cases that give a transition without Compare Match.

- OCR2A changes its value from MAX, like in Figure 18-7. When the OCR2A value is MAX the OCn pin value is the same as the result of a down-counting compare match. To ensure symmetry around BOTTOM the OCn value at MAX must correspond to the result of an up-counting Compare Match.
- The timer starts counting from a value higher than the one in OCR2A, and for that reason misses the Compare Match and hence the OCn change that would have happened on the way up.

## 18.8 Timer/Counter Timing Diagrams

The following figures show the Timer/Counter in synchronous mode, and the timer clock ($clk_{T2}$) is therefore shown as a clock enable signal. In asynchronous mode, $clk_{I/O}$ should be replaced by the Timer/Counter Oscillator clock. The figures include information on when Interrupt Flags are set. Figure 18-8 contains timing data for basic Timer/Counter operation. The figure shows the count sequence close to the MAX value in all modes other than phase correct PWM mode.

**Figure 18-8.** Timer/Counter Timing Diagram, no Prescaling

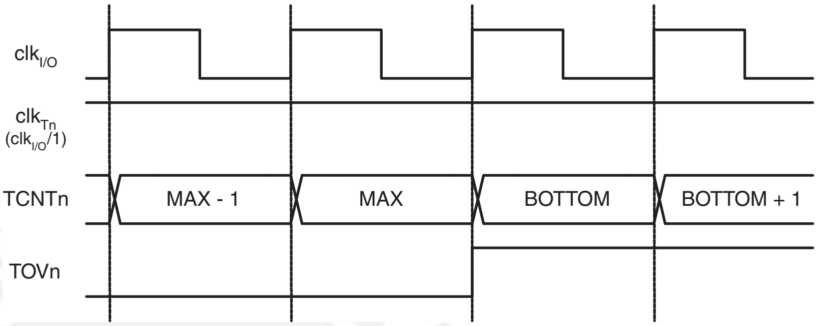


Figure 18-9 shows the same timing data, but with the prescaler enabled.

**Figure 18-9.** Timer/Counter Timing Diagram, with Prescaler ($f_{clk\_I/O}/8$)

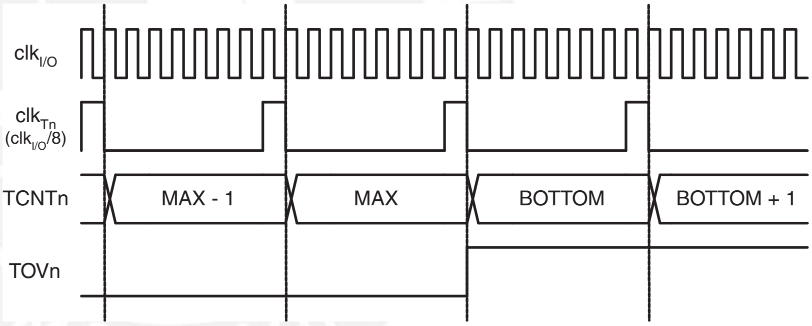


Figure 18-10 shows the setting of OCF2A in all modes except CTC mode.

**Figure 18-10.** Timer/Counter Timing Diagram, Setting of OCF2A, with Prescaler ($f_{clk\_I/O}/8$)

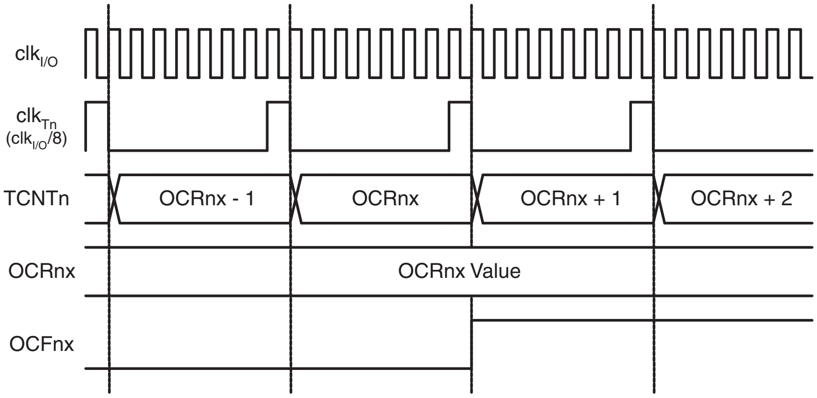
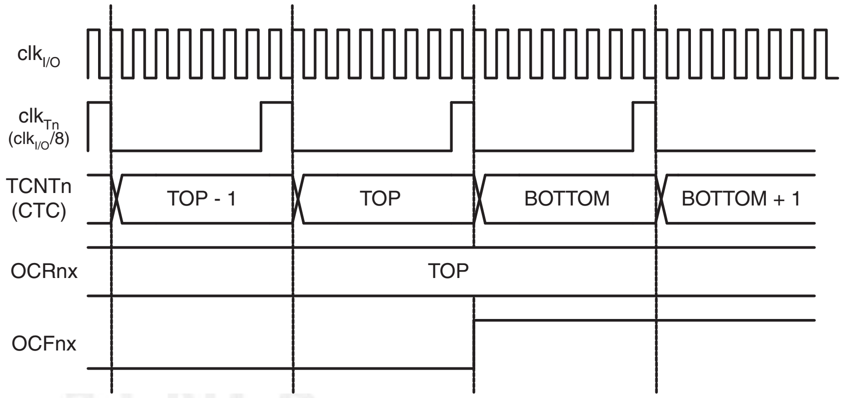


Figure 18-11 shows the setting of OCF2A and the clearing of TCNT2 in CTC mode.

**Figure 18-11.** Timer/Counter Timing Diagram, Clear Timer on Compare Match mode, with Prescaler ($f_{clk\_I/O}/8$)



## 18.9 Asynchronous Operation of Timer/Counter2

When Timer/Counter2 operates asynchronously, some considerations must be taken.

- Warning: When switching between asynchronous and synchronous clocking of Timer/Counter2, the Timer Registers TCNT2, OCR2x, and TCCR2x might be corrupted. A safe procedure for switching clock source is:
  a. Disable the Timer/Counter2 interrupts by clearing OCIE2x and TOIE2.
  b. Select clock source by setting AS2 as appropriate.
  c. Write new values to TCNT2, OCR2x, and TCCR2x.
  d. To switch to asynchronous operation: Wait for TCN2xUB, OCR2xUB, and TCR2xUB.
  e. Clear the Timer/Counter2 Interrupt Flags.
  f. Enable interrupts, if needed.
- The CPU main clock frequency must be more than four times the Oscillator frequency.
- When writing to one of the registers TCNT2, OCR2x, or TCCR2x, the value is transferred to a temporary register, and latched after two positive edges on TOSC1. The user should not write a new value before the contents of the temporary register have been transferred to its destination. Each of the five mentioned registers have their individual temporary register, which means that e.g. writing to TCNT2 does not disturb an OCR2x write in progress. To detect that a transfer to the destination register has taken place, the Asynchronous Status Register – ASSR has been implemented.
- When entering Power-save or ADC Noise Reduction mode after having written to TCNT2, OCR2x, or TCCR2x, the user must wait until the written register has been updated if Timer/Counter2 is used to wake up the device. Otherwise, the MCU will enter sleep mode before the changes are effective. This is particularly important if any of the Output Compare2 interrupt is used to wake up the device, since the Output Compare function is disabled during writing to OCR2x or TCNT2. If the write cycle is not finished, and the MCU enters sleep mode before the corresponding OCR2xUB bit returns to zero, the device will never receive a compare match interrupt, and the MCU will not wake up.

- If Timer/Counter2 is used to wake the device up from Power-save or ADC Noise Reduction mode, precautions must be taken if the user wants to re-enter one of these modes: If re-entering sleep mode within the TOSC1 cycle, the interrupt will immediately occur and the device wake up again. The result is multiple interrupts and wake-ups within one TOSC1 cycle from the first interrupt. If the user is in doubt whether the time before re-entering Power-save or ADC Noise Reduction mode is sufficient, the following algorithm can be used to ensure that one TOSC1 cycle has elapsed:

  a. Write a value to TCCR2x, TCNT2, or OCR2x.

  b. Wait until the corresponding Update Busy Flag in ASSR returns to zero.

  c. Enter Power-save or ADC Noise Reduction mode.

  - When the asynchronous operation is selected, the 32.768kHz Oscillator for Timer/Counter2 is always running, except in Power-down and Standby modes. After a Power-up Reset or wake-up from Power-down or Standby mode, the user should be aware of the fact that this Oscillator might take as long as one second to stabilize. The user is advised to wait for at least one second before using Timer/Counter2 after power-up or wake-up from Power-down or Standby mode. The contents of all Timer/Counter2 Registers must be considered lost after a wake-up from Power-down or Standby mode due to unstable clock signal upon start-up, no matter whether the Oscillator is in use or a clock signal is applied to the TOSC1 pin.

- Description of wake up from Power-save or ADC Noise Reduction mode when the timer is clocked asynchronously: When the interrupt condition is met, the wake up process is started on the following cycle of the timer clock, that is, the timer is always advanced by at least one before the processor can read the counter value. After wake-up, the MCU is halted for four cycles, it executes the interrupt routine, and resumes execution from the instruction following SLEEP.

- Reading of the TCNT2 Register shortly after wake-up from Power-save may give an incorrect result. Since TCNT2 is clocked on the asynchronous TOSC clock, reading TCNT2 must be done through a register synchronized to the internal I/O clock domain. Synchronization takes place for every rising TOSC1 edge. When waking up from Power-save mode, and the I/O clock ($clk_{I/O}$) again becomes active, TCNT2 will read as the previous value (before entering sleep) until the next rising TOSC1 edge. The phase of the TOSC clock after waking up from Power-save mode is essentially unpredictable, as it depends on the wake-up time. The recommended procedure for reading TCNT2 is thus as follows:

  a. Write any value to either of the registers OCR2x or TCCR2x.

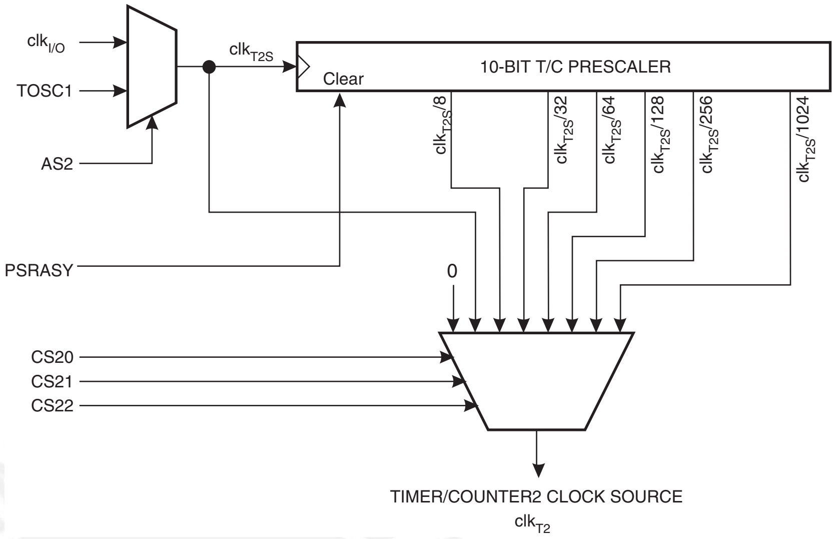
  b. Wait for the corresponding Update Busy Flag to be cleared.

  c. Read TCNT2.

During asynchronous operation, the synchronization of the Interrupt Flags for the asynchronous timer takes 3 processor cycles plus one timer cycle. The timer is therefore advanced by at least one before the processor can read the timer value causing the setting of the Interrupt Flag. The Output Compare pin is changed on the timer clock and is not synchronized to the processor clock.

## 18.10 Timer/Counter Prescaler

**Figure 18-12.** Prescaler for Timer/Counter2



The clock source for Timer/Counter2 is named $clk_{T2S}$. $clk_{T2S}$ is by default connected to the main system I/O clock $clk_{IO}$. By setting the AS2 bit in ASSR, Timer/Counter2 is asynchronously clocked from the TOSC1 pin. This enables use of Timer/Counter2 as a Real Time Counter (RTC). When AS2 is set, pins TOSC1 and TOSC2 are disconnected from Port B. A crystal can then be connected between the TOSC1 and TOSC2 pins to serve as an independent clock source for Timer/Counter2. The Oscillator is optimized for use with a 32.768kHz crystal.

For Timer/Counter2, the possible prescaled selections are: $clk_{T2S}/8$, $clk_{T2S}/32$, $clk_{T2S}/64$, $clk_{T2S}/128$, $clk_{T2S}/256$, and $clk_{T2S}/1024$. Additionally, $clk_{T2S}$ as well as 0 (stop) may be selected. Setting the PSRASY bit in GTCCR resets the prescaler. This allows the user to operate with a predictable prescaler.

## 18.11 Register Description

### 18.11.1 TCCR2A – Timer/Counter Control Register A

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| (0xB0) | COM2A1 | COM2A0 | COM2B1 | COM2B0 | – | – | WGM21 | WGM20 | TCCR2A |
| Read/Write | R/W | R/W | R/W | R/W | R | R | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

- **Bits 7:6 – COM2A1:0: Compare Match Output A Mode**

These bits control the Output Compare pin (OC2A) behavior. If one or both of the COM2A1:0 bits are set, the OC2A output overrides the normal port functionality of the I/O pin it is connected to. However, note that the Data Direction Register (DDR) bit corresponding to the OC2A pin must be set in order to enable the output driver.

When OC2A is connected to the pin, the function of the COM2A1:0 bits depends on the WGM22:0 bit setting. Table 18-2 shows the COM2A1:0 bit functionality when the WGM22:0 bits are set to a normal or CTC mode (non-PWM).

**Table 18-2.** Compare Output Mode, non-PWM Mode

| COM2A1 | COM2A0 | Description |
|---|---|---|
| 0 | 0 | Normal port operation, OC0A disconnected. |
| 0 | 1 | Toggle OC2A on Compare Match |
| 1 | 0 | Clear OC2A on Compare Match |
| 1 | 1 | Set OC2A on Compare Match |

Table 18-3 shows the COM2A1:0 bit functionality when the WGM21:0 bits are set to fast PWM mode.

**Table 18-3.** Compare Output Mode, Fast PWM Mode[(1)]

| COM2A1 | COM2A0 | Description |
|---|---|---|
| 0 | 0 | Normal port operation, OC2A disconnected. |
| 0 | 1 | WGM22 = 0: Normal Port Operation, OC0A Disconnected.<br>WGM22 = 1: Toggle OC2A on Compare Match. |
| 1 | 0 | Clear OC2A on Compare Match, set OC2A at BOTTOM, (non-inverting mode). |
| 1 | 1 | Set OC2A on Compare Match, clear OC2A at BOTTOM, (inverting mode). |

Note: 1. A special case occurs when OCR2A equals TOP and COM2A1 is set. In this case, the Compare Match is ignored, but the set or clear is done at BOTTOM. See "Fast PWM Mode" on page 150 for more details.

Table 18-4 shows the COM2A1:0 bit functionality when the WGM22:0 bits are set to phase correct PWM mode.

**Table 18-4.** Compare Output Mode, Phase Correct PWM Mode[1]

| COM2A1 | COM2A0 | Description |
|---|---|---|
| 0 | 0 | Normal port operation, OC2A disconnected. |
| 0 | 1 | WGM22 = 0: Normal Port Operation, OC2A Disconnected.<br>WGM22 = 1: Toggle OC2A on Compare Match. |
| 1 | 0 | Clear OC2A on Compare Match when up-counting. Set OC2A on Compare Match when down-counting. |
| 1 | 1 | Set OC2A on Compare Match when up-counting. Clear OC2A on Compare Match when down-counting. |

Note: 1. A special case occurs when OCR2A equals TOP and COM2A1 is set. In this case, the Compare Match is ignored, but the set or clear is done at TOP. See "Phase Correct PWM Mode" on page 152 for more details.

- **Bits 5:4 – COM2B1:0: Compare Match Output B Mode**

These bits control the Output Compare pin (OC2B) behavior. If one or both of the COM2B1:0 bits are set, the OC2B output overrides the normal port functionality of the I/O pin it is connected to. However, note that the Data Direction Register (DDR) bit corresponding to the OC2B pin must be set in order to enable the output driver.

When OC2B is connected to the pin, the function of the COM2B1:0 bits depends on the WGM22:0 bit setting. Table 18-5 shows the COM2B1:0 bit functionality when the WGM22:0 bits are set to a normal or CTC mode (non-PWM).

**Table 18-5.** Compare Output Mode, non-PWM Mode

| COM2B1 | COM2B0 | Description |
|---|---|---|
| 0 | 0 | Normal port operation, OC2B disconnected. |
| 0 | 1 | Toggle OC2B on Compare Match |
| 1 | 0 | Clear OC2B on Compare Match |
| 1 | 1 | Set OC2B on Compare Match |

Table 18-6 shows the COM2B1:0 bit functionality when the WGM22:0 bits are set to fast PWM mode.

**Table 18-6.** Compare Output Mode, Fast PWM Mode[1]

| COM2B1 | COM2B0 | Description |
|---|---|---|
| 0 | 0 | Normal port operation, OC2B disconnected. |
| 0 | 1 | Reserved |
| 1 | 0 | Clear OC2B on Compare Match, set OC2B at BOTTOM, (non-inverting mode). |
| 1 | 1 | Set OC2B on Compare Match, clear OC2B at BOTTOM, (inverting mode). |

Note: 1. A special case occurs when OCR2B equals TOP and COM2B1 is set. In this case, the Compare Match is ignored, but the set or clear is done at BOTTOM. See "Phase Correct PWM Mode" on page 152 for more details.

Table 18-7 shows the COM2B1:0 bit functionality when the WGM22:0 bits are set to phase correct PWM mode.

**Table 18-7.** Compare Output Mode, Phase Correct PWM Mode[1]

| COM2B1 | COM2B0 | Description |
|---|---|---|
| 0 | 0 | Normal port operation, OC2B disconnected. |
| 0 | 1 | Reserved |
| 1 | 0 | Clear OC2B on Compare Match when up-counting. Set OC2B on Compare Match when down-counting. |
| 1 | 1 | Set OC2B on Compare Match when up-counting. Clear OC2B on Compare Match when down-counting. |

Note: 1. A special case occurs when OCR2B equals TOP and COM2B1 is set. In this case, the Compare Match is ignored, but the set or clear is done at TOP. See "Phase Correct PWM Mode" on page 152 for more details.

- **Bits 3, 2 – Reserved**

These bits are reserved bits in the Atmel® ATmega48PA/88PA/168PA and will always read as zero.

- **Bits 1:0 – WGM21:0: Waveform Generation Mode**

Combined with the WGM22 bit found in the TCCR2B Register, these bits control the counting sequence of the counter, the source for maximum (TOP) counter value, and what type of waveform generation to be used, see Table 18-8. Modes of operation supported by the Timer/Counter unit are: Normal mode (counter), Clear Timer on Compare Match (CTC) mode, and two types of Pulse Width Modulation (PWM) modes (see "Modes of Operation" on page 149).

**Table 18-8.** Waveform Generation Mode Bit Description

| Mode | WGM2 | WGM1 | WGM0 | Timer/Counter Mode of Operation | TOP | Update of OCRx at | TOV Flag Set on[1][2] |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | Normal | 0xFF | Immediate | MAX |
| 1 | 0 | 0 | 1 | PWM, Phase Correct | 0xFF | TOP | BOTTOM |
| 2 | 0 | 1 | 0 | CTC | OCRA | Immediate | MAX |
| 3 | 0 | 1 | 1 | Fast PWM | 0xFF | BOTTOM | MAX |
| 4 | 1 | 0 | 0 | Reserved | – | – | – |
| 5 | 1 | 0 | 1 | PWM, Phase Correct | OCRA | TOP | BOTTOM |
| 6 | 1 | 1 | 0 | Reserved | – | – | – |
| 7 | 1 | 1 | 1 | Fast PWM | OCRA | BOTTOM | TOP |

Notes: 1. MAX = 0xFF

2. BOTTOM = 0x00

### 18.11.2 TCCR2B – Timer/Counter Control Register B

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| (0xB1) | FOC2A | FOC2B | – | – | WGM22 | CS22 | CS21 | CS20 | TCCR2B |
| Read/Write | W | W | R | R | R | R | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

- **Bit 7 – FOC2A: Force Output Compare A**

The FOC2A bit is only active when the WGM bits specify a non-PWM mode.

However, for ensuring compatibility with future devices, this bit must be set to zero when TCCR2B is written when operating in PWM mode. When writing a logical one to the FOC2A bit, an immediate Compare Match is forced on the Waveform Generation unit. The OC2A output is changed according to its COM2A1:0 bits setting. Note that the FOC2A bit is implemented as a strobe. Therefore it is the value present in the COM2A1:0 bits that determines the effect of the forced compare.

A FOC2A strobe will not generate any interrupt, nor will it clear the timer in CTC mode using OCR2A as TOP.

The FOC2A bit is always read as zero.

- **Bit 6 – FOC2B: Force Output Compare B**

The FOC2B bit is only active when the WGM bits specify a non-PWM mode.

However, for ensuring compatibility with future devices, this bit must be set to zero when TCCR2B is written when operating in PWM mode. When writing a logical one to the FOC2B bit, an immediate Compare Match is forced on the Waveform Generation unit. The OC2B output is changed according to its COM2B1:0 bits setting. Note that the FOC2B bit is implemented as a strobe. Therefore it is the value present in the COM2B1:0 bits that determines the effect of the forced compare.

A FOC2B strobe will not generate any interrupt, nor will it clear the timer in CTC mode using OCR2B as TOP.

The FOC2B bit is always read as zero.

- **Bits 5:4 – Reserved**

These bits are reserved bits in the Atmel® ATmega48PA/88PA/168PA and will always read as zero.

- **Bit 3 – WGM22: Waveform Generation Mode**

See the description in the "TCCR2A – Timer/Counter Control Register A" on page 158.

- **Bit 2:0 – CS22:0: Clock Select**

The three Clock Select bits select the clock source to be used by the Timer/Counter, see Table 18-9 on page 162.

**Table 18-9.** Clock Select Bit Description

| CS22 | CS21 | CS20 | Description |
|---|---|---|---|
| 0 | 0 | 0 | No clock source (Timer/Counter stopped). |
| 0 | 0 | 1 | $clk_{T2S}$/(No prescaling) |
| 0 | 1 | 0 | $clk_{T2S}$/8 (From prescaler) |
| 0 | 1 | 1 | $clk_{T2S}$/32 (From prescaler) |
| 1 | 0 | 0 | $clk_{T2S}$/64 (From prescaler) |
| 1 | 0 | 1 | $clk_{T2S}$/128 (From prescaler) |
| 1 | 1 | 0 | $clk_{T2S}$/256 (From prescaler) |
| 1 | 1 | 1 | $clk_{T2S}$/1024 (From prescaler) |

If external pin modes are used for the Timer/Counter0, transitions on the T0 pin will clock the counter even if the pin is configured as an output. This feature allows software control of the counting.

### 18.11.3 TCNT2 – Timer/Counter Register

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
| (0xB2) | TCNT2[7:0] | | | | | | | | TCNT2 |
| Read/Write | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

The Timer/Counter Register gives direct access, both for read and write operations, to the Timer/Counter unit 8-bit counter. Writing to the TCNT2 Register blocks (removes) the Compare Match on the following timer clock. Modifying the counter (TCNT2) while the counter is running, introduces a risk of missing a Compare Match between TCNT2 and the OCR2x Registers.

### 18.11.4 OCR2A – Output Compare Register A

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
| (0xB3) | OCR2A[7:0] | | | | | | | | OCR2A |
| Read/Write | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

The Output Compare Register A contains an 8-bit value that is continuously compared with the counter value (TCNT2). A match can be used to generate an Output Compare interrupt, or to generate a waveform output on the OC2A pin.

### 18.11.5 OCR2B – Output Compare Register B

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
| (0xB4) | OCR2B[7:0] | | | | | | | | OCR2B |
| Read/Write | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

The Output Compare Register B contains an 8-bit value that is continuously compared with the counter value (TCNT2). A match can be used to generate an Output Compare interrupt, or to generate a waveform output on the OC2B pin.

### 18.11.6 TIMSK2 – Timer/Counter2 Interrupt Mask Register

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| (0x70) | – | – | – | – | – | OCIE2B | OCIE2A | TOIE2 | TIMSK2 |
| Read/Write | R | R | R | R | R | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

- **Bit 2 – OCIE2B: Timer/Counter2 Output Compare Match B Interrupt Enable**

When the OCIE2B bit is written to one and the I-bit in the Status Register is set (one), the Timer/Counter2 Compare Match B interrupt is enabled. The corresponding interrupt is executed if a compare match in Timer/Counter2 occurs, i.e., when the OCF2B bit is set in the Timer/Counter 2 Interrupt Flag Register – TIFR2.

- **Bit 1 – OCIE2A: Timer/Counter2 Output Compare Match A Interrupt Enable**

When the OCIE2A bit is written to one and the I-bit in the Status Register is set (one), the Timer/Counter2 Compare Match A interrupt is enabled. The corresponding interrupt is executed if a compare match in Timer/Counter2 occurs, i.e., when the OCF2A bit is set in the Timer/Counter 2 Interrupt Flag Register – TIFR2.

- **Bit 0 – TOIE2: Timer/Counter2 Overflow Interrupt Enable**

When the TOIE2 bit is written to one and the I-bit in the Status Register is set (one), the Timer/Counter2 Overflow interrupt is enabled. The corresponding interrupt is executed if an overflow in Timer/Counter2 occurs, i.e., when the TOV2 bit is set in the Timer/Counter2 Interrupt Flag Register – TIFR2.

### 18.11.7 TIFR2 – Timer/Counter2 Interrupt Flag Register

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| 0x17 (0x37) | – | – | – | – | – | OCF2B | OCF2A | TOV2 | TIFR2 |
| Read/Write | R | R | R | R | R | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

- **Bit 2 – OCF2B: Output Compare Flag 2 B**

The OCF2B bit is set (one) when a compare match occurs between the Timer/Counter2 and the data in OCR2B – Output Compare Register2. OCF2B is cleared by hardware when executing the corresponding interrupt handling vector. Alternatively, OCF2B is cleared by writing a logic one to the flag. When the I-bit in SREG, OCIE2B (Timer/Counter2 Compare match Interrupt Enable), and OCF2B are set (one), the Timer/Counter2 Compare match Interrupt is executed.

- **Bit 1 – OCF2A: Output Compare Flag 2 A**

The OCF2A bit is set (one) when a compare match occurs between the Timer/Counter2 and the data in OCR2A – Output Compare Register2. OCF2A is cleared by hardware when executing the corresponding interrupt handling vector. Alternatively, OCF2A is cleared by writing a logic one to the flag. When the I-bit in SREG, OCIE2A (Timer/Counter2 Compare match Interrupt Enable), and OCF2A are set (one), the Timer/Counter2 Compare match Interrupt is executed.

- **Bit 0 – TOV2: Timer/Counter2 Overflow Flag**

The TOV2 bit is set (one) when an overflow occurs in Timer/Counter2. TOV2 is cleared by hardware when executing the corresponding interrupt handling vector. Alternatively, TOV2 is cleared by writing a logic one to the flag. When the SREG I-bit, TOIE2A (Timer/Counter2 Overflow Interrupt Enable), and TOV2 are set (one), the Timer/Counter2 Overflow interrupt is executed. In PWM mode, this bit is set when Timer/Counter2 changes counting direction at 0x00.

### 18.11.8 ASSR – Asynchronous Status Register

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| (0xB6) | – | EXCLK | AS2 | TCN2UB | OCR2AUB | OCR2BUB | TCR2AUB | TCR2BUB | ASSR |
| Read/Write | R | R/W | R/W | R | R | R | R | R | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

- **Bit 7 – Reserved**

This bit is reserved and will always read as zero.

- **Bit 6 – EXCLK: Enable External Clock Input**

When EXCLK is written to one, and asynchronous clock is selected, the external clock input buffer is enabled and an external clock can be input on Timer Oscillator 1 (TOSC1) pin instead of a 32kHz crystal. Writing to EXCLK should be done before asynchronous operation is selected. Note that the crystal Oscillator will only run when this bit is zero.

- **Bit 5 – AS2: Asynchronous Timer/Counter2**

When AS2 is written to zero, Timer/Counter2 is clocked from the I/O clock, $clk_{I/O}$. When AS2 is written to one, Timer/Counter2 is clocked from a crystal Oscillator connected to the Timer Oscillator 1 (TOSC1) pin. When the value of AS2 is changed, the contents of TCNT2, OCR2A, OCR2B, TCCR2A and TCCR2B might be corrupted.

- **Bit 4 – TCN2UB: Timer/Counter2 Update Busy**

When Timer/Counter2 operates asynchronously and TCNT2 is written, this bit becomes set. When TCNT2 has been updated from the temporary storage register, this bit is cleared by hardware. A logical zero in this bit indicates that TCNT2 is ready to be updated with a new value.

- **Bit 3 – OCR2AUB: Output Compare Register2 Update Busy**

When Timer/Counter2 operates asynchronously and OCR2A is written, this bit becomes set. When OCR2A has been updated from the temporary storage register, this bit is cleared by hardware. A logical zero in this bit indicates that OCR2A is ready to be updated with a new value.

- **Bit 2 – OCR2BUB: Output Compare Register2 Update Busy**

When Timer/Counter2 operates asynchronously and OCR2B is written, this bit becomes set. When OCR2B has been updated from the temporary storage register, this bit is cleared by hardware. A logical zero in this bit indicates that OCR2B is ready to be updated with a new value.

- **Bit 1 – TCR2AUB: Timer/Counter Control Register2 Update Busy**

When Timer/Counter2 operates asynchronously and TCCR2A is written, this bit becomes set. When TCCR2A has been updated from the temporary storage register, this bit is cleared by hardware. A logical zero in this bit indicates that TCCR2A is ready to be updated with a new value.

- **Bit 0 – TCR2BUB: Timer/Counter Control Register2 Update Busy**

When Timer/Counter2 operates asynchronously and TCCR2B is written, this bit becomes set. When TCCR2B has been updated from the temporary storage register, this bit is cleared by hardware. A logical zero in this bit indicates that TCCR2B is ready to be updated with a new value.

If a write is performed to any of the five Timer/Counter2 Registers while its update busy flag is set, the updated value might get corrupted and cause an unintentional interrupt to occur.

The mechanisms for reading TCNT2, OCR2A, OCR2B, TCCR2A and TCCR2B are different. When reading TCNT2, the actual timer value is read. When reading OCR2A, OCR2B, TCCR2A and TCCR2B the value in the temporary storage register is read.

### 18.11.9 GTCCR – General Timer/Counter Control Register

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| 0x23 (0x43) | TSM | – | – | – | – | – | PSRASY | PSRSYNC | GTCCR |
| Read/Write | R/W | R | R | R | R | R | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

- **Bit 1 – PSRASY: Prescaler Reset Timer/Counter2**

When this bit is one, the Timer/Counter2 prescaler will be reset. This bit is normally cleared immediately by hardware. If the bit is written when Timer/Counter2 is operating in asynchronous mode, the bit will remain one until the prescaler has been reset. The bit will not be cleared by hardware if the TSM bit is set. Refer to the description of the "Bit 7 – TSM: Timer/Counter Synchronization Mode" on page 142 for a description of the Timer/Counter Synchronization mode.

# 19. SPI – Serial Peripheral Interface

## 19.1 Features

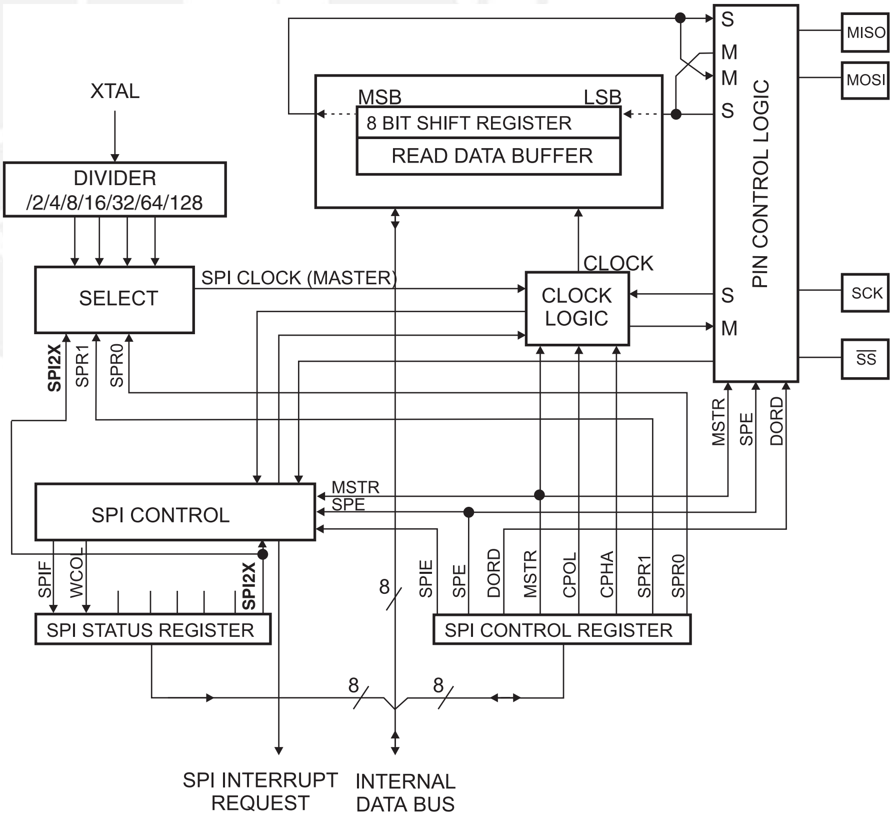
- **Full-duplex, Three-wire Synchronous Data Transfer**
- **Master or Slave Operation**
- **LSB First or MSB First Data Transfer**
- **Seven Programmable Bit Rates**
- **End of Transmission Interrupt Flag**
- **Write Collision Flag Protection**
- **Wake-up from Idle Mode**
- **Double Speed (CK/2) Master SPI Mode**

## 19.2 Overview

The Serial Peripheral Interface (SPI) allows high-speed synchronous data transfer between the Atmel® ATmega48PA/88PA/168PA and peripheral devices or between several AVR devices.

The USART can also be used in Master SPI mode, see “USART in SPI Mode” on page 202. The PRSPI bit in “Minimizing Power Consumption” on page 42 must be written to zero to enable SPI module.

**Figure 19-1.** SPI Block Diagram[1]



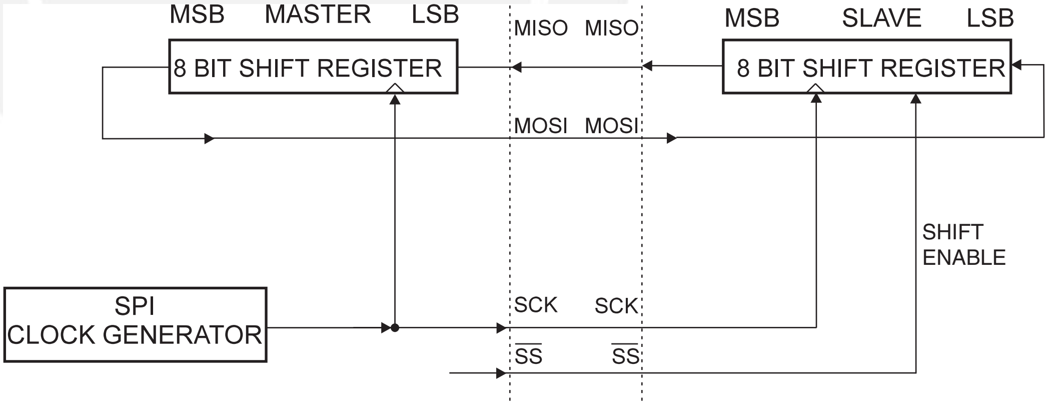
Note: 1. Refer to Figure 1-1 on page 2, and Table 14-3 on page 81 for SPI pin placement.

The interconnection between Master and Slave CPUs with SPI is shown in Figure 19-2 on page 167. The system consists of two shift Registers, and a Master clock generator. The SPI Master initiates the communication cycle when pulling low the Slave Select $\overline{\text{SS}}$ pin of the desired Slave. Master and Slave prepare the data to be sent in their respective shift Registers, and the Master generates the required clock pulses on the SCK line to interchange data. Data is always shifted from Master to Slave on the Master Out – Slave In, MOSI, line, and from Slave to Master on the Master In – Slave Out, MISO, line. After each data packet, the Master will synchronize the Slave by pulling high the Slave Select, $\overline{\text{SS}}$, line.

When configured as a Master, the SPI interface has no automatic control of the $\overline{\text{SS}}$ line. This must be handled by user software before communication can start. When this is done, writing a byte to the SPI Data Register starts the SPI clock generator, and the hardware shifts the eight bits into the Slave. After shifting one byte, the SPI clock generator stops, setting the end of Transmission Flag (SPIF). If the SPI Interrupt Enable bit (SPIE) in the SPCR Register is set, an interrupt is requested. The Master may continue to shift the next byte by writing it into SPDR, or signal the end of packet by pulling high the Slave Select, $\overline{\text{SS}}$ line. The last incoming byte will be kept in the Buffer Register for later use.

When configured as a Slave, the SPI interface will remain sleeping with MISO tri-stated as long as the $\overline{\text{SS}}$ pin is driven high. In this state, software may update the contents of the SPI Data Register, SPDR, but the data will not be shifted out by incoming clock pulses on the SCK pin until the $\overline{\text{SS}}$ pin is driven low. As one byte has been completely shifted, the end of Transmission Flag, SPIF is set. If the SPI Interrupt Enable bit, SPIE, in the SPCR Register is set, an interrupt is requested. The Slave may continue to place new data to be sent into SPDR before reading the incoming data. The last incoming byte will be kept in the Buffer Register for later use.

**Figure 19-2.** SPI Master-slave Interconnection



The system is single buffered in the transmit direction and double buffered in the receive direction. This means that bytes to be transmitted cannot be written to the SPI Data Register before the entire shift cycle is completed. When receiving data, however, a received character must be read from the SPI Data Register before the next character has been completely shifted in. Otherwise, the first byte is lost.

In SPI Slave mode, the control logic will sample the incoming signal of the SCK pin. To ensure correct sampling of the clock signal, the minimum low and high periods should be:

Low periods: Longer than 2 CPU clock cycles.

High periods: Longer than 2 CPU clock cycles.

When the SPI is enabled, the data direction of the MOSI, MISO, SCK, and $\overline{SS}$ pins is overridden according to Table 19-1 on page 168. For more details on automatic port overrides, refer to "Alternate Port Functions" on page 79.

**Table 19-1.** SPI Pin Overrides[1]

| Pin | Direction, Master SPI | Direction, Slave SPI |
|---|---|---|
| MOSI | User Defined | Input |
| MISO | Input | User Defined |
| SCK | User Defined | Input |
| $\overline{SS}$ | User Defined | Input |

Note: 1. See "Alternate Functions of Port B" on page 81 for a detailed description of how to define the direction of the user defined SPI pins.

The following code examples show how to initialize the SPI as a Master and how to perform a simple transmission. DDR_SPI in the examples must be replaced by the actual Data Direction Register controlling the SPI pins. DD_MOSI, DD_MISO and DD_SCK must be replaced by the actual data direction bits for these pins. E.g. if MOSI is placed on pin PB5, replace DD_MOSI with DDB5 and DDR_SPI with DDRB.

Assembly Code Example(1)

```
SPI_MasterInit:
  ; Set MOSI and SCK output, all others input
  ldi   r17,(1<<DD_MOSI)|(1<<DD_SCK)
  out   DDR_SPI,r17
  ; Enable SPI, Master, set clock rate fck/16
  ldi   r17,(1<<SPE)|(1<<MSTR)|(1<<SPR0)
  out   SPCR,r17
  ret

SPI_MasterTransmit:
  ; Start transmission of data (r16)
  out   SPDR,r16
Wait_Transmit:
  ; Wait for transmission complete
  in    r16, SPSR
  sbrsr16, SPIF
  rjmp Wait_Transmit
  ret
```

C Code Example(1)

```
void SPI_MasterInit(void)
{
  /* Set MOSI and SCK output, all others input */
  DDR_SPI = (1<<DD_MOSI)|(1<<DD_SCK);
  /* Enable SPI, Master, set clock rate fck/16 */
  SPCR = (1<<SPE)|(1<<MSTR)|(1<<SPR0);
}

void SPI_MasterTransmit(char cData)
{
  /* Start transmission */
  SPDR = cData;
  /* Wait for transmission complete */
  while(!(SPSR & (1<<SPIF)))
    ;
}
```

Note: 1. See ”About Code Examples” on page 7.

The following code examples show how to initialize the SPI as a Slave and how to perform a simple reception.

Assembly Code Example[1]

```
SPI_SlaveInit:
  ; Set MISO output, all others input
  ldi  r17,(1<<DD_MISO)
  out  DDR_SPI,r17
  ; Enable SPI
  ldi  r17,(1<<SPE)
  out  SPCR,r17
  ret

SPI_SlaveReceive:
  ; Wait for reception complete
  in r16, SPSR
  sbrs r16, SPIF
  rjmp SPI_SlaveReceive
  ; Read received data and return
  in   r16,SPDR
  ret
```

C Code Example[1]

```
void SPI_SlaveInit(void)
{
  /* Set MISO output, all others input */
  DDR_SPI = (1<<DD_MISO);
  /* Enable SPI */
  SPCR = (1<<SPE);
}

char SPI_SlaveReceive(void)
{
  /* Wait for reception complete */
  while(!(SPSR & (1<<SPIF)))
    ;
  /* Return Data Register */
  return SPDR;
}
```

Note: 1. See ”About Code Examples” on page 7.

## 19.3 $\overline{\text{SS}}$ Pin Functionality

### 19.3.1 Slave Mode

When the SPI is configured as a Slave, the Slave Select ($\overline{\text{SS}}$) pin is always input. When $\overline{\text{SS}}$ is held low, the SPI is activated, and MISO becomes an output if configured so by the user. All other pins are inputs. When $\overline{\text{SS}}$ is driven high, all pins are inputs, and the SPI is passive, which means that it will not receive incoming data. Note that the SPI logic will be reset once the $\overline{\text{SS}}$ pin is driven high.

The $\overline{\text{SS}}$ pin is useful for packet/byte synchronization to keep the slave bit counter synchronous with the master clock generator. When the $\overline{\text{SS}}$ pin is driven high, the SPI slave will immediately reset the send and receive logic, and drop any partially received data in the Shift Register.

### 19.3.2 Master Mode

When the SPI is configured as a Master (MSTR in SPCR is set), the user can determine the direction of the $\overline{\text{SS}}$ pin.

If $\overline{\text{SS}}$ is configured as an output, the pin is a general output pin which does not affect the SPI system. Typically, the pin will be driving the $\overline{\text{SS}}$ pin of the SPI Slave.

If $\overline{\text{SS}}$ is configured as an input, it must be held high to ensure Master SPI operation. If the $\overline{\text{SS}}$ pin is driven low by peripheral circuitry when the SPI is configured as a Master with the $\overline{\text{SS}}$ pin defined as an input, the SPI system interprets this as another master selecting the SPI as a slave and starting to send data to it. To avoid bus contention, the SPI system takes the following actions:

1. The MSTR bit in SPCR is cleared and the SPI system becomes a Slave. As a result of the SPI becoming a Slave, the MOSI and SCK pins become inputs.
2. The SPIF Flag in SPSR is set, and if the SPI interrupt is enabled, and the I-bit in SREG is set, the interrupt routine will be executed.

Thus, when interrupt-driven SPI transmission is used in Master mode, and there exists a possibility that $\overline{\text{SS}}$ is driven low, the interrupt should always check that the MSTR bit is still set. If the MSTR bit has been cleared by a slave select, it must be set by the user to re-enable SPI Master mode.

## 19.4 Data Modes

There are four combinations of SCK phase and polarity with respect to serial data, which are determined by control bits CPHA and CPOL. The SPI data transfer formats are shown in Figure 19-3 and Figure 19-4 on page 172. Data bits are shifted out and latched in on opposite edges of the SCK signal, ensuring sufficient time for data signals to stabilize. This is clearly seen by summarizing Table 19-3 on page 173 and Table 19-4 on page 173, as done in Table 19-2.

**Table 19-2.** SPI Modes

| SPI Mode | Conditions | Leading Edge | Trailing eDge |
|---|---|---|---|
| 0 | CPOL=0, CPHA=0 | Sample (Rising) | Setup (Falling) |
| 1 | CPOL=0, CPHA=1 | Setup (Rising) | Sample (Falling) |
| 2 | CPOL=1, CPHA=0 | Sample (Falling) | Setup (Rising) |
| 3 | CPOL=1, CPHA=1 | Setup (Falling) | Sample (Rising) |

**Figure 19-3.** SPI Transfer Format with CPHA = 0

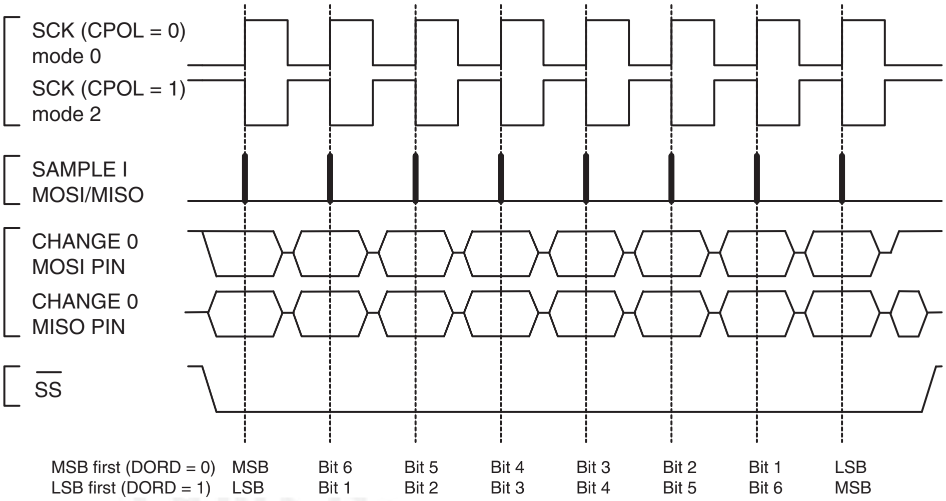
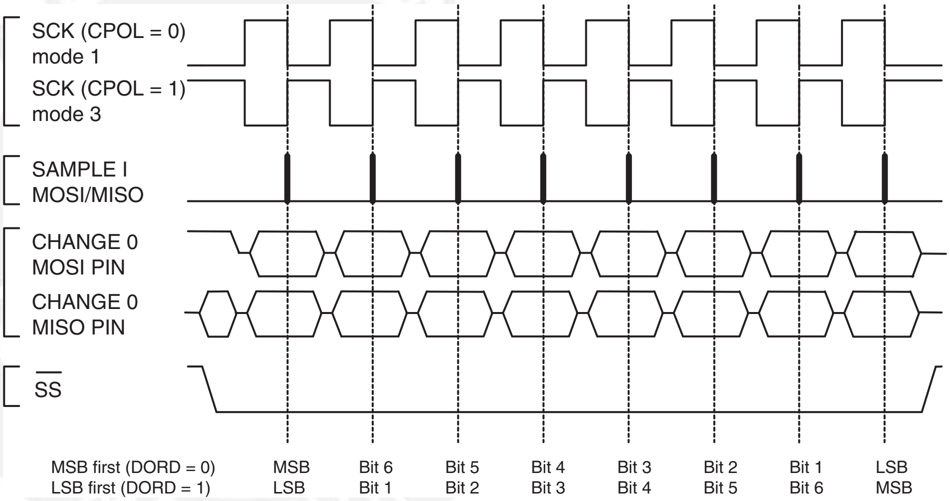


**Figure 19-4.** SPI Transfer Format with CPHA = 1



## 19.5 Register Description

### 19.5.1 SPCR – SPI Control Register

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| 0x2C (0x4C) | SPIE | SPE | DORD | MSTR | CPOL | CPHA | SPR1 | SPR0 | SPCR |
| Read/Write | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

- **Bit 7 – SPIE: SPI Interrupt Enable**

This bit causes the SPI interrupt to be executed if SPIF bit in the SPSR Register is set and the if the Global Interrupt Enable bit in SREG is set.

- **Bit 6 – SPE: SPI Enable**

When the SPE bit is written to one, the SPI is enabled. This bit must be set to enable any SPI operations.

- **Bit 5 – DORD: Data Order**

When the DORD bit is written to one, the LSB of the data word is transmitted first.

When the DORD bit is written to zero, the MSB of the data word is transmitted first.

- **Bit 4 – MSTR: Master/Slave Select**

This bit selects Master SPI mode when written to one, and Slave SPI mode when written logic zero. If $\overline{\text{SS}}$ is configured as an input and is driven low while MSTR is set, MSTR will be cleared, and SPIF in SPSR will become set. The user will then have to set MSTR to re-enable SPI Master mode.

- **Bit 3 – CPOL: Clock Polarity**

When this bit is written to one, SCK is high when idle. When CPOL is written to zero, SCK is low when idle. Refer to Figure 19-3 and Figure 19-4 for an example. The CPOL functionality is summarized below:

**Table 19-3.** CPOL Functionality

| CPOL | Leading Edge | Trailing Edge |
|---|---|---|
| 0 | Rising | Falling |
| 1 | Falling | Rising |

- **Bit 2 – CPHA: Clock Phase**

The settings of the Clock Phase bit (CPHA) determine if data is sampled on the leading (first) or trailing (last) edge of SCK. Refer to Figure 19-3 and Figure 19-4 for an example. The CPOL functionality is summarized below:

**Table 19-4.** CPHA Functionality

| CPHA | Leading Edge | Trailing Edge |
|---|---|---|
| 0 | Sample | Setup |
| 1 | Setup | Sample |

- **Bits 1, 0 – SPR1, SPR0: SPI Clock Rate Select 1 and 0**

These two bits control the SCK rate of the device configured as a Master. SPR1 and SPR0 have no effect on the Slave. The relationship between SCK and the Oscillator Clock frequency $f_{osc}$ is shown in the following table:

**Table 19-5.** Relationship Between SCK and the Oscillator Frequency

| SPI2X | SPR1 | SPR0 | SCK Frequency |
|---|---|---|---|
| 0 | 0 | 0 | $f_{osc}/4$ |
| 0 | 0 | 1 | $f_{osc}/16$ |
| 0 | 1 | 0 | $f_{osc}/64$ |
| 0 | 1 | 1 | $f_{osc}/128$ |
| 1 | 0 | 0 | $f_{osc}/2$ |
| 1 | 0 | 1 | $f_{osc}/8$ |
| 1 | 1 | 0 | $f_{osc}/32$ |
| 1 | 1 | 1 | $f_{osc}/64$ |

### 19.5.2 SPSR – SPI Status Register

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| 0x2D (0x4D) | SPIF | WCOL | – | – | – | – | – | SPI2X | SPSR |
| Read/Write | R | R | R | R | R | R | R | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

- **Bit 7 – SPIF: SPI Interrupt Flag**

When a serial transfer is complete, the SPIF Flag is set. An interrupt is generated if SPIE in SPCR is set and global interrupts are enabled. If $\overline{SS}$ is an input and is driven low when the SPI is in Master mode, this will also set the SPIF Flag. SPIF is cleared by hardware when executing the corresponding interrupt handling vector. Alternatively, the SPIF bit is cleared by first reading the SPI Status Register with SPIF set, then accessing the SPI Data Register (SPDR).

- **Bit 6 – WCOL: Write COLlision Flag**

The WCOL bit is set if the SPI Data Register (SPDR) is written during a data transfer. The WCOL bit (and the SPIF bit) are cleared by first reading the SPI Status Register with WCOL set, and then accessing the SPI Data Register.

- **Bit 5...1 – Reserved**

These bits are reserved bits in the Atmel® ATmega48PA/88PA/168PA and will always read as zero.

- **Bit 0 – SPI2X: Double SPI Speed Bit**

When this bit is written logic one the SPI speed (SCK Frequency) will be doubled when the SPI is in Master mode (see Table 19-5). This means that the minimum SCK period will be two CPU clock periods. When the SPI is configured as Slave, the SPI is only guaranteed to work at $f_{osc}/4$ or lower.

The SPI interface on the Atmel ATmega48PA/88PA/168PA is also used for program memory and EEPROM downloading or uploading. See page 308 for serial programming and verification.

### 19.5.3 SPDR – SPI Data Register

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| 0x2E (0x4E) | **MSB** | | | | | | | **LSB** | **SPDR** |
| Read/Write | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | |
| Initial Value | X | X | X | X | X | X | X | X | Undefined |

The SPI Data Register is a read/write register used for data transfer between the Register File and the SPI Shift Register. Writing to the register initiates data transmission. Reading the register causes the Shift Register Receive buffer to be read.

# 20. USART0

## 20.1 Features

- **Full Duplex Operation (Independent Serial Receive and Transmit Registers)**
- **Asynchronous or Synchronous Operation**
- **Master or Slave Clocked Synchronous Operation**
- **High Resolution Baud Rate Generator**
- **Supports Serial Frames with 5, 6, 7, 8, or 9 Data Bits and 1 or 2 Stop Bits**
- **Odd or Even Parity Generation and Parity Check Supported by Hardware**
- **Data OverRun Detection**
- **Framing Error Detection**
- **Noise Filtering Includes False Start Bit Detection and Digital Low Pass Filter**
- **Three Separate Interrupts on TX Complete, TX Data Register Empty and RX Complete**
- **Multi-processor Communication Mode**
- **Double Speed Asynchronous Communication Mode**

## 20.2 Overview

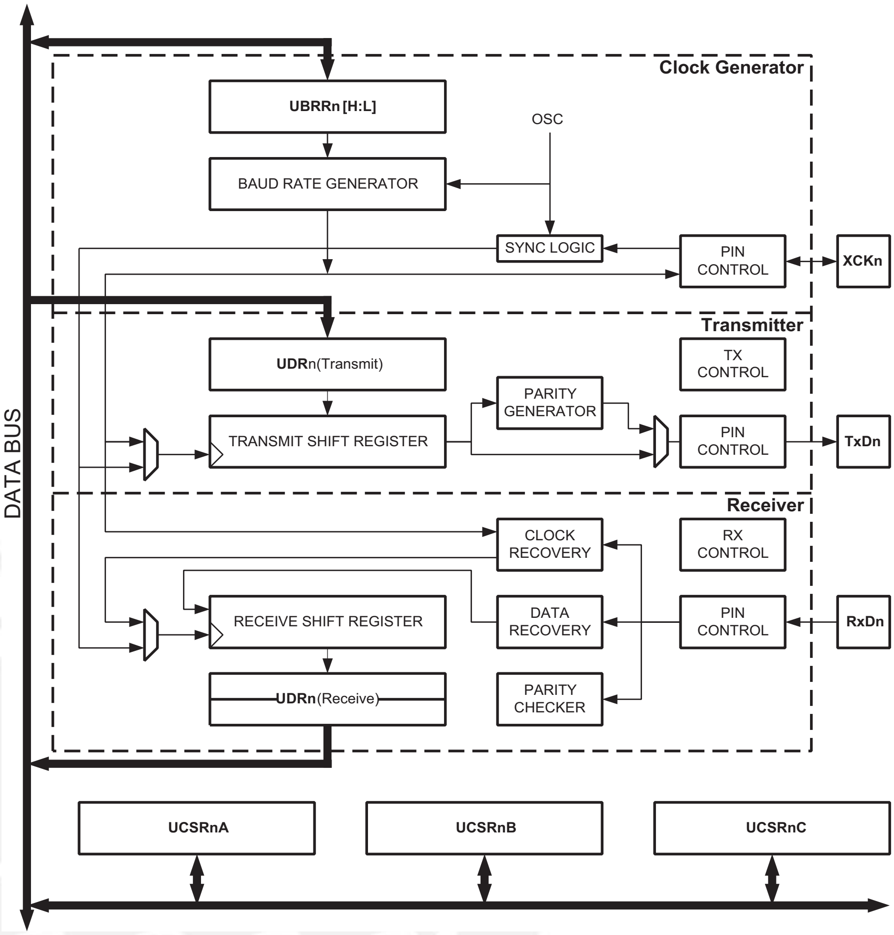
The Universal Synchronous and Asynchronous serial Receiver and Transmitter (USART) is a highly flexible serial communication device.

The USART0 can also be used in Master SPI mode, see "USART in SPI Mode" on page 202. The Power Reduction USART bit, PRUSART0, in "Minimizing Power Consumption" on page 42 must be disabled by writing a logical zero to it.

A simplified block diagram of the USART Transmitter is shown in Figure 20-1 on page 177. CPU accessible I/O Registers and I/O pins are shown in bold.

The dashed boxes in the block diagram separate the three main parts of the USART (listed from the top): Clock Generator, Transmitter and Receiver. Control Registers are shared by all units. The Clock Generation logic consists of synchronization logic for external clock input used by synchronous slave operation, and the baud rate generator. The XCKn (Transfer Clock) pin is only used by synchronous transfer mode. The Transmitter consists of a single write buffer, a serial Shift Register, Parity Generator and Control logic for handling different serial frame formats. The write buffer allows a continuous transfer of data without any delay between frames. The Receiver is the most complex part of the USART module due to its clock and data recovery units. The recovery units are used for asynchronous data reception. In addition to the recovery units, the Receiver includes a Parity Checker, Control logic, a Shift Register and a two level receive buffer (UDRn). The Receiver supports the same frame formats as the Transmitter, and can detect Frame Error, Data OverRun and Parity Errors.

**Figure 20-1.** USART Block Diagram[1]



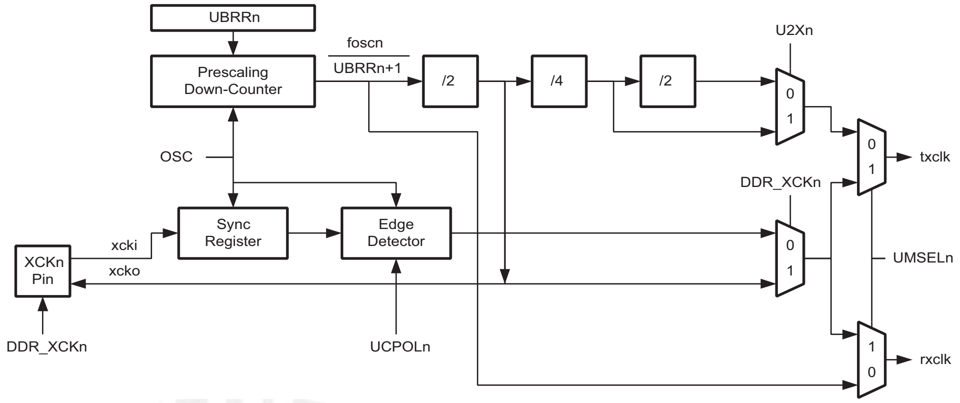
Note: 1. Refer to Figure 1-1 on page 2 and Table 14-9 on page 88 for USART0 pin placement.

## 20.3 Clock Generation

The Clock Generation logic generates the base clock for the Transmitter and Receiver. The USART supports four modes of clock operation: Normal asynchronous, Double Speed asynchronous, Master synchronous and Slave synchronous mode. The UMSELn bit in USART Control and Status Register C (UCSRnC) selects between asynchronous and synchronous operation. Double Speed (asynchronous mode only) is controlled by the U2Xn found in the UCSRnA Register. When using synchronous mode (UMSELn = 1), the Data Direction Register for the XCKn pin (DDR_XCKn) controls whether the clock source is internal (Master mode) or external (Slave mode). The XCKn pin is only active when using synchronous mode.

Figure 20-2 shows a block diagram of the clock generation logic.

**Figure 20-2.** Clock Generation Logic, Block Diagram



Signal description:

**txclk** Transmitter clock (Internal Signal).

**rxclk** Receiver base clock (Internal Signal).

**xcki** Input from XCK pin (internal Signal). Used for synchronous slave operation.

**xcko** Clock output to XCK pin (Internal Signal). Used for synchronous master operation.

**fosc** XTAL pin frequency (System Clock).

### 20.3.1 Internal Clock Generation – The Baud Rate Generator

Internal clock generation is used for the asynchronous and the synchronous master modes of operation. The description in this section refers to Figure 20-2.

The USART Baud Rate Register (UBRRn) and the down-counter connected to it function as a programmable prescaler or baud rate generator. The down-counter, running at system clock ($f_{osc}$), is loaded with the UBRRn value each time the counter has counted down to zero or when the UBRRnL Register is written. A clock is generated each time the counter reaches zero. This clock is the baud rate generator clock output (= $f_{osc}$/(UBRRn+1)). The Transmitter divides the baud rate generator clock output by 2, 8 or 16 depending on mode. The baud rate generator output is used directly by the Receiver’s clock and data recovery units. However, the recovery units use a state machine that uses 2, 8 or 16 states depending on mode set by the state of the UMSELn, U2Xn and DDR_XCKn bits.

Table 20-1 contains equations for calculating the baud rate (in bits per second) and for calculating the UBRRn value for each mode of operation using an internally generated clock source.

**Table 20-1.** Equations for Calculating Baud Rate Register Setting

| Operating Mode | Equation for Calculating Baud Rate[1] | Equation for Calculating UBRRn Value |
|---|---|---|
| Asynchronous Normal mode (U2Xn = 0) | $BAUD = \frac{f_{OSC}}{16(UBRRn + 1)}$ | $UBRRn = \frac{f_{OSC}}{16BAUD} - 1$ |
| Asynchronous Double Speed mode (U2Xn = 1) | $BAUD = \frac{f_{OSC}}{8(UBRRn + 1)}$ | $UBRRn = \frac{f_{OSC}}{8BAUD} - 1$ |
| Synchronous Master mode | $BAUD = \frac{f_{OSC}}{2(UBRRn + 1)}$ | $UBRRn = \frac{f_{OSC}}{2BAUD} - 1$ |

Note: 1. The baud rate is defined to be the transfer rate in bit per second (bps)

| | |
|---|---|
| **BAUD** | Baud rate (in bits per second, bps) |
| **$f_{OSC}$** | System Oscillator clock frequency |
| **UBRRn** | Contents of the UBRRnH and UBRRnL Registers, (0-4095) |

Some examples of UBRRn values for some system clock frequencies are found in Table 20-4 (see page 194).

### 20.3.2 Double Speed Operation (U2Xn)

The transfer rate can be doubled by setting the U2Xn bit in UCSRnA. Setting this bit only has effect for the asynchronous operation. Set this bit to zero when using synchronous operation.

Setting this bit will reduce the divisor of the baud rate divider from 16 to 8, effectively doubling the transfer rate for asynchronous communication. Note however that the Receiver will in this case only use half the number of samples (reduced from 16 to 8) for data sampling and clock recovery, and therefore a more accurate baud rate setting and system clock are required when this mode is used. For the Transmitter, there are no downsides.

### 20.3.3 External Clock

External clocking is used by the synchronous slave modes of operation. The description in this section refers to Figure 20-2 for details.

External clock input from the XCKn pin is sampled by a synchronization register to minimize the chance of meta-stability. The output from the synchronization register must then pass through an edge detector before it can be used by the Transmitter and Receiver. This process introduces a two CPU clock period delay and therefore the maximum external XCKn clock frequency is limited by the following equation:

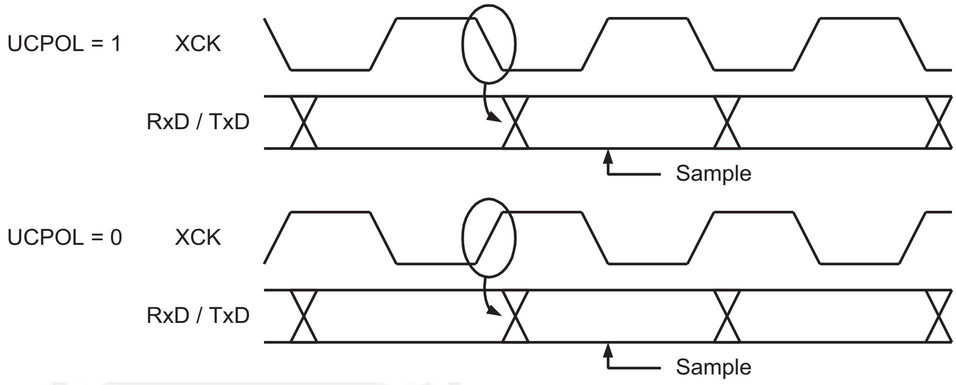
$$f_{XCK} < \frac{f_{OSC}}{4}$$

Note that $f_{osc}$ depends on the stability of the system clock source. It is therefore recommended to add some margin to avoid possible loss of data due to frequency variations.

### 20.3.4 Synchronous Clock Operation

When synchronous mode is used (UMSELn = 1), the XCKn pin will be used as either clock input (Slave) or clock output (Master). The dependency between the clock edges and data sampling or data change is the same. The basic principle is that data input (on RxDn) is sampled at the opposite XCKn clock edge of the edge the data output (TxDn) is changed.

**Figure 20-3.** Synchronous Mode XCKn Timing



The UCPOLn bit UCRSC selects which XCKn clock edge is used for data sampling and which is used for data change. As Figure 20-3 shows, when UCPOLn is zero the data will be changed at rising XCKn edge and sampled at falling XCKn edge. If UCPOLn is set, the data will be changed at falling XCKn edge and sampled at rising XCKn edge.

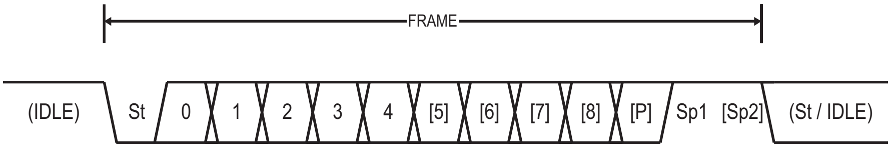
## 20.4 Frame Formats

A serial frame is defined to be one character of data bits with synchronization bits (start and stop bits), and optionally a parity bit for error checking. The USART accepts all 30 combinations of the following as valid frame formats:

- 1 start bit
- 5, 6, 7, 8, or 9 data bits
- no, even or odd parity bit
- 1 or 2 stop bits

A frame starts with the start bit followed by the least significant data bit. Then the next data bits, up to a total of nine, are succeeding, ending with the most significant bit. If enabled, the parity bit is inserted after the data bits, before the stop bits. When a complete frame is transmitted, it can be directly followed by a new frame, or the communication line can be set to an idle (high) state. Figure 20-4 illustrates the possible combinations of the frame formats. Bits inside brackets are optional.

**Figure 20-4.** Frame Formats



**St** Start bit, always low.

**(n)** Data bits (0 to 8).

**P** Parity bit. Can be odd or even.

**Sp** Stop bit, always high.

**IDLE** No transfers on the communication line (RxDn or TxDn). An IDLE line must be high.

The frame format used by the USART is set by the UCSZn2:0, UPMn1:0 and USBSn bits in UCSRnB and UCSRnC. The Receiver and Transmitter use the same setting. Note that changing the setting of any of these bits will corrupt all ongoing communication for both the Receiver and Transmitter.

The USART Character SiZe (UCSZn2:0) bits select the number of data bits in the frame. The USART Parity mode (UPMn1:0) bits enable and set the type of parity bit. The selection between one or two stop bits is done by the USART Stop Bit Select (USBSn) bit. The Receiver ignores the second stop bit. An FE (Frame Error) will therefore only be detected in the cases where the first stop bit is zero.

### 20.4.1 Parity Bit Calculation

The parity bit is calculated by doing an exclusive-or of all the data bits. If odd parity is used, the result of the exclusive or is inverted. The relation between the parity bit and data bits is as follows:

$$P_{even} = d_{n-1} \oplus \ldots \oplus d_3 \oplus d_2 \oplus d_1 \oplus d_0 \oplus 0$$
$$P_{odd} = d_{n-1} \oplus \ldots \oplus d_3 \oplus d_2 \oplus d_1 \oplus d_0 \oplus 1$$

**$P_{even}$** Parity bit using even parity

**$P_{odd}$** Parity bit using odd parity

**$d_n$** Data bit n of the character

If used, the parity bit is located between the last data bit and first stop bit of a serial frame.

## 20.5 USART Initialization

The USART has to be initialized before any communication can take place. The initialization process normally consists of setting the baud rate, setting frame format and enabling the Transmitter or the Receiver depending on the usage. For interrupt driven USART operation, the Global Interrupt Flag should be cleared (and interrupts globally disabled) when doing the initialization.

Before doing a re-initialization with changed baud rate or frame format, be sure that there are no ongoing transmissions during the period the registers are changed. The TXCn Flag can be used to check that the Transmitter has completed all transfers, and the RXC Flag can be used to check that there are no unread data in the receive buffer. Note that the TXCn Flag must be cleared before each transmission (before UDRn is written) if it is used for this purpose.

The following simple USART initialization code examples show one assembly and one C function that are equal in functionality. The examples assume asynchronous operation using polling (no interrupts enabled) and a fixed frame format. The baud rate is given as a function parameter. For the assembly code, the baud rate parameter is assumed to be stored in the r17:r16 Registers.

Assembly Code Example(1)

```
USART_Init:
  ; Set baud rate
  out  UBRRnH, r17
  out  UBRRnL, r16
  ; Enable receiver and transmitter
  ldi  r16, (1<<RXENn)|(1<<TXENn)
  out  UCSRnB,r16
  ; Set frame format: 8data, 2stop bit
  ldi  r16, (1<<USBSn)|(3<<UCSZn0)
  out  UCSRnC,r16
  ret
```

C Code Example(1)

```
#define FOSC 1843200 // Clock Speed
#define BAUD 9600
#define MYUBRR FOSC/16/BAUD-1
void main( void )
{
...
  USART_Init(MYUBRR)
...
}
void USART_Init( unsigned int ubrr)
{
  /*Set baud rate */
  UBRR0H = (unsigned char)(ubrr>>8);
  UBRR0L = (unsigned char)ubrr;
  Enable receiver and transmitter */
  UCSR0B = (1<<RXEN0)|(1<<TXEN0);
  /* Set frame format: 8data, 2stop bit */
  UCSR0C = (1<<USBS0)|(3<<UCSZ00);
}
```

Note: 1. See Section 6. “About Code Examples” on page 7.

More advanced initialization routines can be made that include frame format as parameters, disable interrupts and so on. However, many applications use a fixed setting of the baud and control registers, and for these types of applications the initialization code can be placed directly in the main routine, or be combined with initialization code for other I/O modules.

## 20.6 Data Transmission – The USART Transmitter

The USART Transmitter is enabled by setting the *Transmit Enable* (TXEN) bit in the UCSRnB Register. When the Transmitter is enabled, the normal port operation of the TxDn pin is overridden by the USART and given the function as the Transmitter's serial output. The baud rate, mode of operation and frame format must be set up once before doing any transmissions. If synchronous operation is used, the clock on the XCKn pin will be overridden and used as transmission clock.

### 20.6.1 Sending Frames with 5 to 8 Data Bit

A data transmission is initiated by loading the transmit buffer with the data to be transmitted. The CPU can load the transmit buffer by writing to the UDRn I/O location. The buffered data in the transmit buffer will be moved to the Shift Register when the Shift Register is ready to send a new frame. The Shift Register is loaded with new data if it is in idle state (no ongoing transmission) or immediately after the last stop bit of the previous frame is transmitted. When the Shift Register is loaded with new data, it will transfer one complete frame at the rate given by the Baud Register, U2Xn bit or by XCKn depending on mode of operation.

The following code examples show a simple USART transmit function based on polling of the *Data Register Empty* (UDREn) Flag. When using frames with less than eight bits, the most significant bits written to the UDRn are ignored. The USART has to be initialized before the function can be used. For the assembly code, the data to be sent is assumed to be stored in Register R16

Assembly Code Example[1]

```
USART_Transmit:
  ; Wait for empty transmit buffer
  in r16, UCSRnA
  sbrs r16, UDREn
  rjmp USART_Transmit
  ; Put data (r16) into buffer, sends the data
  out  UDRn,r16
  ret
```

C Code Example[1]

```
void USART_Transmit( unsigned char data )
{
  /* Wait for empty transmit buffer */
  while ( !( UCSRnA & (1<<UDREn)) )
        ;
  /* Put data into buffer, sends the data */
  UDRn = data;
}
```

Note: 1. See Section 6. "About Code Examples" on page 7.

The function simply waits for the transmit buffer to be empty by checking the UDREn Flag, before loading it with new data to be transmitted. If the Data Register Empty interrupt is utilized, the interrupt routine writes the data into the buffer.

### 20.6.2 Sending Frames with 9 Data Bit

If 9-bit characters are used (UCSZn = 7), the ninth bit must be written to the TXB8 bit in UCSRnB before the low byte of the character is written to UDRn. The following code examples show a transmit function that handles 9-bit characters. For the assembly code, the data to be sent is assumed to be stored in registers R17:R16.

Assembly Code Example[1][2]

```
USART_Transmit:
  ; Wait for empty transmit buffer
  in r16, UCSRnA
  sbrs r16, UDREn
  rjmp USART_Transmit
  ; Copy 9th bit from r17 to TXB8
  cbi  UCSRnB,TXB8
  sbrc r17,0
  sbi  UCSRnB,TXB8
  ; Put LSB data (r16) into buffer, sends the data
  out  UDRn,r16
  ret
```

C Code Example[1][2]

```
void USART_Transmit( unsigned int data )
{
  /* Wait for empty transmit buffer */
  while ( !( UCSRnA & (1<<UDREn))) )
        ;
  /* Copy 9th bit to TXB8 */
  UCSRnB &= ~(1<<TXB8);
  if ( data & 0x0100 )
    UCSRnB |= (1<<TXB8);
  /* Put data into buffer, sends the data */
  UDRn = data;
}
```

Notes:
1. These transmit functions are written to be general functions. They can be optimized if the contents of the UCSRnB is static. For example, only the TXB8 bit of the UCSRnB Register is used after initialization.
2. See ”About Code Examples” on page 7.

The ninth bit can be used for indicating an address frame when using multi processor communication mode or for other protocol handling as for example synchronization.

### 20.6.3 Transmitter Flags and Interrupts

The USART Transmitter has two flags that indicate its state: USART Data Register Empty (UDREn) and Transmit Complete (TXCn). Both flags can be used for generating interrupts.

The Data Register Empty (UDREn) Flag indicates whether the transmit buffer is ready to receive new data. This bit is set when the transmit buffer is empty, and cleared when the transmit buffer contains data to be transmitted that has not yet been moved into the Shift Register. For compatibility with future devices, always write this bit to zero when writing the UCSRnA Register.

When the Data Register Empty Interrupt Enable (UDRIEn) bit in UCSRnB is written to one, the USART Data Register Empty Interrupt will be executed as long as UDREn is set (provided that global interrupts are enabled). UDREn is cleared by writing UDRn. When interrupt-driven data transmission is used, the Data Register Empty interrupt routine must either write new data to UDRn in order to clear UDREn or disable the Data Register Empty interrupt, otherwise a new interrupt will occur once the interrupt routine terminates.

The Transmit Complete (TXCn) Flag bit is set one when the entire frame in the Transmit Shift Register has been shifted out and there are no new data currently present in the transmit buffer. The TXCn Flag bit is automatically cleared when a transmit complete interrupt is executed, or it can be cleared by writing a one to its bit location. The TXCn Flag is useful in half-duplex communication interfaces (like the RS-485 standard), where a transmitting application must enter receive mode and free the communication bus immediately after completing the transmission.

When the Transmit Compete Interrupt Enable (TXCIEn) bit in UCSRnB is set, the USART Transmit Complete Interrupt will be executed when the TXCn Flag becomes set (provided that global interrupts are enabled). When the transmit complete interrupt is used, the interrupt handling routine does not have to clear the TXCn Flag, this is done automatically when the interrupt is executed.

### 20.6.4 Parity Generator

The Parity Generator calculates the parity bit for the serial frame data. When parity bit is enabled (UPMn1 = 1), the transmitter control logic inserts the parity bit between the last data bit and the first stop bit of the frame that is sent.

### 20.6.5 Disabling the Transmitter

The disabling of the Transmitter (setting the TXEN to zero) will not become effective until ongoing and pending transmissions are completed, i.e., when the Transmit Shift Register and Transmit Buffer Register do not contain data to be transmitted. When disabled, the Transmitter will no longer override the TxDn pin.

## 20.7 Data Reception – The USART Receiver

The USART Receiver is enabled by writing the Receive Enable (RXENn) bit in the UCSRnB Register to one. When the Receiver is enabled, the normal pin operation of the RxDn pin is overridden by the USART and given the function as the Receiver's serial input. The baud rate, mode of operation and frame format must be set up once before any serial reception can be done. If synchronous operation is used, the clock on the XCKn pin will be used as transfer clock.

### 20.7.1 Receiving Frames with 5 to 8 Data Bits

The Receiver starts data reception when it detects a valid start bit. Each bit that follows the start bit will be sampled at the baud rate or XCKn clock, and shifted into the Receive Shift Register until the first stop bit of a frame is received. A second stop bit will be ignored by the Receiver. When the first stop bit is received, i.e., a complete serial frame is present in the Receive Shift Register, the contents of the Shift Register will be moved into the receive buffer. The receive buffer can then be read by reading the UDRn I/O location.

The following code example shows a simple USART receive function based on polling of the Receive Complete (RXCn) Flag. When using frames with less than eight bits the most significant bits of the data read from the UDRn will be masked to zero. The USART has to be initialized before the function can be used.

Assembly Code Example[1]

```
USART_Receive:
  ; Wait for data to be received
  in r16, UCSRnA
  sbrs r16, UDREn
  rjmp USART_Receive
  ; Get and return received data from buffer
  in   r16, UDRn
  ret
```

C Code Example[1]

```
unsigned char USART_Receive( void )
{
  /* Wait for data to be received */
  while ( !(UCSRnA & (1<<RXCn)) )
        ;
  /* Get and return received data from buffer */
  return UDRn;
}
```

Note: 1. See ”About Code Examples” on page 7.
For I/O Registers located in extended I/O map, “IN”, “OUT”, “SBIS”, “SBIC”, “CBI”, and “SBI” instructions must be replaced with instructions that allow access to extended I/O. Typically “LDS” and “STS” combined with “SBRS”, “SBRC”, “SBR”, and “CBR”.

The function simply waits for data to be present in the receive buffer by checking the RXCn Flag, before reading the buffer and returning the value.

### 20.7.2 Receiving Frames with 9 Data Bits

If 9-bit characters are used (UCSZn=7) the ninth bit must be read from the RXB8n bit in UCSRnB **before** reading the low bits from the UDRn. This rule applies to the FEn, DORn and UPEn Status Flags as well. Read status from UCSRnA, then data from UDRn. Reading the UDRn I/O location will change the state of the receive buffer FIFO and consequently the TXB8n, FEn, DORn and UPEn bits, which all are stored in the FIFO, will change.

The following code example shows a simple USART receive function that handles both nine bit characters and the status bits.

Assembly Code Example(1)

```
USART_Receive:
  ; Wait for data to be received
  in r16, UCSRnA
  sbrs r16, RXCn
  rjmp USART_Receive
  ; Get status and 9th bit, then data from buffer
  in   r18, UCSRnA
  in   r17, UCSRnB
  in   r16, UDRn
  ; If error, return -1
  andi r18,(1<<FEn)|(1<<DORn)|(1<<UPEn)
  breq USART_ReceiveNoError
  ldi  r17, HIGH(-1)
  ldi  r16, LOW(-1)
USART_ReceiveNoError:
  ; Filter the 9th bit, then return
  lsr  r17
  andi r17, 0x01
  ret
```

C Code Example(1)

```
unsigned int USART_Receive( void )
{
  unsigned char status, resh, resl;
  /* Wait for data to be received */
  while ( !(UCSRnA & (1<<RXCn)) )
        ;
  /* Get status and 9th bit, then data */
  /* from buffer */
  status = UCSRnA;
  resh = UCSRnB;
  resl = UDRn;
  /* If error, return -1 */
  if ( status & (1<<FEn)|(1<<DORn)|(1<<UPEn) )
    return -1;
  /* Filter the 9th bit, then return */
  resh = (resh >> 1) & 0x01;
  return ((resh << 8) | resl);
}
```

Note: 1. See Section 6. "About Code Examples" on page 7
For I/O Registers located in extended I/O map, "IN", "OUT", "SBIS", "SBIC", "CBI", and "SBI" instructions must be replaced with instructions that allow access to extended I/O. Typically "LDS" and "STS" combined with "SBRS", "SBRC", "SBR", and "CBR".

The receive function example reads all the I/O Registers into the Register File before any computation is done. This gives an optimal receive buffer utilization since the buffer location read will be free to accept new data as early as possible.

### 20.7.3 Receive Compete Flag and Interrupt

The USART Receiver has one flag that indicates the Receiver state.

The Receive Complete (RXCn) Flag indicates if there are unread data present in the receive buffer. This flag is one when unread data exist in the receive buffer, and zero when the receive buffer is empty (i.e., does not contain any unread data). If the Receiver is disabled (RXENn = 0), the receive buffer will be flushed and consequently the RXCn bit will become zero.

When the Receive Complete Interrupt Enable (RXCIEn) in UCSRnB is set, the USART Receive Complete interrupt will be executed as long as the RXCn Flag is set (provided that global interrupts are enabled). When interrupt-driven data reception is used, the receive complete routine must read the received data from UDRn in order to clear the RXCn Flag, otherwise a new interrupt will occur once the interrupt routine terminates.

### 20.7.4 Receiver Error Flags

The USART Receiver has three Error Flags: Frame Error (FEn), Data OverRun (DORn) and Parity Error (UPEn). All can be accessed by reading UCSRnA. Common for the Error Flags is that they are located in the receive buffer together with the frame for which they indicate the error status. Due to the buffering of the Error Flags, the UCSRnA must be read before the receive buffer (UDRn), since reading the UDRn I/O location changes the buffer read location. Another equality for the Error Flags is that they can not be altered by software doing a write to the flag location. However, all flags must be set to zero when the UCSRnA is written for upward compatibility of future USART implementations. None of the Error Flags can generate interrupts.

The Frame Error (FEn) Flag indicates the state of the first stop bit of the next readable frame stored in the receive buffer. The FEn Flag is zero when the stop bit was correctly read (as one), and the FEn Flag will be one when the stop bit was incorrect (zero). This flag can be used for detecting out-of-sync conditions, detecting break conditions and protocol handling. The FEn Flag is not affected by the setting of the USBSn bit in UCSRnC since the Receiver ignores all, except for the first, stop bits. For compatibility with future devices, always set this bit to zero when writing to UCSRnA.

The Data OverRun (DORn) Flag indicates data loss due to a receiver buffer full condition. A Data OverRun occurs when the receive buffer is full (two characters), it is a new character waiting in the Receive Shift Register, and a new start bit is detected. If the DORn Flag is set there was one or more serial frame lost between the frame last read from UDRn, and the next frame read from UDRn. For compatibility with future devices, always write this bit to zero when writing to UCSRnA. The DORn Flag is cleared when the frame received was successfully moved from the Shift Register to the receive buffer.

The Parity Error (UPEn) Flag indicates that the next frame in the receive buffer had a Parity Error when received. If Parity Check is not enabled the UPEn bit will always be read zero. For compatibility with future devices, always set this bit to zero when writing to UCSRnA. For more details see "Parity Bit Calculation" on page 181 and "Parity Checker" on page 189.

### 20.7.5 Parity Checker

The Parity Checker is active when the high USART Parity mode (UPMn1) bit is set. Type of Parity Check to be performed (odd or even) is selected by the UPMn0 bit. When enabled, the Parity Checker calculates the parity of the data bits in incoming frames and compares the result with the parity bit from the serial frame. The result of the check is stored in the receive buffer together with the received data and stop bits. The Parity Error (UPEn) Flag can then be read by software to check if the frame had a Parity Error.

The UPEn bit is set if the next character that can be read from the receive buffer had a Parity Error when received and the Parity Checking was enabled at that point (UPMn1 = 1). This bit is valid until the receive buffer (UDRn) is read.

### 20.7.6 Disabling the Receiver

In contrast to the Transmitter, disabling of the Receiver will be immediate. Data from ongoing receptions will therefore be lost. When disabled (i.e., the RXENn is set to zero) the Receiver will no longer override the normal function of the RxDn port pin. The Receiver buffer FIFO will be flushed when the Receiver is disabled. Remaining data in the buffer will be lost

### 20.7.7 Flushing the Receive Buffer

The receiver buffer FIFO will be flushed when the Receiver is disabled, i.e., the buffer will be emptied of its contents. Unread data will be lost. If the buffer has to be flushed during normal operation, due to for instance an error condition, read the UDRn I/O location until the RXCn Flag is cleared. The following code example shows how to flush the receive buffer.

Assembly Code Example(1)

```
USART_Flush:
  in r16, UCSRnA
  sbrs r16, RXCn
  ret
  in   r16, UDRn
  rjmp USART_Flush
```

C Code Example(1)

```
void USART_Flush( void )
{
  unsigned char dummy;
  while ( UCSRnA & (1<<RXCn) ) dummy = UDRn;
}
```

Note: 1. See Section 6. “About Code Examples” on page 7
For I/O Registers located in extended I/O map, “IN”, “OUT”, “SBIS”, “SBIC”, “CBI”, and “SBI” instructions must be replaced with instructions that allow access to extended I/O. Typically “LDS” and “STS” combined with “SBRS”, “SBRC”, “SBR”, and “CBR”.

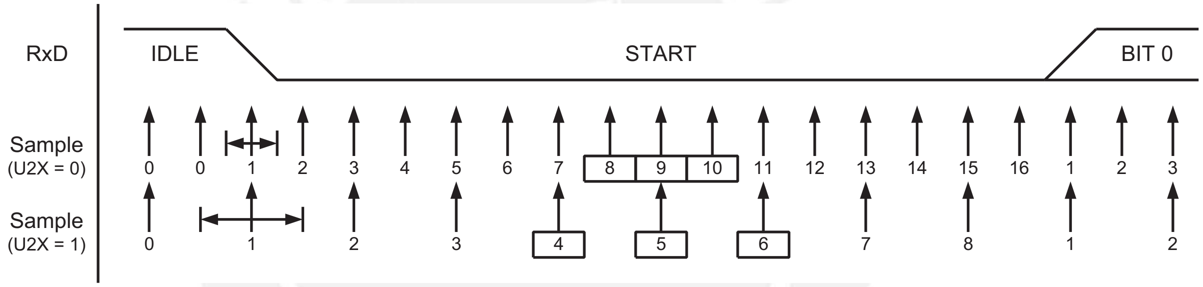
## 20.8 Asynchronous Data Reception

The USART includes a clock recovery and a data recovery unit for handling asynchronous data reception. The clock recovery logic is used for synchronizing the internally generated baud rate clock to the incoming asynchronous serial frames at the RxDn pin. The data recovery logic samples and low pass filters each incoming bit, thereby improving the noise immunity of the Receiver. The asynchronous reception operational range depends on the accuracy of the internal baud rate clock, the rate of the incoming frames, and the frame size in number of bits.

### 20.8.1 Asynchronous Clock Recovery

The clock recovery logic synchronizes internal clock to the incoming serial frames. Figure 20-5 illustrates the sampling process of the start bit of an incoming frame. The sample rate is 16 times the baud rate for Normal mode, and eight times the baud rate for Double Speed mode. The horizontal arrows illustrate the synchronization variation due to the sampling process. Note the larger time variation when using the Double Speed mode (U2Xn = 1) of operation. Samples denoted zero are samples done when the RxDn line is idle (i.e., no communication activity).

**Figure 20-5.** Start Bit Sampling

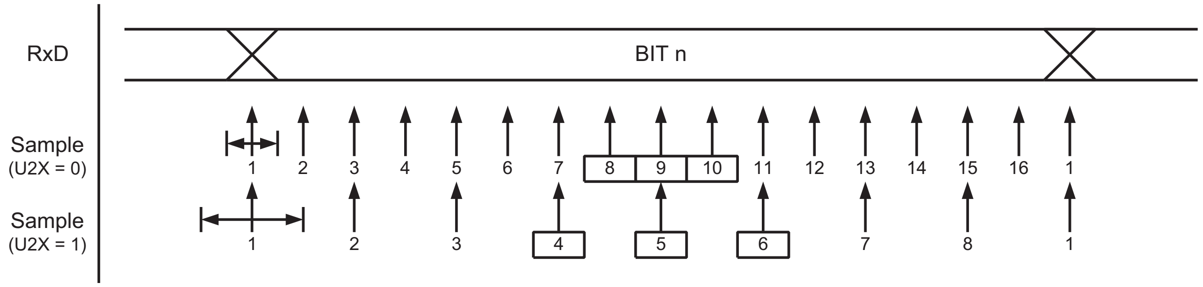


When the clock recovery logic detects a high (idle) to low (start) transition on the RxDn line, the start bit detection sequence is initiated. Let sample 1 denote the first zero-sample as shown in the figure. The clock recovery logic then uses samples 8, 9, and 10 for Normal mode, and samples 4, 5, and 6 for Double Speed mode (indicated with sample numbers inside boxes on the figure), to decide if a valid start bit is received. If two or more of these three samples have logical high levels (the majority wins), the start bit is rejected as a noise spike and the Receiver starts looking for the next high to low-transition. If however, a valid start bit is detected, the clock recovery logic is synchronized and the data recovery can begin. The synchronization process is repeated for each start bit.

### 20.8.2 Asynchronous Data Recovery

When the receiver clock is synchronized to the start bit, the data recovery can begin. The data recovery unit uses a state machine that has 16 states for each bit in Normal mode and eight states for each bit in Double Speed mode. Figure 20-6 shows the sampling of the data bits and the parity bit. Each of the samples is given a number that is equal to the state of the recovery unit.

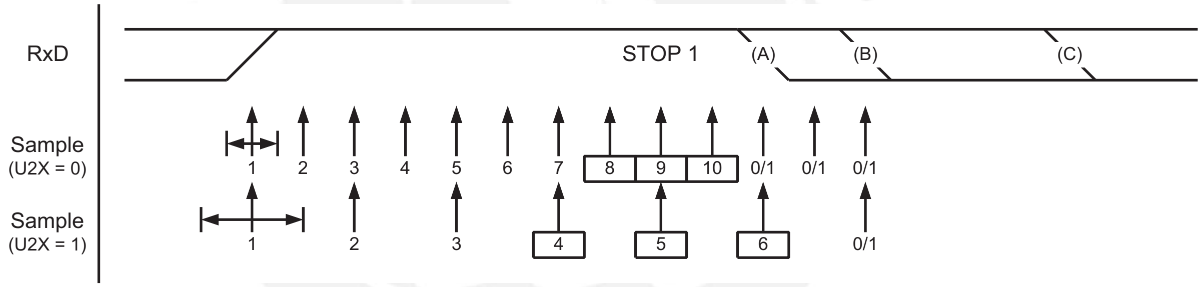
**Figure 20-6.** Sampling of Data and Parity Bit



The decision of the logic level of the received bit is taken by doing a majority voting of the logic value to the three samples in the center of the received bit. The center samples are emphasized on the figure by having the sample number inside boxes. The majority voting process is done as follows: If two or all three samples have high levels, the received bit is registered to be a logic 1. If two or all three samples have low levels, the received bit is registered to be a logic 0. This majority voting process acts as a low pass filter for the incoming signal on the RxDn pin. The recovery process is then repeated until a complete frame is received. Including the first stop bit. Note that the Receiver only uses the first stop bit of a frame.

Figure 20-7 on page 191 shows the sampling of the stop bit and the earliest possible beginning of the start bit of the next frame.

**Figure 20-7.** Stop Bit Sampling and Next Start Bit Sampling



The same majority voting is done to the stop bit as done for the other bits in the frame. If the stop bit is registered to have a logic 0 value, the Frame Error (FEn) Flag will be set.

A new high to low transition indicating the start bit of a new frame can come right after the last of the bits used for majority voting. For Normal Speed mode, the first low level sample can be at point marked (A) in Figure 20-7. For Double Speed mode the first low level must be delayed to (B). (C) marks a stop bit of full length. The early start bit detection influences the operational range of the Receiver.

### 20.8.3 Asynchronous Operational Range

The operational range of the Receiver is dependent on the mismatch between the received bit rate and the internally generated baud rate. If the Transmitter is sending frames at too fast or too slow bit rates, or the internally generated baud rate of the Receiver does not have a similar (see Table 20-2 on page 192) base frequency, the Receiver will not be able to synchronize the frames to the start bit.

The following equations can be used to calculate the ratio of the incoming data rate and internal receiver baud rate.

$$R_{slow} = \frac{(D+1)\times S}{S-1+D\times S+S_F} \qquad R_{fast} = \frac{(D+2)\times S}{(D+1)\times S+S_M}$$

**D** Sum of character size and parity size (D = 5 to 10 bit)

**S** Samples per bit. S = 16 for Normal Speed mode and S = 8 for Double Speed mode.

**$S_F$** First sample number used for majority voting. $S_F = 8$ for normal speed and $S_F = 4$ for Double Speed mode.

**$S_M$** Middle sample number used for majority voting. $S_M = 9$ for normal speed and $S_M = 5$ for Double Speed mode.

**$R_{slow}$** is the ratio of the slowest incoming data rate that can be accepted in relation to the receiver baud rate. $R_{fast}$ is the ratio of the fastest incoming data rate that can be accepted in relation to the receiver baud rate.

Table 20-2 on page 192 and Table 20-3 on page 192 list the maximum receiver baud rate error that can be tolerated. Note that Normal Speed mode has higher toleration of baud rate variations.

**Table 20-2.** Recommended Maximum Receiver Baud Rate Error for Normal Speed Mode (U2Xn = 0)

| D<br># (Data+Parity Bit) | $R_{slow}$ (%) | $R_{fast}$ (%) | Max Total Error (%) | Recommended Max Receiver Error (%) |
|---|---|---|---|---|
| 5 | 93.20 | 106.67 | +6.67/–6.8 | ±3.0 |
| 6 | 94.12 | 105.79 | +5.79/–5.88 | ±2.5 |
| 7 | 94.81 | 105.11 | +5.11/–5.19 | ±2.0 |
| 8 | 95.36 | 104.58 | +4.58/–4.54 | ±2.0 |
| 9 | 95.81 | 104.14 | +4.14/–4.19 | ±1.5 |
| 10 | 96.17 | 103.78 | +3.78/–3.83 | ±1.5 |

**Table 20-3.** Recommended Maximum Receiver Baud Rate Error for Double Speed Mode (U2Xn = 1)

| D<br># (Data+Parity Bit) | $R_{slow}$ (%) | $R_{fast}$ (%) | Max Total Error (%) | Recommended Max Receiver Error (%) |
|---|---|---|---|---|
| 5 | 94.12 | 105.66 | +5.66/–5.88 | ±2.5 |
| 6 | 94.92 | 104.92 | +4.92/–5.08 | ±2.0 |
| 7 | 95.52 | 104,35 | +4.35/–4.48 | ±1.5 |
| 8 | 96.00 | 103.90 | +3.90/–4.00 | ±1.5 |
| 9 | 96.39 | 103.53 | +3.53/–3.61 | ±1.5 |
| 10 | 96.70 | 103.23 | +3.23/–3.30 | ±1.0 |

The recommendations of the maximum receiver baud rate error was made under the assumption that the Receiver and Transmitter equally divides the maximum total error.

There are two possible sources for the receivers baud rate error. The Receiver's system clock (XTAL) will always have some minor instability over the supply voltage range and the temperature range. When using a crystal to generate the system clock, this is rarely a problem, but for a resonator the system clock may differ more than 2% depending of the resonators tolerance. The second source for the error is more controllable. The baud rate generator can not always do an exact division of the system frequency to get the baud rate wanted. In this case an UBRRn value that gives an acceptable low error can be used if possible.

## 20.9 Multi-processor Communication Mode

Setting the Multi-processor Communication mode (MPCMn) bit in UCSRnA enables a filtering function of incoming frames received by the USART Receiver. Frames that do not contain address information will be ignored and not put into the receive buffer. This effectively reduces the number of incoming frames that has to be handled by the CPU, in a system with multiple MCUs that communicate via the same serial bus. The Transmitter is unaffected by the MPCMn setting, but has to be used differently when it is a part of a system utilizing the Multi-processor Communication mode.

If the Receiver is set up to receive frames that contain 5 to 8 data bits, then the first stop bit indicates if the frame contains data or address information. If the Receiver is set up for frames with nine data bits, then the ninth bit (RXB8n) is used for identifying address and data frames. When the frame type bit (the first stop or the ninth bit) is one, the frame contains an address. When the frame type bit is zero the frame is a data frame.

The Multi-processor Communication mode enables several slave MCUs to receive data from a master MCU. This is done by first decoding an address frame to find out which MCU has been addressed. If a particular slave MCU has been addressed, it will receive the following data frames as normal, while the other slave MCUs will ignore the received frames until another address frame is received.

### 20.9.1 Using MPCMn

For an MCU to act as a master MCU, it can use a 9-bit character frame format (UCSZn = 7). The ninth bit (TXB8n) must be set when an address frame (TXB8n = 1) or cleared when a data frame (TXB = 0) is being transmitted. The slave MCUs must in this case be set to use a 9-bit character frame format.

The following procedure should be used to exchange data in Multi-processor Communication mode:

1. All Slave MCUs are in Multi-processor Communication mode (MPCMn in UCSRnA is set).
2. The Master MCU sends an address frame, and all slaves receive and read this frame. In the Slave MCUs, the RXCn Flag in UCSRnA will be set as normal.
3. Each Slave MCU reads the UDRn Register and determines if it has been selected. If so, it clears the MPCMn bit in UCSRnA, otherwise it waits for the next address byte and keeps the MPCMn setting.
4. The addressed MCU will receive all data frames until a new address frame is received. The other Slave MCUs, which still have the MPCMn bit set, will ignore the data frames.

5. When the last data frame is received by the addressed MCU, the addressed MCU sets the MPCMn bit and waits for a new address frame from master. The process then repeats from 2.

Using any of the 5- to 8-bit character frame formats is possible, but impractical since the Receiver must change between using n and n+1 character frame formats. This makes full-duplex operation difficult since the Transmitter and Receiver uses the same character size setting. If 5- to 8-bit character frames are used, the Transmitter must be set to use two stop bit (USBSn = 1) since the first stop bit is used for indicating the frame type.

Do not use Read-Modify-Write instructions (SBI and CBI) to set or clear the MPCMn bit. The MPCMn bit shares the same I/O location as the TXCn Flag and this might accidentally be cleared when using SBI or CBI instructions.

## 20.10 Examples of Baud Rate Setting

For standard crystal and resonator frequencies, the most commonly used baud rates for asynchronous operation can be generated by using the UBRRn settings in Table 20-4. UBRRn values which yield an actual baud rate differing less than 0.5% from the target baud rate, are bold in the table. Higher error ratings are acceptable, but the Receiver will have less noise resistance when the error ratings are high, especially for large serial frames (see "Asynchronous Operational Range" on page 191). The error values are calculated using the following equation:

$$\text{Error}[\%] = \left(\frac{\text{BaudRate}_{\text{Closest Match}}}{\text{BaudRate}} - 1\right) \times 100\%$$

**Table 20-4.** Examples of UBRRn Settings for Commonly Used Oscillator Frequencies

| Baud Rate (bps) | $f_{osc}$ = 1.0000MHz | | | | $f_{osc}$ = 1.8432MHz | | | | $f_{osc}$ = 2.0000MHz | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | U2Xn = 0 | | U2Xn = 1 | | U2Xn = 0 | | U2Xn = 1 | | U2Xn = 0 | | U2Xn = 1 | |
| | UBRRn | Error | UBRRn | Error | UBRRn | Error | UBRRn | Error | UBRRn | Error | UBRRn | Error |
| 2400 | 25 | 0.2% | 51 | 0.2% | 47 | 0.0% | 95 | 0.0% | 51 | 0.2% | 103 | 0.2% |
| 4800 | 12 | 0.2% | 25 | 0.2% | 23 | 0.0% | 47 | 0.0% | 25 | 0.2% | 51 | 0.2% |
| 9600 | 6 | -7.0% | 12 | 0.2% | 11 | 0.0% | 23 | 0.0% | 12 | 0.2% | 25 | 0.2% |
| 14.4k | 3 | 8.5% | 8 | -3.5% | 7 | 0.0% | 15 | 0.0% | 8 | -3.5% | 16 | 2.1% |
| 19.2k | 2 | 8.5% | 6 | -7.0% | 5 | 0.0% | 11 | 0.0% | 6 | -7.0% | 12 | 0.2% |
| 28.8k | 1 | 8.5% | 3 | 8.5% | 3 | 0.0% | 7 | 0.0% | 3 | 8.5% | 8 | -3.5% |
| 38.4k | 1 | -18.6% | 2 | 8.5% | 2 | 0.0% | 5 | 0.0% | 2 | 8.5% | 6 | -7.0% |
| 57.6k | 0 | 8.5% | 1 | 8.5% | 1 | 0.0% | 3 | 0.0% | 1 | 8.5% | 3 | 8.5% |
| 76.8k | – | – | 1 | -18.6% | 1 | -25.0% | 2 | 0.0% | 1 | -18.6% | 2 | 8.5% |
| 115.2k | – | – | 0 | 8.5% | 0 | 0.0% | 1 | 0.0% | 0 | 8.5% | 1 | 8.5% |
| 230.4k | – | – | – | – | – | – | 0 | 0.0% | – | – | – | – |
| 250k | – | – | – | – | – | – | – | – | – | – | 0 | 0.0% |
| Max.(1) | 62.5 kbps | | 125 kbps | | 115.2 kbps | | 230.4 kbps | | 125 kbps | | 250 kbps | |

Note: 1. UBRRn = 0, Error = 0.0%

**Table 20-5.** Examples of UBRRn Settings for Commonly Used Oscillator Frequencies (Continued)

| Baud Rate (bps) | $f_{osc}$ = 3.6864MHz | | | | $f_{osc}$ = 4.0000MHz | | | | $f_{osc}$ = 7.3728MHz | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | U2Xn = 0 | | U2Xn = 1 | | U2Xn = 0 | | U2Xn = 1 | | U2Xn = 0 | | U2Xn = 1 | |
| | UBRRn | Error | UBRRn | Error | UBRRn | Error | UBRRn | Error | UBRRn | Error | UBRRn | Error |
| 2400 | 95 | 0.0% | 191 | 0.0% | 103 | 0.2% | 207 | 0.2% | 191 | 0.0% | 383 | 0.0% |
| 4800 | 47 | 0.0% | 95 | 0.0% | 51 | 0.2% | 103 | 0.2% | 95 | 0.0% | 191 | 0.0% |
| 9600 | 23 | 0.0% | 47 | 0.0% | 25 | 0.2% | 51 | 0.2% | 47 | 0.0% | 95 | 0.0% |
| 14.4k | 15 | 0.0% | 31 | 0.0% | 16 | 2.1% | 34 | -0.8% | 31 | 0.0% | 63 | 0.0% |
| 19.2k | 11 | 0.0% | 23 | 0.0% | 12 | 0.2% | 25 | 0.2% | 23 | 0.0% | 47 | 0.0% |
| 28.8k | 7 | 0.0% | 15 | 0.0% | 8 | -3.5% | 16 | 2.1% | 15 | 0.0% | 31 | 0.0% |
| 38.4k | 5 | 0.0% | 11 | 0.0% | 6 | -7.0% | 12 | 0.2% | 11 | 0.0% | 23 | 0.0% |
| 57.6k | 3 | 0.0% | 7 | 0.0% | 3 | 8.5% | 8 | -3.5% | 7 | 0.0% | 15 | 0.0% |
| 76.8k | 2 | 0.0% | 5 | 0.0% | 2 | 8.5% | 6 | -7.0% | 5 | 0.0% | 11 | 0.0% |
| 115.2k | 1 | 0.0% | 3 | 0.0% | 1 | 8.5% | 3 | 8.5% | 3 | 0.0% | 7 | 0.0% |
| 230.4k | 0 | 0.0% | 1 | 0.0% | 0 | 8.5% | 1 | 8.5% | 1 | 0.0% | 3 | 0.0% |
| 250k | 0 | -7.8% | 1 | -7.8% | 0 | 0.0% | 1 | 0.0% | 1 | -7.8% | 3 | -7.8% |
| 0.5M | – | – | 0 | -7.8% | – | – | 0 | 0.0% | 0 | -7.8% | 1 | -7.8% |
| 1M | – | – | – | – | – | – | – | – | – | – | 0 | -7.8% |
| Max.[1] | 230.4kbps | | 460.8kbps | | 250kbps | | 0.5Mbps | | 460.8kbps | | 921.6kbps | |

Note: 1. UBRRn = 0, Error = 0.0%

**Table 20-6.** Examples of UBRRn Settings for Commonly Used Oscillator Frequencies (Continued)

| Baud Rate (bps) | $f_{osc}$ = 8.0000MHz | | | | $f_{osc}$ = 11.0592MHz | | | | $f_{osc}$ = 14.7456MHz | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | U2Xn = 0 | | U2Xn = 1 | | U2Xn = 0 | | U2Xn = 1 | | U2Xn = 0 | | U2Xn = 1 | |
| | UBRRn | Error | UBRRn | Error | UBRRn | Error | UBRRn | Error | UBRRn | Error | UBRRn | Error |
| 2400 | 207 | 0.2% | 416 | -0.1% | 287 | 0.0% | 575 | 0.0% | 383 | 0.0% | 767 | 0.0% |
| 4800 | 103 | 0.2% | 207 | 0.2% | 143 | 0.0% | 287 | 0.0% | 191 | 0.0% | 383 | 0.0% |
| 9600 | 51 | 0.2% | 103 | 0.2% | 71 | 0.0% | 143 | 0.0% | 95 | 0.0% | 191 | 0.0% |
| 14.4k | 34 | -0.8% | 68 | 0.6% | 47 | 0.0% | 95 | 0.0% | 63 | 0.0% | 127 | 0.0% |
| 19.2k | 25 | 0.2% | 51 | 0.2% | 35 | 0.0% | 71 | 0.0% | 47 | 0.0% | 95 | 0.0% |
| 28.8k | 16 | 2.1% | 34 | -0.8% | 23 | 0.0% | 47 | 0.0% | 31 | 0.0% | 63 | 0.0% |
| 38.4k | 12 | 0.2% | 25 | 0.2% | 17 | 0.0% | 35 | 0.0% | 23 | 0.0% | 47 | 0.0% |
| 57.6k | 8 | -3.5% | 16 | 2.1% | 11 | 0.0% | 23 | 0.0% | 15 | 0.0% | 31 | 0.0% |
| 76.8k | 6 | -7.0% | 12 | 0.2% | 8 | 0.0% | 17 | 0.0% | 11 | 0.0% | 23 | 0.0% |
| 115.2k | 3 | 8.5% | 8 | -3.5% | 5 | 0.0% | 11 | 0.0% | 7 | 0.0% | 15 | 0.0% |
| 230.4k | 1 | 8.5% | 3 | 8.5% | 2 | 0.0% | 5 | 0.0% | 3 | 0.0% | 7 | 0.0% |
| 250k | 1 | 0.0% | 3 | 0.0% | 2 | -7.8% | 5 | -7.8% | 3 | -7.8% | 6 | 5.3% |
| 0.5M | 0 | 0.0% | 1 | 0.0% | – | – | 2 | -7.8% | 1 | -7.8% | 3 | -7.8% |
| 1M | – | – | 0 | 0.0% | – | – | – | – | 0 | -7.8% | 1 | -7.8% |
| Max.[1] | 0.5Mbps | | 1Mbps | | 691.2kbps | | 1.3824Mbps | | 921.6kbps | | 1.8432Mbps | |

Note: 1. UBRRn = 0, Error = 0.0%

Table 20-7. Examples of UBRRn Settings for Commonly Used Oscillator Frequencies (Continued)

| Baud Rate (bps) | $f_{osc}$ = 16.0000MHz | | | |
|---|---|---|---|---|
| | U2Xn = 0 | | U2Xn = 1 | |
| | UBRRn | Error | UBRRn | Error |
| 2400 | 416 | -0.1% | 832 | 0.0% |
| 4800 | 207 | 0.2% | 416 | -0.1% |
| 9600 | 103 | 0.2% | 207 | 0.2% |
| 14.4k | 68 | 0.6% | 138 | -0.1% |
| 19.2k | 51 | 0.2% | 103 | 0.2% |
| 28.8k | 34 | -0.8% | 68 | 0.6% |
| 38.4k | 25 | 0.2% | 51 | 0.2% |
| 57.6k | 16 | 2.1% | 34 | -0.8% |
| 76.8k | 12 | 0.2% | 25 | 0.2% |
| 115.2k | 8 | -3.5% | 16 | 2.1% |
| 230.4k | 3 | 8.5% | 8 | -3.5% |
| 250k | 3 | 0.0% | 7 | 0.0% |
| 0.5M | 1 | 0.0% | 3 | 0.0% |
| 1M | 0 | 0.0% | 1 | 0.0% |
| Max.[1] | 1Mbps | | 2Mbps | |

Note: 1. UBRRn = 0, Error = 0.0%

## 20.11 Register Description

### 20.11.1 UDRn – USART I/O Data Register n

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| | RXB[7:0] | | | | | | | | UDRn (Read) |
| | TXB[7:0] | | | | | | | | UDRn (Write) |
| Read/Write | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

The USART Transmit Data Buffer Register and USART Receive Data Buffer Registers share the same I/O address referred to as USART Data Register or UDRn. The Transmit Data Buffer Register (TXB) will be the destination for data written to the UDRn Register location. Reading the UDRn Register location will return the contents of the Receive Data Buffer Register (RXB).

For 5-, 6-, or 7-bit characters the upper unused bits will be ignored by the Transmitter and set to zero by the Receiver.

The transmit buffer can only be written when the UDREn Flag in the UCSRnA Register is set. Data written to UDRn when the UDREn Flag is not set, will be ignored by the USART Transmitter. When data is written to the transmit buffer, and the Transmitter is enabled, the Transmitter will load the data into the Transmit Shift Register when the Shift Register is empty. Then the data will be serially transmitted on the TxDn pin.

The receive buffer consists of a two level FIFO. The FIFO will change its state whenever the receive buffer is accessed. Due to this behavior of the receive buffer, do not use Read-Modify-Write instructions (SBI and CBI) on this location. Be careful when using bit test instructions (SBIC and SBIS), since these also will change the state of the FIFO.

### 20.11.2 UCSRnA – USART Control and Status Register n A

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| | RXCn | TXCn | UDREn | FEn | DORn | UPEn | U2Xn | MPCMn | UCSRnA |
| Read/Write | R | R/W | R | R | R | R | R/W | R/W | |
| Initial Value | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | |

- **Bit 7 – RXCn: USART Receive Complete**

This flag bit is set when there are unread data in the receive buffer and cleared when the receive buffer is empty (i.e., does not contain any unread data). If the Receiver is disabled, the receive buffer will be flushed and consequently the RXCn bit will become zero. The RXCn Flag can be used to generate a Receive Complete interrupt (see description of the RXCIEn bit).

- **Bit 6 – TXCn: USART Transmit Complete**

This flag bit is set when the entire frame in the Transmit Shift Register has been shifted out and there are no new data currently present in the transmit buffer (UDRn). The TXCn Flag bit is automatically cleared when a transmit complete interrupt is executed, or it can be cleared by writing a one to its bit location. The TXCn Flag can generate a Transmit Complete interrupt (see description of the TXCIEn bit).

- **Bit 5 – UDREn: USART Data Register Empty**

The UDREn Flag indicates if the transmit buffer (UDRn) is ready to receive new data. If UDREn is one, the buffer is empty, and therefore ready to be written. The UDREn Flag can generate a Data Register Empty interrupt (see description of the UDRIEn bit). UDREn is set after a reset to indicate that the Transmitter is ready.

- **Bit 4 – FEn: Frame Error**

This bit is set if the next character in the receive buffer had a Frame Error when received. I.e., when the first stop bit of the next character in the receive buffer is zero. This bit is valid until the receive buffer (UDRn) is read. The FEn bit is zero when the stop bit of received data is one. Always set this bit to zero when writing to UCSRnA.

- **Bit 3 – DORn: Data OverRun**

This bit is set if a Data OverRun condition is detected. A Data OverRun occurs when the receive buffer is full (two characters), it is a new character waiting in the Receive Shift Register, and a new start bit is detected. This bit is valid until the receive buffer (UDRn) is read. Always set this bit to zero when writing to UCSRnA.

- **Bit 2 – UPEn: USART Parity Error**

This bit is set if the next character in the receive buffer had a Parity Error when received and the Parity Checking was enabled at that point (UPMn1 = 1). This bit is valid until the receive buffer (UDRn) is read. Always set this bit to zero when writing to UCSRnA.

- **Bit 1 – U2Xn: Double the USART Transmission Speed**

This bit only has effect for the asynchronous operation. Write this bit to zero when using synchronous operation.

Writing this bit to one will reduce the divisor of the baud rate divider from 16 to 8 effectively doubling the transfer rate for asynchronous communication.

- **Bit 0 – MPCMn: Multi-processor Communication Mode**

This bit enables the Multi-processor Communication mode. When the MPCMn bit is written to one, all the incoming frames received by the USART Receiver that do not contain address information will be ignored. The Transmitter is unaffected by the MPCMn setting. For more detailed information see "Multi-processor Communication Mode" on page 193.

### 20.11.3 UCSRnB – USART Control and Status Register n B

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| | RXCIEn | TXCIEn | UDRIEn | RXENn | TXENn | UCSZn2 | RXB8n | TXB8n | UCSRnB |
| Read/Write | R/W | R/W | R/W | R/W | R/W | R/W | R | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

- **Bit 7 – RXCIEn: RX Complete Interrupt Enable n**

Writing this bit to one enables interrupt on the RXCn Flag. A USART Receive Complete interrupt will be generated only if the RXCIEn bit is written to one, the Global Interrupt Flag in SREG is written to one and the RXCn bit in UCSRnA is set.

- **Bit 6 – TXCIEn: TX Complete Interrupt Enable n**

Writing this bit to one enables interrupt on the TXCn Flag. A USART Transmit Complete interrupt will be generated only if the TXCIEn bit is written to one, the Global Interrupt Flag in SREG is written to one and the TXCn bit in UCSRnA is set.

- **Bit 5 – UDRIEn: USART Data Register Empty Interrupt Enable n**

Writing this bit to one enables interrupt on the UDREn Flag. A Data Register Empty interrupt will be generated only if the UDRIEn bit is written to one, the Global Interrupt Flag in SREG is written to one and the UDREn bit in UCSRnA is set.

- **Bit 4 – RXENn: Receiver Enable n**

Writing this bit to one enables the USART Receiver. The Receiver will override normal port operation for the RxDn pin when enabled. Disabling the Receiver will flush the receive buffer invalidating the FEn, DORn, and UPEn Flags.

- **Bit 3 – TXENn: Transmitter Enable n**

Writing this bit to one enables the USART Transmitter. The Transmitter will override normal port operation for the TxDn pin when enabled. The disabling of the Transmitter (writing TXENn to zero) will not become effective until ongoing and pending transmissions are completed, i.e., when the Transmit Shift Register and Transmit Buffer Register do not contain data to be transmitted. When disabled, the Transmitter will no longer override the TxDn port.

- **Bit 2 – UCSZn2: Character Size n**

The UCSZn2 bits combined with the UCSZn1:0 bit in UCSRnC sets the number of data bits (Character SiZe) in a frame the Receiver and Transmitter use.

- **Bit 1 – RXB8n: Receive Data Bit 8 n**

RXB8n is the ninth data bit of the received character when operating with serial frames with nine data bits. Must be read before reading the low bits from UDRn.

- **Bit 0 – TXB8n: Transmit Data Bit 8 n**

TXB8n is the ninth data bit in the character to be transmitted when operating with serial frames with nine data bits. Must be written before writing the low bits to UDRn.

### 20.11.4 UCSRnC – USART Control and Status Register n C

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
| | UMSELn1 | UMSELn0 | UPMn1 | UPMn0 | USBSn | UCSZn1 | UCSZn0 | UCPOLn | UCSRnC |
| Read/Write | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | |

- **Bits 7:6 – UMSELn1:0 USART Mode Select**

These bits select the mode of operation of the USARTn as shown in Table 20-8.

**Table 20-8.** UMSELn Bits Settings

| UMSELn1 | UMSELn0 | Mode |
|---|---|---|
| 0 | 0 | Asynchronous USART |
| 0 | 1 | Synchronous USART |
| 1 | 0 | (Reserved) |
| 1 | 1 | Master SPI (MSPIM)[1] |

Note: 1. See "USART in SPI Mode" on page 202 for full description of the Master SPI Mode (MSPIM) operation

- **Bits 5:4 – UPMn1:0: Parity Mode**

These bits enable and set type of parity generation and check. If enabled, the Transmitter will automatically generate and send the parity of the transmitted data bits within each frame. The Receiver will generate a parity value for the incoming data and compare it to the UPMn setting. If a mismatch is detected, the UPEn Flag in UCSRnA will be set.

**Table 20-9.** UPMn Bits Settings

| UPMn1 | UPMn0 | Parity Mode |
|---|---|---|
| 0 | 0 | Disabled |
| 0 | 1 | Reserved |
| 1 | 0 | Enabled, Even Parity |
| 1 | 1 | Enabled, Odd Parity |

- **Bit 3 – USBSn: Stop Bit Select**

This bit selects the number of stop bits to be inserted by the Transmitter. The Receiver ignores this setting.

**Table 20-10.** USBS Bit Settings

| USBSn | Stop Bit(s) |
|---|---|
| 0 | 1-bit |
| 1 | 2-bit |

- **Bit 2:1 – UCSZn1:0: Character Size**

The UCSZn1:0 bits combined with the UCSZn2 bit in UCSRnB sets the number of data bits (Character SiZe) in a frame the Receiver and Transmitter use.

**Table 20-11.** UCSZn Bits Settings

| UCSZn2 | UCSZn1 | UCSZn0 | Character Size |
|---|---|---|---|
| 0 | 0 | 0 | 5-bit |
| 0 | 0 | 1 | 6-bit |
| 0 | 1 | 0 | 7-bit |
| 0 | 1 | 1 | 8-bit |
| 1 | 0 | 0 | Reserved |
| 1 | 0 | 1 | Reserved |
| 1 | 1 | 0 | Reserved |
| 1 | 1 | 1 | 9-bit |

- **Bit 0 – UCPOLn: Clock Polarity**

This bit is used for synchronous mode only. Write this bit to zero when asynchronous mode is used. The UCPOLn bit sets the relationship between data output change and data input sample, and the synchronous clock (XCKn).

**Table 20-12.** UCPOLn Bit Settings

| UCPOLn | Transmitted Data Changed (Output of TxDn Pin) | Received Data Sampled (Input on RxDn Pin) |
|---|---|---|
| 0 | Rising XCKn Edge | Falling XCKn Edge |
| 1 | Falling XCKn Edge | Rising XCKn Edge |

### 20.11.5 UBRRnL and UBRRnH – USART Baud Rate Registers

| Bit | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | |
|---|---|---|---|---|---|---|---|---|---|
| | – | – | – | – | UBRRn[11:8] | | | | UBRRnH |
| | UBRRn[7:0] | | | | | | | | UBRRnL |
| | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
| Read/Write | R | R | R | R | R/W | R/W | R/W | R/W | |
| | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

- **Bit 15:12 – Reserved**

These bits are reserved for future use. For compatibility with future devices, these bit must be written to zero when UBRRnH is written.

- **Bit 11:0 – UBRR[11:0]: USART Baud Rate Register**

This is a 12-bit register which contains the USART baud rate. The UBRRnH contains the four most significant bits, and the UBRRnL contains the eight least significant bits of the USART baud rate. Ongoing transmissions by the Transmitter and Receiver will be corrupted if the baud rate is changed. Writing UBRRnL will trigger an immediate update of the baud rate prescaler.

# 21. USART in SPI Mode

## 21.1 Features

- **Full Duplex, Three-wire Synchronous Data Transfer**
- **Master Operation**
- **Supports all four SPI Modes of Operation (Mode 0, 1, 2, and 3)**
- **LSB First or MSB First Data Transfer (Configurable Data Order)**
- **Queued Operation (Double Buffered)**
- **High Resolution Baud Rate Generator**
- **High Speed Operation ($f_{XCKmax} = f_{CK}/2$)**
- **Flexible Interrupt Generation**

## 21.2 Overview

The Universal Synchronous and Asynchronous serial Receiver and Transmitter (USART) can be set to a master SPI compliant mode of operation.

Setting both UMSELn1:0 bits to one enables the USART in MSPIM logic. In this mode of operation the SPI master control logic takes direct control over the USART resources. These resources include the transmitter and receiver shift register and buffers, and the baud rate generator. The parity generator and checker, the data and clock recovery logic, and the RX and TX control logic is disabled. The USART RX and TX control logic is replaced by a common SPI transfer control logic. However, the pin control logic and interrupt generation logic is identical in both modes of operation.

The I/O register locations are the same in both modes. However, some of the functionality of the control registers changes when using MSPIM.

## 21.3 Clock Generation

The Clock Generation logic generates the base clock for the Transmitter and Receiver. For USART MSPIM mode of operation only internal clock generation (i.e. master operation) is supported. The Data Direction Register for the XCKn pin (DDR_XCKn) must therefore be set to one (i.e. as output) for the USART in MSPIM to operate correctly. Preferably the DDR_XCKn should be set up before the USART in MSPIM is enabled (i.e. TXENn and RXENn bit set to one).

The internal clock generation used in MSPIM mode is identical to the USART synchronous master mode. The baud rate or UBRRn setting can therefore be calculated using the same equations, see Table 21-1.

**Table 21-1.** Equations for Calculating Baud Rate Register Setting

| Operating Mode | Equation for Calculating Baud Rate[(1)] | Equation for Calculating UBRRn Value |
|---|---|---|
| Synchronous Master mode | $BAUD = \dfrac{f_{OSC}}{2(\boldsymbol{UBRR}n + 1)}$ | $\boldsymbol{UBRR}n = \dfrac{f_{OSC}}{2BAUD} - 1$ |

Note: 1. The baud rate is defined to be the transfer rate in bit per second (bps)

| | |
|---|---|
| **BAUD** | Baud rate (in bits per second, bps) |
| **$f_{OSC}$** | System Oscillator clock frequency |
| **UBRRn** | Contents of the UBRRnH and UBRRnL Registers, (0-4095) |

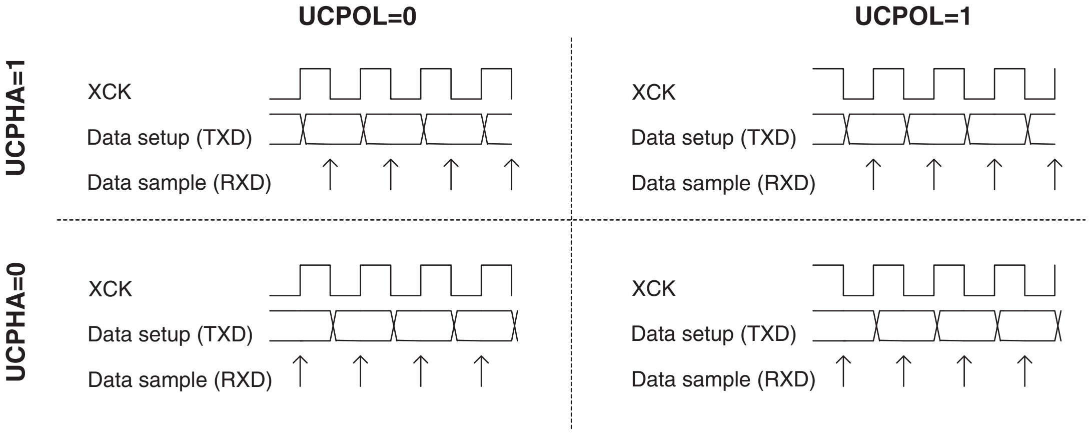
## 21.4 SPI Data Modes and Timing

There are four combinations of XCKn (SCK) phase and polarity with respect to serial data, which are determined by control bits UCPHAn and UCPOLn. The data transfer timing diagrams are shown in Figure 21-1. Data bits are shifted out and latched in on opposite edges of the XCKn signal, ensuring sufficient time for data signals to stabilize. The UCPOLn and UCPHAn functionality is summarized in Table 21-2. Note that changing the setting of any of these bits will corrupt all ongoing communication for both the Receiver and Transmitter.

**Table 21-2.** UCPOLn and UCPHAn Functionality-

| UCPOLn | UCPHAn | SPI Mode | Leading Edge | Trailing Edge |
|---|---|---|---|---|
| 0 | 0 | 0 | Sample (Rising) | Setup (Falling) |
| 0 | 1 | 1 | Setup (Rising) | Sample (Falling) |
| 1 | 0 | 2 | Sample (Falling) | Setup (Rising) |
| 1 | 1 | 3 | Setup (Falling) | Sample (Rising) |

**Figure 21-1.** UCPHAn and UCPOLn data transfer timing diagrams



## 21.5 Frame Formats

A serial frame for the MSPIM is defined to be one character of 8 data bits. The USART in MSPIM mode has two valid frame formats:

- 8-bit data with MSB first
- 8-bit data with LSB first

A frame starts with the least or most significant data bit. Then the next data bits, up to a total of eight, are succeeding, ending with the most or least significant bit accordingly. When a complete frame is transmitted, a new frame can directly follow it, or the communication line can be set to an idle (high) state.

The UDORDn bit in UCSRnC sets the frame format used by the USART in MSPIM mode. The Receiver and Transmitter use the same setting. Note that changing the setting of any of these bits will corrupt all ongoing communication for both the Receiver and Transmitter.

16-bit data transfer can be achieved by writing two data bytes to UDRn. A UART transmit complete interrupt will then signal that the 16-bit value has been shifted out.

### 21.5.1 USART MSPIM Initialization

The USART in MSPIM mode has to be initialized before any communication can take place. The initialization process normally consists of setting the baud rate, setting master mode of operation (by setting DDR_XCKn to one), setting frame format and enabling the Transmitter and the Receiver. Only the transmitter can operate independently. For interrupt driven USART operation, the Global Interrupt Flag should be cleared (and thus interrupts globally disabled) when doing the initialization.

Note: To ensure immediate initialization of the XCKn output the baud-rate register (UBRRn) must be zero at the time the transmitter is enabled. Contrary to the normal mode USART operation the UBRRn must then be written to the desired value after the transmitter is enabled, but before the first transmission is started. Setting UBRRn to zero before enabling the transmitter is not necessary if the initialization is done immediately after a reset since UBRRn is reset to zero.

Before doing a re-initialization with changed baud rate, data mode, or frame format, be sure that there is no ongoing transmissions during the period the registers are changed. The TXCn Flag can be used to check that the Transmitter has completed all transfers, and the RXCn Flag can be used to check that there are no unread data in the receive buffer. Note that the TXCn Flag must be cleared before each transmission (before UDRn is written) if it is used for this purpose.

The following simple USART initialization code examples show one assembly and one C function that are equal in functionality. The examples assume polling (no interrupts enabled). The baud rate is given as a function parameter. For the assembly code, the baud rate parameter is assumed to be stored in the r17:r16 registers.

Assembly Code Example[1]

```
USART_Init:
  clr r18
  out UBRRnH,r18
  out UBRRnL,r18
  ; Setting the XCKn port pin as output, enables master mode.
  sbi XCKn_DDR, XCKn
  ; Set MSPI mode of operation and SPI data mode 0.
  ldi r18, (1<<UMSELn1)|(1<<UMSELn0)|(0<<UCPHAn)|(0<<UCPOLn)
  out UCSRnC,r18
  ; Enable receiver and transmitter.
  ldi r18, (1<<RXENn)|(1<<TXENn)
  out UCSRnB,r18
  ; Set baud rate.
  ; IMPORTANT: The Baud Rate must be set after the transmitter is
enabled!
  out UBRRnH, r17
  out UBRRnL, r18
  ret
```

C Code Example[1]

```
void USART_Init( unsigned int baud )
{
  UBRRn = 0;
  /* Setting the XCKn port pin as output, enables master mode. */
  XCKn_DDR |= (1<<XCKn);
  /* Set MSPI mode of operation and SPI data mode 0. */
  UCSRnC = (1<<UMSELn1)|(1<<UMSELn0)|(0<<UCPHAn)|(0<<UCPOLn);
  /* Enable receiver and transmitter. */
  UCSRnB = (1<<RXENn)|(1<<TXENn);
  /* Set baud rate. */
  /* IMPORTANT: The Baud Rate must be set after the transmitter is
enabled */
  UBRRn = baud;
}
```

Note: 1. See Section 6. "About Code Examples" on page 7.

## 21.6 Data Transfer

Using the USART in MSPI mode requires the Transmitter to be enabled, i.e. the TXENn bit in the UCSRnB register is set to one. When the Transmitter is enabled, the normal port operation of the TxDn pin is overridden and given the function as the Transmitter's serial output. Enabling the receiver is optional and is done by setting the RXENn bit in the UCSRnB register to one. When the receiver is enabled, the normal pin operation of the RxDn pin is overridden and given the function as the Receiver's serial input. The XCKn will in both cases be used as the transfer clock.

After initialization the USART is ready for doing data transfers. A data transfer is initiated by writing to the UDRn I/O location. This is the case for both sending and receiving data since the transmitter controls the transfer clock. The data written to UDRn is moved from the transmit buffer to the shift register when the shift register is ready to send a new frame.

Note: To keep the input buffer in sync with the number of data bytes transmitted, the UDRn register must be read once for each byte transmitted. The input buffer operation is identical to normal USART mode, i.e. if an overflow occurs the character last received will be lost, not the first data in the buffer. This means that if four bytes are transferred, byte 1 first, then byte 2, 3, and 4, and the UDRn is not read before all transfers are completed, then byte 3 to be received will be lost, and not byte 1.

The following code examples show a simple USART in MSPIM mode transfer function based on polling of the Data Register Empty (UDREn) Flag and the Receive Complete (RXCn) Flag. The USART has to be initialized before the function can be used. For the assembly code, the data to be sent is assumed to be stored in Register R16 and the data received will be available in the same register (R16) after the function returns.

The function simply waits for the transmit buffer to be empty by checking the UDREn Flag, before loading it with new data to be transmitted. The function then waits for data to be present in the receive buffer by checking the RXCn Flag, before reading the buffer and returning the value.

Assembly Code Example[1]

```
USART_MSPIM_Transfer:
  ; Wait for empty transmit buffer
  in r16, UCSRnA
  sbrs r16, UDREn
  rjmp USART_MSPIM_Transfer
  ; Put data (r16) into buffer, sends the data
  out UDRn,r16
  ; Wait for data to be received
USART_MSPIM_Wait_RXCn:
  in r16, UCSRnA
  sbrs r16, RXCn
  rjmp USART_MSPIM_Wait_RXCn
  ; Get and return received data from buffer
  in r16, UDRn
  ret
```

C Code Example[1]

```
unsigned char USART_Receive( void )
{
  /* Wait for empty transmit buffer */
  while ( !( UCSRnA & (1<<UDREn)) );
  /* Put data into buffer, sends the data */
  UDRn = data;
  /* Wait for data to be received */
  while ( !(UCSRnA & (1<<RXCn)) );
  /* Get and return received data from buffer */
  return UDRn;
}
```

Note: 1. See ”About Code Examples” on page 7.

### 21.6.1 Transmitter and Receiver Flags and Interrupts

The RXCn, TXCn, and UDREn flags and corresponding interrupts in USART in MSPIM mode are identical in function to the normal USART operation. However, the receiver error status flags (FE, DOR, and PE) are not in use and is always read as zero.

### 21.6.2 Disabling the Transmitter or Receiver

The disabling of the transmitter or receiver in USART in MSPIM mode is identical in function to the normal USART operation.

## 21.7 AVR USART MSPIM versus AVR SPI

The USART in MSPIM mode is fully compatible with the AVR SPI regarding:

- Master mode timing diagram.
- The UCPOLn bit functionality is identical to the SPI CPOL bit.
- The UCPHAn bit functionality is identical to the SPI CPHA bit.
- The UDORDn bit functionality is identical to the SPI DORD bit.

However, since the USART in MSPIM mode reuses the USART resources, the use of the USART in MSPIM mode is somewhat different compared to the SPI. In addition to differences of the control register bits, and that only master operation is supported by the USART in MSPIM mode, the following features differ between the two modules:

- The USART in MSPIM mode includes (double) buffering of the transmitter. The SPI has no buffer.
- The USART in MSPIM mode receiver includes an additional buffer level.
- The SPI WCOL (Write Collision) bit is not included in USART in MSPIM mode.
- The SPI double speed mode (SPI2X) bit is not included. However, the same effect is achieved by setting UBRRn accordingly.
- Interrupt timing is not compatible.
- Pin control differs due to the master only operation of the USART in MSPIM mode.

A comparison of the USART in MSPIM mode and the SPI pins is shown in Table 21-3 on page 208.

**Table 21-3.** Comparison of USART in MSPIM mode and SPI pins.

| USART_MSPIM | SPI | Comment |
|---|---|---|
| TxDn | MOSI | Master Out only |
| RxDn | MISO | Master In only |
| XCKn | SCK | (Functionally identical) |
| (N/A) | $\overline{\text{SS}}$ | Not supported by USART in MSPIM |

## 21.8 Register Description

The following section describes the registers used for SPI operation using the USART.

### 21.8.1 UDRn – USART MSPIM I/O Data Register

The function and bit description of the USART data register (UDRn) in MSPI mode is identical to normal USART operation. See “UDRn – USART I/O Data Register n” on page 196.

### 21.8.2 UCSRnA – USART MSPIM Control and Status Register n A

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
| | RXCn | TXCn | UDREn | – | – | – | – | – | UCSRnA |
| Read/Write | R | R/W | R | R | R | R | R | R | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | |

- **Bit 7 – RXCn: USART Receive Complete**

This flag bit is set when there are unread data in the receive buffer and cleared when the receive buffer is empty (i.e., does not contain any unread data). If the Receiver is disabled, the receive buffer will be flushed and consequently the RXCn bit will become zero. The RXCn Flag can be used to generate a Receive Complete interrupt (see description of the RXCIEn bit).

- **Bit 6 – TXCn: USART Transmit Complete**

This flag bit is set when the entire frame in the Transmit Shift Register has been shifted out and there are no new data currently present in the transmit buffer (UDRn). The TXCn Flag bit is automatically cleared when a transmit complete interrupt is executed, or it can be cleared by writing a one to its bit location. The TXCn Flag can generate a Transmit Complete interrupt (see description of the TXCIEn bit).

- **Bit 5 – UDREn: USART Data Register Empty**

The UDREn Flag indicates if the transmit buffer (UDRn) is ready to receive new data. If UDREn is one, the buffer is empty, and therefore ready to be written. The UDREn Flag can generate a Data Register Empty interrupt (see description of the UDRIE bit). UDREn is set after a reset to indicate that the Transmitter is ready.

- **Bit 4:0 – Reserved Bits in MSPI mode**

When in MSPI mode, these bits are reserved for future use. For compatibility with future devices, these bits must be written to zero when UCSRnA is written.

### 21.8.3 UCSRnB – USART MSPIM Control and Status Register n B

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
| | RXCIEn | TXCIEn | UDRIE | RXENn | TXENn | – | - | - | UCSRnB |
| Read/Write | R/W | R/W | R/W | R/W | R/W | R | R | R | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | |

- **Bit 7 – RXCIEn: RX Complete Interrupt Enable**

Writing this bit to one enables interrupt on the RXCn Flag. A USART Receive Complete interrupt will be generated only if the RXCIEn bit is written to one, the Global Interrupt Flag in SREG is written to one and the RXCn bit in UCSRnA is set.

- **Bit 6 – TXCIEn: TX Complete Interrupt Enable**

Writing this bit to one enables interrupt on the TXCn Flag. A USART Transmit Complete interrupt will be generated only if the TXCIEn bit is written to one, the Global Interrupt Flag in SREG is written to one and the TXCn bit in UCSRnA is set.

- **Bit 5 – UDRIE: USART Data Register Empty Interrupt Enable**

Writing this bit to one enables interrupt on the UDREn Flag. A Data Register Empty interrupt will be generated only if the UDRIE bit is written to one, the Global Interrupt Flag in SREG is written to one and the UDREn bit in UCSRnA is set.

- **Bit 4 – RXENn: Receiver Enable**

Writing this bit to one enables the USART Receiver in MSPIM mode. The Receiver will override normal port operation for the RxDn pin when enabled. Disabling the Receiver will flush the receive buffer. Only enabling the receiver in MSPI mode (i.e. setting RXENn=1 and TXENn=0) has no meaning since it is the transmitter that controls the transfer clock and since only master mode is supported.

- **Bit 3 – TXENn: Transmitter Enable**

Writing this bit to one enables the USART Transmitter. The Transmitter will override normal port operation for the TxDn pin when enabled. The disabling of the Transmitter (writing TXENn to zero) will not become effective until ongoing and pending transmissions are completed, i.e., when the Transmit Shift Register and Transmit Buffer Register do not contain data to be transmitted. When disabled, the Transmitter will no longer override the TxDn port.

- **Bit 2:0 – Reserved Bits in MSPI mode**

When in MSPI mode, these bits are reserved for future use. For compatibility with future devices, these bits must be written to zero when UCSRnB is written.

### 21.8.4 UCSRnC – USART MSPIM Control and Status Register n C

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| | UMSELn1 | UMSELn0 | – | – | – | UDORDn | UCPHAn | UCPOLn | UCSRnC |
| Read/Write | R/W | R/W | R | R | R | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | |

- **Bit 7:6 – UMSELn1:0: USART Mode Select**

These bits select the mode of operation of the USART as shown in Table 21-4. See "UCSRnC – USART Control and Status Register n C" on page 199 for full description of the normal USART operation. The MSPIM is enabled when both UMSELn bits are set to one. The UDORDn, UCPHAn, and UCPOLn can be set in the same write operation where the MSPIM is enabled.

**Table 21-4.** UMSELn Bits Settings

| UMSELn1 | UMSELn0 | Mode |
|---|---|---|
| 0 | 0 | Asynchronous USART |
| 0 | 1 | Synchronous USART |
| 1 | 0 | Reserved |
| 1 | 1 | Master SPI (MSPIM) |

- **Bit 5:3 – Reserved Bits in MSPI mode**

When in MSPI mode, these bits are reserved for future use. For compatibility with future devices, these bits must be written to zero when UCSRnC is written.

- **Bit 2 – UDORDn: Data Order**

When set to one the LSB of the data word is transmitted first. When set to zero the MSB of the data word is transmitted first. Refer to the Frame Formats section page 4 for details.

- **Bit 1 – UCPHAn: Clock Phase**

The UCPHAn bit setting determine if data is sampled on the leasing edge (first) or tailing (last) edge of XCKn. Refer to the SPI Data Modes and Timing section page 4 for details.

- **Bit 0 – UCPOLn: Clock Polarity**

The UCPOLn bit sets the polarity of the XCKn clock. The combination of the UCPOLn and UCPHAn bit settings determine the timing of the data transfer. Refer to the SPI Data Modes and Timing section page 4 for details.

### 21.8.5 USART MSPIM Baud Rate Registers – UBRRnL and UBRRnH

The function and bit description of the baud rate registers in MSPI mode is identical to normal USART operation. See “UBRRnL and UBRRnH – USART Baud Rate Registers” on page 201.

# 22. 2-wire Serial Interface

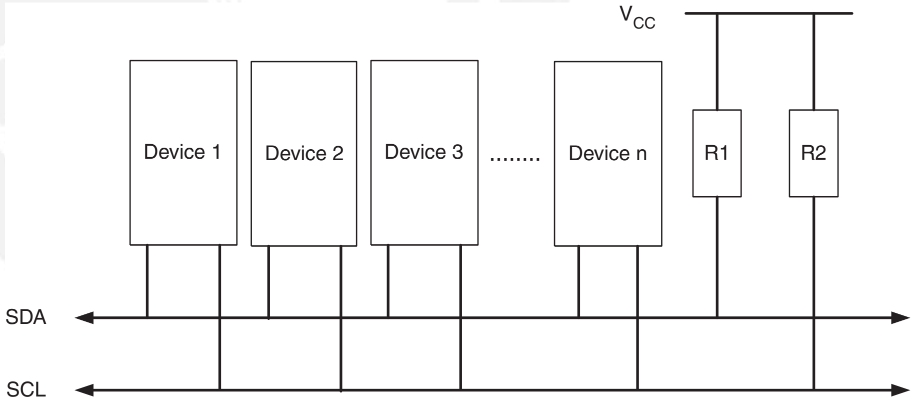
## 22.1 Features

- **Simple Yet Powerful and Flexible Communication Interface, only two Bus Lines Needed**
- **Both Master and Slave Operation Supported**
- **Device can Operate as Transmitter or Receiver**
- **7-bit Address Space Allows up to 128 Different Slave Addresses**
- **Multi-master Arbitration Support**
- **Up to 400kHz Data Transfer Speed**
- **Slew-rate Limited Output Drivers**
- **Noise Suppression Circuitry Rejects Spikes on Bus Lines**
- **Fully Programmable Slave Address with General Call Support**
- **Address Recognition Causes Wake-up When AVR is in Sleep Mode**
- **Compatible with Philips' I$^2$C protocol**

## 22.2 2-wire Serial Interface Bus Definition

The 2-wire Serial Interface (TWI) is ideally suited for typical microcontroller applications. The TWI protocol allows the systems designer to interconnect up to 128 different devices using only two bi-directional bus lines, one for clock (SCL) and one for data (SDA). The only external hardware needed to implement the bus is a single pull-up resistor for each of the TWI bus lines. All devices connected to the bus have individual addresses, and mechanisms for resolving bus contention are inherent in the TWI protocol.

**Figure 22-1.** TWI Bus Interconnection



### 22.2.1 TWI Terminology

The following definitions are frequently encountered in this section.

**Table 22-1.** TWI Terminology

| Term | Description |
|---|---|
| Master | The device that initiates and terminates a transmission. The Master also generates the SCL clock. |
| Slave | The device addressed by a Master. |
| Transmitter | The device placing data on the bus. |
| Receiver | The device reading data from the bus. |

The PRTWI bit in "Minimizing Power Consumption" on page 42 must be written to zero to enable the 2-wire Serial Interface.

### 22.2.2 Electrical Interconnection

As depicted in Figure 22-1, both bus lines are connected to the positive supply voltage through pull-up resistors. The bus drivers of all TWI-compliant devices are open-drain or open-collector. This implements a wired-AND function which is essential to the operation of the interface. A low level on a TWI bus line is generated when one or more TWI devices output a zero. A high level is output when all TWI devices tri-state their outputs, allowing the pull-up resistors to pull the line high. Note that all AVR devices connected to the TWI bus must be powered in order to allow any bus operation.

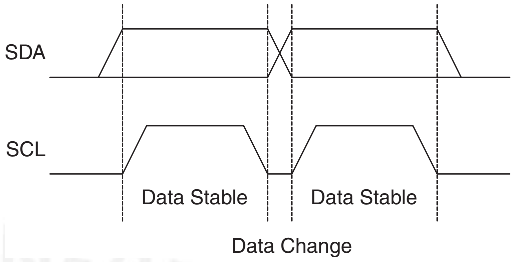
The number of devices that can be connected to the bus is only limited by the bus capacitance limit of 400 pF and the 7-bit slave address space. A detailed specification of the electrical characteristics of the TWI is given in "Two-wire Serial Interface Characteristics" on page 320. Two different sets of specifications are presented there, one relevant for bus speeds below 100kHz, and one valid for bus speeds up to 400kHz.

## 22.3 Data Transfer and Frame Format

### 22.3.1 Transferring Bits

Each data bit transferred on the TWI bus is accompanied by a pulse on the clock line. The level of the data line must be stable when the clock line is high. The only exception to this rule is for generating start and stop conditions.

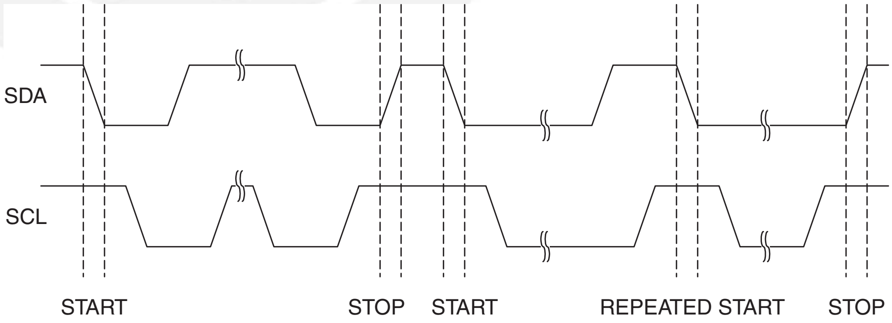
**Figure 22-2.** Data Validity



### 22.3.2 START and STOP Conditions

The Master initiates and terminates a data transmission. The transmission is initiated when the Master issues a START condition on the bus, and it is terminated when the Master issues a STOP condition. Between a START and a STOP condition, the bus is considered busy, and no other master should try to seize control of the bus. A special case occurs when a new START condition is issued between a START and STOP condition. This is referred to as a REPEATED START condition, and is used when the Master wishes to initiate a new transfer without relinquishing control of the bus. After a REPEATED START, the bus is considered busy until the next STOP. This is identical to the START behavior, and therefore START is used to describe both START and REPEATED START for the remainder of this datasheet, unless otherwise noted. As depicted below, START and STOP conditions are signalled by changing the level of the SDA line when the SCL line is high.

**Figure 22-3.** START, REPEATED START and STOP conditions



### 22.3.3 Address Packet Format

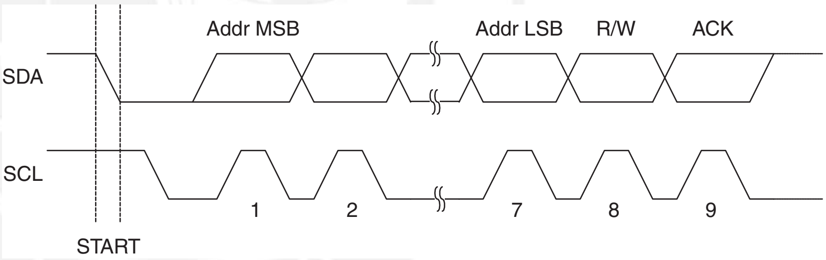
All address packets transmitted on the TWI bus are 9 bits long, consisting of 7 address bits, one READ/WRITE control bit and an acknowledge bit. If the READ/WRITE bit is set, a read operation is to be performed, otherwise a write operation should be performed. When a Slave recognizes that it is being addressed, it should acknowledge by pulling SDA low in the ninth SCL (ACK) cycle. If the addressed Slave is busy, or for some other reason can not service the Master's request, the SDA line should be left high in the ACK clock cycle. The Master can then transmit a STOP condition, or a REPEATED START condition to initiate a new transmission. An address packet consisting of a slave address and a READ or a WRITE bit is called SLA+R or SLA+W, respectively.

The MSB of the address byte is transmitted first. Slave addresses can freely be allocated by the designer, but the address 0000 000 is reserved for a general call.

When a general call is issued, all slaves should respond by pulling the SDA line low in the ACK cycle. A general call is used when a Master wishes to transmit the same message to several slaves in the system. When the general call address followed by a Write bit is transmitted on the bus, all slaves set up to acknowledge the general call will pull the SDA line low in the ack cycle. The following data packets will then be received by all the slaves that acknowledged the general call. Note that transmitting the general call address followed by a Read bit is meaningless, as this would cause contention if several slaves started transmitting different data.

All addresses of the format 1111 xxx should be reserved for future purposes.

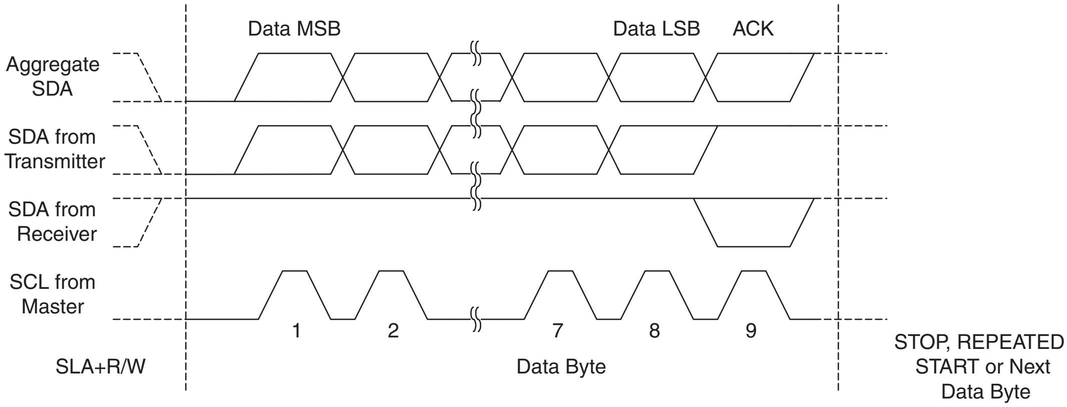
**Figure 22-4.** Address Packet Format



### 22.3.4 Data Packet Format

All data packets transmitted on the TWI bus are nine bits long, consisting of one data byte and an acknowledge bit. During a data transfer, the Master generates the clock and the START and STOP conditions, while the Receiver is responsible for acknowledging the reception. An Acknowledge (ACK) is signalled by the Receiver pulling the SDA line low during the ninth SCL cycle. If the Receiver leaves the SDA line high, a NACK is signalled. When the Receiver has received the last byte, or for some reason cannot receive any more bytes, it should inform the Transmitter by sending a NACK after the final byte. The MSB of the data byte is transmitted first.

Figure 22-5. Data Packet Format

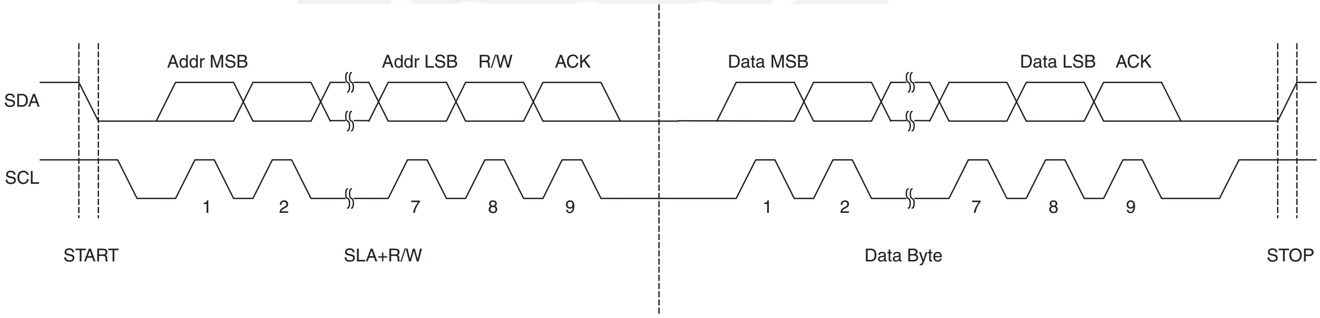


### 22.3.5 Combining Address and Data Packets into a Transmission

A transmission basically consists of a START condition, a SLA+R/W, one or more data packets and a STOP condition. An empty message, consisting of a START followed by a STOP condition, is illegal. Note that the Wired-ANDing of the SCL line can be used to implement handshaking between the Master and the Slave. The Slave can extend the SCL low period by pulling the SCL line low. This is useful if the clock speed set up by the Master is too fast for the Slave, or the Slave needs extra time for processing between the data transmissions. The Slave extending the SCL low period will not affect the SCL high period, which is determined by the Master. As a consequence, the Slave can reduce the TWI data transfer speed by prolonging the SCL duty cycle.

Figure 22-6 shows a typical data transmission. Note that several data bytes can be transmitted between the SLA+R/W and the STOP condition, depending on the software protocol implemented by the application software.

Figure 22-6. Typical Data Transmission



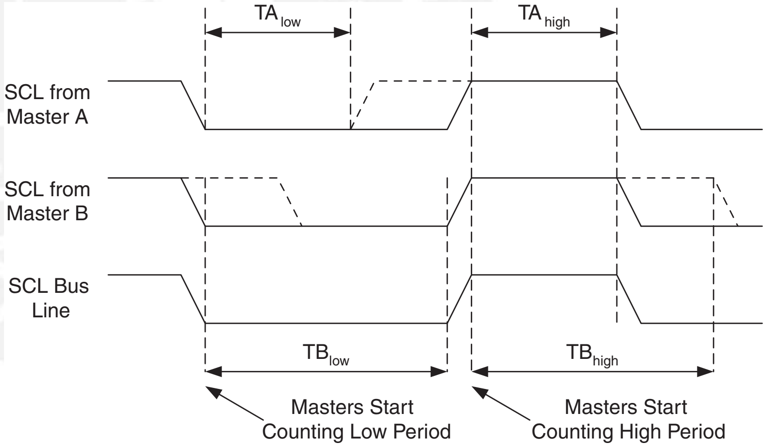
## 22.4 Multi-master Bus Systems, Arbitration and Synchronization

The TWI protocol allows bus systems with several masters. Special concerns have been taken in order to ensure that transmissions will proceed as normal, even if two or more masters initiate a transmission at the same time. Two problems arise in multi-master systems:

- An algorithm must be implemented allowing only one of the masters to complete the transmission. All other masters should cease transmission when they discover that they have lost the selection process. This selection process is called arbitration. When a contending master discovers that it has lost the arbitration process, it should immediately switch to Slave mode to check whether it is being addressed by the winning master. The fact that multiple masters have started transmission at the same time should not be detectable to the slaves, i.e. the data being transferred on the bus must not be corrupted.
- Different masters may use different SCL frequencies. A scheme must be devised to synchronize the serial clocks from all masters, in order to let the transmission proceed in a lockstep fashion. This will facilitate the arbitration process.

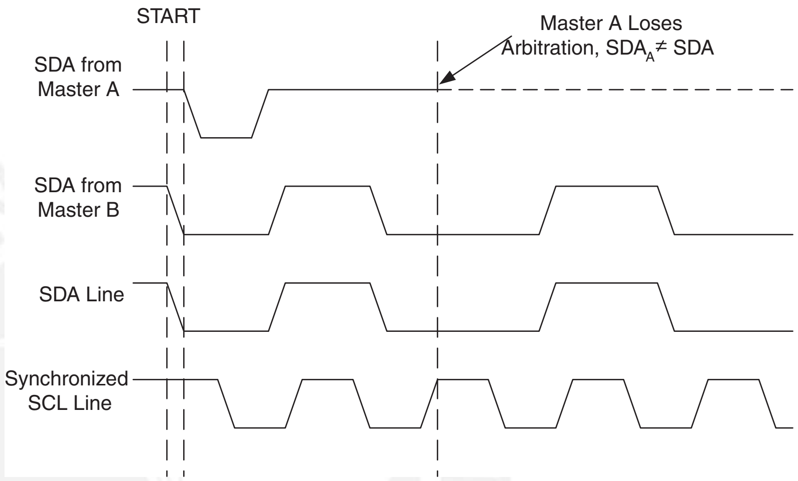
The wired-ANDing of the bus lines is used to solve both these problems. The serial clocks from all masters will be wired-ANDed, yielding a combined clock with a high period equal to the one from the Master with the shortest high period. The low period of the combined clock is equal to the low period of the Master with the longest low period. Note that all masters listen to the SCL line, effectively starting to count their SCL high and low time-out periods when the combined SCL line goes high or low, respectively.

**Figure 22-7.** SCL Synchronization Between Multiple Masters



Arbitration is carried out by all masters continuously monitoring the SDA line after outputting data. If the value read from the SDA line does not match the value the Master had output, it has lost the arbitration. Note that a Master can only lose arbitration when it outputs a high SDA value while another Master outputs a low value. The losing Master should immediately go to Slave mode, checking if it is being addressed by the winning Master. The SDA line should be left high, but losing masters are allowed to generate a clock signal until the end of the current data or address packet. Arbitration will continue until only one Master remains, and this may take many bits. If several masters are trying to address the same Slave, arbitration will continue into the data packet.

**Figure 22-8.** Arbitration Between Two Masters



Note that arbitration is not allowed between:

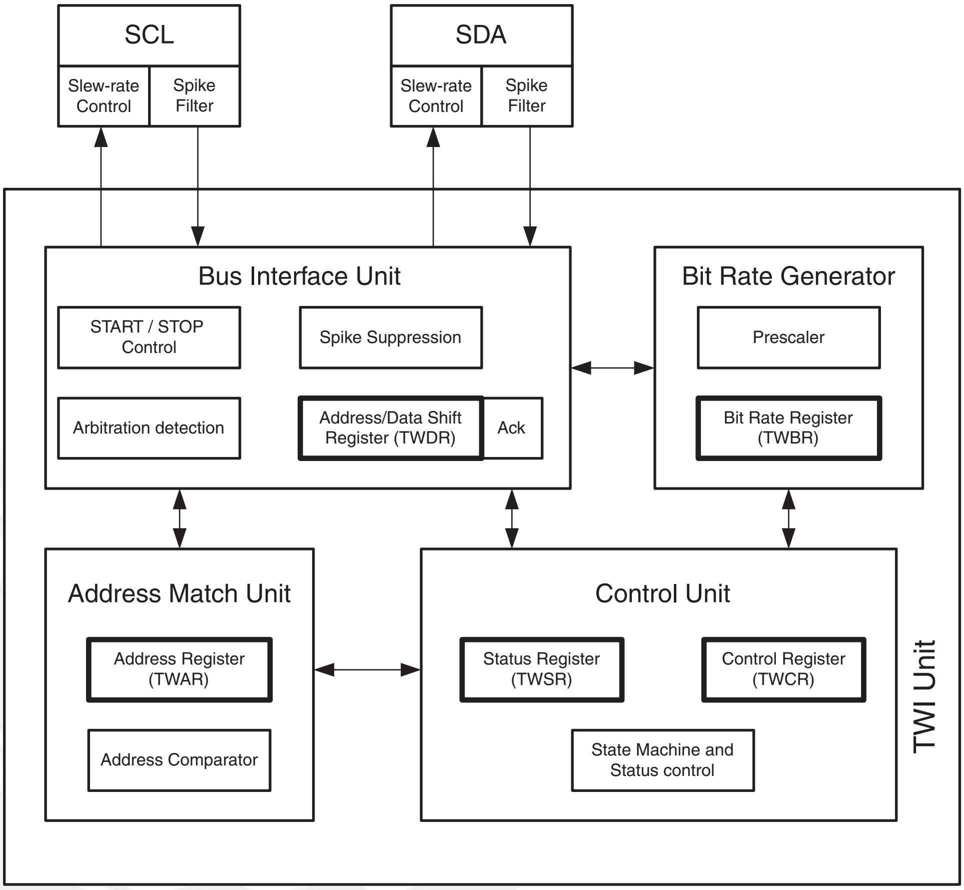
- A REPEATED START condition and a data bit.
- A STOP condition and a data bit.
- A REPEATED START and a STOP condition.

It is the user software's responsibility to ensure that these illegal arbitration conditions never occur. This implies that in multi-master systems, all data transfers must use the same composition of SLA+R/W and data packets. In other words: All transmissions must contain the same number of data packets, otherwise the result of the arbitration is undefined.

## 22.5 Overview of the TWI Module

The TWI module is comprised of several submodules, as shown in Figure 22-9. All registers drawn in a thick line are accessible through the AVR data bus.

**Figure 22-9.** Overview of the TWI Module



### 22.5.1 SCL and SDA Pins

These pins interface the AVR TWI with the rest of the MCU system. The output drivers contain a slew-rate limiter in order to conform to the TWI specification. The input stages contain a spike suppression unit removing spikes shorter than 50 ns. Note that the internal pull-ups in the AVR pads can be enabled by setting the PORT bits corresponding to the SCL and SDA pins, as explained in the I/O Port section. The internal pull-ups can in some systems eliminate the need for external ones.

### 22.5.2 Bit Rate Generator Unit

This unit controls the period of SCL when operating in a Master mode. The SCL period is controlled by settings in the TWI Bit Rate Register (TWBR) and the Prescaler bits in the TWI Status Register (TWSR). Slave operation does not depend on Bit Rate or Prescaler settings, but the CPU clock frequency in the Slave must be at least 16 times higher than the SCL frequency. Note that slaves may prolong the SCL low period, thereby reducing the average TWI bus clock period. The SCL frequency is generated according to the following equation:

$$\text{SCL frequency} = \frac{\text{CPU Clock frequency}}{16 + 2(\text{TWBR}) \times (\text{PrescalerValue})}$$

- TWBR = Value of the TWI Bit Rate Register.
- *PrescalerValue* = Value of the prescaler, see Table 22-8 on page 242.

Note: Pull-up resistor values should be selected according to the SCL frequency and the capacitive bus line load. See Table 29-8 on page 320 for value of pull-up resistor.

### 22.5.3 Bus Interface Unit

This unit contains the Data and Address Shift Register (TWDR), a START/STOP Controller and Arbitration detection hardware. The TWDR contains the address or data bytes to be transmitted, or the address or data bytes received. In addition to the 8-bit TWDR, the Bus Interface Unit also contains a register containing the (N)ACK bit to be transmitted or received. This (N)ACK Register is not directly accessible by the application software. However, when receiving, it can be set or cleared by manipulating the TWI Control Register (TWCR). When in Transmitter mode, the value of the received (N)ACK bit can be determined by the value in the TWSR.

The START/STOP Controller is responsible for generation and detection of START, REPEATED START, and STOP conditions. The START/STOP controller is able to detect START and STOP conditions even when the AVR MCU is in one of the sleep modes, enabling the MCU to wake up if addressed by a Master.

If the TWI has initiated a transmission as Master, the Arbitration Detection hardware continuously monitors the transmission trying to determine if arbitration is in process. If the TWI has lost an arbitration, the Control Unit is informed. Correct action can then be taken and appropriate status codes generated.

### 22.5.4 Address Match Unit

The Address Match unit checks if received address bytes match the seven-bit address in the TWI Address Register (TWAR). If the TWI General Call Recognition Enable (TWGCE) bit in the TWAR is written to one, all incoming address bits will also be compared against the General Call address. Upon an address match, the Control Unit is informed, allowing correct action to be taken. The TWI may or may not acknowledge its address, depending on settings in the TWCR. The Address Match unit is able to compare addresses even when the AVR MCU is in sleep mode, enabling the MCU to wake up if addressed by a Master.

### 22.5.5 Control Unit

The Control unit monitors the TWI bus and generates responses corresponding to settings in the TWI Control Register (TWCR). When an event requiring the attention of the application occurs on the TWI bus, the TWI Interrupt Flag (TWINT) is asserted. In the next clock cycle, the TWI Status Register (TWSR) is updated with a status code identifying the event. The TWSR only contains relevant status information when the TWI Interrupt Flag is asserted. At all other times, the TWSR contains a special status code indicating that no relevant status information is available. As long as the TWINT Flag is set, the SCL line is held low. This allows the application software to complete its tasks before allowing the TWI transmission to continue.

The TWINT Flag is set in the following situations:

- After the TWI has transmitted a START/REPEATED START condition.
- After the TWI has transmitted SLA+R/W.
- After the TWI has transmitted an address byte.
- After the TWI has lost arbitration.
- After the TWI has been addressed by own slave address or general call.
- After the TWI has received a data byte.
- After a STOP or REPEATED START has been received while still addressed as a Slave.
- When a bus error has occurred due to an illegal START or STOP condition.

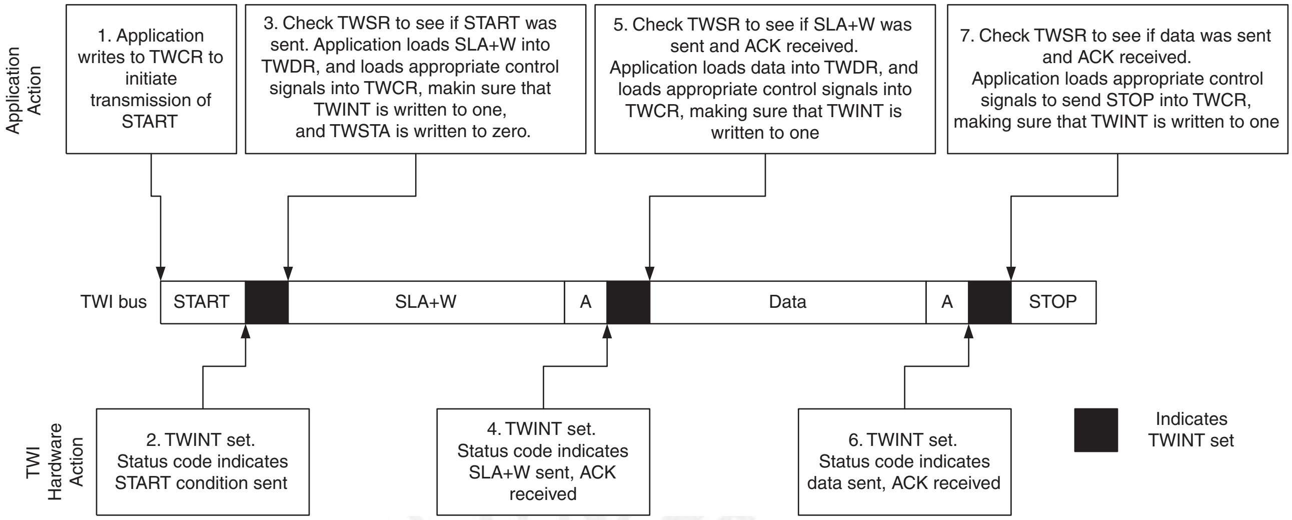
## 22.6 Using the TWI

The AVR TWI is byte-oriented and interrupt based. Interrupts are issued after all bus events, like reception of a byte or transmission of a START condition. Because the TWI is interrupt-based, the application software is free to carry on other operations during a TWI byte transfer. Note that the TWI Interrupt Enable (TWIE) bit in TWCR together with the Global Interrupt Enable bit in SREG allow the application to decide whether or not assertion of the TWINT Flag should generate an interrupt request. If the TWIE bit is cleared, the application must poll the TWINT Flag in order to detect actions on the TWI bus.

When the TWINT Flag is asserted, the TWI has finished an operation and awaits application response. In this case, the TWI Status Register (TWSR) contains a value indicating the current state of the TWI bus. The application software can then decide how the TWI should behave in the next TWI bus cycle by manipulating the TWCR and TWDR Registers.

Figure 22-10 is a simple example of how the application can interface to the TWI hardware. In this example, a Master wishes to transmit a single data byte to a Slave. This description is quite abstract, a more detailed explanation follows later in this section. A simple code example implementing the desired behavior is also presented.

**Figure 22-10.** Interfacing the Application to the TWI in a Typical Transmission


Application Action
1. Application writes to TWCR to initiate transmission of START
3. Check TWSR to see if START was sent. Application loads SLA+W into TWDR, and loads appropriate control signals into TWCR, makin sure that TWINT is written to one, and TWSTA is written to zero.
5. Check TWSR to see if SLA+W was sent and ACK received. Application loads data into TWDR, and loads appropriate control signals into TWCR, making sure that TWINT is written to one
7. Check TWSR to see if data was sent and ACK received. Application loads appropriate control signals to send STOP into TWCR, making sure that TWINT is written to one
TWI bus
START
SLA+W
A
Data
A
STOP
TWI Hardware Action
2. TWINT set. Status code indicates START condition sent
4. TWINT set. Status code indicates SLA+W sent, ACK received
6. TWINT set. Status code indicates data sent, ACK received
Indicates TWINT set

1. The first step in a TWI transmission is to transmit a START condition. This is done by writing a specific value into TWCR, instructing the TWI hardware to transmit a START condition. Which value to write is described later on. However, it is important that the TWINT bit is set in the value written. Writing a one to TWINT clears the flag. The TWI will not start any operation as long as the TWINT bit in TWCR is set. Immediately after the application has cleared TWINT, the TWI will initiate transmission of the START condition.
2. When the START condition has been transmitted, the TWINT Flag in TWCR is set, and TWSR is updated with a status code indicating that the START condition has successfully been sent.
3. The application software should now examine the value of TWSR, to make sure that the START condition was successfully transmitted. If TWSR indicates otherwise, the application software might take some special action, like calling an error routine. Assuming that the status code is as expected, the application must load SLA+W into TWDR. Remember that TWDR is used both for address and data. After TWDR has been loaded with the desired SLA+W, a specific value must be written to TWCR, instructing the TWI hardware to transmit the SLA+W present in TWDR. Which value to write is described later on. However, it is important that the TWINT bit is set in the value written. Writing a one to TWINT clears the flag. The TWI will not start any operation as long as the TWINT bit in TWCR is set. Immediately after the application has cleared TWINT, the TWI will initiate transmission of the address packet.
4. When the address packet has been transmitted, the TWINT Flag in TWCR is set, and TWSR is updated with a status code indicating that the address packet has successfully been sent. The status code will also reflect whether a Slave acknowledged the packet or not.

5. The application software should now examine the value of TWSR, to make sure that the address packet was successfully transmitted, and that the value of the ACK bit was as expected. If TWSR indicates otherwise, the application software might take some special action, like calling an error routine. Assuming that the status code is as expected, the application must load a data packet into TWDR. Subsequently, a specific value must be written to TWCR, instructing the TWI hardware to transmit the data packet present in TWDR. Which value to write is described later on. However, it is important that the TWINT bit is set in the value written. Writing a one to TWINT clears the flag. The TWI will not start any operation as long as the TWINT bit in TWCR is set. Immediately after the application has cleared TWINT, the TWI will initiate transmission of the data packet.
6. When the data packet has been transmitted, the TWINT Flag in TWCR is set, and TWSR is updated with a status code indicating that the data packet has successfully been sent. The status code will also reflect whether a Slave acknowledged the packet or not.
7. The application software should now examine the value of TWSR, to make sure that the data packet was successfully transmitted, and that the value of the ACK bit was as expected. If TWSR indicates otherwise, the application software might take some special action, like calling an error routine. Assuming that the status code is as expected, the application must write a specific value to TWCR, instructing the TWI hardware to transmit a STOP condition. Which value to write is described later on. However, it is important that the TWINT bit is set in the value written. Writing a one to TWINT clears the flag. The TWI will not start any operation as long as the TWINT bit in TWCR is set. Immediately after the application has cleared TWINT, the TWI will initiate transmission of the STOP condition. Note that TWINT is NOT set after a STOP condition has been sent.

Even though this example is simple, it shows the principles involved in all TWI transmissions. These can be summarized as follows:

- When the TWI has finished an operation and expects application response, the TWINT Flag is set. The SCL line is pulled low until TWINT is cleared.
- When the TWINT Flag is set, the user must update all TWI Registers with the value relevant for the next TWI bus cycle. As an example, TWDR must be loaded with the value to be transmitted in the next bus cycle.
- After all TWI Register updates and other pending application software tasks have been completed, TWCR is written. When writing TWCR, the TWINT bit should be set. Writing a one to TWINT clears the flag. The TWI will then commence executing whatever operation was specified by the TWCR setting.

In the following an assembly and C implementation of the example is given. Note that the code below assumes that several definitions have been made, for example by using include-files.

**Table 22-2.**

| | Assembly Code Example | C Example | Comments |
|---|---|---|---|
| 1 | **ldi** r16,<br>(1<<TWINT)\|(1<<TWSTA)\|<br>(1<<TWEN)<br>**out** TWCR, r16 | TWCR = (1<<TWINT)\|(1<<TWSTA)\|<br>(1<<TWEN) | Send START condition |
| 2 | wait1:<br>**in** r16,TWCR<br>**sbrs** r16,TWINT<br>**rjmp** wait1 | **while** (!(TWCR & (1<<TWINT)))<br>; | Wait for TWINT Flag set. This indicates that the START condition has been transmitted |
| 3 | **in** r16,TWSR<br>**andi** r16, 0xF8<br>**cpi** r16, START<br>**brne** ERROR | **if** ((TWSR & 0xF8) != START)<br>ERROR(); | Check value of TWI Status Register. Mask prescaler bits. If status different from START go to ERROR |
| | **ldi** r16, SLA_W<br>**out** TWDR, r16<br>**ldi** r16, (1<<TWINT) \|<br>(1<<TWEN)<br>**out** TWCR, r16 | TWDR = SLA_W;<br>TWCR = (1<<TWINT) \| (1<<TWEN); | Load SLA_W into TWDR Register. Clear TWINT bit in TWCR to start transmission of address |
| 4 | wait2:<br>**in** r16,TWCR<br>**sbrs** r16,TWINT<br>**rjmp** wait2 | **while** (!(TWCR & (1<<TWINT)))<br>; | Wait for TWINT Flag set. This indicates that the SLA+W has been transmitted, and ACK/NACK has been received. |
| 5 | **in** r16,TWSR<br>**andi** r16, 0xF8<br>**cpi** r16, MT_SLA_ACK<br>**brne** ERROR | **if** ((TWSR & 0xF8) !=<br>MT_SLA_ACK)<br>ERROR(); | Check value of TWI Status Register. Mask prescaler bits. If status different from MT_SLA_ACK go to ERROR |
| | **ldi** r16, DATA<br>**out** TWDR, r16<br>**ldi** r16, (1<<TWINT) \|<br>(1<<TWEN)<br>**out** TWCR, r16 | TWDR = DATA;<br>TWCR = (1<<TWINT) \| (1<<TWEN); | Load DATA into TWDR Register. Clear TWINT bit in TWCR to start transmission of data |
| 6 | wait3:<br>**in** r16,TWCR<br>**sbrs** r16,TWINT<br>**rjmp** wait3 | **while** (!(TWCR & (1<<TWINT)))<br>; | Wait for TWINT Flag set. This indicates that the DATA has been transmitted, and ACK/NACK has been received. |
| 7 | **in** r16,TWSR<br>**andi** r16, 0xF8<br>**cpi** r16, MT_DATA_ACK<br>**brne** ERROR | **if** ((TWSR & 0xF8) !=<br>MT_DATA_ACK)<br>ERROR(); | Check value of TWI Status Register. Mask prescaler bits. If status different from MT_DATA_ACK go to ERROR |
| | **ldi** r16,<br>(1<<TWINT)\|(1<<TWEN)\|<br>(1<<TWSTO)<br>**out** TWCR, r16 | TWCR = (1<<TWINT)\|(1<<TWEN)\|<br>(1<<TWSTO); | Transmit STOP condition |

## 22.7 Transmission Modes

The TWI can operate in one of four major modes. These are named Master Transmitter (MT), Master Receiver (MR), Slave Transmitter (ST) and Slave Receiver (SR). Several of these modes can be used in the same application. As an example, the TWI can use MT mode to write data into a TWI EEPROM, MR mode to read the data back from the EEPROM. If other masters are present in the system, some of these might transmit data to the TWI, and then SR mode would be used. It is the application software that decides which modes are legal.

The following sections describe each of these modes. Possible status codes are described along with figures detailing data transmission in each of the modes. These figures contain the following abbreviations:

S: START condition

Rs: REPEATED START condition

R: Read bit (high level at SDA)

W: Write bit (low level at SDA)

A: Acknowledge bit (low level at SDA)

$\overline{A}$: Not acknowledge bit (high level at SDA)

Data: 8-bit data byte

P: STOP condition

SLA: Slave Address

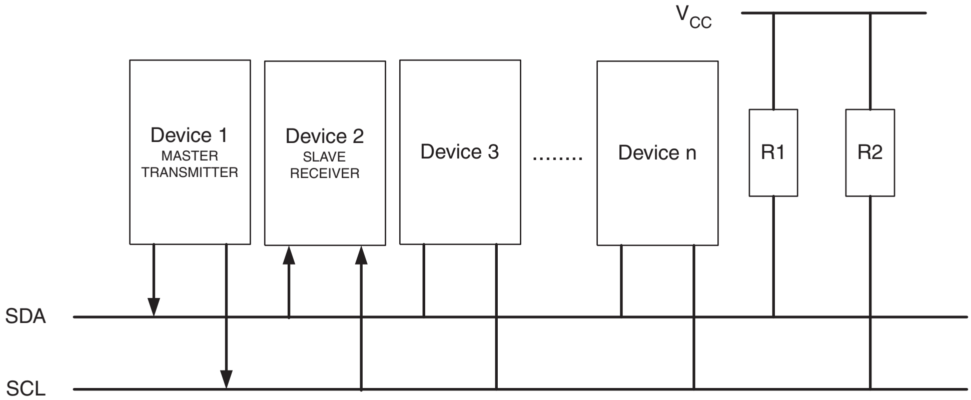
In Figure 22-12 to Figure 22-18, circles are used to indicate that the TWINT Flag is set. The numbers in the circles show the status code held in TWSR, with the prescaler bits masked to zero. At these points, actions must be taken by the application to continue or complete the TWI transfer. The TWI transfer is suspended until the TWINT Flag is cleared by software.

When the TWINT Flag is set, the status code in TWSR is used to determine the appropriate software action. For each status code, the required software action and details of the following serial transfer are given in Table 22-3 to Table 22-6. Note that the prescaler bits are masked to zero in these tables.

### 22.7.1 Master Transmitter Mode

In the Master Transmitter mode, a number of data bytes are transmitted to a Slave Receiver (see Figure 22-11). In order to enter a Master mode, a START condition must be transmitted. The format of the following address packet determines whether Master Transmitter or Master Receiver mode is to be entered. If SLA+W is transmitted, MT mode is entered, if SLA+R is transmitted, MR mode is entered. All the status codes mentioned in this section assume that the prescaler bits are zero or are masked to zero.

**Figure 22-11.** Data Transfer in Master Transmitter Mode



A START condition is sent by writing the following value to TWCR:

| TWCR | TWINT | TWEA | TWSTA | TWSTO | TWWC | TWEN | – | TWIE |
|---|---|---|---|---|---|---|---|---|
| value | 1 | X | 1 | 0 | X | 1 | 0 | X |

TWEN must be set to enable the 2-wire Serial Interface, TWSTA must be written to one to transmit a START condition and TWINT must be written to one to clear the TWINT Flag. The TWI will then test the 2-wire Serial Bus and generate a START condition as soon as the bus becomes free. After a START condition has been transmitted, the TWINT Flag is set by hardware, and the status code in TWSR will be 0x08 (see Table 22-3). In order to enter MT mode, SLA+W must be transmitted. This is done by writing SLA+W to TWDR. Thereafter the TWINT bit should be cleared (by writing it to one) to continue the transfer. This is accomplished by writing the following value to TWCR:

| TWCR | TWINT | TWEA | TWSTA | TWSTO | TWWC | TWEN | – | TWIE |
|---|---|---|---|---|---|---|---|---|
| value | 1 | X | 0 | 0 | X | 1 | 0 | X |

When SLA+W have been transmitted and an acknowledgement bit has been received, TWINT is set again and a number of status codes in TWSR are possible. Possible status codes in Master mode are 0x18, 0x20, or 0x38. The appropriate action to be taken for each of these status codes is detailed in Table 22-3.

When SLA+W has been successfully transmitted, a data packet should be transmitted. This is done by writing the data byte to TWDR. TWDR must only be written when TWINT is high. If not, the access will be discarded, and the Write Collision bit (TWWC) will be set in the TWCR Register. After updating TWDR, the TWINT bit should be cleared (by writing it to one) to continue the transfer. This is accomplished by writing the following value to TWCR:

| TWCR | TWINT | TWEA | TWSTA | TWSTO | TWWC | TWEN | – | TWIE |
|---|---|---|---|---|---|---|---|---|
| value | 1 | X | 0 | 0 | X | 1 | 0 | X |

This scheme is repeated until the last byte has been sent and the transfer is ended by generating a STOP condition or a repeated START condition. A STOP condition is generated by writing the following value to TWCR:

| TWCR | TWINT | TWEA | TWSTA | TWSTO | TWWC | TWEN | – | TWIE |
|---|---|---|---|---|---|---|---|---|
| value | 1 | X | 0 | 1 | X | 1 | 0 | X |

A REPEATED START condition is generated by writing the following value to TWCR:

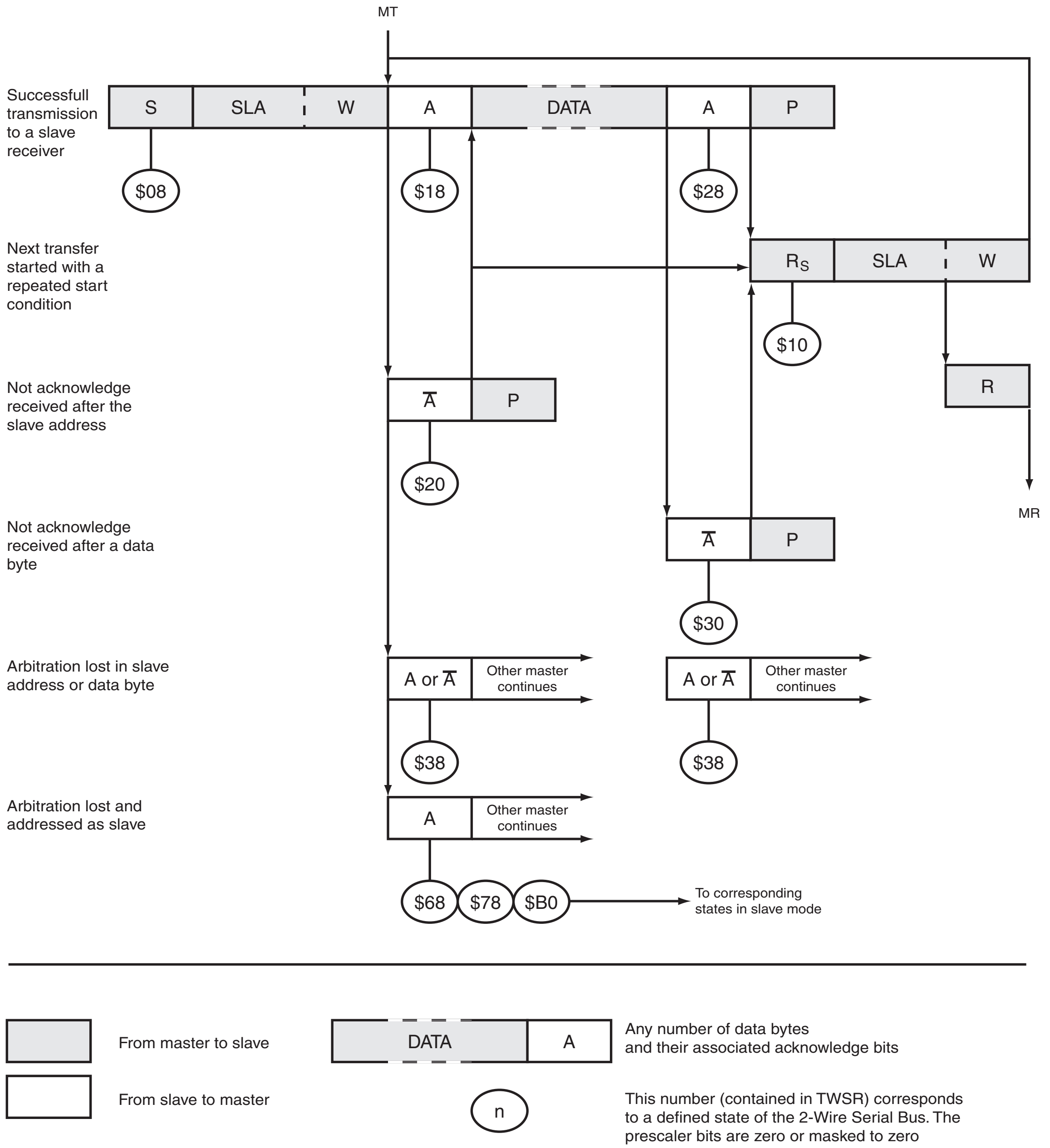
| TWCR | TWINT | TWEA | TWSTA | TWSTO | TWWC | TWEN | – | TWIE |
|---|---|---|---|---|---|---|---|---|
| value | 1 | X | 1 | 0 | X | 1 | 0 | X |

After a repeated START condition (state 0x10) the 2-wire Serial Interface can access the same Slave again, or a new Slave without transmitting a STOP condition. Repeated START enables the Master to switch between Slaves, Master Transmitter mode and Master Receiver mode without losing control of the bus.

**Table 22-3.** Status codes for Master Transmitter Mode

| Status Code (TWSR) Prescaler Bits are 0 | Status of the 2-wire Serial Bus and 2-wire Serial Interface Hardware | Application Software Response | | | | | Next Action Taken by TWI Hardware |
|---|---|---|---|---|---|---|---|
| | | | To TWCR | | | | |
| | | To/from TWDR | STA | STO | TWINT | TWEA | |
| 0x08 | A START condition has been transmitted | Load SLA+W | 0 | 0 | 1 | X | SLA+W will be transmitted;<br>ACK or NOT ACK will be received |
| 0x10 | A repeated START condition has been transmitted | Load SLA+W or | 0 | 0 | 1 | X | SLA+W will be transmitted;<br>ACK or NOT ACK will be received |
| | | Load SLA+R | 0 | 0 | 1 | X | SLA+R will be transmitted;<br>Logic will switch to Master Receiver mode |
| 0x18 | SLA+W has been transmitted;<br>ACK has been received | Load data byte or | 0 | 0 | 1 | X | Data byte will be transmitted and ACK or NOT ACK will be received |
| | | No TWDR action or | 1 | 0 | 1 | X | Repeated START will be transmitted |
| | | No TWDR action or | 0 | 1 | 1 | X | STOP condition will be transmitted and<br>TWSTO Flag will be reset |
| | | No TWDR action | 1 | 1 | 1 | X | STOP condition followed by a START condition will be transmitted and TWSTO Flag will be reset |
| 0x20 | SLA+W has been transmitted;<br>NOT ACK has been received | Load data byte or | 0 | 0 | 1 | X | Data byte will be transmitted and ACK or NOT ACK will be received |
| | | No TWDR action or | 1 | 0 | 1 | X | Repeated START will be transmitted |
| | | No TWDR action or | 0 | 1 | 1 | X | STOP condition will be transmitted and<br>TWSTO Flag will be reset |
| | | No TWDR action | 1 | 1 | 1 | X | STOP condition followed by a START condition will be transmitted and TWSTO Flag will be reset |
| 0x28 | Data byte has been transmitted;<br>ACK has been received | Load data byte or | 0 | 0 | 1 | X | Data byte will be transmitted and ACK or NOT ACK will be received |
| | | No TWDR action or | 1 | 0 | 1 | X | Repeated START will be transmitted |
| | | No TWDR action or | 0 | 1 | 1 | X | STOP condition will be transmitted and<br>TWSTO Flag will be reset |
| | | No TWDR action | 1 | 1 | 1 | X | STOP condition followed by a START condition will be transmitted and TWSTO Flag will be reset |
| 0x30 | Data byte has been transmitted;<br>NOT ACK has been received | Load data byte or | 0 | 0 | 1 | X | Data byte will be transmitted and ACK or NOT ACK will be received |
| | | No TWDR action or | 1 | 0 | 1 | X | Repeated START will be transmitted |
| | | No TWDR action or | 0 | 1 | 1 | X | STOP condition will be transmitted and<br>TWSTO Flag will be reset |
| | | No TWDR action | 1 | 1 | 1 | X | STOP condition followed by a START condition will be transmitted and TWSTO Flag will be reset |
| 0x38 | Arbitration lost in SLA+W or data bytes | No TWDR action or | 0 | 0 | 1 | X | 2-wire Serial Bus will be released and not addressed Slave mode entered |
| | | No TWDR action | 1 | 0 | 1 | X | A START condition will be transmitted when the bus becomes free |

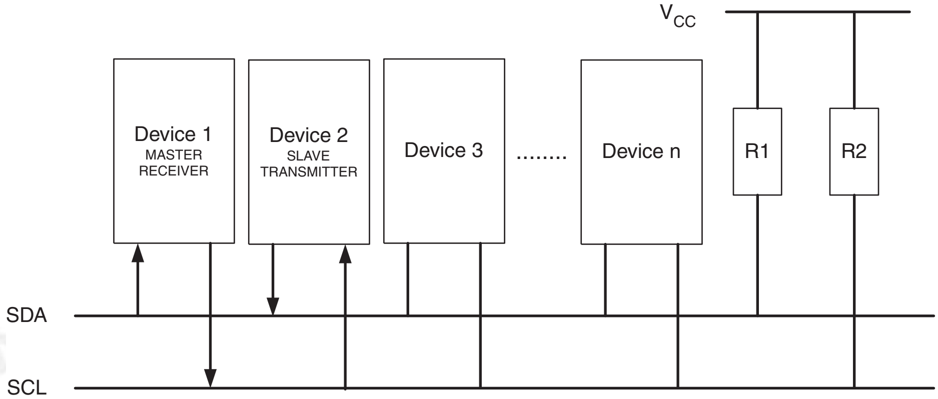
**Figure 22-12.** Formats and States in the Master Transmitter Mode


MT
Successfull transmission to a slave receiver
S
SLA
W
A
DATA
A
P
$08
$18
$28
Next transfer started with a repeated start condition
RS
SLA
W
$10
R
MR
Not acknowledge received after the slave address
Ā
P
$20
Not acknowledge received after a data byte
Ā
P
$30
Arbitration lost in slave address or data byte
A or Ā
Other master continues
A or Ā
Other master continues
$38
$38
Arbitration lost and addressed as slave
A
Other master continues
$68
$78
$B0
To corresponding states in slave mode
From master to slave
From slave to master
DATA
A
Any number of data bytes and their associated acknowledge bits
n
This number (contained in TWSR) corresponds to a defined state of the 2-Wire Serial Bus. The prescaler bits are zero or masked to zero

### 22.7.2 Master Receiver Mode

In the Master Receiver mode, a number of data bytes are received from a Slave Transmitter (Slave see Figure 22-13). In order to enter a Master mode, a START condition must be transmitted. The format of the following address packet determines whether Master Transmitter or Master Receiver mode is to be entered. If SLA+W is transmitted, MT mode is entered, if SLA+R is transmitted, MR mode is entered. All the status codes mentioned in this section assume that the prescaler bits are zero or are masked to zero.

**Figure 22-13.** Data Transfer in Master Receiver Mode



A START condition is sent by writing the following value to TWCR:

| TWCR | TWINT | TWEA | TWSTA | TWSTO | TWWC | TWEN | – | TWIE |
|---|---|---|---|---|---|---|---|---|
| value | 1 | X | 1 | 0 | X | 1 | 0 | X |

TWEN must be written to one to enable the 2-wire Serial Interface, TWSTA must be written to one to transmit a START condition and TWINT must be set to clear the TWINT Flag. The TWI will then test the 2-wire Serial Bus and generate a START condition as soon as the bus becomes free. After a START condition has been transmitted, the TWINT Flag is set by hardware, and the status code in TWSR will be 0x08 (See Table 22-3). In order to enter MR mode, SLA+R must be transmitted. This is done by writing SLA+R to TWDR. Thereafter the TWINT bit should be cleared (by writing it to one) to continue the transfer. This is accomplished by writing the following value to TWCR:

| TWCR | TWINT | TWEA | TWSTA | TWSTO | TWWC | TWEN | – | TWIE |
|---|---|---|---|---|---|---|---|---|
| value | 1 | X | 0 | 0 | X | 1 | 0 | X |

When SLA+R have been transmitted and an acknowledgement bit has been received, TWINT is set again and a number of status codes in TWSR are possible. Possible status codes in Master mode are 0x38, 0x40, or 0x48. The appropriate action to be taken for each of these status codes is detailed in Table 22-4. Received data can be read from the TWDR Register when the TWINT Flag is set high by hardware. This scheme is repeated until the last byte has been received. After the last byte has been received, the MR should inform the ST by sending a NACK after the last received data byte. The transfer is ended by generating a STOP condition or a repeated START condition. A STOP condition is generated by writing the following value to TWCR:

| TWCR | TWINT | TWEA | TWSTA | TWSTO | TWWC | TWEN | – | TWIE |
|---|---|---|---|---|---|---|---|---|
| value | 1 | X | 0 | 1 | X | 1 | 0 | X |

A REPEATED START condition is generated by writing the following value to TWCR:

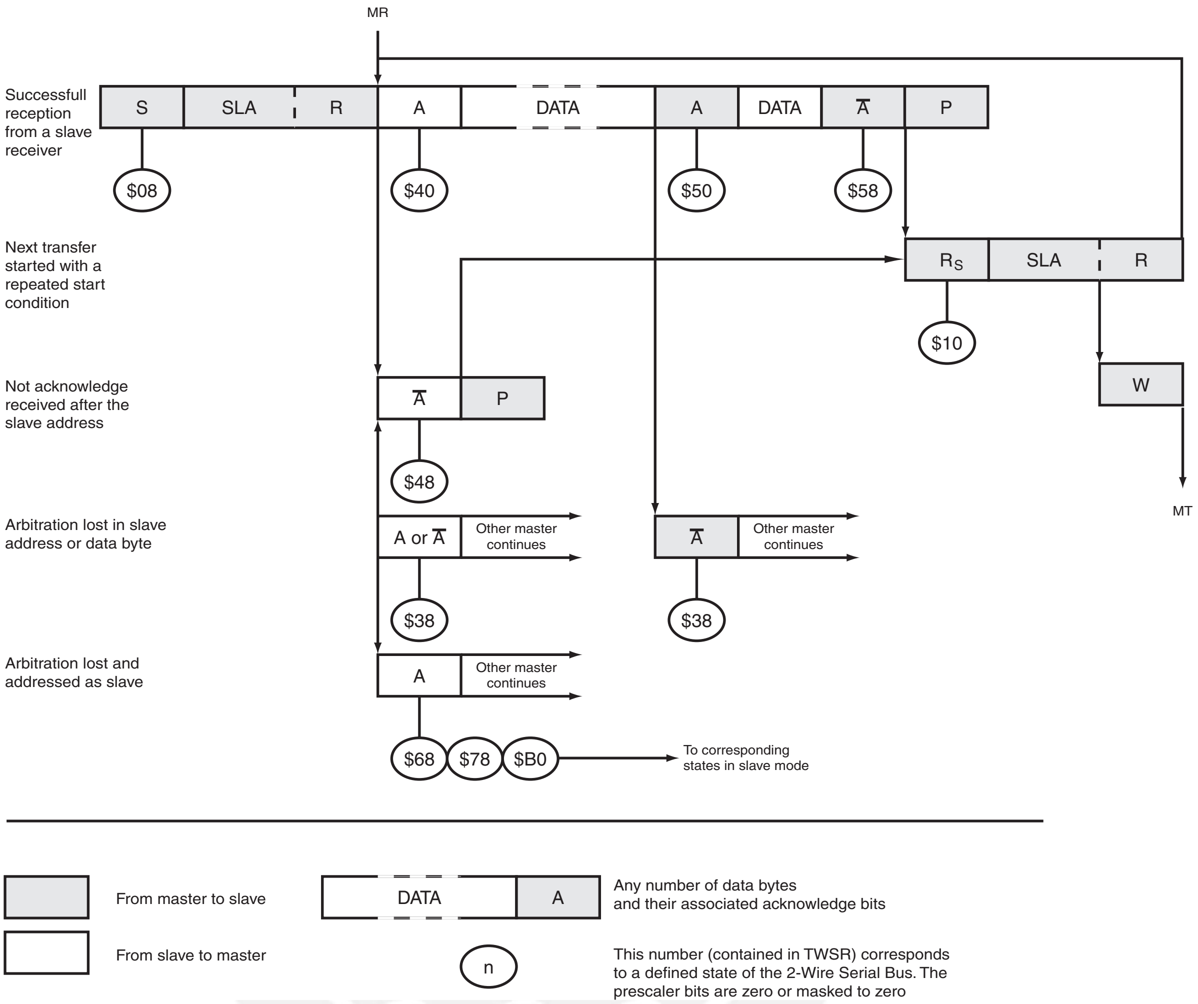
| TWCR | TWINT | TWEA | TWSTA | TWSTO | TWWC | TWEN | – | TWIE |
|---|---|---|---|---|---|---|---|---|
| value | 1 | X | 1 | 0 | X | 1 | 0 | X |

After a repeated START condition (state 0x10) the 2-wire Serial Interface can access the same Slave again, or a new Slave without transmitting a STOP condition. Repeated START enables the Master to switch between Slaves, Master Transmitter mode and Master Receiver mode without losing control over the bus.

**Table 22-4.** Status codes for Master Receiver Mode

| Status Code (TWSR) Prescaler Bits are 0 | Status of the 2-wire Serial Bus and 2-wire Serial Interface Hardware | Application Software Response | | | | | Next Action Taken by TWI Hardware |
|---|---|---|---|---|---|---|---|
| | | | To TWCR | | | | |
| | | To/from TWDR | STA | STO | TWINT | TWEA | |
| 0x08 | A START condition has been transmitted | Load SLA+R | 0 | 0 | 1 | X | SLA+R will be transmitted<br>ACK or NOT ACK will be received |
| 0x10 | A repeated START condition has been transmitted | Load SLA+R or<br>Load SLA+W | 0<br>0 | 0<br>0 | 1<br>1 | X<br>X | SLA+R will be transmitted<br>ACK or NOT ACK will be received<br>SLA+W will be transmitted<br>Logic will switch to Master Transmitter mode |
| 0x38 | Arbitration lost in SLA+R or NOT ACK bit | No TWDR action or<br>No TWDR action | 0<br>1 | 0<br>0 | 1<br>1 | X<br>X | 2-wire Serial Bus will be released and not addressed Slave mode will be entered<br>A START condition will be transmitted when the bus<br>becomes free |
| 0x40 | SLA+R has been transmitted;<br>ACK has been received | No TWDR action or<br>No TWDR action | 0<br>0 | 0<br>0 | 1<br>1 | 0<br>1 | Data byte will be received and NOT ACK will be returned<br>Data byte will be received and ACK will be returned |
| 0x48 | SLA+R has been transmitted;<br>NOT ACK has been received | No TWDR action or<br>No TWDR action or<br>No TWDR action | 1<br>0<br>1 | 0<br>1<br>1 | 1<br>1<br>1 | X<br>X<br>X | Repeated START will be transmitted<br>STOP condition will be transmitted and TWSTO Flag will be reset<br>STOP condition followed by a START condition will be transmitted and TWSTO Flag will be reset |
| 0x50 | Data byte has been received;<br>ACK has been returned | Read data byte or<br>Read data byte | 0<br>0 | 0<br>0 | 1<br>1 | 0<br>1 | Data byte will be received and NOT ACK will be returned<br>Data byte will be received and ACK will be returned |
| 0x58 | Data byte has been received;<br>NOT ACK has been returned | Read data byte or<br>Read data byte or<br>Read data byte | 1<br>0<br>1 | 0<br>1<br>1 | 1<br>1<br>1 | X<br>X<br>X | Repeated START will be transmitted<br>STOP condition will be transmitted and TWSTO Flag will be reset<br>STOP condition followed by a START condition will be transmitted and TWSTO Flag will be reset |

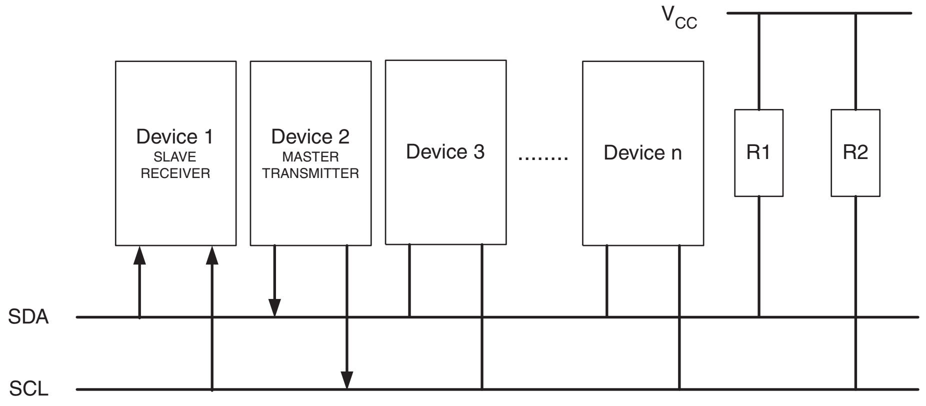
**Figure 22-14.** Formats and States in the Master Receiver Mode


MR
Successfull reception from a slave receiver
S
SLA
R
A
DATA
A
DATA
A̅
P
$08
$40
$50
$58
Next transfer started with a repeated start condition
R<sub>S</sub>
SLA
R
$10
W
MT
Not acknowledge received after the slave address
A̅
P
$48
Arbitration lost in slave address or data byte
A or A̅
Other master continues
A̅
Other master continues
$38
$38
Arbitration lost and addressed as slave
A
Other master continues
$68
$78
$B0
To corresponding states in slave mode
From master to slave
DATA
A
Any number of data bytes and their associated acknowledge bits
From slave to master
n
This number (contained in TWSR) corresponds to a defined state of the 2-Wire Serial Bus. The prescaler bits are zero or masked to zero

### 22.7.3 Slave Receiver Mode

In the Slave Receiver mode, a number of data bytes are received from a Master Transmitter (see Figure 22-15). All the status codes mentioned in this section assume that the prescaler bits are zero or are masked to zero.

**Figure 22-15.** Data Transfer in Slave Receiver Mode



To initiate the Slave Receiver mode, TWAR and TWCR must be initialized as follows:

| TWAR | TWA6 | TWA5 | TWA4 | TWA3 | TWA2 | TWA1 | TWA0 | TWGCE |
|---|---|---|---|---|---|---|---|---|
| value | Device's Own Slave Address | | | | | | | |

The upper 7 bits are the address to which the 2-wire Serial Interface will respond when addressed by a Master. If the LSB is set, the TWI will respond to the general call address (0x00), otherwise it will ignore the general call address.

| TWCR | TWINT | TWEA | TWSTA | TWSTO | TWWC | TWEN | – | TWIE |
|---|---|---|---|---|---|---|---|---|
| value | 0 | 1 | 0 | 0 | 0 | 1 | 0 | X |

TWEN must be written to one to enable the TWI. The TWEA bit must be written to one to enable the acknowledgement of the device's own slave address or the general call address. TWSTA and TWSTO must be written to zero.

When TWAR and TWCR have been initialized, the TWI waits until it is addressed by its own slave address (or the general call address if enabled) followed by the data direction bit. If the direction bit is "0" (write), the TWI will operate in SR mode, otherwise ST mode is entered. After its own slave address and the write bit have been received, the TWINT Flag is set and a valid status code can be read from TWSR. The status code is used to determine the appropriate software action. The appropriate action to be taken for each status code is detailed in Table 22-5. The Slave Receiver mode may also be entered if arbitration is lost while the TWI is in the Master mode (see states 0x68 and 0x78).

If the TWEA bit is reset during a transfer, the TWI will return a "Not Acknowledge" ("1") to SDA after the next received data byte. This can be used to indicate that the Slave is not able to receive any more bytes. While TWEA is zero, the TWI does not acknowledge its own slave address. However, the 2-wire Serial Bus is still monitored and address recognition may resume at any time by setting TWEA. This implies that the TWEA bit may be used to temporarily isolate the TWI from the 2-wire Serial Bus.

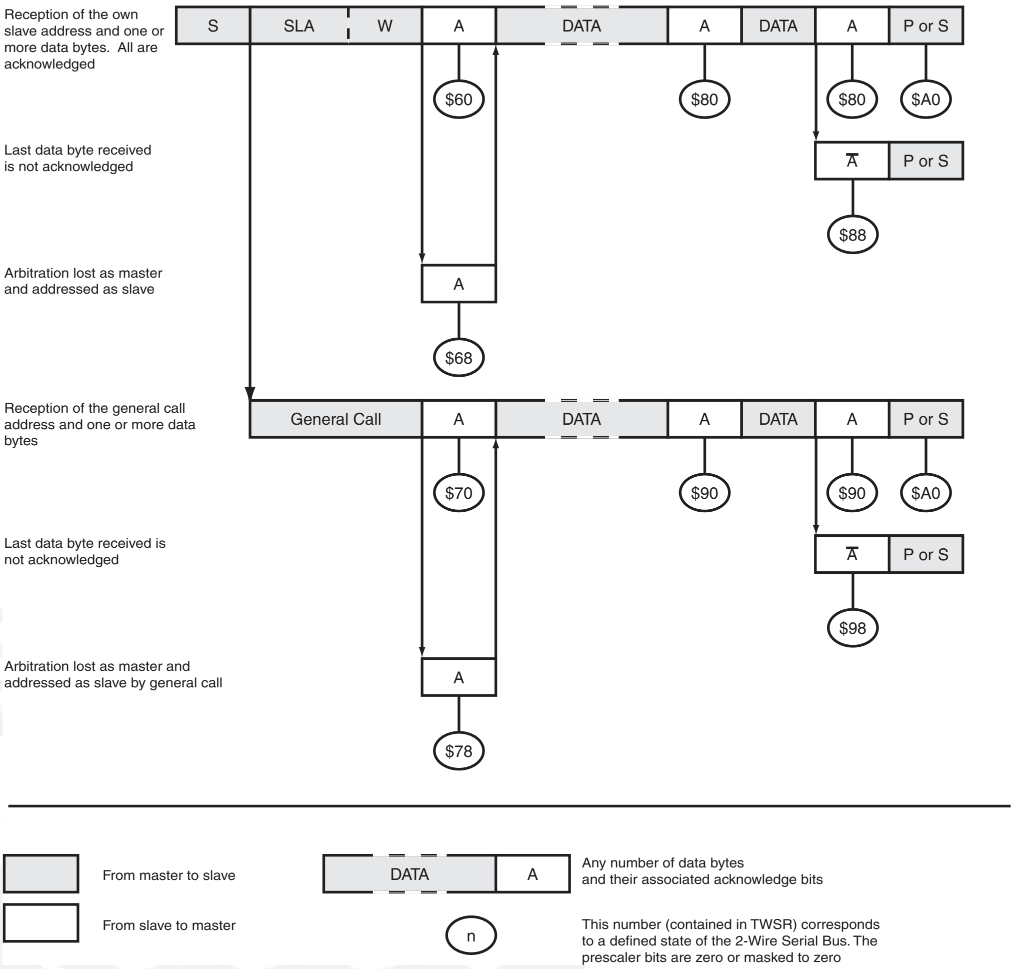
In all sleep modes other than Idle mode, the clock system to the TWI is turned off. If the TWEA bit is set, the interface can still acknowledge its own slave address or the general call address by using the 2-wire Serial Bus clock as a clock source. The part will then wake up from sleep and the TWI will hold the SCL clock low during the wake up and until the TWINT Flag is cleared (by writing it to one). Further data reception will be carried out as normal, with the AVR clocks running as normal. Observe that if the AVR is set up with a long start-up time, the SCL line may be held low for a long time, blocking other data transmissions.

Note that the 2-wire Serial Interface Data Register – TWDR does not reflect the last byte present on the bus when waking up from these Sleep modes.

**Table 22-5.** Status Codes for Slave Receiver Mode

| Status Code (TWSR) Prescaler Bits are 0 | Status of the 2-wire Serial Bus and 2-wire Serial Interface Hardware | Application Software Response | | | | | Next Action Taken by TWI Hardware |
|---|---|---|---|---|---|---|---|
| | | | To TWCR | | | | |
| | | To/from TWDR | STA | STO | TWINT | TWEA | |
| 0x60 | Own SLA+W has been received; ACK has been returned | No TWDR action or | X | 0 | 1 | 0 | Data byte will be received and NOT ACK will be returned |
| | | No TWDR action | X | 0 | 1 | 1 | Data byte will be received and ACK will be returned |
| 0x68 | Arbitration lost in SLA+R/W as Master; own SLA+W has been received; ACK has been returned | No TWDR action or | X | 0 | 1 | 0 | Data byte will be received and NOT ACK will be returned |
| | | No TWDR action | X | 0 | 1 | 1 | Data byte will be received and ACK will be returned |
| 0x70 | General call address has been received; ACK has been returned | No TWDR action or | X | 0 | 1 | 0 | Data byte will be received and NOT ACK will be returned |
| | | No TWDR action | X | 0 | 1 | 1 | Data byte will be received and ACK will be returned |
| 0x78 | Arbitration lost in SLA+R/W as Master; General call address has been received; ACK has been returned | No TWDR action or | X | 0 | 1 | 0 | Data byte will be received and NOT ACK will be returned |
| | | No TWDR action | X | 0 | 1 | 1 | Data byte will be received and ACK will be returned |
| 0x80 | Previously addressed with own SLA+W; data has been received; ACK has been returned | Read data byte or | X | 0 | 1 | 0 | Data byte will be received and NOT ACK will be returned |
| | | Read data byte | X | 0 | 1 | 1 | Data byte will be received and ACK will be returned |
| 0x88 | Previously addressed with own SLA+W; data has been received; NOT ACK has been returned | Read data byte or | 0 | 0 | 1 | 0 | Switched to the not addressed Slave mode; no recognition of own SLA or GCA |
| | | Read data byte or | 0 | 0 | 1 | 1 | Switched to the not addressed Slave mode; own SLA will be recognized; GCA will be recognized if TWGCE = “1” |
| | | Read data byte or | 1 | 0 | 1 | 0 | Switched to the not addressed Slave mode; no recognition of own SLA or GCA; a START condition will be transmitted when the bus becomes free |
| | | Read data byte | 1 | 0 | 1 | 1 | Switched to the not addressed Slave mode; own SLA will be recognized; GCA will be recognized if TWGCE = “1”; a START condition will be transmitted when the bus becomes free |
| 0x90 | Previously addressed with general call; data has been received; ACK has been returned | Read data byte or | X | 0 | 1 | 0 | Data byte will be received and NOT ACK will be returned |
| | | Read data byte | X | 0 | 1 | 1 | Data byte will be received and ACK will be returned |
| 0x98 | Previously addressed with general call; data has been received; NOT ACK has been returned | Read data byte or | 0 | 0 | 1 | 0 | Switched to the not addressed Slave mode; no recognition of own SLA or GCA |
| | | Read data byte or | 0 | 0 | 1 | 1 | Switched to the not addressed Slave mode; own SLA will be recognized; GCA will be recognized if TWGCE = “1” |
| | | Read data byte or | 1 | 0 | 1 | 0 | Switched to the not addressed Slave mode; no recognition of own SLA or GCA; a START condition will be transmitted when the bus becomes free |
| | | Read data byte | 1 | 0 | 1 | 1 | Switched to the not addressed Slave mode; own SLA will be recognized; GCA will be recognized if TWGCE = “1”; a START condition will be transmitted when the bus becomes free |
| 0xA0 | A STOP condition or repeated START condition has been received while still addressed as Slave | No action | 0 | 0 | 1 | 0 | Switched to the not addressed Slave mode; no recognition of own SLA or GCA |
| | | | 0 | 0 | 1 | 1 | Switched to the not addressed Slave mode; own SLA will be recognized; GCA will be recognized if TWGCE = “1” |
| | | | 1 | 0 | 1 | 0 | Switched to the not addressed Slave mode; no recognition of own SLA or GCA; a START condition will be transmitted when the bus becomes free |
| | | | 1 | 0 | 1 | 1 | Switched to the not addressed Slave mode; own SLA will be recognized; GCA will be recognized if TWGCE = “1”; a START condition will be transmitted when the bus becomes free |

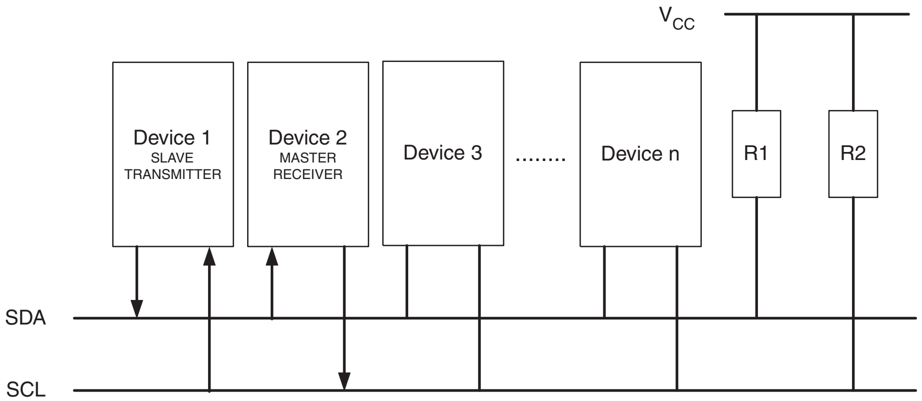
**Figure 22-16.** Formats and States in the Slave Receiver Mode


Reception of the own slave address and one or more data bytes. All are acknowledged
S
SLA
W
A
DATA
A
DATA
A
P or S
$60
$80
$80
$A0
Last data byte received is not acknowledged
A̅
P or S
$88
Arbitration lost as master and addressed as slave
A
$68
Reception of the general call address and one or more data bytes
General Call
A
DATA
A
DATA
A
P or S
$70
$90
$90
$A0
Last data byte received is not acknowledged
A̅
P or S
$98
Arbitration lost as master and addressed as slave by general call
A
$78
From master to slave
From slave to master
DATA
A
Any number of data bytes and their associated acknowledge bits
n
This number (contained in TWSR) corresponds to a defined state of the 2-Wire Serial Bus. The prescaler bits are zero or masked to zero

### 22.7.4 Slave Transmitter Mode

In the Slave Transmitter mode, a number of data bytes are transmitted to a Master Receiver (see Figure 22-17). All the status codes mentioned in this section assume that the prescaler bits are zero or are masked to zero.

**Figure 22-17.** Data Transfer in Slave Transmitter Mode



To initiate the Slave Transmitter mode, TWAR and TWCR must be initialized as follows:

| TWAR | TWA6 | TWA5 | TWA4 | TWA3 | TWA2 | TWA1 | TWA0 | TWGCE |
|---|---|---|---|---|---|---|---|---|
| value | Device’s Own Slave Address | | | | | | | |

The upper seven bits are the address to which the 2-wire Serial Interface will respond when addressed by a Master. If the LSB is set, the TWI will respond to the general call address (0x00), otherwise it will ignore the general call address.

| TWCR | TWINT | TWEA | TWSTA | TWSTO | TWWC | TWEN | – | TWIE |
|---|---|---|---|---|---|---|---|---|
| value | 0 | 1 | 0 | 0 | 0 | 1 | 0 | X |

TWEN must be written to one to enable the TWI. The TWEA bit must be written to one to enable the acknowledgement of the device’s own slave address or the general call address. TWSTA and TWSTO must be written to zero.

When TWAR and TWCR have been initialized, the TWI waits until it is addressed by its own slave address (or the general call address if enabled) followed by the data direction bit. If the direction bit is “1” (read), the TWI will operate in ST mode, otherwise SR mode is entered. After its own slave address and the write bit have been received, the TWINT Flag is set and a valid status code can be read from TWSR. The status code is used to determine the appropriate software action. The appropriate action to be taken for each status code is detailed in Table 22-6. The Slave Transmitter mode may also be entered if arbitration is lost while the TWI is in the Master mode (see state 0xB0).

If the TWEA bit is written to zero during a transfer, the TWI will transmit the last byte of the transfer. State 0xC0 or state 0xC8 will be entered, depending on whether the Master Receiver transmits a NACK or ACK after the final byte. The TWI is switched to the not addressed Slave mode, and will ignore the Master if it continues the transfer. Thus the Master Receiver receives all “1” as serial data. State 0xC8 is entered if the Master demands additional data bytes (by transmitting ACK), even though the Slave has transmitted the last byte (TWEA zero and expecting NACK from the Master).

While TWEA is zero, the TWI does not respond to its own slave address. However, the 2-wire Serial Bus is still monitored and address recognition may resume at any time by setting TWEA. This implies that the TWEA bit may be used to temporarily isolate the TWI from the 2-wire Serial Bus.

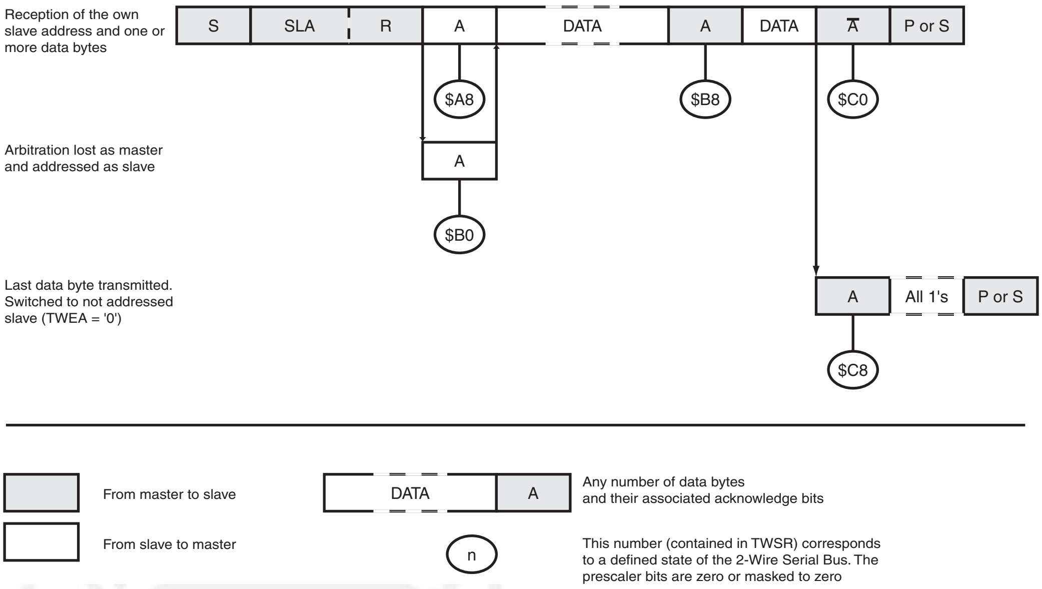
In all sleep modes other than Idle mode, the clock system to the TWI is turned off. If the TWEA bit is set, the interface can still acknowledge its own slave address or the general call address by using the 2-wire Serial Bus clock as a clock source. The part will then wake up from sleep and the TWI will hold the SCL clock will low during the wake up and until the TWINT Flag is cleared (by writing it to one). Further data transmission will be carried out as normal, with the AVR clocks running as normal. Observe that if the AVR is set up with a long start-up time, the SCL line may be held low for a long time, blocking other data transmissions.

Note that the 2-wire Serial Interface Data Register – TWDR does not reflect the last byte present on the bus when waking up from these sleep modes.

**Table 22-6.** Status Codes for Slave Transmitter Mode

| Status Code (TWSR) Prescaler Bits are 0 | Status of the 2-wire Serial Bus and 2-wire Serial Interface Hardware | Application Software Response | | | | | Next Action Taken by TWI Hardware |
|---|---|---|---|---|---|---|---|
| | | | To TWCR | | | | |
| | | To/from TWDR | STA | STO | TWINT | TWEA | |
| 0xA8 | Own SLA+R has been received; ACK has been returned | Load data byte or | X | 0 | 1 | 0 | Last data byte will be transmitted and NOT ACK should be received |
| | | Load data byte | X | 0 | 1 | 1 | Data byte will be transmitted and ACK should be received |
| 0xB0 | Arbitration lost in SLA+R/W as Master; own SLA+R has been received; ACK has been returned | Load data byte or | X | 0 | 1 | 0 | Last data byte will be transmitted and NOT ACK should be received |
| | | Load data byte | X | 0 | 1 | 1 | Data byte will be transmitted and ACK should be received |
| 0xB8 | Data byte in TWDR has been transmitted; ACK has been received | Load data byte or | X | 0 | 1 | 0 | Last data byte will be transmitted and NOT ACK should be received |
| | | Load data byte | X | 0 | 1 | 1 | Data byte will be transmitted and ACK should be received |
| 0xC0 | Data byte in TWDR has been transmitted; NOT ACK has been received | No TWDR action or | 0 | 0 | 1 | 0 | Switched to the not addressed Slave mode; no recognition of own SLA or GCA |
| | | No TWDR action or | 0 | 0 | 1 | 1 | Switched to the not addressed Slave mode; own SLA will be recognized; GCA will be recognized if TWGCE = “1” |
| | | No TWDR action or | 1 | 0 | 1 | 0 | Switched to the not addressed Slave mode; no recognition of own SLA or GCA; a START condition will be transmitted when the bus becomes free |
| | | No TWDR action | 1 | 0 | 1 | 1 | Switched to the not addressed Slave mode; own SLA will be recognized; GCA will be recognized if TWGCE = “1”; a START condition will be transmitted when the bus becomes free |
| 0xC8 | Last data byte in TWDR has been transmitted (TWEA = “0”); ACK has been received | No TWDR action or | 0 | 0 | 1 | 0 | Switched to the not addressed Slave mode; no recognition of own SLA or GCA |
| | | No TWDR action or | 0 | 0 | 1 | 1 | Switched to the not addressed Slave mode; own SLA will be recognized; GCA will be recognized if TWGCE = “1” |
| | | No TWDR action or | 1 | 0 | 1 | 0 | Switched to the not addressed Slave mode; no recognition of own SLA or GCA; a START condition will be transmitted when the bus becomes free |
| | | No TWDR action | 1 | 0 | 1 | 1 | Switched to the not addressed Slave mode; own SLA will be recognized; GCA will be recognized if TWGCE = “1”; a START condition will be transmitted when the bus becomes free |

**Figure 22-18.** Formats and States in the Slave Transmitter Mode


Reception of the own slave address and one or more data bytes
S
SLA
R
A
DATA
A
DATA
A̅
P or S
$A8
$B8
$C0
Arbitration lost as master and addressed as slave
A
$B0
Last data byte transmitted. Switched to not addressed slave (TWEA = '0')
A
All 1's
P or S
$C8
From master to slave
From slave to master
DATA
A
Any number of data bytes and their associated acknowledge bits
n
This number (contained in TWSR) corresponds to a defined state of the 2-Wire Serial Bus. The prescaler bits are zero or masked to zero

### 22.7.5 Miscellaneous States

There are two status codes that do not correspond to a defined TWI state, see Table 22-7.

Status 0xF8 indicates that no relevant information is available because the TWINT Flag is not set. This occurs between other states, and when the TWI is not involved in a serial transfer.

Status 0x00 indicates that a bus error has occurred during a 2-wire Serial Bus transfer. A bus error occurs when a START or STOP condition occurs at an illegal position in the format frame. Examples of such illegal positions are during the serial transfer of an address byte, a data byte, or an acknowledge bit. When a bus error occurs, TWINT is set. To recover from a bus error, the TWSTO Flag must set and TWINT must be cleared by writing a logic one to it. This causes the TWI to enter the not addressed Slave mode and to clear the TWSTO Flag (no other bits in TWCR are affected). The SDA and SCL lines are released, and no STOP condition is transmitted.

**Table 22-7.** Miscellaneous States

| Status Code (TWSR) Prescaler Bits are 0 | Status of the 2-wire Serial Bus and 2-wire Serial Interface Hardware | Application Software Response | | | | | Next Action Taken by TWI Hardware |
|---|---|---|---|---|---|---|---|
| | | | To TWCR | | | | |
| | | To/from TWDR | STA | STO | TWINT | TWEA | |
| 0xF8 | No relevant state information available; TWINT = “0” | No TWDR action | No TWCR action | | | | Wait or proceed current transfer |
| 0x00 | Bus error due to an illegal START or STOP condition | No TWDR action | 0 | 1 | 1 | X | Only the internal hardware is affected, no STOP condition is sent on the bus. In all cases, the bus is released and TWSTO is cleared. |

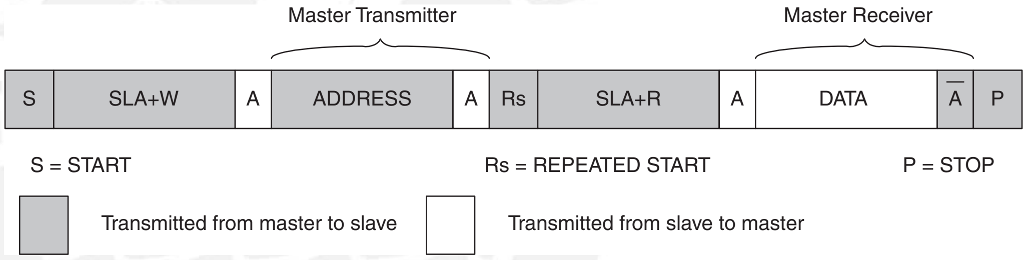
### 22.7.6 Combining Several TWI Modes

In some cases, several TWI modes must be combined in order to complete the desired action. Consider for example reading data from a serial EEPROM. Typically, such a transfer involves the following steps:

1. The transfer must be initiated.
2. The EEPROM must be instructed what location should be read.
3. The reading must be performed.
4. The transfer must be finished.

Note that data is transmitted both from Master to Slave and vice versa. The Master must instruct the Slave what location it wants to read, requiring the use of the MT mode. Subsequently, data must be read from the Slave, implying the use of the MR mode. Thus, the transfer direction must be changed. The Master must keep control of the bus during all these steps, and the steps should be carried out as an atomical operation. If this principle is violated in a multi master system, another Master can alter the data pointer in the EEPROM between steps 2 and 3, and the Master will read the wrong data location. Such a change in transfer direction is accomplished by transmitting a REPEATED START between the transmission of the address byte and reception of the data. After a REPEATED START, the Master keeps ownership of the bus. The following figure shows the flow in this transfer.

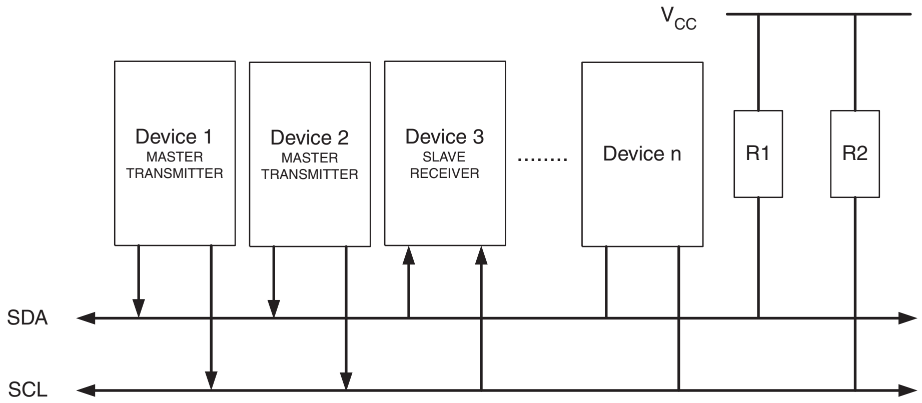
**Figure 22-19.** Combining Several TWI Modes to Access a Serial EEPROM



## 22.8 Multi-master Systems and Arbitration

If multiple masters are connected to the same bus, transmissions may be initiated simultaneously by one or more of them. The TWI standard ensures that such situations are handled in such a way that one of the masters will be allowed to proceed with the transfer, and that no data will be lost in the process. An example of an arbitration situation is depicted below, where two masters are trying to transmit data to a Slave Receiver.

**Figure 22-20.** An Arbitration Example

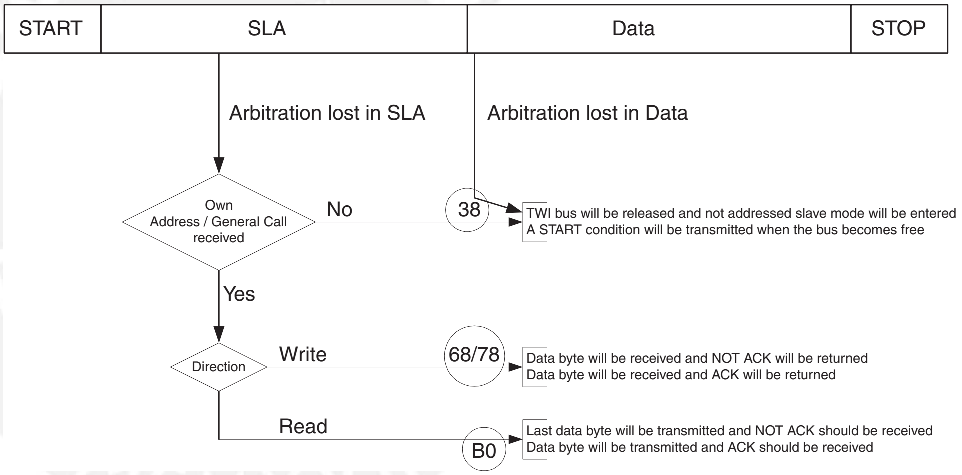


Several different scenarios may arise during arbitration, as described below:

- Two or more masters are performing identical communication with the same Slave. In this case, neither the Slave nor any of the masters will know about the bus contention.
- Two or more masters are accessing the same Slave with different data or direction bit. In this case, arbitration will occur, either in the READ/WRITE bit or in the data bits. The masters trying to output a one on SDA while another Master outputs a zero will lose the arbitration. Losing masters will switch to not addressed Slave mode or wait until the bus is free and transmit a new START condition, depending on application software action.
- Two or more masters are accessing different slaves. In this case, arbitration will occur in the SLA bits. Masters trying to output a one on SDA while another Master outputs a zero will lose the arbitration. Masters losing arbitration in SLA will switch to Slave mode to check if they are being addressed by the winning Master. If addressed, they will switch to SR or ST mode, depending on the value of the READ/WRITE bit. If they are not being addressed, they will switch to not addressed Slave mode or wait until the bus is free and transmit a new START condition, depending on application software action.

This is summarized in Figure 22-21. Possible status values are given in circles.

**Figure 22-21.** Possible Status Codes Caused by Arbitration


START
SLA
Data
STOP
Arbitration lost in SLA
Arbitration lost in Data
Own Address / General Call received
No
38
TWI bus will be released and not addressed slave mode will be entered
A START condition will be transmitted when the bus becomes free
Yes
Direction
Write
68/78
Data byte will be received and NOT ACK will be returned
Data byte will be received and ACK will be returned
Read
B0
Last data byte will be transmitted and NOT ACK should be received
Data byte will be transmitted and ACK should be received

## 22.9 Register Description

### 22.9.1 TWBR – TWI Bit Rate Register

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| (0xB8) | TWBR7 | TWBR6 | TWBR5 | TWBR4 | TWBR3 | TWBR2 | TWBR1 | TWBR0 | TWBR |
| Read/Write | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

- **Bits 7...0 – TWI Bit Rate Register**

TWBR selects the division factor for the bit rate generator. The bit rate generator is a frequency divider which generates the SCL clock frequency in the Master modes. See "Bit Rate Generator Unit" on page 220 for calculating bit rates.

### 22.9.2 TWCR – TWI Control Register

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| (0xBC) | TWINT | TWEA | TWSTA | TWSTO | TWWC | TWEN | – | TWIE | TWCR |
| Read/Write | R/W | R/W | R/W | R/W | R | R/W | R | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

The TWCR is used to control the operation of the TWI. It is used to enable the TWI, to initiate a Master access by applying a START condition to the bus, to generate a Receiver acknowledge, to generate a stop condition, and to control halting of the bus while the data to be written to the bus are written to the TWDR. It also indicates a write collision if data is attempted written to TWDR while the register is inaccessible.

- **Bit 7 – TWINT: TWI Interrupt Flag**

This bit is set by hardware when the TWI has finished its current job and expects application software response. If the I-bit in SREG and TWIE in TWCR are set, the MCU will jump to the TWI Interrupt Vector. While the TWINT Flag is set, the SCL low period is stretched. The TWINT Flag must be cleared by software by writing a logic one to it. Note that this flag is not automatically cleared by hardware when executing the interrupt routine. Also note that clearing this flag starts the operation of the TWI, so all accesses to the TWI Address Register (TWAR), TWI Status Register (TWSR), and TWI Data Register (TWDR) must be complete before clearing this flag.

- **Bit 6 – TWEA: TWI Enable Acknowledge Bit**

The TWEA bit controls the generation of the acknowledge pulse. If the TWEA bit is written to one, the ACK pulse is generated on the TWI bus if the following conditions are met:

1. The device's own slave address has been received.
2. A general call has been received, while the TWGCE bit in the TWAR is set.
3. A data byte has been received in Master Receiver or Slave Receiver mode.

By writing the TWEA bit to zero, the device can be virtually disconnected from the 2-wire Serial Bus temporarily. Address recognition can then be resumed by writing the TWEA bit to one again.

- **Bit 5 – TWSTA: TWI START Condition Bit**

The application writes the TWSTA bit to one when it desires to become a Master on the 2-wire Serial Bus. The TWI hardware checks if the bus is available, and generates a START condition on the bus if it is free. However, if the bus is not free, the TWI waits until a STOP condition is detected, and then generates a new START condition to claim the bus Master status. TWSTA must be cleared by software when the START condition has been transmitted.

- **Bit 4 – TWSTO: TWI STOP Condition Bit**

Writing the TWSTO bit to one in Master mode will generate a STOP condition on the 2-wire Serial Bus. When the STOP condition is executed on the bus, the TWSTO bit is cleared automatically. In Slave mode, setting the TWSTO bit can be used to recover from an error condition. This will not generate a STOP condition, but the TWI returns to a well-defined unaddressed Slave mode and releases the SCL and SDA lines to a high impedance state.

- **Bit 3 – TWWC: TWI Write Collision Flag**

The TWWC bit is set when attempting to write to the TWI Data Register – TWDR when TWINT is low. This flag is cleared by writing the TWDR Register when TWINT is high.

- **Bit 2 – TWEN: TWI Enable Bit**

The TWEN bit enables TWI operation and activates the TWI interface. When TWEN is written to one, the TWI takes control over the I/O pins connected to the SCL and SDA pins, enabling the slew-rate limiters and spike filters. If this bit is written to zero, the TWI is switched off and all TWI transmissions are terminated, regardless of any ongoing operation.

- **Bit 1 – Reserved**

This bit is a reserved bit and will always read as zero.

- **Bit 0 – TWIE: TWI Interrupt Enable**

When this bit is written to one, and the I-bit in SREG is set, the TWI interrupt request will be activated for as long as the TWINT Flag is high.

### 22.9.3 TWSR – TWI Status Register

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| (0xB9) | TWS7 | TWS6 | TWS5 | TWS4 | TWS3 | – | TWPS1 | TWPS0 | TWSR |
| Read/Write | R | R | R | R | R | R | R/W | R/W | |
| Initial Value | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | |

- **Bits 7:3 – TWS: TWI Status**

These 5 bits reflect the status of the TWI logic and the 2-wire Serial Bus. The different status codes are described later in this section. Note that the value read from TWSR contains both the 5-bit status value and the 2-bit prescaler value. The application designer should mask the prescaler bits to zero when checking the Status bits. This makes status checking independent of prescaler setting. This approach is used in this datasheet, unless otherwise noted.

- **Bit 2 – Reserved**

This bit is reserved and will always read as zero.

- **Bits 1:0 – TWPS: TWI Prescaler Bits**

These bits can be read and written, and control the bit rate prescaler.

**Table 22-8.** TWI Bit Rate Prescaler

| TWPS1 | TWPS0 | Prescaler Value |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 4 |
| 1 | 0 | 16 |
| 1 | 1 | 64 |

To calculate bit rates, see "Bit Rate Generator Unit" on page 220. The value of TWPS1...0 is used in the equation.

### 22.9.4 TWDR – TWI Data Register

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
| (0xBB) | TWD7 | TWD6 | TWD5 | TWD4 | TWD3 | TWD2 | TWD1 | TWD0 | TWDR |
| Read/Write | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | |
| Initial Value | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | |

In Transmit mode, TWDR contains the next byte to be transmitted. In Receive mode, the TWDR contains the last byte received. It is writable while the TWI is not in the process of shifting a byte. This occurs when the TWI Interrupt Flag (TWINT) is set by hardware. Note that the Data Register cannot be initialized by the user before the first interrupt occurs. The data in TWDR remains stable as long as TWINT is set. While data is shifted out, data on the bus is simultaneously shifted in. TWDR always contains the last byte present on the bus, except after a wake up from a sleep mode by the TWI interrupt. In this case, the contents of TWDR is undefined.

In the case of a lost bus arbitration, no data is lost in the transition from Master to Slave. Handling of the ACK bit is controlled automatically by the TWI logic, the CPU cannot access the ACK bit directly.

- **Bits 7:0 – TWD: TWI Data Register**

These eight bits constitute the next data byte to be transmitted, or the latest data byte received on the 2-wire Serial Bus.

### 22.9.5 TWAR – TWI (Slave) Address Register

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| (0xBA) | TWA6 | TWA5 | TWA4 | TWA3 | TWA2 | TWA1 | TWA0 | TWGCE | TWAR |
| Read/Write | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | |
| Initial Value | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | |

The TWAR should be loaded with the 7-bit Slave address (in the seven most significant bits of TWAR) to which the TWI will respond when programmed as a Slave Transmitter or Receiver, and not needed in the Master modes. In multi master systems, TWAR must be set in masters which can be addressed as Slaves by other Masters.

The LSB of TWAR is used to enable recognition of the general call address (0x00). There is an associated address comparator that looks for the slave address (or general call address if enabled) in the received serial address. If a match is found, an interrupt request is generated.

- **Bits 7:1 – TWA: TWI (Slave) Address Register**

These seven bits constitute the slave address of the TWI unit.

- **Bit 0 – TWGCE: TWI General Call Recognition Enable Bit**

If set, this bit enables the recognition of a General Call given over the 2-wire Serial Bus.

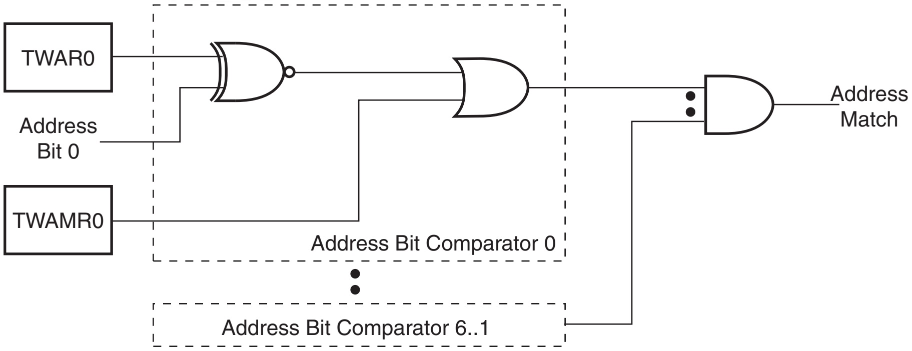
### 22.9.6 TWAMR – TWI (Slave) Address Mask Register

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| (0xBD) | TWAM[6:0] | | | | | | | – | TWAMR |
| Read/Write | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

- **Bits 7:1 – TWAM: TWI Address Mask**

The TWAMR can be loaded with a 7-bit Salve Address mask. Each of the bits in TWAMR can mask (disable) the corresponding address bits in the TWI Address Register (TWAR). If the mask bit is set to one then the address match logic ignores the compare between the incoming address bit and the corresponding bit in TWAR. Figure 22-22 shown the address match logic in detail.

**Figure 22-22.** TWI Address Match Logic, Block Diagram



- **Bit 0 – Reserved**

This bit is an unused bit in the Atmel® ATmega48PA/88PA/168PA, and will always read as zero.

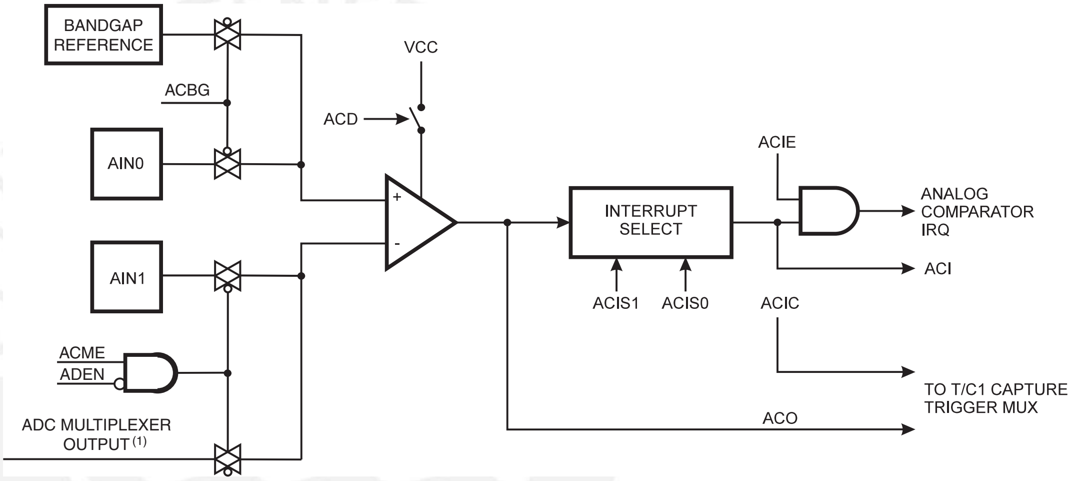
# 23. Analog Comparator

## 23.1 Overview

The Analog Comparator compares the input values on the positive pin AIN0 and negative pin AIN1. When the voltage on the positive pin AIN0 is higher than the voltage on the negative pin AIN1, the Analog Comparator output, ACO, is set. The comparator's output can be set to trigger the Timer/Counter1 Input Capture function. In addition, the comparator can trigger a separate interrupt, exclusive to the Analog Comparator. The user can select Interrupt triggering on comparator output rise, fall or toggle. A block diagram of the comparator and its surrounding logic is shown in Figure 23-1.

The Power Reduction ADC bit, PRADC, in "Minimizing Power Consumption" on page 42 must be disabled by writing a logical zero to be able to use the ADC input MUX.

**Figure 23-1.** Analog Comparator Block Diagram[2]



Notes: 1. See Table 23-1 on page 245.

2. Refer to Figure 1-1 on page 2 and Table 14-9 on page 88 for Analog Comparator pin placement.

## 23.2 Analog Comparator Multiplexed Input

It is possible to select any of the ADC7...0 pins to replace the negative input to the Analog Comparator. The ADC multiplexer is used to select this input, and consequently, the ADC must be switched off to utilize this feature. If the Analog Comparator Multiplexer Enable bit (ACME in ADCSRB) is set and the ADC is switched off (ADEN in ADCSRA is zero), MUX2...0 in ADMUX select the input pin to replace the negative input to the Analog Comparator, as shown in Table 23-1. If ACME is cleared or ADEN is set, AIN1 is applied to the negative input to the Analog Comparator

**Table 23-1.** Analog Comparator Multiplexed Input

| ACME | ADEN | MUX2...0 | Analog Comparator Negative Input |
|---|---|---|---|
| 0 | x | xxx | AIN1 |
| 1 | 1 | xxx | AIN1 |
| 1 | 0 | 000 | ADC0 |
| 1 | 0 | 001 | ADC1 |
| 1 | 0 | 010 | ADC2 |
| 1 | 0 | 011 | ADC3 |
| 1 | 0 | 100 | ADC4 |
| 1 | 0 | 101 | ADC5 |
| 1 | 0 | 110 | ADC6 |
| 1 | 0 | 111 | ADC7 |

## 23.3 Register Description

### 23.3.1 ADCSRB – ADC Control and Status Register B

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| (0x7B) | – | ACME | – | – | – | ADTS2 | ADTS1 | ADTS0 | ADCSRB |
| Read/Write | R | R/W | R | R | R | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

- **Bit 6 – ACME: Analog Comparator Multiplexer Enable**

When this bit is written logic one and the ADC is switched off (ADEN in ADCSRA is zero), the ADC multiplexer selects the negative input to the Analog Comparator. When this bit is written logic zero, AIN1 is applied to the negative input of the Analog Comparator. For a detailed description of this bit, see "Analog Comparator Multiplexed Input" on page 244.

### 23.3.2 ACSR – Analog Comparator Control and Status Register

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| 0x30 (0x50) | ACD | ACBG | ACO | ACI | ACIE | ACIC | ACIS1 | ACIS0 | ACSR |
| Read/Write | R/W | R/W | R | R/W | R/W | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | N/A | 0 | 0 | 0 | 0 | 0 | |

- **Bit 7 – ACD: Analog Comparator Disable**

When this bit is written logic one, the power to the Analog Comparator is switched off. This bit can be set at any time to turn off the Analog Comparator. This will reduce power consumption in Active and Idle mode. When changing the ACD bit, the Analog Comparator Interrupt must be disabled by clearing the ACIE bit in ACSR. Otherwise an interrupt can occur when the bit is changed.

- **Bit 6 – ACBG: Analog Comparator Bandgap Select**

When this bit is set, a fixed bandgap reference voltage replaces the positive input to the Analog Comparator. When this bit is cleared, AIN0 is applied to the positive input of the Analog Comparator. When the bandgap reference is used as input to the Analog Comparator, it will take a certain time for the voltage to stabilize. If not stabilized, the first conversion may give a wrong value. See "Internal Voltage Reference" on page 51

- **Bit 5 – ACO: Analog Comparator Output**

The output of the Analog Comparator is synchronized and then directly connected to ACO. The synchronization introduces a delay of 1 - 2 clock cycles.

- **Bit 4 – ACI: Analog Comparator Interrupt Flag**

This bit is set by hardware when a comparator output event triggers the interrupt mode defined by ACIS1 and ACIS0. The Analog Comparator interrupt routine is executed if the ACIE bit is set and the I-bit in SREG is set. ACI is cleared by hardware when executing the corresponding interrupt handling vector. Alternatively, ACI is cleared by writing a logic one to the flag.

- **Bit 3 – ACIE: Analog Comparator Interrupt Enable**

When the ACIE bit is written logic one and the I-bit in the Status Register is set, the Analog Comparator interrupt is activated. When written logic zero, the interrupt is disabled.

- **Bit 2 – ACIC: Analog Comparator Input Capture Enable**

When written logic one, this bit enables the input capture function in Timer/Counter1 to be triggered by the Analog Comparator. The comparator output is in this case directly connected to the input capture front-end logic, making the comparator utilize the noise canceler and edge select features of the Timer/Counter1 Input Capture interrupt. When written logic zero, no connection between the Analog Comparator and the input capture function exists. To make the comparator trigger the Timer/Counter1 Input Capture interrupt, the ICIE1 bit in the Timer Interrupt Mask Register (TIMSK1) must be set.

- **Bits 1, 0 – ACIS1, ACIS0: Analog Comparator Interrupt Mode Select**

These bits determine which comparator events that trigger the Analog Comparator interrupt. The different settings are shown in Table 23-2.

**Table 23-2.** ACIS1/ACIS0 Settings

| ACIS1 | ACIS0 | Interrupt Mode |
|---|---|---|
| 0 | 0 | Comparator Interrupt on Output Toggle. |
| 0 | 1 | Reserved |
| 1 | 0 | Comparator Interrupt on Falling Output Edge. |
| 1 | 1 | Comparator Interrupt on Rising Output Edge. |

When changing the ACIS1/ACIS0 bits, the Analog Comparator Interrupt must be disabled by clearing its Interrupt Enable bit in the ACSR Register. Otherwise an interrupt can occur when the bits are changed.

### 23.3.3 DIDR1 – Digital Input Disable Register 1

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| (0x7F) | – | – | – | – | – | – | AIN1D | AIN0D | DIDR1 |
| Read/Write | R | R | R | R | R | R | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

- **Bit 7:2 – Reserved**

These bits are unused bits in the Atmel® ATmega48PA/88PA/168PA, and will always read as zero.

- **Bit 1, 0 – AIN1D, AIN0D: AIN1, AIN0 Digital Input Disable**

When this bit is written logic one, the digital input buffer on the AIN1/0 pin is disabled. The corresponding PIN Register bit will always read as zero when this bit is set. When an analog signal is applied to the AIN1/0 pin and the digital input from this pin is not needed, this bit should be written logic one to reduce power consumption in the digital input buffer.

# 24. Analog-to-Digital Converter

## 24.1 Features

- **10-bit Resolution**
- **0.5 LSB Integral Non-linearity**
- **± 2 LSB Absolute Accuracy**
- **13 - 260 µs Conversion Time**
- **Up to 76.9 kSPS (Up to 15 kSPS at Maximum Resolution)**
- **6 Multiplexed Single Ended Input Channels**
- **2 Additional Multiplexed Single Ended Input Channels**
- **Temperature Sensor Input Channel**
- **Optional Left Adjustment for ADC Result Readout**
- **0 - $V_{CC}$ ADC Input Voltage Range**
- **Selectable 1.1V ADC Reference Voltage**
- **Free Running or Single Conversion Mode**
- **Interrupt on ADC Conversion Complete**
- **Sleep Mode Noise Canceler**

## 24.2 Overview

The Atmel® ATmega48PA/88PA/168PA features a 10-bit successive approximation ADC. The ADC is connected to an 8-channel Analog Multiplexer which allows eight single-ended voltage inputs constructed from the pins of Port A. The single-ended voltage inputs refer to 0V (GND).

The ADC contains a Sample and Hold circuit which ensures that the input voltage to the ADC is held at a constant level during conversion. A block diagram of the ADC is shown in Figure 24-1 on page 249.

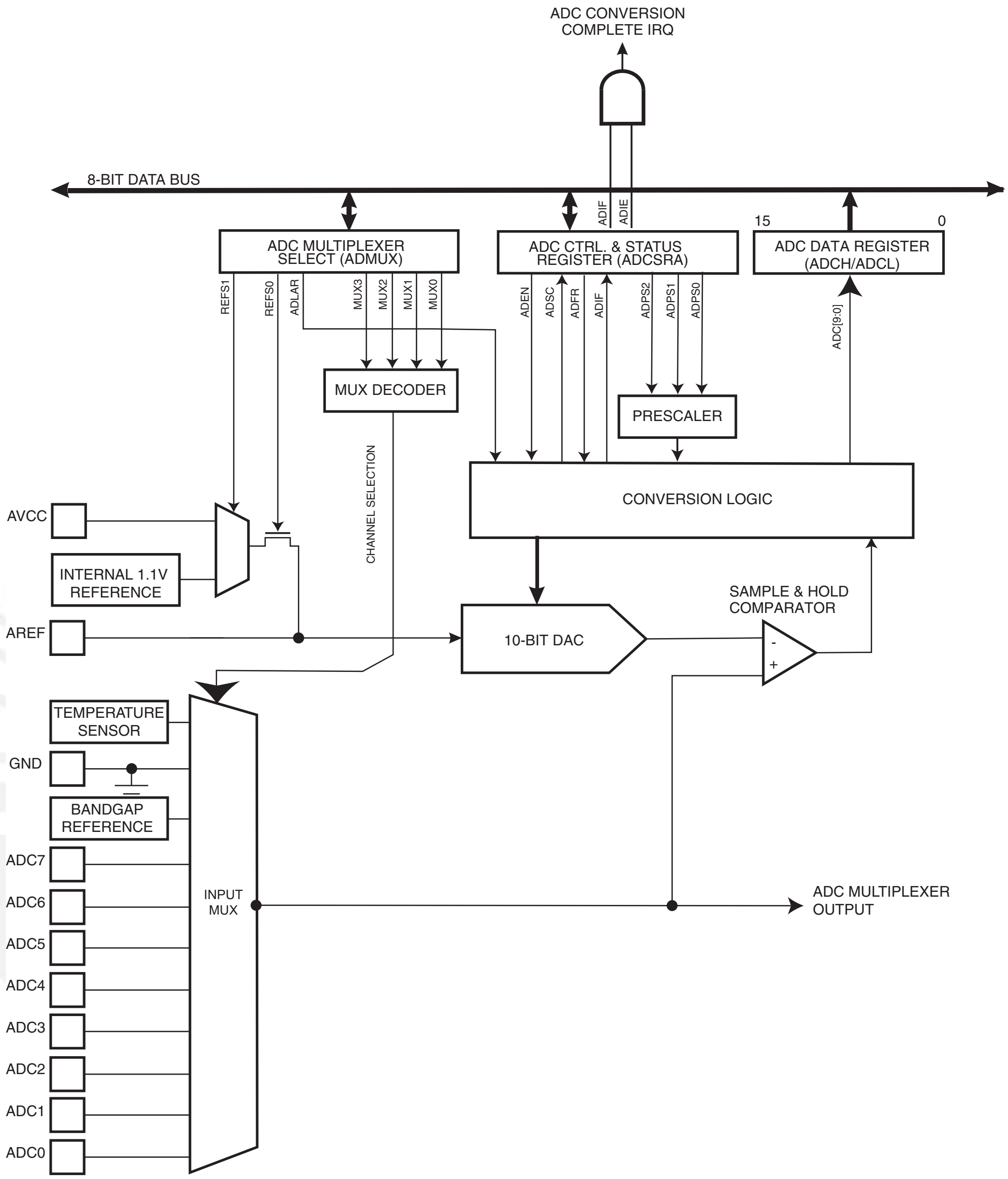
The ADC has a separate analog supply voltage pin, $AV_{CC}$. $AV_{CC}$ must not differ more than ±0.3V from $V_{CC}$. See the paragraph "ADC Noise Canceler" on page 255 on how to connect this pin.

Internal reference voltages of nominally 1.1V or $AV_{CC}$ are provided On-chip. The voltage reference may be externally decoupled at the AREF pin by a capacitor for better noise performance.

The Power Reduction ADC bit, PRADC, in "Minimizing Power Consumption" on page 42 must be disabled by writing a logical zero to enable the ADC.

The ADC converts an analog input voltage to a 10-bit digital value through successive approximation. The minimum value represents GND and the maximum value represents the voltage on the AREF pin minus 1 LSB. Optionally, $AV_{CC}$ or an internal 1.1V reference voltage may be connected to the AREF pin by writing to the REFSn bits in the ADMUX Register. The internal voltage reference may thus be decoupled by an external capacitor at the AREF pin to improve noise immunity.

**Figure 24-1.** Analog to Digital Converter Block Schematic Operation



The analog input channel is selected by writing to the MUX bits in ADMUX. Any of the ADC input pins, as well as GND and a fixed bandgap voltage reference, can be selected as single ended inputs to the ADC. The ADC is enabled by setting the ADC Enable bit, ADEN in ADCSRA. Voltage reference and input channel selections will not go into effect until ADEN is set. The ADC does not consume power when ADEN is cleared, so it is recommended to switch off the ADC before entering power saving sleep modes.

The ADC generates a 10-bit result which is presented in the ADC Data Registers, ADCH and ADCL. By default, the result is presented right adjusted, but can optionally be presented left adjusted by setting the ADLAR bit in ADMUX.

If the result is left adjusted and no more than 8-bit precision is required, it is sufficient to read ADCH. Otherwise, ADCL must be read first, then ADCH, to ensure that the content of the Data Registers belongs to the same conversion. Once ADCL is read, ADC access to Data Registers is blocked. This means that if ADCL has been read, and a conversion completes before ADCH is read, neither register is updated and the result from the conversion is lost. When ADCH is read, ADC access to the ADCH and ADCL Registers is re-enabled.

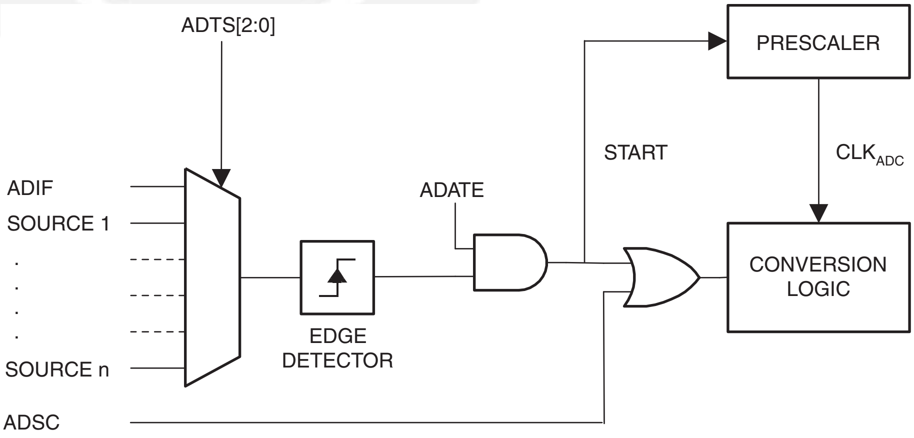
The ADC has its own interrupt which can be triggered when a conversion completes. When ADC access to the Data Registers is prohibited between reading of ADCH and ADCL, the interrupt will trigger even if the result is lost.

## 24.3 Starting a Conversion

A single conversion is started by disabling the Power Reduction ADC bit, PRADC, in "Minimizing Power Consumption" on page 42 by writing a logical zero to it and writing a logical one to the ADC Start Conversion bit, ADSC. This bit stays high as long as the conversion is in progress and will be cleared by hardware when the conversion is completed. If a different data channel is selected while a conversion is in progress, the ADC will finish the current conversion before performing the channel change.

Alternatively, a conversion can be triggered automatically by various sources. Auto Triggering is enabled by setting the ADC Auto Trigger Enable bit, ADATE in ADCSRA. The trigger source is selected by setting the ADC Trigger Select bits, ADTS in ADCSRB (See description of the ADTS bits for a list of the trigger sources). When a positive edge occurs on the selected trigger signal, the ADC prescaler is reset and a conversion is started. This provides a method of starting conversions at fixed intervals. If the trigger signal still is set when the conversion completes, a new conversion will not be started. If another positive edge occurs on the trigger signal during conversion, the edge will be ignored. Note that an Interrupt Flag will be set even if the specific interrupt is disabled or the Global Interrupt Enable bit in SREG is cleared. A conversion can thus be triggered without causing an interrupt. However, the Interrupt Flag must be cleared in order to trigger a new conversion at the next interrupt event.

**Figure 24-2.** ADC Auto Trigger Logic

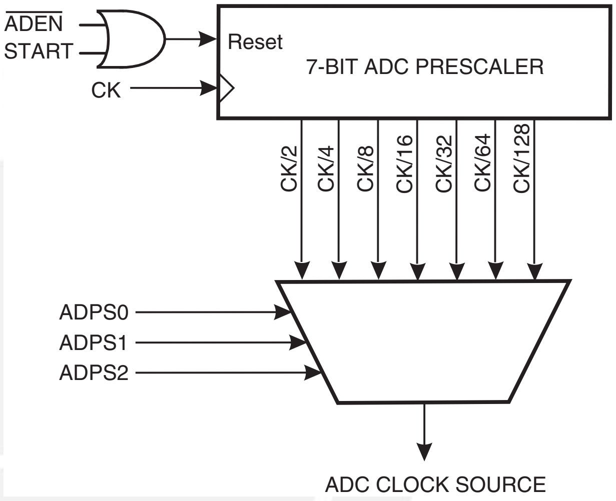


Using the ADC Interrupt Flag as a trigger source makes the ADC start a new conversion as soon as the ongoing conversion has finished. The ADC then operates in Free Running mode, constantly sampling and updating the ADC Data Register. The first conversion must be started by writing a logical one to the ADSC bit in ADCSRA. In this mode the ADC will perform successive conversions independently of whether the ADC Interrupt Flag, ADIF is cleared or not.

If Auto Triggering is enabled, single conversions can be started by writing ADSC in ADCSRA to one. ADSC can also be used to determine if a conversion is in progress. The ADSC bit will be read as one during a conversion, independently of how the conversion was started.

## 24.4 Prescaling and Conversion Timing

**Figure 24-3.** ADC Prescaler



By default, the successive approximation circuitry requires an input clock frequency between 50kHz and 200kHz to get maximum resolution. If a lower resolution than 10 bits is needed, the input clock frequency to the ADC can be higher than 200kHz to get a higher sample rate.

The ADC module contains a prescaler, which generates an acceptable ADC clock frequency from any CPU frequency above 100kHz. The prescaling is set by the ADPS bits in ADCSRA. The prescaler starts counting from the moment the ADC is switched on by setting the ADEN bit in ADCSRA. The prescaler keeps running for as long as the ADEN bit is set, and is continuously reset when ADEN is low.

When initiating a single ended conversion by setting the ADSC bit in ADCSRA, the conversion starts at the following rising edge of the ADC clock cycle.

A normal conversion takes 13 ADC clock cycles. The first conversion after the ADC is switched on (ADEN in ADCSRA is set) takes 25 ADC clock cycles in order to initialize the analog circuitry.

When the bandgap reference voltage is used as input to the ADC, it will take a certain time for the voltage to stabilize. If not stabilized, the first value read after the first conversion may be wrong.

The actual sample-and-hold takes place 1.5 ADC clock cycles after the start of a normal conversion and 13.5 ADC clock cycles after the start of an first conversion. When a conversion is complete, the result is written to the ADC Data Registers, and ADIF is set. In Single Conversion mode, ADSC is cleared simultaneously. The software may then set ADSC again, and a new conversion will be initiated on the first rising ADC clock edge.

When Auto Triggering is used, the prescaler is reset when the trigger event occurs. This assures a fixed delay from the trigger event to the start of conversion. In this mode, the sample-and-hold takes place two ADC clock cycles after the rising edge on the trigger source signal. Three additional CPU clock cycles are used for synchronization logic.

In Free Running mode, a new conversion will be started immediately after the conversion completes, while ADSC remains high. For a summary of conversion times, see Table 24-1 on page 253.

**Figure 24-4.** ADC Timing Diagram, First Conversion (Single Conversion Mode)

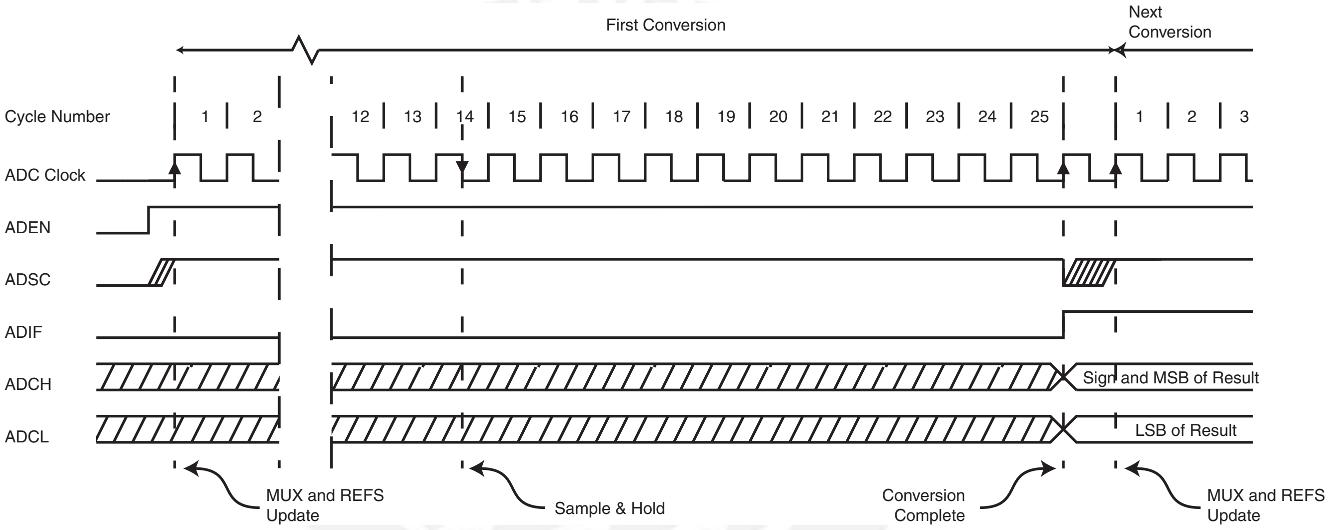


**Figure 24-5.** ADC Timing Diagram, Single Conversion

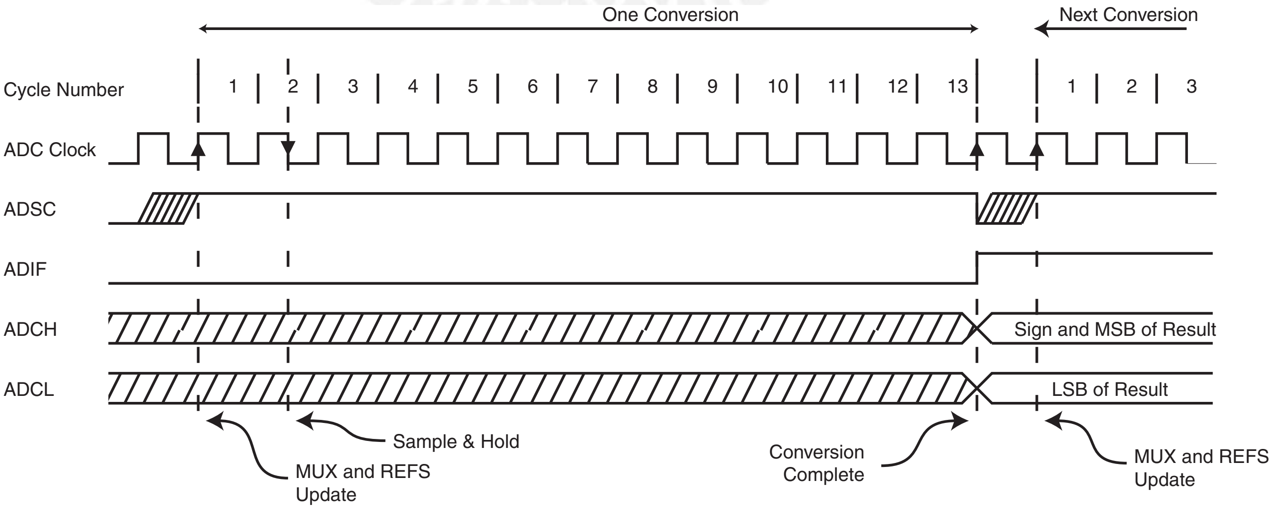


**Figure 24-6.** ADC Timing Diagram, Auto Triggered Conversion

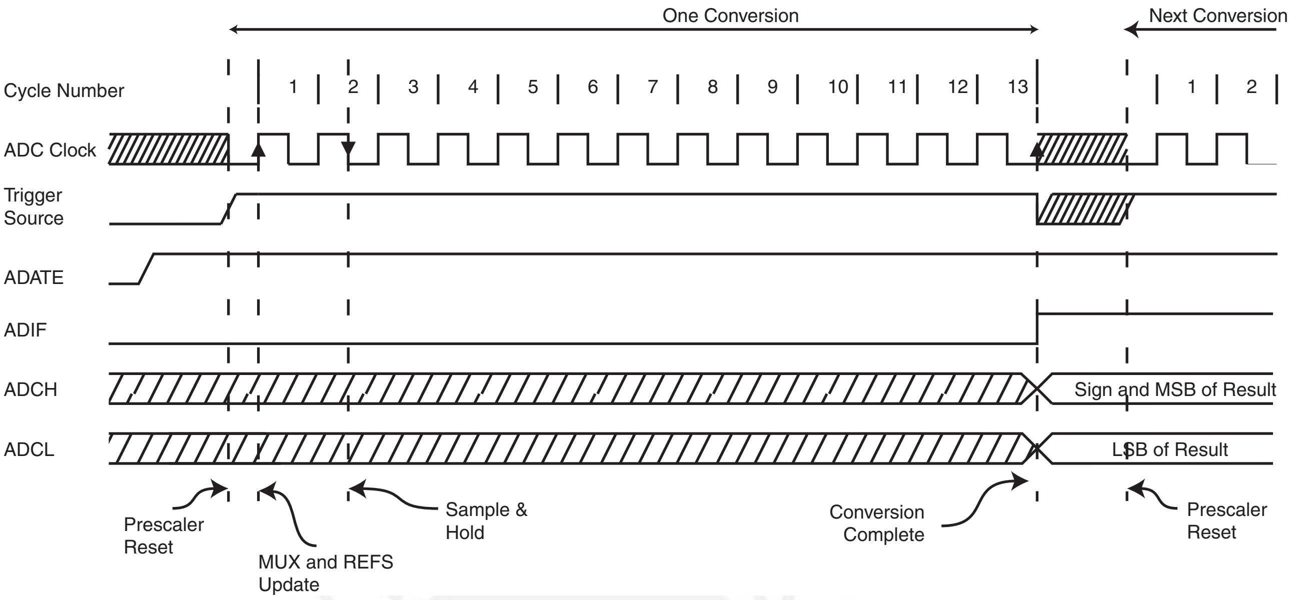


**Figure 24-7.** ADC Timing Diagram, Free Running Conversion

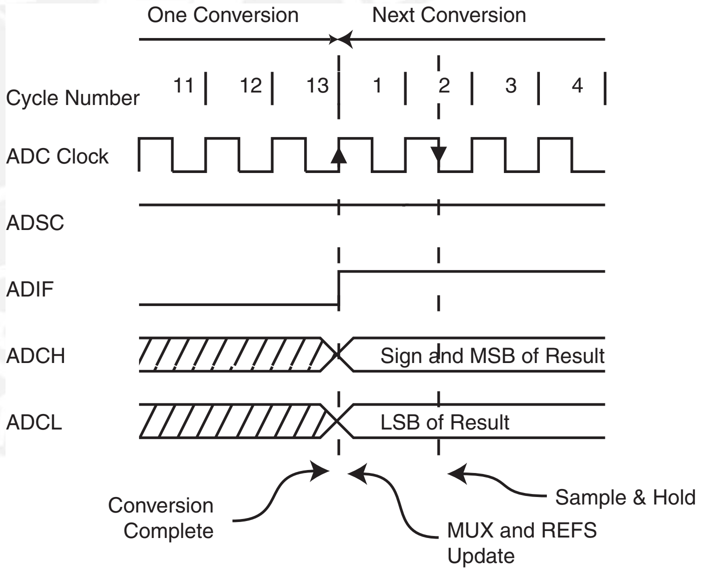


**Table 24-1.** ADC Conversion Time

| Condition | Sample & Hold (Cycles from Start of Conversion) | Conversion Time (Cycles) |
|---|---|---|
| First conversion | 13.5 | 25 |
| Normal conversions, single ended | 1.5 | 13 |
| Auto Triggered conversions | 2 | 13.5 |

## 24.5 Changing Channel or Reference Selection

The MUXn and REFS1:0 bits in the ADMUX Register are single buffered through a temporary register to which the CPU has random access. This ensures that the channels and reference selection only takes place at a safe point during the conversion. The channel and reference selection is continuously updated until a conversion is started. Once the conversion starts, the channel and reference selection is locked to ensure a sufficient sampling time for the ADC. Continuous updating resumes in the last ADC clock cycle before the conversion completes (ADIF in ADCSRA is set). Note that the conversion starts on the following rising ADC clock edge after ADSC is written. The user is thus advised not to write new channel or reference selection values to ADMUX until one ADC clock cycle after ADSC is written.

If Auto Triggering is used, the exact time of the triggering event can be indeterministic. Special care must be taken when updating the ADMUX Register, in order to control which conversion will be affected by the new settings.

If both ADATE and ADEN is written to one, an interrupt event can occur at any time. If the ADMUX Register is changed in this period, the user cannot tell if the next conversion is based on the old or the new settings. ADMUX can be safely updated in the following ways:

a. When ADATE or ADEN is cleared.

b. During conversion, minimum one ADC clock cycle after the trigger event.

c. After a conversion, before the Interrupt Flag used as trigger source is cleared.

When updating ADMUX in one of these conditions, the new settings will affect the next ADC conversion.

### 24.5.1 ADC Input Channels

When changing channel selections, the user should observe the following guidelines to ensure that the correct channel is selected:

In Single Conversion mode, always select the channel before starting the conversion. The channel selection may be changed one ADC clock cycle after writing one to ADSC. However, the simplest method is to wait for the conversion to complete before changing the channel selection.

In Free Running mode, always select the channel before starting the first conversion. The channel selection may be changed one ADC clock cycle after writing one to ADSC. However, the simplest method is to wait for the first conversion to complete, and then change the channel selection. Since the next conversion has already started automatically, the next result will reflect the previous channel selection. Subsequent conversions will reflect the new channel selection.

### 24.5.2 ADC Voltage Reference

The reference voltage for the ADC ($V_{REF}$) indicates the conversion range for the ADC. Single ended channels that exceed $V_{REF}$ will result in codes close to 0x3FF. $V_{REF}$ can be selected as either $AV_{CC}$, internal 1.1V reference, or external AREF pin.

$AV_{CC}$ is connected to the ADC through a passive switch. The internal 1.1V reference is generated from the internal bandgap reference ($V_{BG}$) through an internal amplifier. In either case, the external AREF pin is directly connected to the ADC, and the reference voltage can be made more immune to noise by connecting a capacitor between the AREF pin and ground. $V_{REF}$ can also be measured at the AREF pin with a high impedance voltmeter. Note that $V_{REF}$ is a high impedance source, and only a capacitive load should be connected in a system.

If the user has a fixed voltage source connected to the AREF pin, the user may not use the other reference voltage options in the application, as they will be shorted to the external voltage. If no external voltage is applied to the AREF pin, the user may switch between $AV_{CC}$ and 1.1V as reference selection. The first ADC conversion result after switching reference voltage source may be inaccurate, and the user is advised to discard this result.

## 24.6 ADC Noise Canceler

The ADC features a noise canceler that enables conversion during sleep mode to reduce noise induced from the CPU core and other I/O peripherals. The noise canceler can be used with ADC Noise Reduction and Idle mode. To make use of this feature, the following procedure should be used:

a. Make sure that the ADC is enabled and is not busy converting. Single Conversion mode must be selected and the ADC conversion complete interrupt must be enabled.

b. Enter ADC Noise Reduction mode (or Idle mode). The ADC will start a conversion once the CPU has been halted.

c. If no other interrupts occur before the ADC conversion completes, the ADC interrupt will wake up the CPU and execute the ADC Conversion Complete interrupt routine. If another interrupt wakes up the CPU before the ADC conversion is complete, that interrupt will be executed, and an ADC Conversion Complete interrupt request will be generated when the ADC conversion completes. The CPU will remain in active mode until a new sleep command is executed.

Note that the ADC will not be automatically turned off when entering other sleep modes than Idle mode and ADC Noise Reduction mode. The user is advised to write zero to ADEN before entering such sleep modes to avoid excessive power consumption.

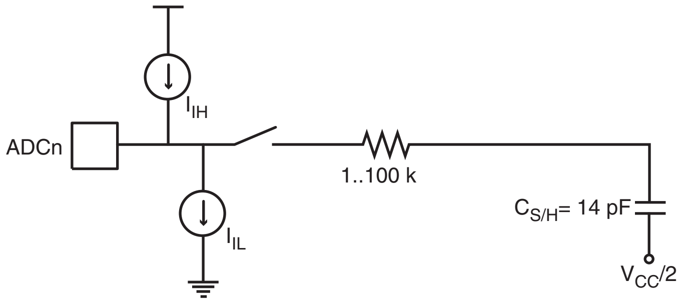
### 24.6.1 Analog Input Circuitry

The analog input circuitry for single ended channels is illustrated in Figure 24-8. An analog source applied to ADCn is subjected to the pin capacitance and input leakage of that pin, regardless of whether that channel is selected as input for the ADC. When the channel is selected, the source must drive the S/H capacitor through the series resistance (combined resistance in the input path).

The ADC is optimized for analog signals with an output impedance of approximately 10 kΩ or less. If such a source is used, the sampling time will be negligible. If a source with higher impedance is used, the sampling time will depend on how long time the source needs to charge the S/H capacitor, with can vary widely. The user is recommended to only use low impedance sources with slowly varying signals, since this minimizes the required charge transfer to the S/H capacitor.

Signal components higher than the Nyquist frequency ($f_{ADC}/2$) should not be present for either kind of channels, to avoid distortion from unpredictable signal convolution. The user is advised to remove high frequency components with a low-pass filter before applying the signals as inputs to the ADC.

**Figure 24-8.** Analog Input Circuitry



### 24.6.2 Analog Noise Canceling Techniques

Digital circuitry inside and outside the device generates EMI which might affect the accuracy of analog measurements. If conversion accuracy is critical, the noise level can be reduced by applying the following techniques:

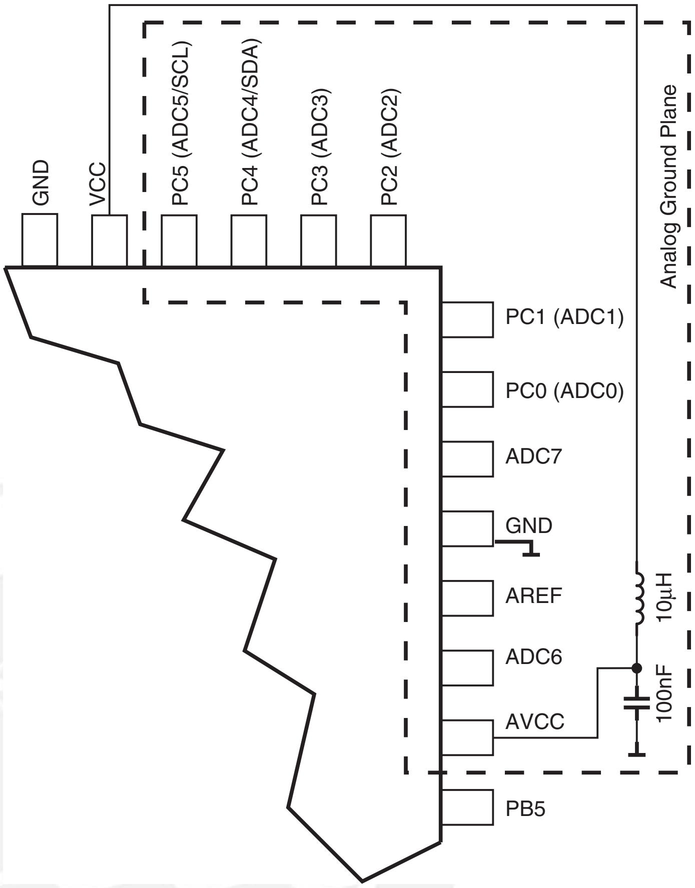
a. Keep analog signal paths as short as possible. Make sure analog tracks run over the analog ground plane, and keep them well away from high-speed switching digital tracks.

b. The $AV_{CC}$ pin on the device should be connected to the digital $V_{CC}$ supply voltage via an LC network as shown in Figure 24-9 on page 257.

c. Use the ADC noise canceler function to reduce induced noise from the CPU.

d. If any ADC [3...0] port pins are used as digital outputs, it is essential that these do not switch while a conversion is in progress. However, using the 2-wire Interface (ADC4 and ADC5) will only affect the conversion on ADC4 and ADC5 and not the other ADC channels.

**Figure 24-9.** ADC Power Connections



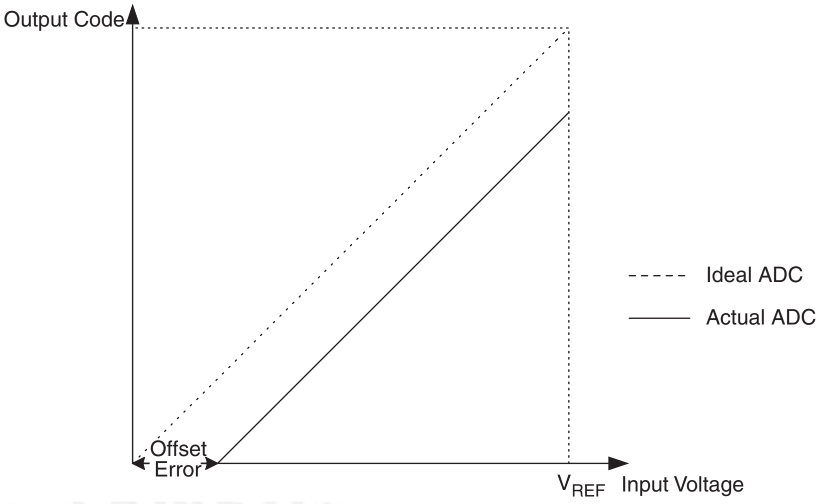
### 24.6.3 ADC Accuracy Definitions

An n-bit single-ended ADC converts a voltage linearly between GND and $V_{REF}$ in $2^n$ steps (LSBs). The lowest code is read as 0, and the highest code is read as $2^n$-1.

Several parameters describe the deviation from the ideal behavior:

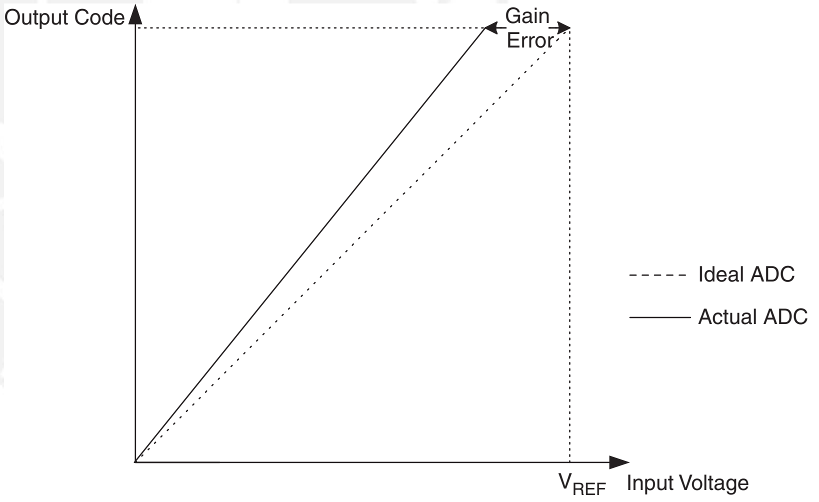
- Offset: The deviation of the first transition (0x000 to 0x001) compared to the ideal transition (at 0.5 LSB). Ideal value: 0 LSB.

**Figure 24-10.** Offset Error



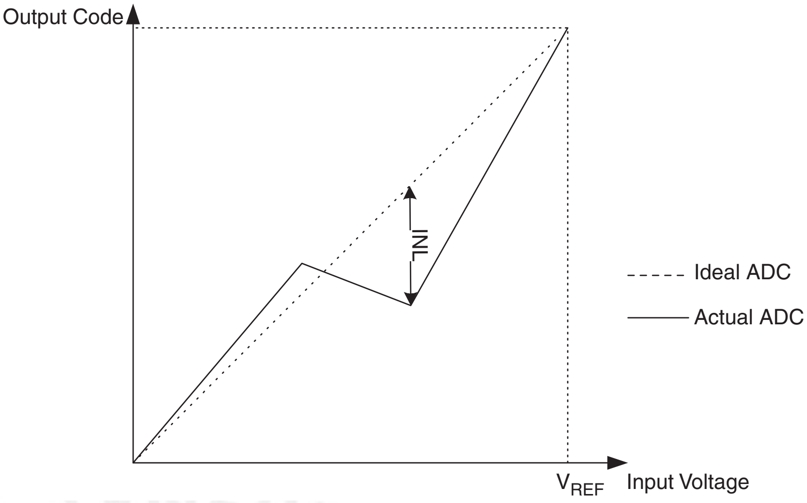
- Gain error: After adjusting for offset, the gain error is found as the deviation of the last transition (0x3FE to 0x3FF) compared to the ideal transition (at 1.5 LSB below maximum). Ideal value: 0 LSB

**Figure 24-11.** Gain Error



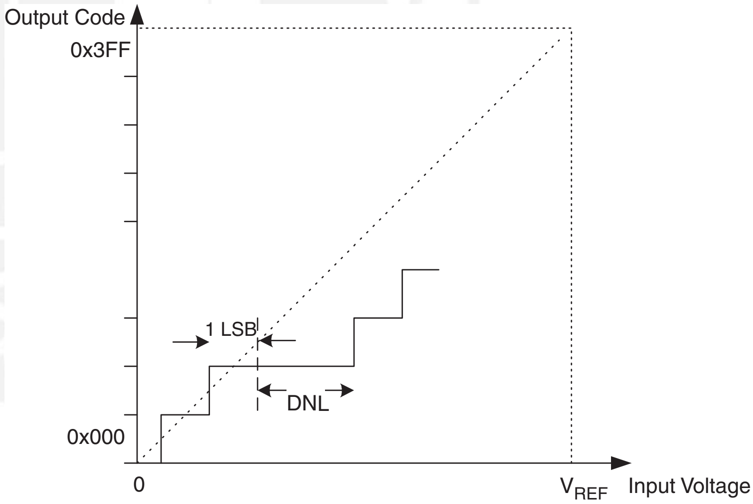
- Integral Non-linearity (INL): After adjusting for offset and gain error, the INL is the maximum deviation of an actual transition compared to an ideal transition for any code. Ideal value: 0 LSB.

**Figure 24-12.** Integral Non-linearity (INL)



- Differential Non-linearity (DNL): The maximum deviation of the actual code width (the interval between two adjacent transitions) from the ideal code width (1 LSB). Ideal value: 0 LSB.

**Figure 24-13.** Differential Non-linearity (DNL)



- Quantization Error: Due to the quantization of the input voltage into a finite number of codes, a range of input voltages (1 LSB wide) will code to the same value. Always ±0.5 LSB.
- Absolute accuracy: The maximum deviation of an actual (unadjusted) transition compared to an ideal transition for any code. This is the compound effect of offset, gain error, differential error, non-linearity, and quantization error. Ideal value: ±0.5 LSB.

## 24.7 ADC Conversion Result

After the conversion is complete (ADIF is high), the conversion result can be found in the ADC Result Registers (ADCL, ADCH).

For single ended conversion, the result is

$$ADC = \frac{V_{IN} \times 1024}{V_{REF}}$$

where $V_{IN}$ is the voltage on the selected input pin and $V_{REF}$ the selected voltage reference (see Table 24-3 on page 262 and Table 24-4 on page 263). 0x000 represents analog ground, and 0x3FF represents the selected reference voltage minus one LSB.

## 24.8 Temperature Measurement

The temperature measurement is based on an on-chip temperature sensor that is coupled to a single ended ADC input. 1000 setting in MUX[3..0] bits of ADMUX register selects the temperature sensor. The internal 1.1V voltage reference, a 11 setting in REFS[1..0] of ADMUX, must also be selected for the ADC voltage reference source during the temperature sensor measurement. When the temperature sensor is enabled, the ADC converter can be used in single conversion mode to measure the voltage over the temperature sensor.

The measured voltage has a linear relationship to the temperature as described in Table 24-2.

The voltage sensitivity is approximately 1LSB/°C (142/128) and the accuracy of the temperature measurement is ±20°C using the manufacturing calibration offset value (TS_ADC_25[H..L]).

The values described in Table 24-2 are typical values. However, due to the process variation the temperature sensor output varies from one chip to another.

**Table 24-2.** Sensor Output Code versus Temperature (Typical Values)

| Temperature/°C | –40°C | +25°C | +125°C |
|---|---|---|---|
| | 0x010D | 0x0160 | 0x01E0 |

### 24.8.1 Manufacturing Calibration

Calibration values determined during test are available in the signature row.

The temperature in degrees Celsius can be calculated using the formula:

$$\frac{(((\mathrm{ADCH}{<<}8) + \mathrm{ADCL}) - ((\mathrm{TS\_ADC\_25\_H}{<<}8) + \mathrm{TS\_ADC\_25\_L}) \times 128)}{142} + 25$$

Where:.

a. ADCH & ADCL are the ADC data register values obtained during temperature sensor reading.

b. TS_ADC_25_H and _L is the 10-bit ADC temp sensor reading stored as two byte values during factory calibration at 25°C in the signature row.

c. The ratio 128/142 is the design compensation factor for the temperature sensor gain, as the temp sensor is slightly more sensitive than 1°C/unit.

d. The +25 is the offset compensation for the fact that the calibration values are obtained at +25°C during factory calibration.

See Section 27.8.10 “Reading the Signature Row from Software” on page 286 and Table 27-5 on page 286 for signature row access and parameter addresses.

## 24.9 Register Description

### 24.9.1 ADMUX – ADC Multiplexer Selection Register

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| (0x7C) | REFS1 | REFS0 | ADLAR | – | MUX3 | MUX2 | MUX1 | MUX0 | ADMUX |
| Read/Write | R/W | R/W | R/W | R | R/W | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

- **Bit 7:6 – REFS[1:0]: Reference Selection Bits**

These bits select the voltage reference for the ADC, as shown in Table 24-3. If these bits are changed during a conversion, the change will not go in effect until this conversion is complete (ADIF in ADCSRA is set). The internal voltage reference options may not be used if an external reference voltage is being applied to the AREF pin.

**Table 24-3.** Voltage Reference Selections for ADC

| REFS1 | REFS0 | Voltage Reference Selection |
|---|---|---|
| 0 | 0 | AREF, Internal $V_{ref}$ turned off |
| 0 | 1 | $AV_{CC}$ with external capacitor[1] at AREF pin |
| 1 | 0 | Reserved |
| 1 | 1 | Internal 1.1V Voltage Reference with external capacitor[1] at AREF pin |

Note: 1. Note the value used for the external ARef capacitor (e.g. 10nF) should be very much smaller than the decoupling capacitor used on the AVcc pin (e.g. 100nF) to prevent possible switching glitches.

- **Bit 5 – ADLAR: ADC Left Adjust Result**

The ADLAR bit affects the presentation of the ADC conversion result in the ADC Data Register. Write one to ADLAR to left adjust the result. Otherwise, the result is right adjusted. Changing the ADLAR bit will affect the ADC Data Register immediately, regardless of any ongoing conversions. For a complete description of this bit, see "ADCL and ADCH – The ADC Data Register" on page 265.

- **Bit 4 – Reserved**

This bit is an unused bit in the Atmel® ATmega48PA/88PA/168PA, and will always read as zero.

- **Bits 3:0 – MUX[3:0]: Analog Channel Selection Bits**

The value of these bits selects which analog inputs are connected to the ADC. See Table 24-4 for details. If these bits are changed during a conversion, the change will not go in effect until this conversion is complete (ADIF in ADCSRA is set).

**Table 24-4.** Input Channel Selections

| MUX3...0 | Single Ended Input |
|---|---|
| 0000 | ADC0 |
| 0001 | ADC1 |
| 0010 | ADC2 |
| 0011 | ADC3 |
| 0100 | ADC4 |
| 0101 | ADC5 |
| 0110 | ADC6 |
| 0111 | ADC7 |
| 1000 | ADC8[1] |
| 1001 | (reserved) |
| 1010 | (reserved) |
| 1011 | (reserved) |
| 1100 | (reserved) |
| 1101 | (reserved) |
| 1110 | 1.1V ($V_{BG}$) |
| 1111 | 0V (GND) |

Note: 1. For Temperature Sensor.

### 24.9.2 ADCSRA – ADC Control and Status Register A

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
| (0x7A) | ADEN | ADSC | ADATE | ADIF | ADIE | ADPS2 | ADPS1 | ADPS0 | ADCSRA |
| Read/Write | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

- **Bit 7 – ADEN: ADC Enable**

Writing this bit to one enables the ADC. By writing it to zero, the ADC is turned off. Turning the ADC off while a conversion is in progress, will terminate this conversion.

- **Bit 6 – ADSC: ADC Start Conversion**

In Single Conversion mode, write this bit to one to start each conversion. In Free Running mode, write this bit to one to start the first conversion. The first conversion after ADSC has been written after the ADC has been enabled, or if ADSC is written at the same time as the ADC is enabled, will take 25 ADC clock cycles instead of the normal 13. This first conversion performs initialization of the ADC.

ADSC will read as one as long as a conversion is in progress. When the conversion is complete, it returns to zero. Writing zero to this bit has no effect.

- **Bit 5 – ADATE: ADC Auto Trigger Enable**

When this bit is written to one, Auto Triggering of the ADC is enabled. The ADC will start a conversion on a positive edge of the selected trigger signal. The trigger source is selected by setting the ADC Trigger Select bits, ADTS in ADCSRB.

- **Bit 4 – ADIF: ADC Interrupt Flag**

This bit is set when an ADC conversion completes and the Data Registers are updated. The ADC Conversion Complete Interrupt is executed if the ADIE bit and the I-bit in SREG are set. ADIF is cleared by hardware when executing the corresponding interrupt handling vector. Alternatively, ADIF is cleared by writing a logical one to the flag. Beware that if doing a Read-Modify-Write on ADCSRA, a pending interrupt can be disabled. This also applies if the SBI and CBI instructions are used.

- **Bit 3 – ADIE: ADC Interrupt Enable**

When this bit is written to one and the I-bit in SREG is set, the ADC Conversion Complete Interrupt is activated.

- **Bits 2:0 – ADPS[2:0]: ADC Prescaler Select Bits**

These bits determine the division factor between the system clock frequency and the input clock to the ADC.

**Table 24-5.** ADC Prescaler Selections

| ADPS2 | ADPS1 | ADPS0 | Division Factor |
|---|---|---|---|
| 0 | 0 | 0 | 2 |
| 0 | 0 | 1 | 2 |
| 0 | 1 | 0 | 4 |
| 0 | 1 | 1 | 8 |
| 1 | 0 | 0 | 16 |
| 1 | 0 | 1 | 32 |
| 1 | 1 | 0 | 64 |
| 1 | 1 | 1 | 128 |

### 24.9.3 ADCL and ADCH – The ADC Data Register

#### *24.9.3.1 ADLAR = 0*

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Bit | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | |
| (0x79) | – | – | – | – | – | – | ADC9 | ADC8 | ADCH |
| (0x78) | ADC7 | ADC6 | ADC5 | ADC4 | ADC3 | ADC2 | ADC1 | ADC0 | ADCL |
| | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
| Read/Write | R | R | R | R | R | R | R | R | |
| | R | R | R | R | R | R | R | R | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

#### *24.9.3.2 ADLAR = 1*

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Bit | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | |
| (0x79) | ADC9 | ADC8 | ADC7 | ADC6 | ADC5 | ADC4 | ADC3 | ADC2 | ADCH |
| (0x78) | ADC1 | ADC0 | – | – | – | – | – | – | ADCL |
| | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
| Read/Write | R | R | R | R | R | R | R | R | |
| | R | R | R | R | R | R | R | R | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

When an ADC conversion is complete, the result is found in these two registers.

When ADCL is read, the ADC Data Register is not updated until ADCH is read. Consequently, if the result is left adjusted and no more than 8-bit precision is required, it is sufficient to read ADCH. Otherwise, ADCL must be read first, then ADCH.

The ADLAR bit in ADMUX, and the MUXn bits in ADMUX affect the way the result is read from the registers. If ADLAR is set, the result is left adjusted. If ADLAR is cleared (default), the result is right adjusted.

- **ADC9:0: ADC Conversion Result**

These bits represent the result from the conversion, as detailed in "ADC Conversion Result" on page 260.

### 24.9.4 ADCSRB – ADC Control and Status Register B

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
| (0x7B) | – | ACME | – | – | – | ADTS2 | ADTS1 | ADTS0 | ADCSRB |
| Read/Write | R | R/W | R | R | R | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

- **Bit 7, 5:3 – Reserved**

These bits are reserved for future use. To ensure compatibility with future devices, these bits must be written to zero when ADCSRB is written.

- **Bit 2:0 – ADTS[2:0]: ADC Auto Trigger Source**

If ADATE in ADCSRA is written to one, the value of these bits selects which source will trigger an ADC conversion. If ADATE is cleared, the ADTS[2:0] settings will have no effect. A conversion will be triggered by the rising edge of the selected Interrupt Flag. Note that switching from a trigger source that is cleared to a trigger source that is set, will generate a positive edge on the trigger signal. If ADEN in ADCSRA is set, this will start a conversion. Switching to Free Running mode (ADTS[2:0]=0) will not cause a trigger event, even if the ADC Interrupt Flag is set.

**Table 24-6.** ADC Auto Trigger Source Selections

| ADTS2 | ADTS1 | ADTS0 | Trigger Source |
|---|---|---|---|
| 0 | 0 | 0 | Free Running mode |
| 0 | 0 | 1 | Analog Comparator |
| 0 | 1 | 0 | External Interrupt Request 0 |
| 0 | 1 | 1 | Timer/Counter0 Compare Match A |
| 1 | 0 | 0 | Timer/Counter0 Overflow |
| 1 | 0 | 1 | Timer/Counter1 Compare Match B |
| 1 | 1 | 0 | Timer/Counter1 Overflow |
| 1 | 1 | 1 | Timer/Counter1 Capture Event |

### 24.9.5 DIDR0 – Digital Input Disable Register 0

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| (0x7E) | – | – | ADC5D | ADC4D | ADC3D | ADC2D | ADC1D | ADC0D | DIDR0 |
| Read/Write | R | R | R/W | R/W | R/W | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

- **Bits 7:6 – Reserved**

These bits are reserved for future use. To ensure compatibility with future devices, these bits must be written to zero when DIDR0 is written.

- **Bit 5:0 – ADC5D...ADC0D: ADC5...0 Digital Input Disable**

When this bit is written logic one, the digital input buffer on the corresponding ADC pin is disabled. The corresponding PIN Register bit will always read as zero when this bit is set. When an analog signal is applied to the ADC5...0 pin and the digital input from this pin is not needed, this bit should be written logic one to reduce power consumption in the digital input buffer.

Note that ADC pins ADC7 and ADC6 do not have digital input buffers, and therefore do not require Digital Input Disable bits.

# 25. debugWIRE On-chip Debug System

## 25.1 Features

- **Complete Program Flow Control**
- **Emulates All On-chip Functions, Both Digital and Analog, except RESET Pin**
- **Real-time Operation**
- **Symbolic Debugging Support (Both at C and Assembler Source Level, or for Other HLLs)**
- **Unlimited Number of Program Break Points (Using Software Break Points)**
- **Non-intrusive Operation**
- **Electrical Characteristics Identical to Real Device**
- **Automatic Configuration System**
- **High-Speed Operation**
- **Programming of Non-volatile Memories**

## 25.2 Overview

The debugWIRE On-chip debug system uses a One-wire, bi-directional interface to control the program flow, execute AVR instructions in the CPU and to program the different non-volatile memories.

## 25.3 Physical Interface

When the debugWIRE Enable (DWEN) Fuse is programmed and Lock bits are unprogrammed, the debugWIRE system within the target device is activated. The RESET port pin is configured as a wire-AND (open-drain) bi-directional I/O pin with pull-up enabled and becomes the communication gateway between target and emulator.

**Figure 25-1.** The debugWIRE Setup

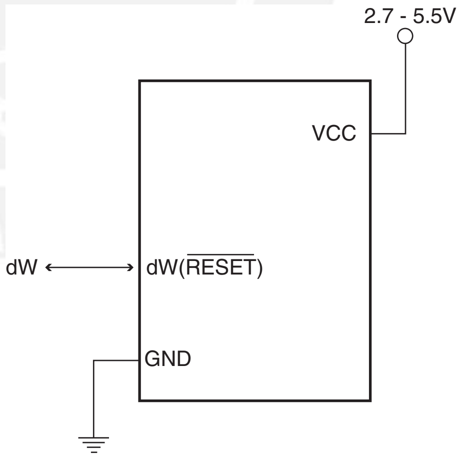


Figure 25-1 shows the schematic of a target MCU, with debugWIRE enabled, and the emulator connector. The system clock is not affected by debugWIRE and will always be the clock source selected by the CKSEL Fuses.

When designing a system where debugWIRE will be used, the following observations must be made for correct operation:

- Pull-up resistors on the dW/(RESET) line must not be smaller than 10kΩ. The pull-up resistor is not required for debugWIRE functionality.
- Connecting the RESET pin directly to $V_{CC}$ will not work.
- Capacitors connected to the RESET pin must be disconnected when using debugWire.
- All external reset sources must be disconnected.

## 25.4 Software Break Points

debugWIRE supports Program memory Break Points by the AVR Break instruction. Setting a Break Point in AVR Studio® will insert a BREAK instruction in the Program memory. The instruction replaced by the BREAK instruction will be stored. When program execution is continued, the stored instruction will be executed before continuing from the Program memory. A break can be inserted manually by putting the BREAK instruction in the program.

The Flash must be re-programmed each time a Break Point is changed. This is automatically handled by AVR Studio through the debugWIRE interface. The use of Break Points will therefore reduce the Flash Data retention. Devices used for debugging purposes should not be shipped to end customers.

## 25.5 Limitations of debugWIRE

The debugWIRE communication pin (dW) is physically located on the same pin as External Reset (RESET). An External Reset source is therefore not supported when the debugWIRE is enabled.

A programmed DWEN Fuse enables some parts of the clock system to be running in all sleep modes. This will increase the power consumption while in sleep. Thus, the DWEN Fuse should be disabled when debugWire is not used.

## 25.6 Register Description

The following section describes the registers used with the debugWire.

### 25.6.1 DWDR – debugWire Data Register

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| | DWDR[7:0] | | | | | | | | DWDR |
| Read/Write | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

The DWDR Register provides a communication channel from the running program in the MCU to the debugger. This register is only accessible by the debugWIRE and can therefore not be used as a general purpose register in the normal operations.

# 26. Self-Programming the Flash, Atmel ATmega48PA

## 26.1 Overview

In Atmel® ATmega48PA there is no Read-While-Write support, and no separate Boot Loader Section. The SPM instruction can be executed from the entire Flash.

The device provides a Self-Programming mechanism for downloading and uploading program code by the MCU itself. The Self-Programming can use any available data interface and associated protocol to read code and write (program) that code into the Program memory.

The Program memory is updated in a page by page fashion. Before programming a page with the data stored in the temporary page buffer, the page must be erased. The temporary page buffer is filled one word at a time using SPM and the buffer can be filled either before the Page Erase command or between a Page Erase and a Page Write operation:

Alternative 1, fill the buffer before a Page Erase

- Fill temporary page buffer
- Perform a Page Erase
- Perform a Page Write

Alternative 2, fill the buffer after Page Erase

- Perform a Page Erase
- Fill temporary page buffer
- Perform a Page Write

If only a part of the page needs to be changed, the rest of the page must be stored (for example in the temporary page buffer) before the erase, and then be re-written. When using alternative 1, the Boot Loader provides an effective Read-Modify-Write feature which allows the user software to first read the page, do the necessary changes, and then write back the modified data. If alternative 2 is used, it is not possible to read the old data while loading since the page is already erased. The temporary page buffer can be accessed in a random sequence. It is essential that the page address used in both the Page Erase and Page Write operation is addressing the same page.

### 26.1.1 Performing Page Erase by SPM

To execute Page Erase, set up the address in the Z-pointer, write “00000011” to SPMCSR and execute SPM within four clock cycles after writing SPMCSR. The data in R1 and R0 is ignored. The page address must be written to PCPAGE in the Z-register. Other bits in the Z-pointer will be ignored during this operation.

- The CPU is halted during the Page Erase operation.

Note: If an interrupt occurs in the time sequence the four cycle access cannot be guaranteed. In order to ensure atomic operation you should disable interrupts before writing to SPMCSR.

### 26.1.2 Filling the Temporary Buffer (Page Loading)

To write an instruction word, set up the address in the Z-pointer and data in R1:R0, write "00000001" to SPMCSR and execute SPM within four clock cycles after writing SPMCSR. The content of PCWORD in the Z-register is used to address the data in the temporary buffer. The temporary buffer will auto-erase after a Page Write operation or by writing the RWWSRE bit in SPMCSR. It is also erased after a system reset. Note that it is not possible to write more than one time to each address without erasing the temporary buffer.

If the EEPROM is written in the middle of an SPM Page Load operation, all data loaded will be lost.

### 26.1.3 Performing a Page Write

To execute Page Write, set up the address in the Z-pointer, write "00000101" to SPMCSR and execute SPM within four clock cycles after writing SPMCSR. The data in R1 and R0 is ignored. The page address must be written to PCPAGE. Other bits in the Z-pointer must be written to zero during this operation.

- The CPU is halted during the Page Write operation.

## 26.2 Addressing the Flash During Self-Programming

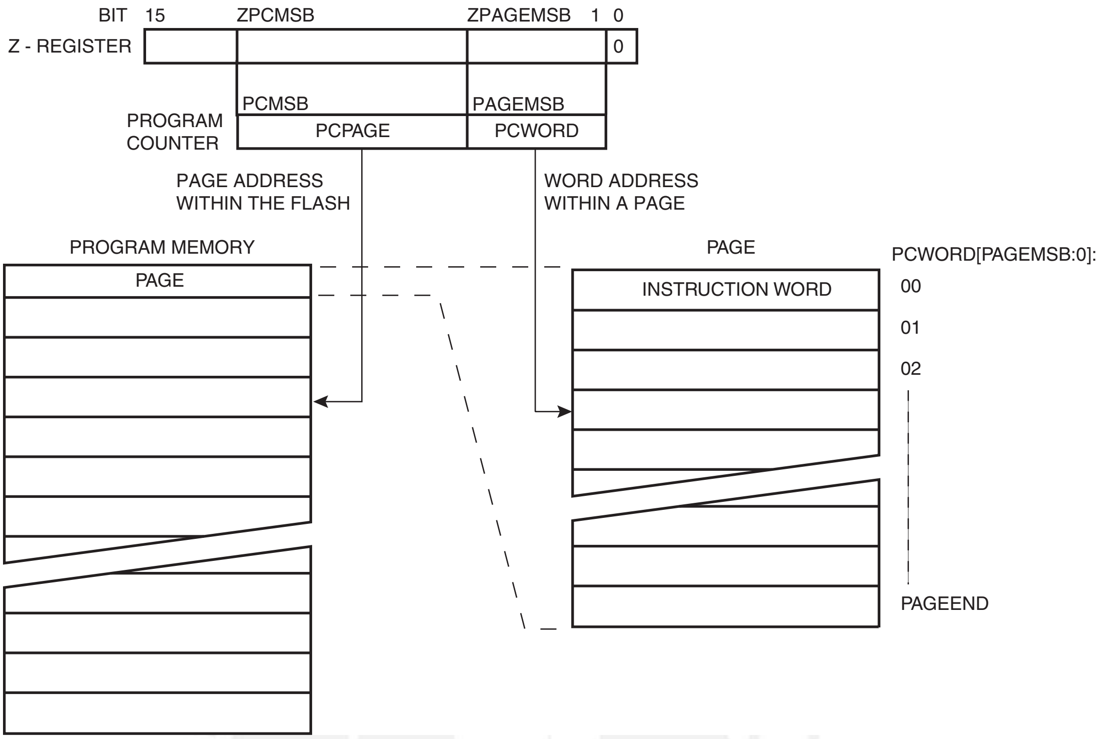
The Z-pointer is used to address the SPM commands.

| Bit | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 |
|---|---|---|---|---|---|---|---|---|
| **ZH (R31)** | **Z15** | **Z14** | **Z13** | **Z12** | **Z11** | **Z10** | **Z9** | **Z8** |
| **ZL (R30)** | **Z7** | **Z6** | **Z5** | **Z4** | **Z3** | **Z2** | **Z1** | **Z0** |
| | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

Since the Flash is organized in pages (see Table 28-9 on page 298), the Program Counter can be treated as having two different sections. One section, consisting of the least significant bits, is addressing the words within a page, while the most significant bits are addressing the pages. This is shown in Figure 27-3 on page 283. Note that the Page Erase and Page Write operations are addressed independently. Therefore it is of major importance that the software addresses the same page in both the Page Erase and Page Write operation.

The LPM instruction uses the Z-pointer to store the address. Since this instruction addresses the Flash byte-by-byte, also the LSB (bit Z0) of the Z-pointer is used.

**Figure 26-1.** Addressing the Flash During SPM[1]



Note: 1. The different variables used in Figure 27-3 are listed in Table 28-9 on page 298.

### 26.2.1 EEPROM Write Prevents Writing to SPMCSR

Note that an EEPROM write operation will block all software programming to Flash. Reading the Fuses and Lock bits from software will also be prevented during the EEPROM write operation. It is recommended that the user checks the status bit (EEPE) in the EECR Register and verifies that the bit is cleared before writing to the SPMCSR Register.

### 26.2.2 Reading the Fuse and Lock Bits from Software

It is possible to read both the Fuse and Lock bits from software. To read the Lock bits, load the Z-pointer with 0x0001 and set the BLBSET and SELFPRGEN bits in SPMCSR. When an LPM instruction is executed within three CPU cycles after the BLBSET and SELFPRGEN bits are set in SPMCSR, the value of the Lock bits will be loaded in the destination register. The BLBSET and SELFPRGEN bits will auto-clear upon completion of reading the Lock bits or if no LPM instruction is executed within three CPU cycles or no SPM instruction is executed within four CPU cycles. When BLBSET and SELFPRGEN are cleared, LPM will work as described in the Instruction set Manual.

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| Rd | – | – | – | – | – | – | **LB2** | **LB1** |

The algorithm for reading the Fuse Low byte is similar to the one described above for reading the Lock bits. To read the Fuse Low byte, load the Z-pointer with 0x0000 and set the BLBSET and SELFPRGEN bits in SPMCSR. When an LPM instruction is executed within three cycles after the BLBSET and SELFPRGEN bits are set in the SPMCSR, the value of the Fuse Low byte (FLB) will be loaded in the destination register as shown below.See Table 28-5 on page 296 for a detailed description and mapping of the Fuse Low byte.

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| Rd | FLB7 | FLB6 | FLB5 | FLB4 | FLB3 | FLB2 | FLB1 | FLB0 |

Similarly, when reading the Fuse High byte (FHB), load 0x0003 in the Z-pointer. When an LPM instruction is executed within three cycles after the BLBSET and SELFPRGEN bits are set in the SPMCSR, the value of the Fuse High byte will be loaded in the destination register as shown below. See Table 28-5 on page 296 for detailed description and mapping of the Extended Fuse byte.

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| Rd | FHB7 | FHB6 | FHB5 | FHB4 | FHB3 | FHB2 | FHB1 | FHB0 |

Similarly, when reading the Extended Fuse byte (EFB), load 0x0002 in the Z-pointer. When an LPM instruction is executed within three cycles after the BLBSET and SELFPRGEN bits are set in the SPMCSR, the value of the Extended Fuse byte will be loaded in the destination register as shown below. See Table 28-5 on page 296 for detailed description and mapping of the Extended Fuse byte.

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| Rd | FHB7 | FHB6 | FHB5 | FHB4 | FHB3 | FHB2 | FHB1 | FHB0 |

Fuse and Lock bits that are programmed, will be read as zero. Fuse and Lock bits that are unprogrammed, will be read as one.

### 26.2.3 Preventing Flash Corruption

During periods of low $V_{CC}$, the Flash program can be corrupted because the supply voltage is too low for the CPU and the Flash to operate properly. These issues are the same as for board level systems using the Flash, and the same design solutions should be applied.

A Flash program corruption can be caused by two situations when the voltage is too low. First, a regular write sequence to the Flash requires a minimum voltage to operate correctly. Secondly, the CPU itself can execute instructions incorrectly, if the supply voltage for executing instructions is too low.

Flash corruption can easily be avoided by following these design recommendations (one is sufficient):

1. Keep the AVR RESET active (low) during periods of insufficient power supply voltage. This can be done by enabling the internal Brown-out Detector (BOD) if the operating voltage matches the detection level. If not, an external low $V_{CC}$ reset protection circuit can be used. If a reset occurs while a write operation is in progress, the write operation will be completed provided that the power supply voltage is sufficient.
2. Keep the AVR core in Power-down sleep mode during periods of low $V_{CC}$. This will prevent the CPU from attempting to decode and execute instructions, effectively protecting the SPMCSR Register and thus the Flash from unintentional writes.

### 26.2.4 Programming Time for Flash when Using SPM

The calibrated RC Oscillator is used to time Flash accesses. Table 27-6 shows the typical programming time for Flash accesses from the CPU.

**Table 26-1.** SPM Programming Time[1]

| Symbol | Min. Programming Time | Max Programming Time |
|---|---|---|
| Flash write (Page Erase, Page Write, and write Lock bits by SPM) | 3.7ms | 4.5ms |

Note: 1. Minimum and maximum programming time is per individual operation.

### 26.2.5 Simple Assembly Code Example for a Boot Loader

Note that the RWWSB bit will always be read as zero in Atmel® ATmega48PA. Nevertheless, it is recommended to check this bit as shown in the code example, to ensure compatibility with devices supporting Read-While-Write.

```
  ;-the routine writes one page of data from RAM to Flash
  ; the first data location in RAM is pointed to by the Y pointer
  ; the first data location in Flash is pointed to by the Z-pointer
  ;-error handling is not included
  ;-the routine must be placed inside the Boot space
  ; (at least the Do_spm sub routine). Only code inside NRWW section can
  ; be read during Self-Programming (Page Erase and Page Write).
  ;-registers used: r0, r1, temp1 (r16), temp2 (r17), looplo (r24),
  ; loophi (r25), spmcrval (r20)
  ; storing and restoring of registers is not included in the routine
  ; register usage can be optimized at the expense of code size
  ;-It is assumed that either the interrupt table is moved to the Boot
  ; loader section or that the interrupts are disabled.
.equ PAGESIZEB = PAGESIZE*2   ;PAGESIZEB is page size in BYTES, not words
.org SMALLBOOTSTART
Write_page:
  ; Page Erase
  ldi  spmcrval, (1<<PGERS) | (1<<SELFPRGEN)
  rcallDo_spm

  ; re-enable the RWW section
  ldi  spmcrval, (1<<RWWSRE) | (1<<SELFPRGEN)
  rcallDo_spm

  ; transfer data from RAM to Flash page buffer
  ldi  looplo, low(PAGESIZEB)   ;init loop variable
  ldi  loophi, high(PAGESIZEB)  ;not required for PAGESIZEB<=256
Wrloop:
  ld   r0, Y+
  ld   r1, Y+
  ldi  spmcrval, (1<<SELFPRGEN)
  rcallDo_spm
  adiw ZH:ZL, 2
  sbiw loophi:looplo, 2         ;use subi for PAGESIZEB<=256
  brne Wrloop

  ; execute Page Write
  subi ZL, low(PAGESIZEB)       ;restore pointer
  sbci ZH, high(PAGESIZEB)      ;not required for PAGESIZEB<=256
  ldi  spmcrval, (1<<PGWRT) | (1<<SELFPRGEN)
  rcallDo_spm

  ; re-enable the RWW section
  ldi  spmcrval, (1<<RWWSRE) | (1<<SELFPRGEN)
  rcallDo_spm

  ; read back and check, optional
  ldi  looplo, low(PAGESIZEB)   ;init loop variable
  ldi  loophi, high(PAGESIZEB)  ;not required for PAGESIZEB<=256
  subi YL, low(PAGESIZEB)       ;restore pointer
  sbci YH, high(PAGESIZEB)
Rdloop:
  lpm  r0, Z+
  ld   r1, Y+
  cpse r0, r1
  rjmp Error
  sbiw loophi:looplo, 1         ;use subi for PAGESIZEB<=256
  brne Rdloop

  ; return to RWW section
  ; verify that RWW section is safe to read
Return:
  in   temp1, SPMCSR
  sbrs temp1, RWWSB     ; If RWWSB is set, the RWW section is not ready yet
  ret
  ; re-enable the RWW section
  ldi  spmcrval, (1<<RWWSRE) | (1<<SELFPRGEN)
  rcallDo_spm
  rjmp Return

Do_spm:
  ; check for previous SPM complete
Wait_spm:
  in   temp1, SPMCSR
  sbrc temp1, SELFPRGEN
  rjmp Wait_spm
  ; input: spmcrval determines SPM action
  ; disable interrupts if enabled, store status
  in   temp2, SREG
  cli
  ; check that no EEPROM write access is present
Wait_ee:
  sbic EECR, EEPE
  rjmp Wait_ee
  ; SPM timed sequence
  out  SPMCSR, spmcrval
  spm
  ; restore SREG (to enable interrupts if originally enabled)
  out  SREG, temp2
  ret
```

## 26.3 Register Description

### 26.3.1 SPMCSR – Store Program Memory Control and Status Register

The Store Program Memory Control and Status Register contains the control bits needed to control the Program memory operations.

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| 0x37 (0x57) | SPMIE | RWWSB | SIGRD | RWWSRE | BLBSET | PGWRT | PGERS | SELFPRGEN | SPMCSR |
| Read/Write | R/W | R | R/W | R/W | R/W | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

- **Bit 7 – SPMIE: SPM Interrupt Enable**

When the SPMIE bit is written to one, and the I-bit in the Status Register is set (one), the SPM ready interrupt will be enabled. The SPM ready Interrupt will be executed as long as the SELFPRGEN bit in the SPMCSR Register is cleared. The interrupt will not be generated during EEPROM write or SPM.

- **Bit 6 – RWWSB: Read-While-Write Section Busy**

This bit is for compatibility with devices supporting Read-While-Write. It will always read as zero in Atmel® ATmega48PA.

- **Bit 5 – SIGRD: Signature Row Read**

If this bit is written to one at the same time as SELFPRGEN, the next LPM instruction within three clock cycles will read a byte from the signature row into the destination register. see “Reading the Signature Row from Software” on page 286 for details. An SPM instruction within four cycles after SIGRD and SELFPRGEN are set will have no effect. This operation is reserved for future use and should not be used.

- **Bit 4 – RWWSRE: Read-While-Write Section Read Enable**

The functionality of this bit in Atmel ATmega48PA is a subset of the functionality in the Atmel® ATmega48PA/88PA/168PA. If the RWWSRE bit is written while filling the temporary page buffer, the temporary page buffer will be cleared and the data will be lost.

- **Bit 3 – BLBSET: Boot Lock Bit Set**

The functionality of this bit in Atmel ATmega48PA is a subset of the functionality in the Atmel ATmega48PA/88PA/168PA. An LPM instruction within three cycles after BLBSET and SELFPRGEN are set in the SPMCSR Register, will read either the Lock bits or the Fuse bits (depending on Z0 in the Z-pointer) into the destination register. See “Reading the Fuse and Lock Bits from Software” on page 271 for details.

- **Bit 2 – PGWRT: Page Write**

If this bit is written to one at the same time as SELFPRGEN, the next SPM instruction within four clock cycles executes Page Write, with the data stored in the temporary buffer. The page address is taken from the high part of the Z-pointer. The data in R1 and R0 are ignored. The PGWRT bit will auto-clear upon completion of a Page Write, or if no SPM instruction is executed within four clock cycles. The CPU is halted during the entire Page Write operation.

- **Bit 1 – PGERS: Page Erase**

If this bit is written to one at the same time as SELFPRGEN, the next SPM instruction within four clock cycles executes Page Erase. The page address is taken from the high part of the Z-pointer. The data in R1 and R0 are ignored. The PGERS bit will auto-clear upon completion of a Page Erase, or if no SPM instruction is executed within four clock cycles. The CPU is halted during the entire Page Write operation.

- **Bit 0 – SELFPRGEN: Self Programming Enable**

This bit enables the SPM instruction for the next four clock cycles. If written to one together with either RWWSRE, BLBSET, PGWRT, or PGERS, the following SPM instruction will have a special meaning, see description above. If only SELFPRGEN is written, the following SPM instruction will store the value in R1:R0 in the temporary page buffer addressed by the Z-pointer. The LSB of the Z-pointer is ignored. The SELFPRGEN bit will auto-clear upon completion of an SPM instruction, or if no SPM instruction is executed within four clock cycles. During Page Erase and Page Write, the SELFPRGEN bit remains high until the operation is completed.

Writing any other combination than “10001”, “01001”, “00101”, “00011” or “00001” in the lower five bits will have no effect.

# 27. Boot Loader Support – Read-While-Write Self-Programming

The Boot Loader Support applies to Atmel® ATmega48PA/88PA/168PA

## 27.1 Features

- **Read-While-Write Self-Programming**
- **Flexible Boot Memory Size**
- **High Security (Separate Boot Lock Bits for a Flexible Protection)**
- **Separate Fuse to Select Reset Vector**
- **Optimized Page[1] Size**
- **Code Efficient Algorithm**
- **Efficient Read-Modify-Write Support**

Note: 1. A page is a section in the Flash consisting of several bytes (see Table 28-9 on page 298) used during programming. The page organization does not affect normal operation.

## 27.2 Overview

In the Atmel ATmega48PA/88PA/168PA the Boot Loader Support provides a real Read-While-Write Self-Programming mechanism for downloading and uploading program code by the MCU itself. This feature allows flexible application software updates controlled by the MCU using a Flash-resident Boot Loader program. The Boot Loader program can use any available data interface and associated protocol to read code and write (program) that code into the Flash memory, or read the code from the program memory. The program code within the Boot Loader section has the capability to write into the entire Flash, including the Boot Loader memory. The Boot Loader can thus even modify itself, and it can also erase itself from the code if the feature is not needed anymore. The size of the Boot Loader memory is configurable with fuses and the Boot Loader has two separate sets of Boot Lock bits which can be set independently. This gives the user a unique flexibility to select different levels of protection.

## 27.3 Application and Boot Loader Flash Sections

The Flash memory is organized in two main sections, the Application section and the Boot Loader section (see Figure 27-2). The size of the different sections is configured by the BOOTSZ Fuses as shown in Table 27-7 on page 290 and Figure 27-2. These two sections can have different level of protection since they have different sets of Lock bits.

### 27.3.1 Application Section

The Application section is the section of the Flash that is used for storing the application code. The protection level for the Application section can be selected by the application Boot Lock bits (Boot Lock bits 0), see Table 27-2 on page 281. The Application section can never store any Boot Loader code since the SPM instruction is disabled when executed from the Application section.

### 27.3.2 BLS – Boot Loader Section

While the Application section is used for storing the application code, the The Boot Loader software must be located in the BLS since the SPM instruction can initiate a programming when executing from the BLS only. The SPM instruction can access the entire Flash, including the BLS itself. The protection level for the Boot Loader section can be selected by the Boot Loader Lock bits (Boot Lock bits 1), see Table 27-3 on page 281.

## 27.4 Read-While-Write and No Read-While-Write Flash Sections

Whether the CPU supports Read-While-Write or if the CPU is halted during a Boot Loader software update is dependent on which address that is being programmed. In addition to the two sections that are configurable by the BOOTSZ Fuses as described above, the Flash is also divided into two fixed sections, the Read-While-Write (RWW) section and the No Read-While-Write (NRWW) section. The limit between the RWW- and NRWW sections is given in Table 27-8 on page 290 and Figure 27-2 on page 280. The main difference between the two sections is:

- When erasing or writing a page located inside the RWW section, the NRWW section can be read during the operation.
- When erasing or writing a page located inside the NRWW section, the CPU is halted during the entire operation.

Note that the user software can never read any code that is located inside the RWW section during a Boot Loader software operation. The syntax "Read-While-Write section" refers to which section that is being programmed (erased or written), not which section that actually is being read during a Boot Loader software update.

### 27.4.1 RWW – Read-While-Write Section

If a Boot Loader software update is programming a page inside the RWW section, it is possible to read code from the Flash, but only code that is located in the NRWW section. During an on-going programming, the software must ensure that the RWW section never is being read. If the user software is trying to read code that is located inside the RWW section (i.e., by a call/jmp/lpm or an interrupt) during programming, the software might end up in an unknown state. To avoid this, the interrupts should either be disabled or moved to the Boot Loader section. The Boot Loader section is always located in the NRWW section. The RWW Section Busy bit (RWWSB) in the Store Program Memory Control and Status Register (SPMCSR) will be read as logical one as long as the RWW section is blocked for reading. After a programming is completed, the RWWSB must be cleared by software before reading code located in the RWW section. See "SPMCSR – Store Program Memory Control and Status Register" on page 292. for details on how to clear RWWSB.

### 27.4.2 NRWW – No Read-While-Write Section

The code located in the NRWW section can be read when the Boot Loader software is updating a page in the RWW section. When the Boot Loader code updates the NRWW section, the CPU is halted during the entire Page Erase or Page Write operation.

**Table 27-1.** Read-While-Write Features

| Which Section does the Z-pointer Address during the Programming? | Which Section can be read during Programming? | CPU Halted? | Read-While-Write Supported? |
|---|---|---|---|
| RWW Section | NRWW Section | No | Yes |
| NRWW Section | None | Yes | No |

**Figure 27-1.** Read-While-Write versus No Read-While-Write

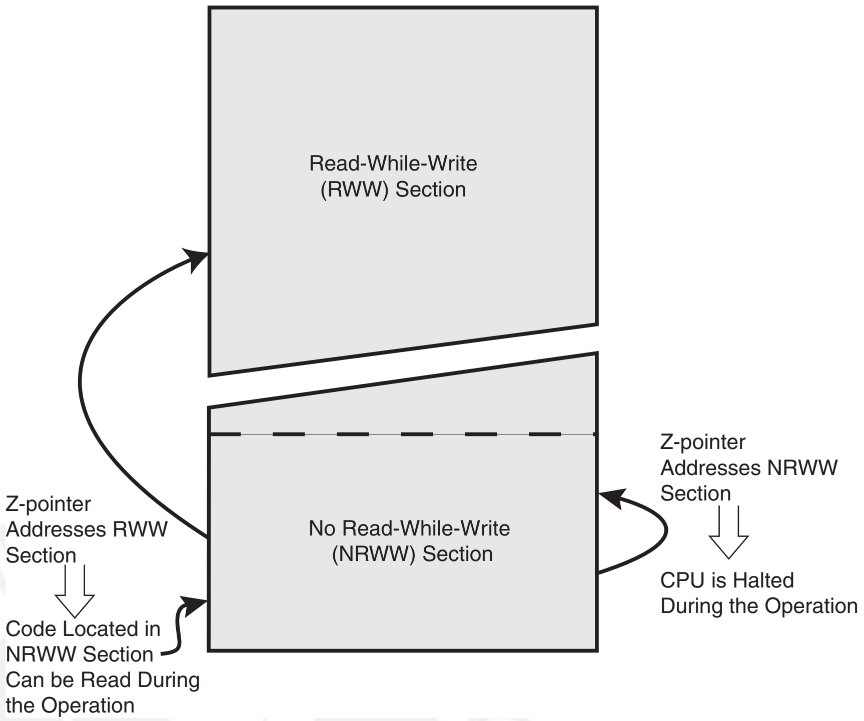
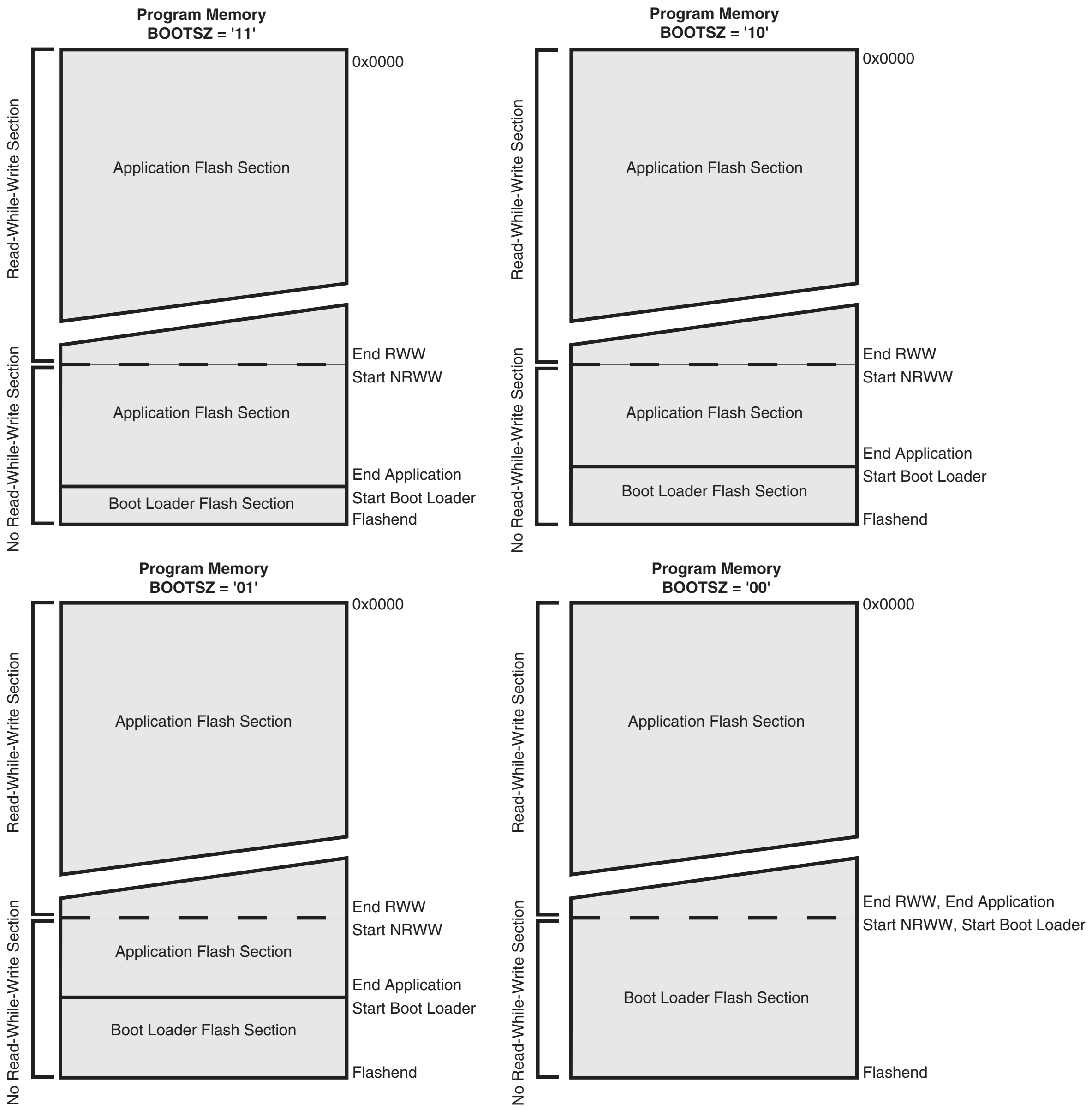


**Figure 27-2.** Memory Sections



Note: 1. The parameters in the figure above are given in Table 27-7 on page 290.

## 27.5 Boot Loader Lock Bits

If no Boot Loader capability is needed, the entire Flash is available for application code. The Boot Loader has two separate sets of Boot Lock bits which can be set independently. This gives the user a unique flexibility to select different levels of protection.

The user can select:

- To protect the entire Flash from a software update by the MCU.
- To protect only the Boot Loader Flash section from a software update by the MCU.
- To protect only the Application Flash section from a software update by the MCU.
- Allow software update in the entire Flash.

See Table 27-2 and Table 27-3 for further details. The Boot Lock bits can be set in software and in Serial or Parallel Programming mode, but they can be cleared by a Chip Erase command only. The general Write Lock (Lock Bit mode 2) does not control the programming of the Flash memory by SPM instruction. Similarly, the general Read/Write Lock (Lock Bit mode 1) does not control reading nor writing by LPM/SPM, if it is attempted.

**Table 27-2.** Boot Lock Bit0 Protection Modes (Application Section)[1]

| BLB0 Mode | BLB02 | BLB01 | Protection |
|---|---|---|---|
| 1 | 1 | 1 | No restrictions for SPM or LPM accessing the Application section. |
| 2 | 1 | 0 | SPM is not allowed to write to the Application section. |
| 3 | 0 | 0 | SPM is not allowed to write to the Application section, and LPM executing from the Boot Loader section is not allowed to read from the Application section. If Interrupt Vectors are placed in the Boot Loader section, interrupts are disabled while executing from the Application section. |
| 4 | 0 | 1 | LPM executing from the Boot Loader section is not allowed to read from the Application section. If Interrupt Vectors are placed in the Boot Loader section, interrupts are disabled while executing from the Application section. |

Note: 1. "1" means unprogrammed, "0" means programmed

**Table 27-3.** Boot Lock Bit1 Protection Modes (Boot Loader Section)[1]

| BLB1 Mode | BLB12 | BLB11 | Protection |
|---|---|---|---|
| 1 | 1 | 1 | No restrictions for SPM or LPM accessing the Boot Loader section. |
| 2 | 1 | 0 | SPM is not allowed to write to the Boot Loader section. |
| 3 | 0 | 0 | SPM is not allowed to write to the Boot Loader section, and LPM executing from the Application section is not allowed to read from the Boot Loader section. If Interrupt Vectors are placed in the Application section, interrupts are disabled while executing from the Boot Loader section. |
| 4 | 0 | 1 | LPM executing from the Application section is not allowed to read from the Boot Loader section. If Interrupt Vectors are placed in the Application section, interrupts are disabled while executing from the Boot Loader section. |

Note: 1. "1" means unprogrammed, "0" means programmed

## 27.6 Entering the Boot Loader Program

Entering the Boot Loader takes place by a jump or call from the application program. This may be initiated by a trigger such as a command received via USART, or SPI interface. Alternatively, the Boot Reset Fuse can be programmed so that the Reset Vector is pointing to the Boot Flash start address after a reset. In this case, the Boot Loader is started after a reset. After the application code is loaded, the program can start executing the application code. Note that the fuses cannot be changed by the MCU itself. This means that once the Boot Reset Fuse is programmed, the Reset Vector will always point to the Boot Loader Reset and the fuse can only be changed through the serial or parallel programming interface.

**Table 27-4.** Boot Reset Fuse[1]

| BOOTRST | Reset Address |
|---|---|
| 1 | Reset Vector = Application Reset (address 0x0000) |
| 0 | Reset Vector = Boot Loader Reset (see Table 27-7 on page 290) |

Note: 1. "1" means unprogrammed, "0" means programmed

## 27.7 Addressing the Flash During Self-Programming

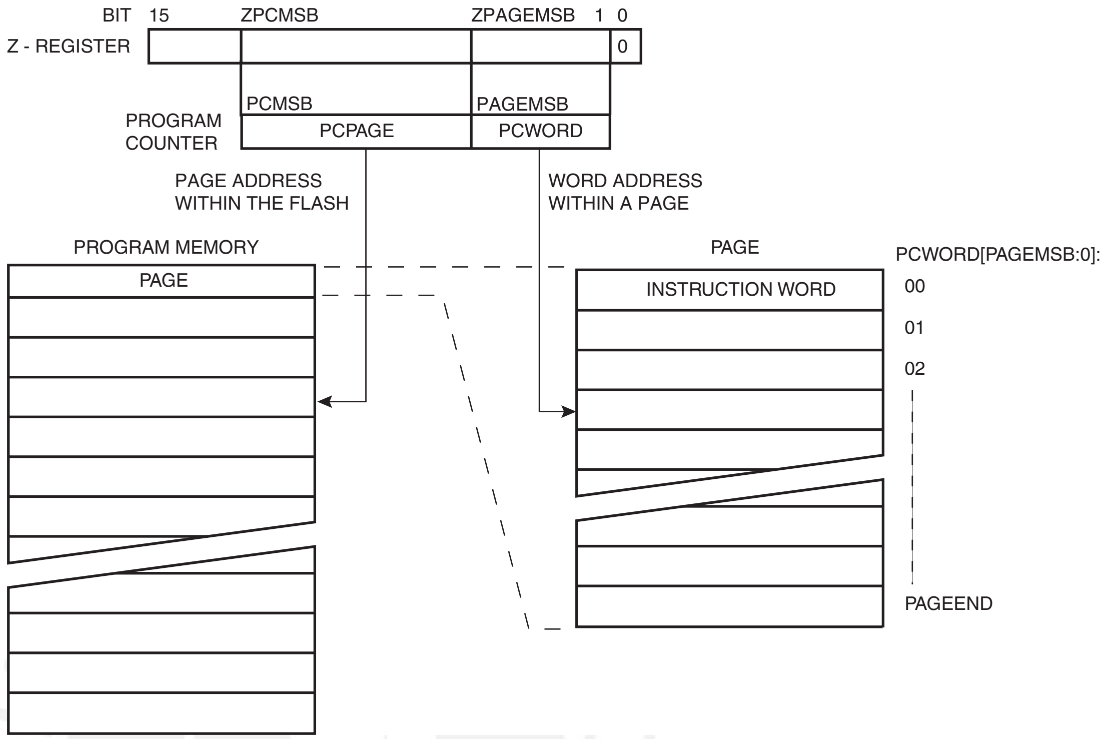
The Z-pointer is used to address the SPM commands.

| Bit | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 |
|---|---|---|---|---|---|---|---|---|
| **ZH (R31)** | **Z15** | **Z14** | **Z13** | **Z12** | **Z11** | **Z10** | **Z9** | **Z8** |
| **ZL (R30)** | **Z7** | **Z6** | **Z5** | **Z4** | **Z3** | **Z2** | **Z1** | **Z0** |
| | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

Since the Flash is organized in pages (see Table 28-9 on page 298), the Program Counter can be treated as having two different sections. One section, consisting of the least significant bits, is addressing the words within a page, while the most significant bits are addressing the pages. This is1 shown in Figure 27-3. Note that the Page Erase and Page Write operations are addressed independently. Therefore it is of major importance that the Boot Loader software addresses the same page in both the Page Erase and Page Write operation. Once a programming operation is initiated, the address is latched and the Z-pointer can be used for other operations.

The only SPM operation that does not use the Z-pointer is Setting the Boot Loader Lock bits. The content of the Z-pointer is ignored and will have no effect on the operation. The LPM instruction does also use the Z-pointer to store the address. Since this instruction addresses the Flash byte-by-byte, also the LSB (bit Z0) of the Z-pointer is used.

**Figure 27-3.** Addressing the Flash During SPM[1]



Note: 1. The different variables used in Figure 27-3 are listed in Table 27-9 on page 290.

## 27.8 Self-Programming the Flash

The program memory is updated in a page by page fashion. Before programming a page with the data stored in the temporary page buffer, the page must be erased. The temporary page buffer is filled one word at a time using SPM and the buffer can be filled either before the Page Erase command or between a Page Erase and a Page Write operation:

Alternative 1, fill the buffer before a Page Erase

- Fill temporary page buffer
- Perform a Page Erase
- Perform a Page Write

Alternative 2, fill the buffer after Page Erase

- Perform a Page Erase
- Fill temporary page buffer
- Perform a Page Write

If only a part of the page needs to be changed, the rest of the page must be stored (for example in the temporary page buffer) before the erase, and then be rewritten. When using alternative 1, the Boot Loader provides an effective Read-Modify-Write feature which allows the user software to first read the page, do the necessary changes, and then write back the modified data. If alternative 2 is used, it is not possible to read the old data while loading since the page is already erased. The temporary page buffer can be accessed in a random sequence.

It is essential that the page address used in both the Page Erase and Page Write operation is addressing the same page. See "Simple Assembly Code Example for a Boot Loader" on page 288 for an assembly code example.

### 27.8.1 Performing Page Erase by SPM

To execute Page Erase, set up the address in the Z-pointer, write "X0000011" to SPMCSR and execute SPM within four clock cycles after writing SPMCSR. The data in R1 and R0 is ignored. The page address must be written to PCPAGE in the Z-register. Other bits in the Z-pointer will be ignored during this operation.

- Page Erase to the RWW section: The NRWW section can be read during the Page Erase.
- Page Erase to the NRWW section: The CPU is halted during the operation.

### 27.8.2 Filling the Temporary Buffer (Page Loading)

To write an instruction word, set up the address in the Z-pointer and data in R1:R0, write "00000001" to SPMCSR and execute SPM within four clock cycles after writing SPMCSR. The content of PCWORD in the Z-register is used to address the data in the temporary buffer. The temporary buffer will auto-erase after a Page Write operation or by writing the RWWSRE bit in SPMCSR. It is also erased after a system reset. Note that it is not possible to write more than one time to each address without erasing the temporary buffer.

If the EEPROM is written in the middle of an SPM Page Load operation, all data loaded will be lost.

### 27.8.3 Performing a Page Write

To execute Page Write, set up the address in the Z-pointer, write "X0000101" to SPMCSR and execute SPM within four clock cycles after writing SPMCSR. The data in R1 and R0 is ignored. The page address must be written to PCPAGE. Other bits in the Z-pointer must be written to zero during this operation.

- Page Write to the RWW section: The NRWW section can be read during the Page Write.
- Page Write to the NRWW section: The CPU is halted during the operation.

### 27.8.4 Using the SPM Interrupt

If the SPM interrupt is enabled, the SPM interrupt will generate a constant interrupt when the SELFPRGEN bit in SPMCSR is cleared. This means that the interrupt can be used instead of polling the SPMCSR Register in software. When using the SPM interrupt, the Interrupt Vectors should be moved to the BLS section to avoid that an interrupt is accessing the RWW section when it is blocked for reading. How to move the interrupts is described in "Interrupts" on page 58.

### 27.8.5 Consideration While Updating BLS

Special care must be taken if the user allows the Boot Loader section to be updated by leaving Boot Lock bit11 unprogrammed. An accidental write to the Boot Loader itself can corrupt the entire Boot Loader, and further software updates might be impossible. If it is not necessary to change the Boot Loader software itself, it is recommended to program the Boot Lock bit11 to protect the Boot Loader software from any internal software changes.

### 27.8.6 Prevent Reading the RWW Section During Self-Programming

During Self-Programming (either Page Erase or Page Write), the RWW section is always blocked for reading. The user software itself must prevent that this section is addressed during the self programming operation. The RWWSB in the SPMCSR will be set as long as the RWW section is busy. During Self-Programming the Interrupt Vector table should be moved to the BLS as described in “Watchdog Timer” on page 51, or the interrupts must be disabled. Before addressing the RWW section after the programming is completed, the user software must clear the RWWSB by writing the RWWSRE. See “Simple Assembly Code Example for a Boot Loader” on page 288 for an example.

### 27.8.7 Setting the Boot Loader Lock Bits by SPM

To set the Boot Loader Lock bits and general Lock Bits, write the desired data to R0, write “X0001001” to SPMCSR and execute SPM within four clock cycles after writing SPMCSR.

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| **R0** | **1** | **1** | **BLB12** | **BLB11** | **BLB02** | **BLB01** | **LB2** | **LB1** |

See Table 27-2 and Table 27-3 for how the different settings of the Boot Loader bits affect the Flash access.

If bits 5...0 in R0 are cleared (zero), the corresponding Lock bit will be programmed if an SPM instruction is executed within four cycles after BLBSET and SELFPRGEN are set in SPMCSR. The Z-pointer is don’t care during this operation, but for future compatibility it is recommended to load the Z-pointer with 0x0001 (same as used for reading the $IO_{ck}$ bits). For future compatibility it is also recommended to set bits 7 and 6 in R0 to “1” when writing the Lock bits. When programming the Lock bits the entire Flash can be read during the operation.

### 27.8.8 EEPROM Write Prevents Writing to SPMCSR

Note that an EEPROM write operation will block all software programming to Flash. Reading the Fuses and Lock bits from software will also be prevented during the EEPROM write operation. It is recommended that the user checks the status bit (EEPE) in the EECR Register and verifies that the bit is cleared before writing to the SPMCSR Register.

### 27.8.9 Reading the Fuse and Lock Bits from Software

It is possible to read both the Fuse and Lock bits from software. To read the Lock bits, load the Z-pointer with 0x0001 and set the BLBSET and SELFPRGEN bits in SPMCSR. When an LPM instruction is executed within three CPU cycles after the BLBSET and SELFPRGEN bits are set in SPMCSR, the value of the Lock bits will be loaded in the destination register. The BLBSET and SELFPRGEN bits will auto-clear upon completion of reading the Lock bits or if no LPM instruction is executed within three CPU cycles or no SPM instruction is executed within four CPU cycles. When BLBSET and SELFPRGEN are cleared, LPM will work as described in the Instruction set Manual.

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| Rd | **–** | **–** | **BLB12** | **BLB11** | **BLB02** | **BLB01** | **LB2** | **LB1** |

The algorithm for reading the Fuse Low byte is similar to the one described above for reading the Lock bits. To read the Fuse Low byte, load the Z-pointer with 0x0000 and set the BLBSET and SELFPRGEN bits in SPMCSR. When an LPM instruction is executed within three cycles after the BLBSET and SELFPRGEN bits are set in the SPMCSR, the value of the Fuse Low byte (FLB) will be loaded in the destination register as shown below. Refer to Table 28-5 on page 296 for a detailed description and mapping of the Fuse Low byte.

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| Rd | **FLB7** | **FLB6** | **FLB5** | **FLB4** | **FLB3** | **FLB2** | **FLB1** | **FLB0** |

Similarly, when reading the Fuse High byte, load 0x0003 in the Z-pointer. When an LPM instruction is executed within three cycles after the BLBSET and SELFPRGEN bits are set in the SPMCSR, the value of the Fuse High byte (FHB) will be loaded in the destination register as shown below. Refer to Table 28-6 on page 296 for detailed description and mapping of the Fuse High byte.

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| Rd | **FHB7** | **FHB6** | **FHB5** | **FHB4** | **FHB3** | **FHB2** | **FHB1** | **FHB0** |

When reading the Extended Fuse byte, load 0x0002 in the Z-pointer. When an LPM instruction is executed within three cycles after the BLBSET and SELFPRGEN bits are set in the SPMCSR, the value of the Extended Fuse byte (EFB) will be loaded in the destination register as shown below. Refer to Table 28-5 on page 296 for detailed description and mapping of the Extended Fuse byte.

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| Rd | – | – | – | – | **EFB3** | **EFB2** | **EFB1** | **EFB0** |

Fuse and Lock bits that are programmed, will be read as zero. Fuse and Lock bits that are unprogrammed, will be read as one.

### 27.8.10 Reading the Signature Row from Software

To read the Signature Row from software, load the Z-pointer with the signature byte address given in Table 27-5 and set the SIGRD and SELFPRGEN bits in SPMCSR. When an LPM instruction is executed within three CPU cycles after the SIGRD and SELFPRGEN bits are set in SPMCSR, the signature byte value will be loaded in the destination register. The SIGRD and SELFPRGEN bits will auto-clear upon completion of reading the Signature Row Lock bits or if no LPM instruction is executed within three CPU cycles. When SIGRD and SELFPRGEN are cleared, LPM will work as described in the Instruction set Manual.

**Table 27-5.** Signature Row Addressing

| Signature Byte | Z-Pointer Address |
|---|---|
| Device Signature Byte 1 | 0x0000 |
| Device Signature Byte 2 | 0x0002 |
| Device Signature Byte 3 | 0x0004 |
| RC Oscillator Calibration Byte 3V | 0x0001 |
| TS_ADC_25_L - Temp Sensor value at 25°C - Low byte | 0x0005 |
| TS_ADC_25_H - Temp Sensor value at 25°C - High byte | 0x0007 |
| RC Oscillator Calibration Byte 5V | 0x0009 |

Note: All other addresses are reserved for future use

### 27.8.11 Preventing Flash Corruption

During periods of low $V_{CC}$, the Flash program can be corrupted because the supply voltage is too low for the CPU and the Flash to operate properly. These issues are the same as for board level systems using the Flash, and the same design solutions should be applied.

A Flash program corruption can be caused by two situations when the voltage is too low. First, a regular write sequence to the Flash requires a minimum voltage to operate correctly. Secondly, the CPU itself can execute instructions incorrectly, if the supply voltage for executing instructions is too low.

Flash corruption can easily be avoided by following these design recommendations (one is sufficient):

1. If there is no need for a Boot Loader update in the system, program the Boot Loader Lock bits to prevent any Boot Loader software updates.
2. Keep the AVR RESET active (low) during periods of insufficient power supply voltage. This can be done by enabling the internal Brown-out Detector (BOD) if the operating voltage matches the detection level. If not, an external low $V_{CC}$ reset protection circuit can be used. If a reset occurs while a write operation is in progress, the write operation will be completed provided that the power supply voltage is sufficient.
3. Keep the AVR core in Power-down sleep mode during periods of low $V_{CC}$. This will prevent the CPU from attempting to decode and execute instructions, effectively protecting the SPMCSR Register and thus the Flash from unintentional writes.

### 27.8.12 Programming Time for Flash when Using SPM

The calibrated RC Oscillator is used to time Flash accesses. Table 27-6 shows the typical programming time for Flash accesses from the CPU.

**Table 27-6.** SPM Programming Time(27.8.13)

| Symbol | Min. Programming Time | Max Programming Time |
|---|---|---|
| Flash write (Page Erase, Page Write, and write Lock bits by SPM) | 3.7ms | 4.5ms |

Note: 1. Minimum and maximum programming time is per individual operation.

### 27.8.13 Simple Assembly Code Example for a Boot Loader

```
  ;-the routine writes one page of data from RAM to Flash
  ; the first data location in RAM is pointed to by the Y pointer
  ; the first data location in Flash is pointed to by the Z-pointer
  ;-error handling is not included
  ;-the routine must be placed inside the Boot space
  ; (at least the Do_spm sub routine). Only code inside NRWW section can
  ; be read during Self-Programming (Page Erase and Page Write).
  ;-registers used: r0, r1, temp1 (r16), temp2 (r17), looplo (r24),
  ; loophi (r25), spmcrval (r20)
  ; storing and restoring of registers is not included in the routine
  ; register usage can be optimized at the expense of code size
  ;-It is assumed that either the interrupt table is moved to the Boot
  ; loader section or that the interrupts are disabled.
.equ PAGESIZEB = PAGESIZE*2    ;PAGESIZEB is page size in BYTES, not words
.org SMALLBOOTSTART
Write_page:
  ; Page Erase
  ldi  spmcrval, (1<<PGERS) | (1<<SELFPRGEN)
  call Do_spm

  ; re-enable the RWW section
  ldi  spmcrval, (1<<RWWSRE) | (1<<SELFPRGEN)
  call Do_spm

  ; transfer data from RAM to Flash page buffer
  ldi  looplo, low(PAGESIZEB)   ;init loop variable
  ldi  loophi, high(PAGESIZEB)  ;not required for PAGESIZEB<=256
Wrloop:
  ld   r0, Y+
  ld   r1, Y+
  ldi  spmcrval, (1<<SELFPRGEN)
  call Do_spm
  adiw ZH:ZL, 2
  sbiw loophi:looplo, 2         ;use subi for PAGESIZEB<=256
  brne Wrloop

  ; execute Page Write
  subi ZL, low(PAGESIZEB)       ;restore pointer
  sbci ZH, high(PAGESIZEB)      ;not required for PAGESIZEB<=256
  ldi  spmcrval, (1<<PGWRT) | (1<<SELFPRGEN)
  call Do_spm

  ; re-enable the RWW section
  ldi  spmcrval, (1<<RWWSRE) | (1<<SELFPRGEN)
  call Do_spm

  ; read back and check, optional
  ldi  looplo, low(PAGESIZEB)   ;init loop variable
  ldi  loophi, high(PAGESIZEB)  ;not required for PAGESIZEB<=256
  subi YL, low(PAGESIZEB)       ;restore pointer
  sbci YH, high(PAGESIZEB)
Rdloop:
  lpm  r0, Z+
  ld   r1, Y+
  cpse r0, r1
  jmp  Error
  sbiw loophi:looplo, 1         ;use subi for PAGESIZEB<=256
  brne Rdloop
  ; return to RWW section
  ; verify that RWW section is safe to read
Return:
  in   temp1, SPMCSR
  sbrs temp1, RWWSB     ; If RWWSB is set, the RWW section is not ready yet
  ret
  ; re-enable the RWW section
  ldi  spmcrval, (1<<RWWSRE) | (1<<SELFPRGEN)
  call Do_spm
  rjmp Return

Do_spm:
  ; check for previous SPM complete
Wait_spm:
  in   temp1, SPMCSR
  sbrc temp1, SELFPRGEN
  rjmp Wait_spm
  ; input: spmcrval determines SPM action
  ; disable interrupts if enabled, store status
  in   temp2, SREG
  cli
  ; check that no EEPROM write access is present
Wait_ee:
  sbic EECR, EEPE
  rjmp Wait_ee
  ; SPM timed sequence
  out  SPMCSR, spmcrval
  spm
  ; restore SREG (to enable interrupts if originally enabled)
  out  SREG, temp2
  ret
```

### 27.8.14 Atmel ATmega88PA Boot Loader Parameters

In Table 27-7 through Table 27-9, the parameters used in the description of the self programming are given.

**Table 27-7.** Boot Size Configuration, Atmel ATmega88PA

| BOOTSZ1 | BOOTSZ0 | Boot Size | Pages | Application Flash Section | Boot Loader Flash Section | End Application Section | Boot Reset Address (Start Boot Loader Section) |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 128 words | 4 | 0x000 - 0xF7F | 0xF80 - 0xFFF | 0xF7F | 0xF80 |
| 1 | 0 | 256 words | 8 | 0x000 - 0xEFF | 0xF00 - 0xFFF | 0xEFF | 0xF00 |
| 0 | 1 | 512 words | 16 | 0x000 - 0xDFF | 0xE00 - 0xFFF | 0xDFF | 0xE00 |
| 0 | 0 | 1024 words | 32 | 0x000 - 0xBFF | 0xC00 - 0xFFF | 0xBFF | 0xC00 |

Note: The different BOOTSZ Fuse configurations are shown in Figure 27-2 on page 280.

**Table 27-8.** Read-While-Write Limit, Atmel ATmega88PA

| Section | Pages | Address |
|---|---|---|
| Read-While-Write section (RWW) | 96 | 0x000 - 0xBFF |
| No Read-While-Write section (NRWW) | 32 | 0xC00 - 0xFFF |

For details about these two section, see "NRWW – No Read-While-Write Section" on page 278 and "RWW – Read-While-Write Section" on page 278.

**Table 27-9.** Explanation of Different Variables used in Figure 27-3 and the Mapping to the Z-pointer, Atmel ATmega88PA

| Variable | | Corresponding Z-value[1] | Description |
|---|---|---|---|
| PCMSB | 11 | | Most significant bit in the Program Counter. (The Program Counter is 12 bits PC[11:0]) |
| PAGEMSB | 4 | | Most significant bit which is used to address the words within one page (32 words in a page requires 5 bits PC [4:0]). |
| ZPCMSB | | Z12 | Bit in Z-register that is mapped to PCMSB. Because Z0 is not used, the ZPCMSB equals PCMSB + 1. |
| ZPAGEMSB | | Z5 | Bit in Z-register that is mapped to PAGEMSB. Because Z0 is not used, the ZPAGEMSB equals PAGEMSB + 1. |
| PCPAGE | PC[11:5] | Z12:Z6 | Program counter page address: Page select, for page erase and page write |
| PCWORD | PC[4:0] | Z5:Z1 | Program counter word address: Word select, for filling temporary buffer (must be zero during page write operation) |

Note: 1. Z15:Z13: always ignored
Z0: should be zero for all SPM commands, byte select for the LPM instruction.
See "Addressing the Flash During Self-Programming" on page 282 for details about the use of Z-pointer during Self-Programming.

### 27.8.15 Atmel ATmega168PA Boot Loader Parameters

In Table 27-10 through Table 27-12, the parameters used in the description of the self programming are given.

**Table 27-10.** Boot Size Configuration, Atmel ATmega168PA

| BOOTSZ1 | BOOTSZ0 | Boot Size | Pages | Application Flash Section | Boot Loader Flash Section | End Application Section | Boot Reset Address (Start Boot Loader Section) |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 128 words | 2 | 0x0000 - 0x1F7F | 0x1F80 - 0x1FFF | 0x1F7F | 0x1F80 |
| 1 | 0 | 256 words | 4 | 0x0000 - 0x1EFF | 0x1F00 - 0x1FFF | 0x1EFF | 0x1F00 |
| 0 | 1 | 512 words | 8 | 0x0000 - 0x1DFF | 0x1E00 - 0x1FFF | 0x1DFF | 0x1E00 |
| 0 | 0 | 1024 words | 16 | 0x0000 - 0x1BFF | 0x1C00 - 0x1FFF | 0x1BFF | 0x1C00 |

Note: The different BOOTSZ Fuse configurations are shown in Figure 27-2 on page 280.

**Table 27-11.** Read-While-Write Limit, Atmel ATmega168PA

| Section | Pages | Address |
|---|---|---|
| Read-While-Write section (RWW) | 112 | 0x0000 - 0x1BFF |
| No Read-While-Write section (NRWW) | 16 | 0x1C00 - 0x1FFF |

For details about these two section, see "NRWW – No Read-While-Write Section" on page 278 and "RWW – Read-While-Write Section" on page 278

**Table 27-12.** Explanation of Different Variables used in Figure 27-3 and the Mapping to the Z-pointer, Atmel ATmega168PA

| Variable | | Corresponding Z-value[1] | Description |
|---|---|---|---|
| PCMSB | 12 | | Most significant bit in the Program Counter. (The Program Counter is 13 bits PC[12:0]) |
| PAGEMSB | 5 | | Most significant bit which is used to address the words within one page (64 words in a page requires 6 bits PC [5:0]) |
| ZPCMSB | | Z13 | Bit in Z-register that is mapped to PCMSB. Because Z0 is not used, the ZPCMSB equals PCMSB + 1. |
| ZPAGEMSB | | Z6 | Bit in Z-register that is mapped to PAGEMSB. Because Z0 is not used, the ZPAGEMSB equals PAGEMSB + 1. |
| PCPAGE | PC[12:6] | Z13:Z7 | Program counter page address: Page select, for page erase and page write |
| PCWORD | PC[5:0] | Z6:Z1 | Program counter word address: Word select, for filling temporary buffer (must be zero during page write operation) |

Note: 1. Z15:Z14: always ignored
Z0: should be zero for all SPM commands, byte select for the LPM instruction.
See "Addressing the Flash During Self-Programming" on page 282 for details about the use of Z-pointer during Self-Programming.

## 27.9 Register Description

### 27.9.1 SPMCSR – Store Program Memory Control and Status Register

The Store Program Memory Control and Status Register contains the control bits needed to control the Boot Loader operations.

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| 0x37 (0x57) | SPMIE | RWWSB | – | RWWSRE | BLBSET | PGWRT | PGERS | SELFPRGEN | SPMCSR |
| Read/Write | R/W | R | R | R/W | R/W | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

- **Bit 7 – SPMIE: SPM Interrupt Enable**

When the SPMIE bit is written to one, and the I-bit in the Status Register is set (one), the SPM ready interrupt will be enabled. The SPM ready Interrupt will be executed as long as the SELFPRGEN bit in the SPMCSR Register is cleared.

- **Bit 6 – RWWSB: Read-While-Write Section Busy**

When a Self-Programming (Page Erase or Page Write) operation to the RWW section is initiated, the RWWSB will be set (one) by hardware. When the RWWSB bit is set, the RWW section cannot be accessed. The RWWSB bit will be cleared if the RWWSRE bit is written to one after a Self-Programming operation is completed. Alternatively the RWWSB bit will automatically be cleared if a page load operation is initiated.

- **Bit 5 – Reserved**

This bit is a reserved bit in the Atmel® ATmega48PA/88PA/168PA and always read as zero.

- **Bit 4 – RWWSRE: Read-While-Write Section Read Enable**

When programming (Page Erase or Page Write) to the RWW section, the RWW section is blocked for reading (the RWWSB will be set by hardware). To re-enable the RWW section, the user software must wait until the programming is completed (SELFPRGEN will be cleared). Then, if the RWWSRE bit is written to one at the same time as SELFPRGEN, the next SPM instruction within four clock cycles re-enables the RWW section. The RWW section cannot be re-enabled while the Flash is busy with a Page Erase or a Page Write (SELFPRGEN is set). If the RWWSRE bit is written while the Flash is being loaded, the Flash load operation will abort and the data loaded will be lost.

- **Bit 3 – BLBSET: Boot Lock Bit Set**

If this bit is written to one at the same time as SELFPRGEN, the next SPM instruction within four clock cycles sets Boot Lock bits and Memory Lock bits, according to the data in R0. The data in R1 and the address in the Z-pointer are ignored. The BLBSET bit will automatically be cleared upon completion of the Lock bit set, or if no SPM instruction is executed within four clock cycles.

An LPM instruction within three cycles after BLBSET and SELFPRGEN are set in the SPMCSR Register, will read either the Lock bits or the Fuse bits (depending on Z0 in the Z-pointer) into the destination register. See “Reading the Fuse and Lock Bits from Software” on page 285 for details.

- **Bit 2 – PGWRT: Page Write**

If this bit is written to one at the same time as SELFPRGEN, the next SPM instruction within four clock cycles executes Page Write, with the data stored in the temporary buffer. The page address is taken from the high part of the Z-pointer. The data in R1 and R0 are ignored. The PGWRT bit will auto-clear upon completion of a Page Write, or if no SPM instruction is executed within four clock cycles. The CPU is halted during the entire Page Write operation if the NRWW section is addressed.

- **Bit 1 – PGERS: Page Erase**

If this bit is written to one at the same time as SELFPRGEN, the next SPM instruction within four clock cycles executes Page Erase. The page address is taken from the high part of the Z-pointer. The data in R1 and R0 are ignored. The PGERS bit will auto-clear upon completion of a Page Erase, or if no SPM instruction is executed within four clock cycles. The CPU is halted during the entire Page Write operation if the NRWW section is addressed.

- **Bit 0 – SELFPRGEN: Self Programming Enable**

This bit enables the SPM instruction for the next four clock cycles. If written to one together with either RWWSRE, BLBSET, PGWRT or PGERS, the following SPM instruction will have a special meaning, see description above. If only SELFPRGEN is written, the following SPM instruction will store the value in R1:R0 in the temporary page buffer addressed by the Z-pointer. The LSB of the Z-pointer is ignored. The SELFPRGEN bit will auto-clear upon completion of an SPM instruction, or if no SPM instruction is executed within four clock cycles. During Page Erase and Page Write, the SELFPRGEN bit remains high until the operation is completed.

Writing any other combination than “10001”, “01001”, “00101”, “00011” or “00001” in the lower five bits will have no effect.

# 28. Memory Programming

## 28.1 Program And Data Memory Lock Bits

The Atmel® ATmega48PA provides two Lock bits and the Atmel ATmega48PA/88PA/168PA provides six Lock bits. These can be left unprogrammed (“1”) or can be programmed (“0”) to obtain the additional features listed in Table 28-2. The Lock bits can only be erased to “1” with the Chip Erase command.

The Atmel ATmega48PA has no separate Boot Loader section, and the SPM instruction is enabled for the whole Flash if the SELFPRGEN fuse is programmed (“0”). Otherwise the SPM instruction is disabled.

**Table 28-1.** Lock Bit Byte[1]

| Lock Bit Byte | Bit No | Description | Default Value |
|---|---|---|---|
| | 7 | – | 1 (unprogrammed) |
| | 6 | – | 1 (unprogrammed) |
| BLB12() | 5 | Boot Lock bit | 1 (unprogrammed) |
| BLB11() | 4 | Boot Lock bit | 1 (unprogrammed) |
| BLB02() | 3 | Boot Lock bit | 1 (unprogrammed) |
| BLB01() | 2 | Boot Lock bit | 1 (unprogrammed) |
| LB2 | 1 | Lock bit | 1 (unprogrammed) |
| LB1 | 0 | Lock bit | 1 (unprogrammed) |

Notes: 1. “1” means unprogrammed, “0” means programmed.
2. Only on Atmel ATmega48PA/88PA/168PA.

Notes: 1. “1” means unprogrammed, “0” means programmed.
2. Only on Atmel ATmega48PA/88PA/168PA.

**Table 28-2.** Lock Bit Protection Modes[1][2]

| Memory Lock Bits | | | Protection Type |
|---|---|---|---|
| LB Mode | LB2 | LB1 | |
| 1 | 1 | 1 | No memory lock features enabled. |
| 2 | 1 | 0 | Further programming of the Flash and EEPROM is disabled in Parallel and Serial Programming mode. The Fuse bits are locked in both Serial and Parallel Programming mode.[1] |
| 3 | 0 | 0 | Further programming and verification of the Flash and EEPROM is disabled in Parallel and Serial Programming mode. The Boot Lock bits and Fuse bits are locked in both Serial and Parallel Programming mode.[1] |

Notes: 1. “Program the Fuse bits and Boot Lock bits before programming the LB1 and LB2.
2. “1” means unprogrammed, “0” means programmed

**Table 28-3.** Lock Bit Protection Modes[1][2] (only Atmel ATmega48PA/88PA/168PA)

| BLB0 Mode | BLB02 | BLB01 | |
|---|---|---|---|
| 1 | 1 | 1 | No restrictions for SPM or LPM accessing the Application section. |
| 2 | 1 | 0 | SPM is not allowed to write to the Application section. |
| 3 | 0 | 0 | SPM is not allowed to write to the Application section, and LPM executing from the Boot Loader section is not allowed to read from the Application section. If Interrupt Vectors are placed in the Boot Loader section, interrupts are disabled while executing from the Application section. |
| 4 | 0 | 1 | LPM executing from the Boot Loader section is not allowed to read from the Application section. If Interrupt Vectors are placed in the Boot Loader section, interrupts are disabled while executing from the Application section. |
| **BLB1 Mode** | **BLB12** | **BLB11** | |
| 1 | 1 | 1 | No restrictions for SPM or LPM accessing the Boot Loader section. |
| 2 | 1 | 0 | SPM is not allowed to write to the Boot Loader section. |
| 3 | 0 | 0 | SPM is not allowed to write to the Boot Loader section, and LPM executing from the Application section is not allowed to read from the Boot Loader section. If Interrupt Vectors are placed in the Application section, interrupts are disabled while executing from the Boot Loader section. |
| 4 | 0 | 1 | LPM executing from the Application section is not allowed to read from the Boot Loader section. If Interrupt Vectors are placed in the Application section, interrupts are disabled while executing from the Boot Loader section. |

Notes: 1. Program the Fuse bits and Boot Lock bits before programming the LB1 and LB2.

2. “1” means unprogrammed, “0” means programmed

## 28.2 Fuse Bits

The Atmel® ATmega48PA/88PA/168PA has three Fuse bytes. Table 28-5 - Table 28-7 describe briefly the functionality of all the fuses and how they are mapped into the Fuse bytes. Note that the fuses are read as logical zero, “0”, if they are programmed.

**Table 28-4.** Extended Fuse Byte for the Atmel ATmega48PA

| Extended Fuse Byte | Bit No | Description | Default Value |
|---|---|---|---|
| – | 7 | – | 1 |
| – | 6 | – | 1 |
| – | 5 | – | 1 |
| – | 4 | – | 1 |
| – | 3 | – | 1 |
| – | 2 | – | 1 |
| – | 1 | – | 1 |
| SELFPRGEN | 0 | Self Programming Enable | 1 (unprogrammed) |

**Table 28-5.** Extended Fuse Byte for Atmel ATmega88PA/168PA

| Extended Fuse Byte | Bit No | Description | Default Value |
|---|---|---|---|
| – | 7 | – | 1 |
| – | 6 | – | 1 |
| – | 5 | – | 1 |
| – | 4 | – | 1 |
| – | 3 | – | 1 |
| BOOTSZ1 | 2 | Select Boot Size (see Table 27-7 on page 290 and Table 27-10 on page 291 for details) | 0 (programmed)[1] |
| BOOTSZ0 | 1 | Select Boot Size (see Table 27-7 on page 290 and Table 27-10 on page 291 for details) | 0 (programmed)[1] |
| BOOTRST | 0 | Select Reset Vector | 1 (unprogrammed) |

Note: 1. The default value of BOOTSZ[1:0] results in maximum Boot Size. See "Pin Name Mapping" on page 299.

**Table 28-6.** Fuse High Byte for the Atmel ATmega48PA/88PA/168PA

| High Fuse Byte | Bit No | Description | Default Value |
|---|---|---|---|
| RSTDISBL[1] | 7 | External Reset Disable | 1 (unprogrammed) |
| DWEN | 6 | debugWIRE Enable | 1 (unprogrammed) |
| SPIEN[2] | 5 | Enable Serial Program and Data Downloading | 0 (programmed, SPI programming enabled) |
| WDTON[3] | 4 | Watchdog Timer Always On | 1 (unprogrammed) |
| EESAVE | 3 | EEPROM memory is preserved through the Chip Erase | 1 (unprogrammed), EEPROM not reserved |
| BODLEVEL2[4] | 2 | Brown-out Detector trigger level | 1 (unprogrammed) |
| BODLEVEL1[4] | 1 | Brown-out Detector trigger level | 0 (programmed) |
| BODLEVEL0[4] | 0 | Brown-out Detector trigger level | 1 (unprogrammed) |

Notes: 1. See "Alternate Functions of Port C" on page 85 for description of RSTDISBL Fuse.

2. The SPIEN Fuse is not accessible in serial programming mode.

3. See "WDTCSR – Watchdog Timer Control Register" on page 56 for details.

4. See Table 29-6 on page 317 for BODLEVEL Fuse decoding (default = 2.7V).

**Table 28-7.** Fuse Low Byte

| Low Fuse Byte | Bit No | Description | Default Value |
|---|---|---|---|
| CKDIV8[4] | 7 | Divide clock by 8 | 0 (programmed) |
| CKOUT[3] | 6 | Clock output | 1 (unprogrammed) |
| SUT1 | 5 | Select start-up time | 1 (unprogrammed)[1] |
| SUT0 | 4 | Select start-up time | 0 (programmed)[1] |
| CKSEL3 | 3 | Select Clock source | 0 (programmed)[2] |
| CKSEL2 | 2 | Select Clock source | 0 (programmed)[2] |
| CKSEL1 | 1 | Select Clock source | 1 (unprogrammed)[2] |
| CKSEL0 | 0 | Select Clock source | 0 (programmed)[2] |

Note:
1. The default value of SUT1...0 results in maximum start-up time for the default clock source. See Table 9-12 on page 33 for details.
2. The default setting of CKSEL3...0 results in internal RC Oscillator at 8MHz. See Table 9-11 on page 33 for details.
3. The CKOUT Fuse allows the system clock to be output on PORTB0. See “Clock Output Buffer” on page 35 for details.
4. See “System Clock Prescaler” on page 36 for details.

The status of the Fuse bits is not affected by Chip Erase. Note that the Fuse bits are locked if Lock bit1 (LB1) is programmed. Program the Fuse bits before programming the Lock bits.

### 28.2.1 Latching of Fuses

The fuse values are latched when the device enters programming mode and changes of the fuse values will have no effect until the part leaves Programming mode. This does not apply to the EESAVE Fuse which will take effect once it is programmed. The fuses are also latched on Power-up in Normal mode.

## 28.3 Signature Bytes

All Atmel microcontrollers have a three-byte signature code which identifies the device. This code can be read in both serial and parallel mode, also when the device is locked. The three bytes reside in a separate address space. For the Atmel® ATmega48PA/88PA/168PA the signature bytes are given in Table 28-8.

**Table 28-8.** Device ID

| | Signature Bytes Address | | |
|---|---|---|---|
| Part | 0x000 | 0x002 | 0x004 |
| Atmel ATmega48PA/88PA/168PA | 0x1E | 0x92 | 0x0A |
| Atmel ATmega88PA | 0x1E | 0x93 | 0x0F |
| Atmel ATmega168PA | 0x1E | 0x94 | 0x0B |

## 28.4 Calibration Byte

The Atmel ATmega48PA/88PA/168PA has 2 calibration values for the Internal RC Oscillator. The 3V calibration byte resides in the address 0x0001 in the signature address space and the 5V calibration byte resides in the address 0x0009. During reset, the 3V calibration byte is automatically written into the OSCCAL Register to ensure correct frequency of the calibrated RC Oscillator.

## 28.5 Page Size

**Table 28-9.** No. of Words in a Page and No. of Pages in the Flash

| Device | Flash Size | Page Size | PCWORD | No. of Pages | PCPAGE | PCMSB |
|---|---|---|---|---|---|---|
| Atmel ATmega48PA/ 88PA/168PA | 2K words (4K bytes) | 32 words | PC[4:0] | 64 | PC[10:5] | 10 |
| Atmel ATmega88PA | 4K words (8K bytes) | 32 words | PC[4:0] | 128 | PC[11:5] | 11 |
| Atmel ATmega168PA | 8K words (16K bytes) | 64 words | PC[5:0] | 128 | PC[12:6] | 12 |

**Table 28-10.** No. of Words in a Page and No. of Pages in the EEPROM

| Device | EEPROM Size | Page Size | PCWORD | No. of Pages | PCPAGE | EEAMSB |
|---|---|---|---|---|---|---|
| Atmel ATmega48PA/ 88PA/168PA | 256 bytes | 4 bytes | EEA[1:0] | 64 | EEA[7:2] | 7 |
| Atmel ATmega88PA | 512 bytes | 4 bytes | EEA[1:0] | 128 | EEA[8:2] | 8 |
| Atmel ATmega168PA | 512 bytes | 4 bytes | EEA[1:0] | 128 | EEA[8:2] | 8 |

## 28.6 Parallel Programming Parameters, Pin Mapping, and Commands

This section describes how to parallel program and verify Flash Program memory, EEPROM Data memory, Memory Lock bits, and Fuse bits in the Atmel® ATmega48PA/88PA/168PA. Pulses are assumed to be at least 250 ns unless otherwise noted.

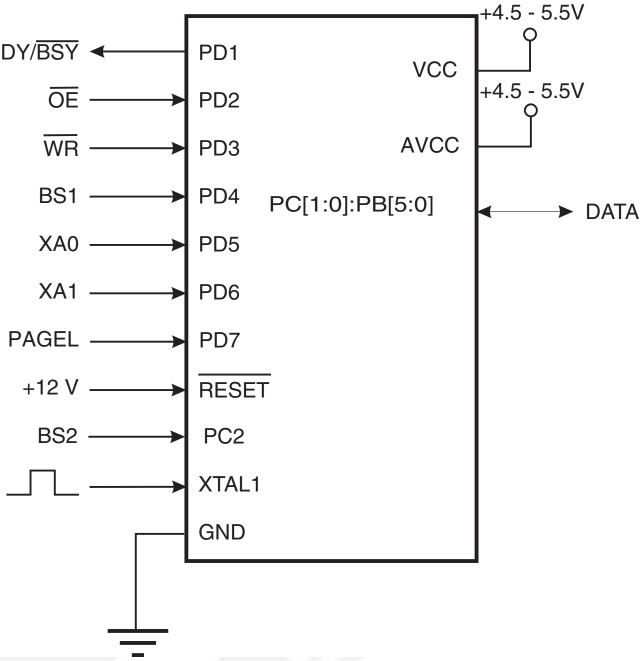
### 28.6.1 Signal Names

In this section, some pins of the Atmel ATmega48PA/88PA/168PA are referenced by signal names describing their functionality during parallel programming, see Figure 28-1 and Table 28-11. Pins not described in the following table are referenced by pin names.

The XA1/XA0 pins determine the action executed when the XTAL1 pin is given a positive pulse. The bit coding is shown in Table 28-13.

When pulsing $\overline{\text{WR}}$ or $\overline{\text{OE}}$, the command loaded determines the action executed. The different Commands are shown in Table 28-14.

**Figure 28-1.** Parallel Programming



Note: $V_{CC}$ - 0.3V < $AV_{CC}$ < $V_{CC}$ + 0.3V, however, $AV_{CC}$ should always be within 4.5 - 5.5V

**Table 28-11.** Pin Name Mapping

| Signal Name in Programming Mode | Pin Name | I/O | Function |
|---|---|---|---|
| RDY/$\overline{BSY}$ | PD1 | O | 0: Device is busy programming, 1: Device is ready for new command |
| $\overline{OE}$ | PD2 | I | Output Enable (Active low) |
| $\overline{WR}$ | PD3 | I | Write Pulse (Active low) |
| BS1 | PD4 | I | Byte Select 1 (“0” selects Low byte, “1” selects High byte) |
| XA0 | PD5 | I | XTAL Action Bit 0 |
| XA1 | PD6 | I | XTAL Action Bit 1 |
| PAGEL | PD7 | I | Program memory and EEPROM Data Page Load |
| BS2 | PC2 | I | Byte Select 2 (“0” selects Low byte, “1” selects 2’nd High byte) |
| DATA | {PC[1:0]: PB[5:0]} | I/O | Bi-directional Data bus (Output when $\overline{OE}$ is low) |

**Table 28-12.** Pin Values Used to Enter Programming Mode

| Pin | Symbol | Value |
|---|---|---|
| PAGEL | Prog_enable[3] | 0 |
| XA1 | Prog_enable[2] | 0 |
| XA0 | Prog_enable[1] | 0 |
| BS1 | Prog_enable[0] | 0 |

**Table 28-13.** XA1 and XA0 Coding

| XA1 | XA0 | Action when XTAL1 is Pulsed |
|---|---|---|
| 0 | 0 | Load Flash or EEPROM Address (High or low address byte determined by BS1). |
| 0 | 1 | Load Data (High or Low data byte for Flash determined by BS1). |
| 1 | 0 | Load Command |
| 1 | 1 | No Action, Idle |

**Table 28-14.** Command Byte Bit Coding

| Command Byte | Command Executed |
|---|---|
| 1000 0000 | Chip Erase |
| 0100 0000 | Write Fuse bits |
| 0010 0000 | Write Lock bits |
| 0001 0000 | Write Flash |
| 0001 0001 | Write EEPROM |
| 0000 1000 | Read Signature Bytes and Calibration byte |
| 0000 0100 | Read Fuse and Lock bits |
| 0000 0010 | Read Flash |
| 0000 0011 | Read EEPROM |

## 28.7 Parallel Programming

### 28.7.1 Enter Programming Mode

The following algorithm puts the device in Parallel (High-voltage) Programming mode:

1. Set Prog_enable pins listed in Table 28-12 on page 299 to “0000”, RESET pin to 0V and $V_{CC}$ to 0V.
2. Apply 4.5 - 5.5V between $V_{CC}$ and GND.

Ensure that $V_{CC}$ reaches at least 1.8V within the next 20 µs.

3. Wait 20 - 60 µs, and apply 11.5 - 12.5V to RESET.
4. Keep the Prog_enable pins unchanged for at least 10µs after the High-voltage has been applied to ensure the Prog_enable Signature has been latched.
5. Wait at least 300 µs before giving any parallel programming commands.
6. Exit Programming mode by power the device down or by bringing RESET pin to 0V.

If the rise time of the $V_{CC}$ is unable to fulfill the requirements listed above, the following alternative algorithm can be used.

1. Set Prog_enable pins listed in Table 28-12 on page 299 to “0000”, RESET pin to 0V and $V_{CC}$ to 0V.
2. Apply 4.5 - 5.5V between $V_{CC}$ and GND.
3. Monitor $V_{CC}$, and as soon as $V_{CC}$ reaches 0.9 - 1.1V, apply 11.5 - 12.5V to RESET.
4. Keep the Prog_enable pins unchanged for at least 10µs after the High-voltage has been applied to ensure the Prog_enable Signature has been latched.
5. Wait until $V_{CC}$ actually reaches 4.5 -5.5V before giving any parallel programming commands.
6. Exit Programming mode by power the device down or by bringing RESET pin to 0V.

### 28.7.2 Considerations for Efficient Programming

The loaded command and address are retained in the device during programming. For efficient programming, the following should be considered.

- The command needs only be loaded once when writing or reading multiple memory locations.
- Skip writing the data value 0xFF, that is the contents of the entire EEPROM (unless the EESAVE Fuse is programmed) and Flash after a Chip Erase.
- Address high byte needs only be loaded before programming or reading a new 256 word window in Flash or 256 byte EEPROM. This consideration also applies to Signature bytes reading.

### 28.7.3 Chip Erase

The Chip Erase will erase the Flash and EEPROM[1] memories plus Lock bits. The Lock bits are not reset until the program memory has been completely erased. The Fuse bits are not changed. A Chip Erase must be performed before the Flash and/or EEPROM are reprogrammed.

Note: 1. The EEPRPOM memory is preserved during Chip Erase if the EESAVE Fuse is programmed.

Load Command "Chip Erase"

1. Set XA1, XA0 to "10". This enables command loading.
2. Set BS1 to "0".
3. Set DATA to "1000 0000". This is the command for Chip Erase.
4. Give XTAL1 a positive pulse. This loads the command.
5. Give $\overline{\text{WR}}$ a negative pulse. This starts the Chip Erase. RDY/$\overline{\text{BSY}}$ goes low.
6. Wait until RDY/$\overline{\text{BSY}}$ goes high before loading a new command.

### 28.7.4 Programming the Flash

The Flash is organized in pages, see Table 28-9 on page 298. When programming the Flash, the program data is latched into a page buffer. This allows one page of program data to be programmed simultaneously. The following procedure describes how to program the entire Flash memory:

A. Load Command "Write Flash"

1. Set XA1, XA0 to "10". This enables command loading.
2. Set BS1 to "0".
3. Set DATA to "0001 0000". This is the command for Write Flash.
4. Give XTAL1 a positive pulse. This loads the command.

B. Load Address Low byte

1. Set XA1, XA0 to "00". This enables address loading.
2. Set BS1 to "0". This selects low address.
3. Set DATA = Address low byte (0x00 - 0xFF).
4. Give XTAL1 a positive pulse. This loads the address low byte.

C. Load Data Low Byte

1. Set XA1, XA0 to “01”. This enables data loading.
2. Set DATA = Data low byte (0x00 - 0xFF).
3. Give XTAL1 a positive pulse. This loads the data byte.

D. Load Data High Byte

1. Set BS1 to “1”. This selects high data byte.
2. Set XA1, XA0 to “01”. This enables data loading.
3. Set DATA = Data high byte (0x00 - 0xFF).
4. Give XTAL1 a positive pulse. This loads the data byte.

E. Latch Data

1. Set BS1 to “1”. This selects high data byte.
2. Give PAGEL a positive pulse. This latches the data bytes. (See Figure 28-3 for signal waveforms)

F. Repeat B through E until the entire buffer is filled or until all data within the page is loaded.

While the lower bits in the address are mapped to words within the page, the higher bits address the pages within the FLASH. This is illustrated in Figure 28-2 on page 303. Note that if less than eight bits are required to address words in the page (pagesize < 256), the most significant bit(s) in the address low byte are used to address the page when performing a Page Write.

G. Load Address High byte

1. Set XA1, XA0 to “00”. This enables address loading.
2. Set BS1 to “1”. This selects high address.
3. Set DATA = Address high byte (0x00 - 0xFF).
4. Give XTAL1 a positive pulse. This loads the address high byte.

H. Program Page

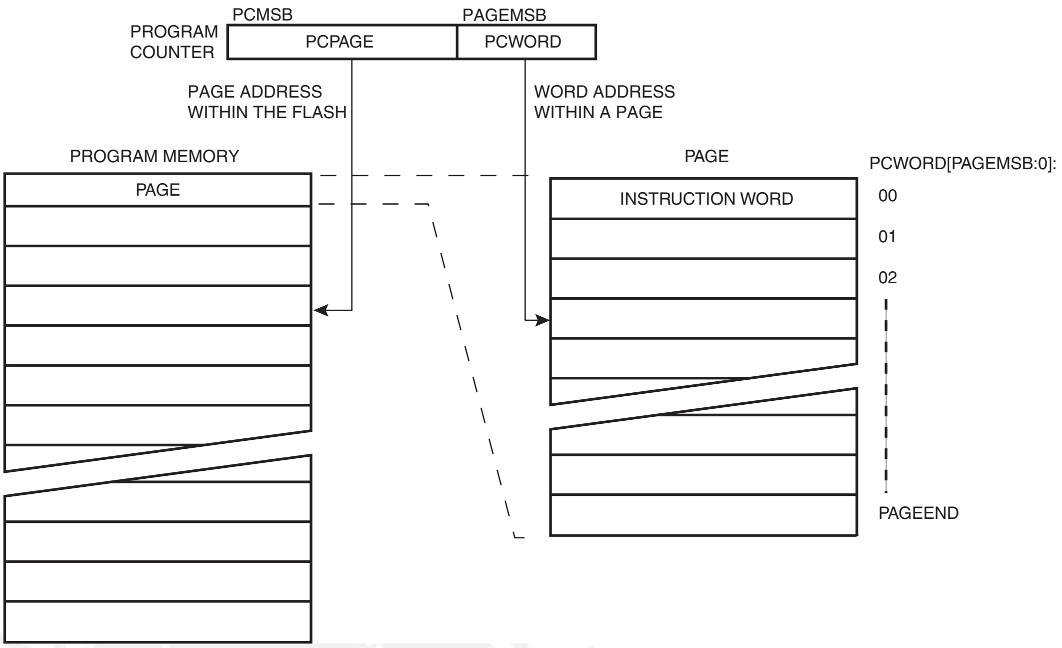
1. Give $\overline{\text{WR}}$ a negative pulse. This starts programming of the entire page of data. RDY/$\overline{\text{BSY}}$ goes low.
2. Wait until RDY/$\overline{\text{BSY}}$ goes high (See Figure 28-3 for signal waveforms).

I. Repeat B through H until the entire Flash is programmed or until all data has been programmed.

J. End Page Programming

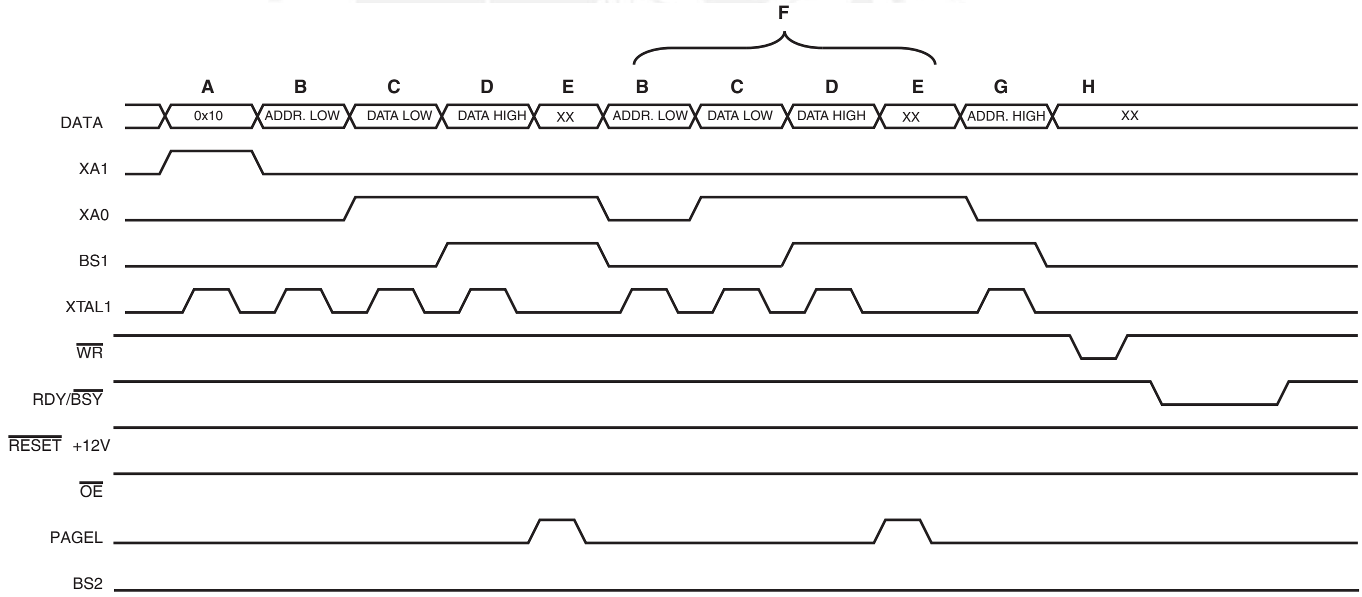
1. 1. Set XA1, XA0 to “10”. This enables command loading.
2. Set DATA to “0000 0000”. This is the command for No Operation.
3. Give XTAL1 a positive pulse. This loads the command, and the internal write signals are reset.

**Figure 28-2.** Addressing the Flash Which is Organized in Pages[1]



Note: 1. PCPAGE and PCWORD are listed in Table 28-9 on page 298.

**Figure 28-3.** Programming the Flash Waveforms[1]



Note: 1. “XX” is don’t care. The letters refer to the programming description above.

### 28.7.5 Programming the EEPROM

The EEPROM is organized in pages, see Table 28-10 on page 298. When programming the EEPROM, the program data is latched into a page buffer. This allows one page of data to be programmed simultaneously. The programming algorithm for the EEPROM data memory is as follows (refer to “Programming the Flash” on page 301 for details on Command, Address and Data loading):

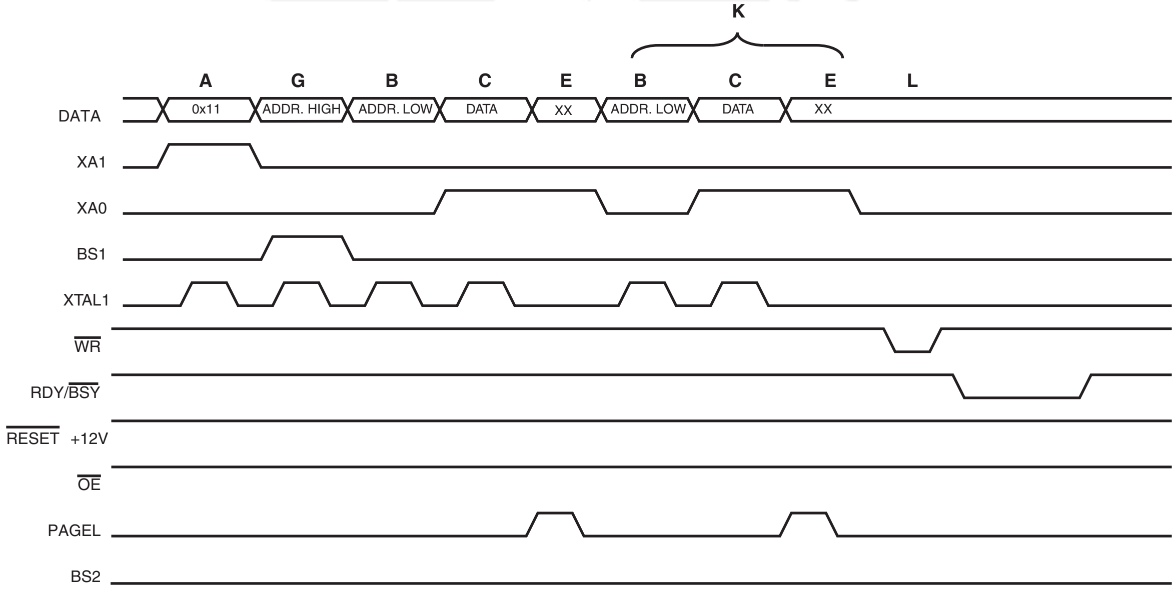
1. A: Load Command “0001 0001”.
2. G: Load Address High Byte (0x00 - 0xFF).
3. B: Load Address Low Byte (0x00 - 0xFF).
4. C: Load Data (0x00 - 0xFF).
5. E: Latch data (give PAGEL a positive pulse).

K: Repeat 3 through 5 until the entire buffer is filled.

L: Program EEPROM page

1. Set BS1 to “0”.
2. Give $\overline{\text{WR}}$ a negative pulse. This starts programming of the EEPROM page. RDY/$\overline{\text{BSY}}$ goes low.
3. Wait until to RDY/$\overline{\text{BSY}}$ goes high before programming the next page (See Figure 28-4 for signal waveforms).

**Figure 28-4.** Programming the EEPROM Waveforms



### 28.7.6 Reading the Flash

The algorithm for reading the Flash memory is as follows (refer to "Programming the Flash" on page 301 for details on Command and Address loading):

1. A: Load Command "0000 0010".
2. G: Load Address High Byte (0x00 - 0xFF).
3. B: Load Address Low Byte (0x00 - 0xFF).
4. Set $\overline{\text{OE}}$ to "0", and BS1 to "0". The Flash word low byte can now be read at DATA.
5. Set BS1 to "1". The Flash word high byte can now be read at DATA.
6. Set $\overline{\text{OE}}$ to "1".

### 28.7.7 Reading the EEPROM

The algorithm for reading the EEPROM memory is as follows (refer to "Programming the Flash" on page 301 for details on Command and Address loading):

1. A: Load Command "0000 0011".
2. G: Load Address High Byte (0x00 - 0xFF).
3. B: Load Address Low Byte (0x00 - 0xFF).
4. Set $\overline{\text{OE}}$ to "0", and BS1 to "0". The EEPROM Data byte can now be read at DATA.
5. Set $\overline{\text{OE}}$ to "1".

### 28.7.8 Programming the Fuse Low Bits

The algorithm for programming the Fuse Low bits is as follows (refer to "Programming the Flash" on page 301 for details on Command and Data loading):

1. A: Load Command "0100 0000".
2. C: Load Data Low Byte. Bit n = "0" programs and bit n = "1" erases the Fuse bit.
3. Give $\overline{\text{WR}}$ a negative pulse and wait for RDY/$\overline{\text{BSY}}$ to go high.

### 28.7.9 Programming the Fuse High Bits

The algorithm for programming the Fuse High bits is as follows (refer to "Programming the Flash" on page 301 for details on Command and Data loading):

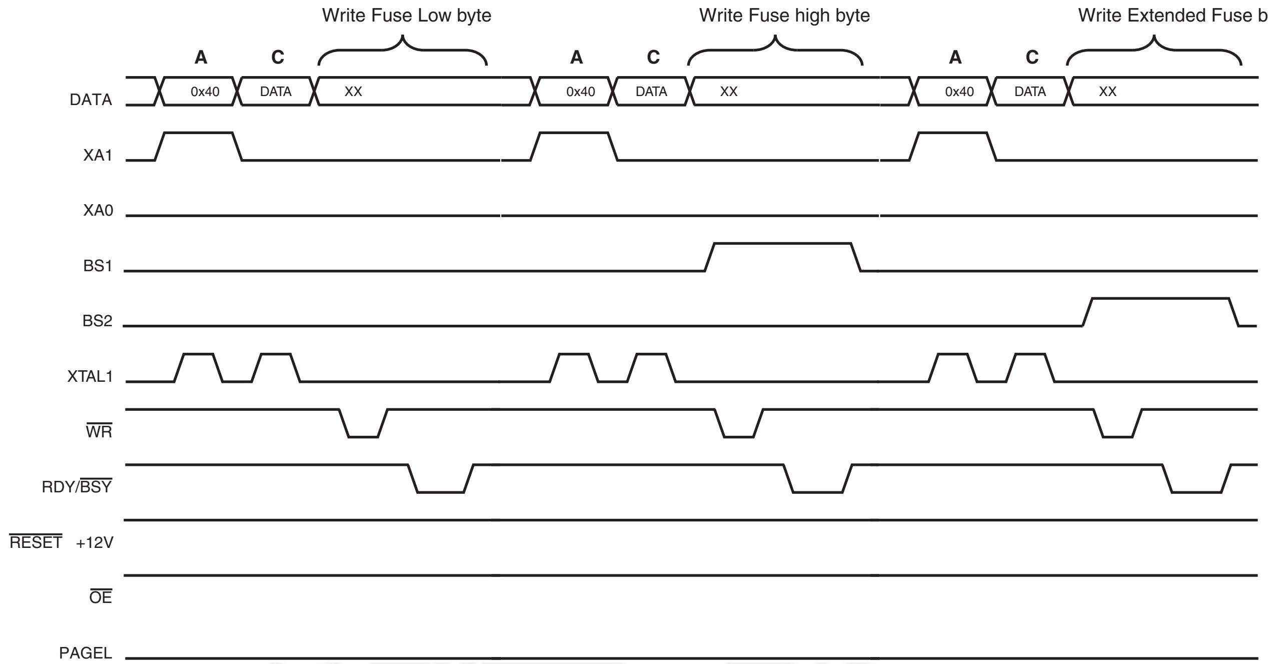
1. A: Load Command "0100 0000".
2. C: Load Data Low Byte. Bit n = "0" programs and bit n = "1" erases the Fuse bit.
3. Set BS1 to "1" and BS2 to "0". This selects high data byte.
4. Give $\overline{\text{WR}}$ a negative pulse and wait for RDY/$\overline{\text{BSY}}$ to go high.
5. Set BS1 to "0". This selects low data byte.

### 28.7.10 Programming the Extended Fuse Bits

The algorithm for programming the Extended Fuse bits is as follows (refer to "Programming the Flash" on page 301 for details on Command and Data loading):

1. 1. A: Load Command "0100 0000".
2. 2. C: Load Data Low Byte. Bit n = "0" programs and bit n = "1" erases the Fuse bit.
3. 3. Set BS1 to "0" and BS2 to "1". This selects extended data byte.
4. 4. Give $\overline{\text{WR}}$ a negative pulse and wait for RDY/$\overline{\text{BSY}}$ to go high.
5. 5. Set BS2 to "0". This selects low data byte.

**Figure 28-5.** Programming the FUSES Waveforms



### 28.7.11 Programming the Lock Bits

The algorithm for programming the Lock bits is as follows (refer to "Programming the Flash" on page 301 for details on Command and Data loading):

1. A: Load Command "0010 0000".
2. C: Load Data Low Byte. Bit n = "0" programs the Lock bit. If LB mode 3 is programmed (LB1 and LB2 is programmed), it is not possible to program the Boot Lock bits by any External Programming mode.
3. Give WR a negative pulse and wait for RDY/BSY to go high.

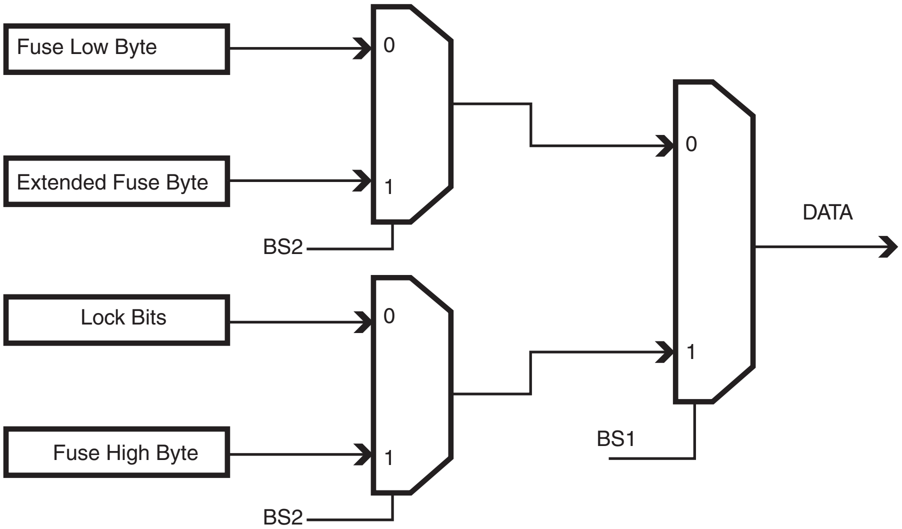
The Lock bits can only be cleared by executing Chip Erase.

### 28.7.12 Reading the Fuse and Lock Bits

The algorithm for reading the Fuse and Lock bits is as follows (refer to "Programming the Flash" on page 301 for details on Command loading):

1. A: Load Command "0000 0100".
2. Set OE to "0", BS2 to "0" and BS1 to "0". The status of the Fuse Low bits can now be read at DATA ("0" means programmed).
3. Set OE to "0", BS2 to "1" and BS1 to "1". The status of the Fuse High bits can now be read at DATA ("0" means programmed).
4. Set OE to "0", BS2 to "1", and BS1 to "0". The status of the Extended Fuse bits can now be read at DATA ("0" means programmed).
5. Set OE to "0", BS2 to "0" and BS1 to "1". The status of the Lock bits can now be read at DATA ("0" means programmed).
6. Set OE to "1".

**Figure 28-6.** Mapping Between BS1, BS2 and the Fuse and Lock Bits During Read



### 28.7.13 Reading the Signature Bytes

The algorithm for reading the Signature bytes is as follows (refer to "Programming the Flash" on page 301 for details on Command and Address loading):

1. A: Load Command "0000 1000".
2. B: Load Address Low Byte (0x00 - 0x02).
3. Set $\overline{\text{OE}}$ to "0", and BS1 to "0". The selected Signature byte can now be read at DATA.
4. Set $\overline{\text{OE}}$ to "1".

### 28.7.14 Reading the Calibration Byte

The algorithm for reading the Calibration byte is as follows (refer to "Programming the Flash" on page 301 for details on Command and Address loading):

1. A: Load Command "0000 1000".
2. B: Load Address Low Byte, 0x00.
3. Set $\overline{\text{OE}}$ to "0", and BS1 to "1". The Calibration byte can now be read at DATA.
4. Set $\overline{\text{OE}}$ to "1".

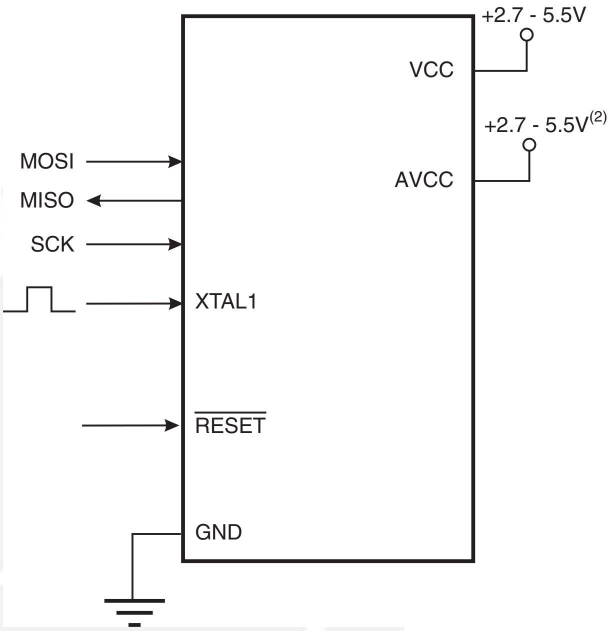
### 28.7.15 Parallel Programming Characteristics

For characteristics of the Parallel Programming, see "Parallel Programming Characteristics" on page 322.

## 28.8 Serial Downloading

Both the Flash and EEPROM memory arrays can be programmed using the serial SPI bus while $\overline{\text{RESET}}$ is pulled to GND. The serial interface consists of pins SCK, MOSI (input) and MISO (output). After $\overline{\text{RESET}}$ is set low, the Programming Enable instruction needs to be executed first before program/erase operations can be executed. NOTE, in Table 28-15 on page 309, the pin mapping for SPI programming is listed. Not all parts use the SPI pins dedicated for the internal SPI interface.

**Figure 28-7.** Serial Programming and Verify[1]



Notes: 1. If the device is clocked by the internal Oscillator, it is no need to connect a clock source to the XTAL1 pin.

2. $V_{CC}$ - 0.3V < $AV_{CC}$ < $V_{CC}$ + 0.3V, however, $AV_{CC}$ should always be within 2.7 - 5.5V

When programming the EEPROM, an auto-erase cycle is built into the self-timed programming operation (in the Serial mode ONLY) and there is no need to first execute the Chip Erase instruction. The Chip Erase operation turns the content of every memory location in both the Program and EEPROM arrays into 0xFF.

Depending on CKSEL Fuses, a valid clock must be present. The minimum low and high periods for the serial clock (SCK) input are defined as follows:

Low: > 2 CPU clock cycles for $f_{ck}$ < 12MHz, 3 CPU clock cycles for $f_{ck}$ >= 12MHz

High: > 2 CPU clock cycles for $f_{ck}$ < 12MHz, 3 CPU clock cycles for $f_{ck}$ >= 12MHz

### 28.8.1 Serial Programming Pin Mapping

**Table 28-15.** Pin Mapping Serial Programming

| Symbol | Pins | I/O | Description |
|---|---|---|---|
| MOSI | PB3 | I | Serial Data in |
| MISO | PB4 | O | Serial Data out |
| SCK | PB5 | I | Serial Clock |

### 28.8.2 Serial Programming Algorithm

When writing serial data to the Atmel® ATmega48PA/88PA/168PA, data is clocked on the rising edge of SCK.

When reading data from the Atmel ATmega48PA/88PA/168PA, data is clocked on the falling edge of SCK. See Figure 28-9 for timing details.

To program and verify the Atmel ATmega48PA/88PA/168PA in the serial programming mode, the following sequence is recommended (See Serial Programming Instruction set in Table 28-17 on page 311):

1. Power-up sequence:
   Apply power between $V_{CC}$ and GND while $\overline{\text{RESET}}$ and SCK are set to "0". In some systems, the programmer can not guarantee that SCK is held low during power-up. In this case, $\overline{\text{RESET}}$ must be given a positive pulse of at least two CPU clock cycles duration after SCK has been set to "0".
2. Wait for at least 20ms and enable serial programming by sending the Programming Enable serial instruction to pin MOSI.
3. The serial programming instructions will not work if the communication is out of synchronization. When in sync. the second byte (0x53), will echo back when issuing the third byte of the Programming Enable instruction. Whether the echo is correct or not, all four bytes of the instruction must be transmitted. If the 0x53 did not echo back, give $\overline{\text{RESET}}$ a positive pulse and issue a new Programming Enable command.
4. The Flash is programmed one page at a time. The memory page is loaded one byte at a time by supplying the 6 LSB of the address and data together with the Load Program Memory Page instruction. To ensure correct loading of the page, the data low byte must be loaded before data high byte is applied for a given address. The Program Memory Page is stored by loading the Write Program Memory Page instruction with the 7 MSB of the address. If polling (RDY/$\overline{\text{BSY}}$) is not used, the user must wait at least $t_{WD\_FLASH}$ before issuing the next page (See Table 28-16). Accessing the serial programming interface before the Flash write operation completes can result in incorrect programming.

5. **A**: The EEPROM array is programmed one byte at a time by supplying the address and data together with the appropriate Write instruction. An EEPROM memory location is first automatically erased before new data is written. If polling (RDY/$\overline{\text{BSY}}$) is not used, the user must wait at least $t_{WD\_EEPROM}$ before issuing the next byte (See Table 28-16). In a chip erased device, no 0xFFs in the data file(s) need to be programmed.
   **B**: The EEPROM array is programmed one page at a time. The Memory page is loaded one byte at a time by supplying the 6 LSB of the address and data together with the Load EEPROM Memory Page instruction. The EEPROM Memory Page is stored by loading the Write EEPROM Memory Page Instruction with the 7 MSB of the address. When using EEPROM page access only byte locations loaded with the Load EEPROM Memory Page instruction is altered. The remaining locations remain unchanged. If polling (RDY/$\overline{\text{BSY}}$) is not used, the used must wait at least $t_{WD\_EEPROM}$ before issuing the next byte (See Table 28-16). In a chip erased device, no 0xFF in the data file(s) need to be programmed.
6. Any memory location can be verified by using the Read instruction which returns the content at the selected address at serial output MISO.
7. At the end of the programming session, $\overline{\text{RESET}}$ can be set high to commence normal operation.
8. Power-off sequence (if needed):
   Set $\overline{\text{RESET}}$ to “1”.
   Turn $V_{CC}$ power off.

**Table 28-16.** Typical Wait Delay Before Writing the Next Flash or EEPROM Location

| Symbol | Minimum Wait Delay |
|---|---|
| $t_{WD\_FLASH}$ | 4.5ms |
| $t_{WD\_EEPROM}$ | 3.6ms |
| $t_{WD\_ERASE}$ | 9.0ms |

### 28.8.3 Serial Programming Instruction set

Table 28-17 on page 311 and Figure 28-8 on page 312 describes the Instruction set.

**Table 28-17.** Serial Programming Instruction Set (Hexadecimal values)

| | Instruction Format | | | |
|---|---|---|---|---|
| **Instruction/Operation** | **Byte 1** | **Byte 2** | **Byte 3** | **Byte4** |
| Programming Enable | $AC | $53 | $00 | $00 |
| Chip Erase (Program Memory/EEPROM) | $AC | $80 | $00 | $00 |
| Poll RDY/$\overline{\text{BSY}}$ | $F0 | $00 | $00 | data byte out |
| **Load Instructions** | | | | |
| Load Extended Address byte[1] | $4D | $00 | Extended adr | $00 |
| Load Program Memory Page, High byte | $48 | $00 | adr LSB | high data byte in |
| Load Program Memory Page, Low byte | $40 | $00 | adr LSB | low data byte in |
| Load EEPROM Memory Page (page access) | $C1 | $00 | 0000 000aa | data byte in |
| **Read Instructions** | | | | |
| Read Program Memory, High byte | $28 | adr MSB | adr LSB | high data byte out |
| Read Program Memory, Low byte | $20 | adr MSB | adr LSB | low data byte out |
| Read EEPROM Memory | $A0 | 0000 00aa | aaaa aaaa | data byte out |
| Read Lock bits | $58 | $00 | $00 | data byte out |
| Read Signature Byte | $30 | $00 | 0000 000aa | data byte out |
| Read Fuse bits | $50 | $00 | $00 | data byte out |
| Read Fuse High bits | $58 | $08 | $00 | data byte out |
| Read Extended Fuse Bits | $50 | $08 | $00 | data byte out |
| Read Calibration Byte | $38 | $00 | $00 | data byte out |
| **Write Instructions**[6] | | | | |
| Write Program Memory Page | $4C | adr MSB[8] | adr LSB[8] | $00 |
| Write EEPROM Memory | $C0 | 0000 00aa | aaaa aaaa | data byte in |
| Write EEPROM Memory Page (page access) | $C2 | 0000 00aa | aaaa aa00 | $00 |
| Write Lock bits | $AC | $E0 | $00 | data byte in |
| Write Fuse bits | $AC | $A0 | $00 | data byte in |
| Write Fuse High bits | $AC | $A8 | $00 | data byte in |
| Write Extended Fuse Bits | $AC | $A4 | $00 | data byte in |

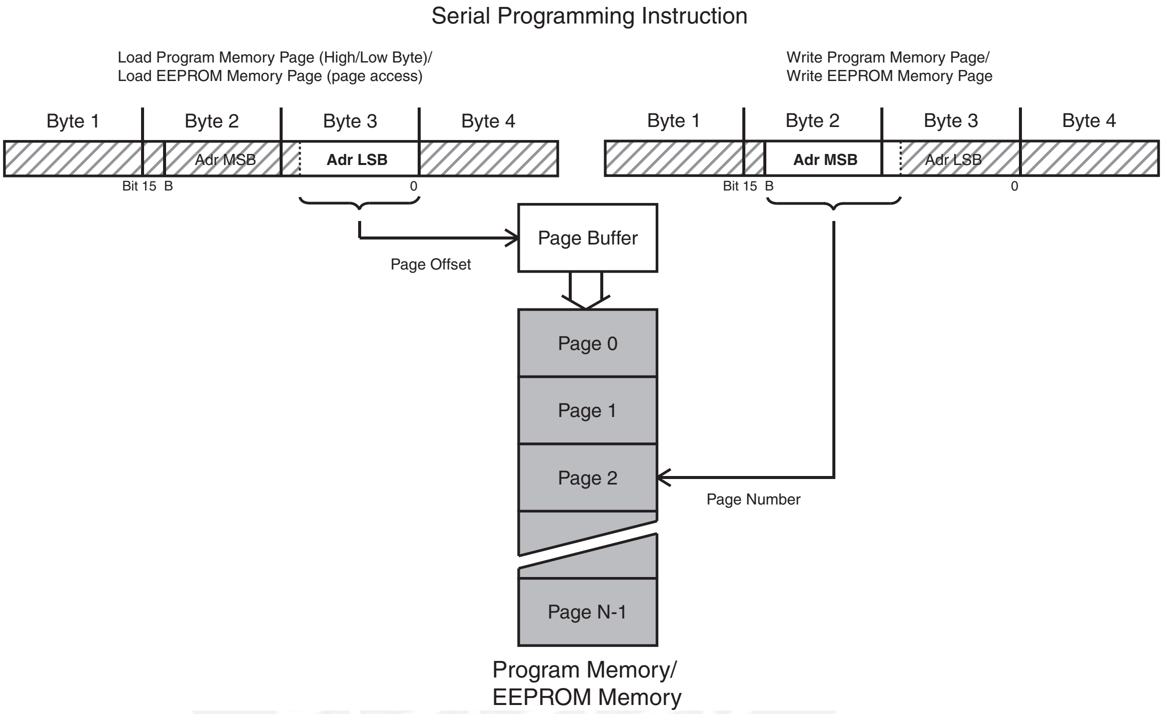
Notes:
1. Not all instructions are applicable for all parts.
2. a = address.
3. Bits are programmed ‘0’, unprogrammed ‘1’.
4. To ensure future compatibility, unused Fuses and Lock bits should be unprogrammed (‘1’).
5. Refer to the corresponding section for Fuse and Lock bits, Calibration and Signature bytes and Page size.
6. Instructions accessing program memory use a word address. This address may be random within the page range.
7. See http://www.atmel.com/avr for Application Notes regarding programming and programmers.
8. WORDS

If the LSB in RDY/BSY data byte out is ‘1’, a programming operation is still pending. Wait until this bit returns ‘0’ before the next instruction is carried out.

Within the same page, the low data byte must be loaded prior to the high data byte.

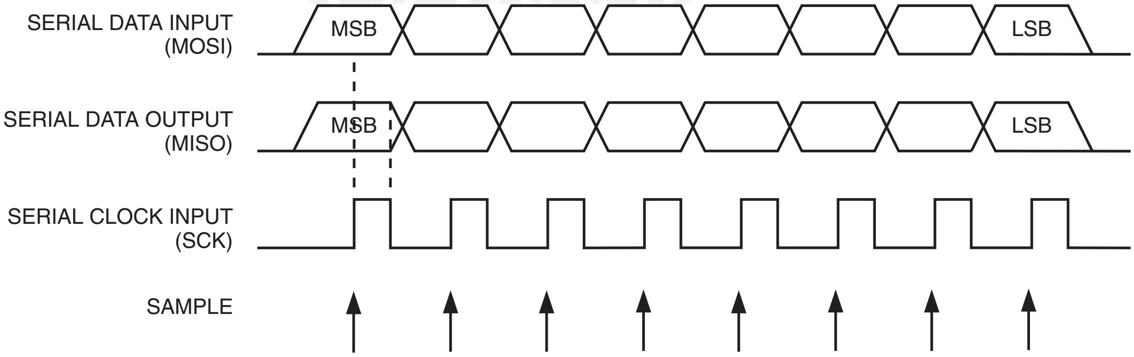
After data is loaded to the page buffer, program the EEPROM page, see Figure 28-8 on page 312.

**Figure 28-8.** Serial Programming Instruction example



### 28.8.4 SPI Serial Programming Characteristics

**Figure 28-9.** Serial Programming Waveforms



For characteristics of the SPI module see “SPI Timing Characteristics” on page 318.

# 29. Electrical Characteristics

All AC/DC characteristics contained in this datasheet are based on characterization of the Atmel® ATmega48PA/88PA/168PA AVR microcontroller manufactured in an automotive process technology.

## 29.1 Absolute Maximum Ratings[1]

| | |
|---|---|
| Operating Temperature | –55°C to +125°C |
| Storage Temperature | –65°C to +150°C |
| Voltage on any Pin except $\overline{\text{RESET}}$ with respect to Ground | –0.5V to $V_{CC}$+0.5V |
| Voltage on $\overline{\text{RESET}}$ with respect to Ground | –0.5V to +13.0V |
| Maximum Operating Voltage | 6.0V |
| DC Current per I/O Pin | 40.0mA |
| DC Current $V_{CC}$ and GND Pins | 200.0mA |

Note: 1. Stresses beyond those listed under “Absolute Maximum Ratings” may cause permanent damage to the device. This is a stress rating only and functional operation of the device at these or other conditions beyond those indicated in the operational sections of this specification is not implied. Exposure to absolute maximum rating conditions for extended periods may affect device reliability.

## 29.2 DC Characteristics

**Table 29-1.** Common DC characteristics $T_A$ = -40°C to 125°C, $V_{CC}$ = 2.7V to 5.5V (unless otherwise noted)

| Symbol | Parameter | Condition | Min. | Typ. | Max. | Units |
|---|---|---|---|---|---|---|
| $V_{IL}$ | Input Low Voltage, except XTAL1 and $\overline{\text{RESET}}$ pin | $V_{CC}$ = 2.7V - 5.5V | –0.3 | | $0.3V_{CC}$[(1)] | V |
| $V_{IH}$ | Input High Voltage, except XTAL1 and $\overline{\text{RESET}}$ pins | $V_{CC}$ = 2.4V - 5.5V | $0.6V_{CC}$[(2)] | | $V_{CC}$ + 0.5 | V |
| $V_{IL1}$ | Input Low Voltage, XTAL1 pin | $V_{CC}$ = 2.7V - 5.5V | –0.3 | | $0.1V_{CC}$[(1)] | V |
| $V_{IH1}$ | Input High Voltage, XTAL1 pin | $V_{CC}$ = 2.4V - 5.5V | $0.7V_{CC}$[(2)] | | $V_{CC}$ + 0.5 | V |
| $V_{IL2}$ | Input Low Voltage, $\overline{\text{RESET}}$ pin | $V_{CC}$ = 2.7V - 5.5V | –0.3 | | $0.1V_{CC}$[(1)] | V |
| $V_{IH2}$ | Input High Voltage, $\overline{\text{RESET}}$ pin | $V_{CC}$ = 2.7V - 5.5V | $0.9V_{CC}$[(2)] | | $V_{CC}$ + 0.5 | V |
| $V_{OL}$ | Output Low Voltage[(4)] except RESET pin | $I_{OL}$ = 20mA, $V_{CC}$ = 5V<br>$I_{OL}$ = 5mA, $V_{CC}$ = 3V | | | 0.8<br>0.5 | V |
| $V_{OH}$ | Output High Voltage[(3)] except Reset pin | $I_{OH}$ = –20mA, $V_{CC}$ = 5V<br>$I_{OH}$ = –10mA, $V_{CC}$ = 3V | 4.1<br>2.3 | | | V |
| $I_{IL}$ | Input Leakage Current I/O Pin | $V_{CC}$ = 5.5V, pin low (absolute value) | | | 1 | µA |
| $I_{IH}$ | Input Leakage Current I/O Pin | $V_{CC}$ = 5.5V, pin high (absolute value) | | | 1 | µA |
| $R_{RST}$ | Reset Pull-up Resistor | $V_{CC}$ = 5V, $V_{in}$ = 0V | 30 | | 60 | kΩ |
| $R_{PU}$ | I/O Pin Pull-up Resistor | | 20 | | 50 | kΩ |
| $V_{ACIO}$ | Analog Comparator Input Offset Voltage | $V_{CC}$ = 5V, $0.1V_{CC} < V_{in} < V_{CC}$ - 100mV | | <10 | 40 | mV |
| $I_{ACLK}$ | Analog Comparator Input Leakage Current | $0.1V_{CC}$ < Vin < $V_{CC}$ – 100mV | –50 | | 50 | nA |
| $t_{ACID}$ | Analog Comparator Propagation Delay | $V_{CC}$ = 4.5V | | 140 | | ns |

Notes:

1. "Max" means the highest value where the pin is guaranteed to be read as low
2. "Min." means the lowest value where the pin is guaranteed to be read as high
3. Although each I/O port can source more than the test conditions (20mA at $V_{CC}$ = 5V, 10mA at $V_{CC}$ = 3V) under steady state conditions (non-transient), the following must be observed:
   Atmel ATmega48PA/88PA/168PA:
   1] The sum of all $I_{OH}$, for ports C0 - C5, D0- D4, ADC7, $\overline{\text{RESET}}$ should not exceed 150mA.
   2] The sum of all $I_{OH}$, for ports B0 - B5, D5 - D7, ADC6, XTAL1, XTAL2 should not exceed 150mA.
   If I$I_{OH}$ exceeds the test condition, $V_{OH}$ may exceed the related specification. Pins are not guaranteed to source current greater than the listed test condition.
4. Although each I/O port can sink more than the test conditions (20mA at $V_{CC}$ = 5V, 10mA at $V_{CC}$ = 3V) under steady state conditions (non-transient), the following must be observed:
   Atmel ATmega48PA/88PA/168PA:
   1] The sum of all $I_{OL}$, for ports C0 - C5, ADC7, ADC6 should not exceed 100mA.
   2] The sum of all $I_{OL}$, for ports B0 - B5, D5 - D7, XTAL1, XTAL2 should not exceed 100mA.
   3] The sum of all $I_{OL}$, for ports D0 - D4, $\overline{\text{RESET}}$ should not exceed 100mA.
   If $I_{OL}$ exceeds the test condition, $V_{OL}$ may exceed the related specification. Pins are not guaranteed to sink current greater than the listed test condition.

### 29.2.1 DC Characteristics

**Table 29-2.** DC characteristics - $T_A$ = –40°C to +125°C, $V_{CC}$ = 2.7V to 5.5V (unless otherwise noted)

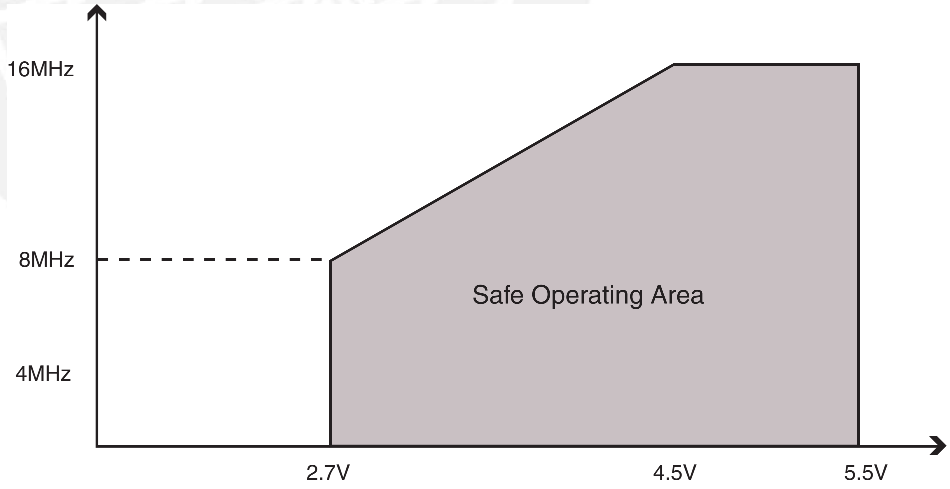
| Symbol | Parameter | Condition | Min. | Typ.[2] | Max. | Units |
|---|---|---|---|---|---|---|
| $I_{CC}$ | Power Supply Current[1] | Active 4MHz, $V_{CC}$ = 3V | | 1.3 | 2.4 | mA |
| | | Active 8MHz, $V_{CC}$ = 5V | | 4.6 | 10 | mA |
| | | Active 16MHz, $V_{CC}$ = 5V | | 8.4 | 16 | mA |
| | | Idle 4MHz, $V_{CC}$ = 3V | | 0.2 | 0.6 | mA |
| | | Idle 8MHz, $V_{CC}$ = 5V | | 0.9 | 1.6 | mA |
| | | Idle 16MHz, $V_{CC}$ = 5V | | 1.8 | 4 | mA |
| | Power-down mode[3] | WDT enabled, $V_{CC}$ = 3V | | 4.2 | 44 | µA |
| | | WDT enabled, $V_{CC}$ = 5V | | 6.2 | 66 | µA |
| | | WDT disabled, $V_{CC}$ = 3V | | 0.8 | 40 | µA |
| | | WDT disabled, $V_{CC}$ = 5V | | 1.1 | 60 | µA |

Notes: 1. Values with “Minimizing Power Consumption” enabled (0xFF).
2. Typical values at 25°C. Maximum values are test limits in production.
3. The current consumption values include input leakage current

## 29.3 Speed Grades

Maximum frequency is dependent on $V_{CC.}$ As shown in Figure 29-1.

**Figure 29-1.** Maximum Frequency versus $V_{CC}$



## 29.4 Clock Characteristics

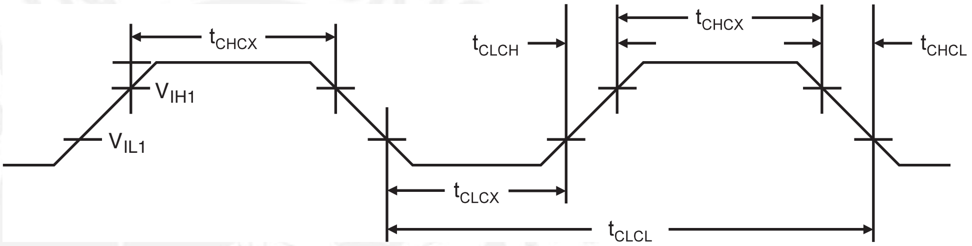
### 29.4.1 Calibrated Internal RC Oscillator Accuracy

**Table 29-3.** Calibration Accuracy of Internal RC Oscillator

| | Frequency | $V_{CC}$ | Temperature | Calibration Accuracy |
|---|---|---|---|---|
| **Default 3V Factory Calibration** | 8.0MHz | 3V | 25°C | ±1% |
| | | 2.7V - 5.5V | –40°C - 125°C | ±14% |
| **5V Factory Calibration** | 8.0MHz | 5V | 25°C | ±1% |
| | | 4.5V - 5.5V | –40°C - 125°C | ±10% |
| **Watchdog Oscillator** | 128kHz | 2.7V - 5.5V | –40°C - 125°C | ±40% |

### 29.4.2 External Clock Drive Waveforms

**Figure 29-2.** External Clock Drive Waveforms



### 29.4.3 External Clock Drive

**Table 29-4.** External Clock Drive

| | | $V_{CC}$ = 2.7 - 5.5V | | $V_{CC}$ = 4.5 - 5.5V | | |
|---|---|---|---|---|---|---|
| **Symbol** | **Parameter** | **Min.** | **Max.** | **Min.** | **Max.** | **Units** |
| $1/t_{CLCL}$ | Oscillator Frequency | 0 | 8 | 0 | 16 | MHz |
| $t_{CLCL}$ | Clock Period | 125 | | 62.5 | | ns |
| $t_{CHCX}$ | High Time | 50 | | 25 | | ns |
| $t_{CLCX}$ | Low Time | 50 | | 25 | | ns |
| $t_{CLCH}$ | Rise Time | | 1.6 | | 0.5 | μs |
| $t_{CHCL}$ | Fall Time | | 1.6 | | 0.5 | μs |
| $\Delta t_{CLCL}$ | Change in period from one clock cycle to the next | | 2 | | 2 | % |

Note: All DC Characteristics contained in this datasheet are based on simulation and characterization of other AVR microcontrollers manufactured in the same process technology. These values are preliminary values representing design targets, and will be updated after characterization of actual silicon.

## 29.5 System and Reset Characteristics

**Table 29-5.** Power on Reset specifications[1]

| Symbol | Parameter | | Min. | Typ | Max | Units |
|---|---|---|---|---|---|---|
| $V_{POT}$ | Power-on Reset Threshold Voltage (rising) | | | 1.4 | | V |
| | Power-on Reset Threshold Voltage (falling)[2] | | 0.6 | 1.3 | 1.6 | V |
| Vpormax | VCC Max. start voltage to ensure internal Power-on Reset signal | | | | 0.4 | V |
| Vpormin | VCC Min. start voltage to ensure internal Power-on Reset signal | | –0.1 | | | V |
| VCCRR | VCC Rise Rate to ensure Power-on Reset | | 0.01 | | | V/ms |
| VRST | RESET Pin Threshold Voltage | | 0.2Vcc | | 0.9Vcc | V |
| $t_{RST}$ | Minimum pulse width on $\overline{\text{RESET}}$ Pin | | 2.5 | | | µs |
| $V_{BG}$ | Bandgap reference voltage | | 1.0 | 1.1 | 1.2 | V |
| $t_{BG}$ | Bandgap reference start-up time | | | 40 | 70 | µs |
| $V_{HYST}$ | Brown-out Detector Hysteresis | | | 80 | | mV |

Notes: 1. Values are guidelines only.

2. Before rising, the supply has to be between VPORMIN and VPORMAX to ensure a Reset.

**Table 29-6.** BODLEVEL Fuse Coding[1]

| BODLEVEL 2:0 Fuses | Min. $V_{BOT}$ | Typ $V_{BOT}$ | Max $V_{BOT}$ | Units |
|---|---|---|---|---|
| 111 | BOD Disabled | | | |
| 110 | 1.6 | 1.8 | 2.0 | V |
| 101 | 2.5 | 2.7 | 2.9 | |
| 100 | 3.9 | 4.3 | 4.6 | |
| 011 | | 2.3[2] | | |
| 010 | | 2.2[2] | | |
| 000 | | 2.0[2] | | |
| 001 | | 1.9[2] | | |

Notes: 1. $V_{BOT}$ may be below nominal minimum operating voltage for some devices. For devices where this is the case, the device is tested down to $V_{CC} = V_{BOT}$ during the production test. This guarantees that a Brown-Out Reset will occur before $V_{CC}$ drops to a voltage where correct operation of the microcontroller is no longer guaranteed. The test is performed using BODLEVEL = 110, 101 and 100.

2. Not Tested in production

## 29.6 SPI Timing Characteristics

See Figure 29-3 and Figure 29-4 for details.

**Table 29-7.** SPI Timing Parameters

| | Description | Mode | Min. | Typ | Max | |
|---|---|---|---|---|---|---|
| 1 | SCK period | Master | | See Table 19-5 | | ns |
| 2 | SCK high/low | Master | | 50% duty cycle | | |
| 3 | Rise/Fall time | Master | | 3.6 | | |
| 4 | Setup | Master | | 10 | | |
| 5 | Hold | Master | | 10 | | |
| 6 | Out to SCK | Master | | $0.5 \times t_{sck}$ | | |
| 7 | SCK to out | Master | | 10 | | |
| 8 | SCK to out high | Master | | 10 | | |
| 9 | $\overline{\text{SS}}$ low to out | Slave | | 15 | | |
| 10 | SCK period | Slave | $4 \times t_{ck}$ | | | |
| 11 | SCK high/low[1] | Slave | $2 \times t_{ck}$ | | | |
| 12 | Rise/Fall time | Slave | | | 1600 | |
| 13 | Setup | Slave | 10 | | | |
| 14 | Hold | Slave | $t_{ck}$ | | | |
| 15 | SCK to out | Slave | | 15 | | |
| 16 | SCK to $\overline{\text{SS}}$ high | Slave | 20 | | | |
| 17 | $\overline{\text{SS}}$ high to tri-state | Slave | | 10 | | |
| 18 | $\overline{\text{SS}}$ low to SCK | Slave | 20 | | | |

Notes:

1. In SPI Programming mode the minimum SCK high/low period is:
   - 2 $t_{CLCL}$ for $f_{CK} < 12\text{MHz}$
   - 3 $t_{CLCL}$ for $f_{CK} > 12\text{MHz}$
2. All DC Characteristics contained in this datasheet are based on simulation and characterization of other AVR microcontrollers manufactured in the same process technology. These values are preliminary values representing design targets, and will be updated after characterization of actual silicon.

**Figure 29-3.** SPI Interface Timing Requirements (Master Mode)

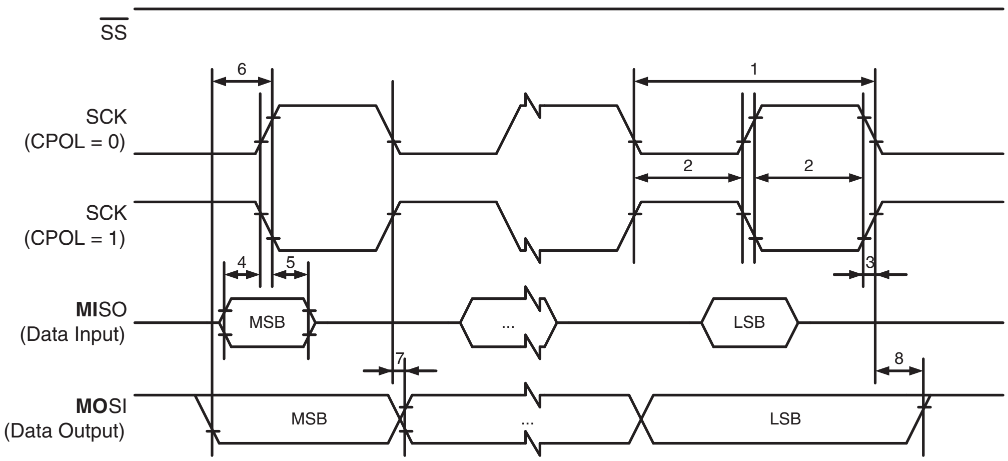
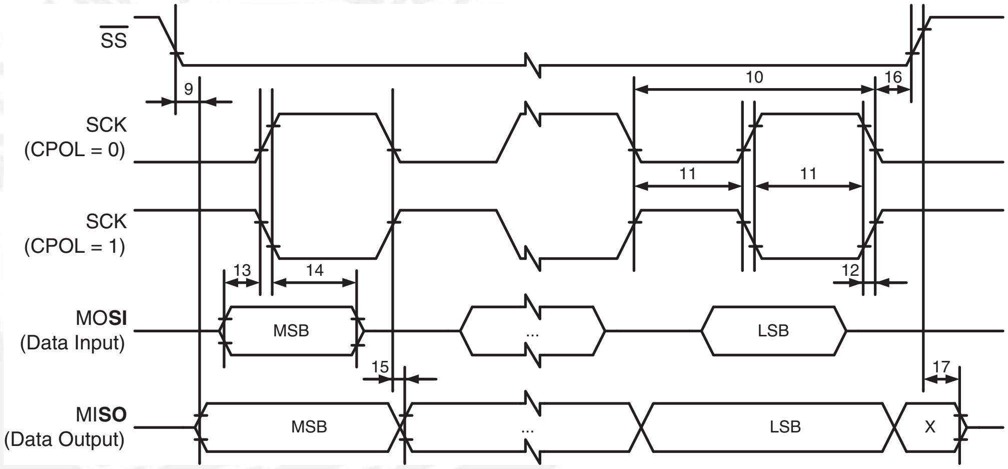


**Figure 29-4.** SPI Interface Timing Requirements (Slave Mode)



## 29.7 Two-wire Serial Interface Characteristics

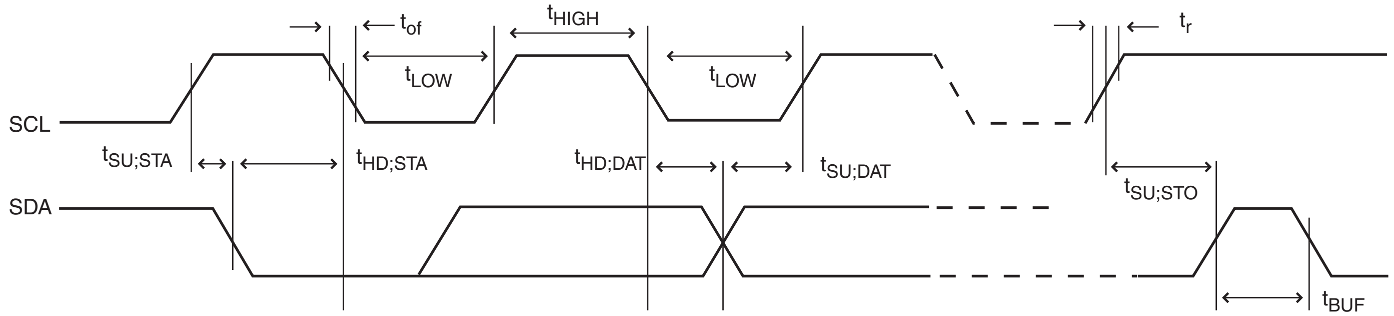
Table 29-8 describes the requirements for devices connected to the 2-wire Serial Bus. The Atmel® ATmega48PA/88PA/168PA 2-wire Serial Interface meets or exceeds these requirements under the noted conditions. Timing symbols refer to Figure 29-5.

**Table 29-8.** Two-wire Serial Bus Requirements

| Symbol | Parameter | Condition | Min. | Max | Units |
|---|---|---|---|---|---|
| $V_{IL}$ | Input Low-voltage | | -0.3 | $0.3\ V_{CC}$ | V |
| $V_{IH}$ | Input High-voltage | | $0.7\ V_{CC}$ | $V_{CC} + 0.5$ | V |
| $V_{hys}$(1) | Hysteresis of Schmitt Trigger Inputs | | $0.05\ V_{CC}$(2) | – | V |
| $V_{OL}$(1) | Output Low-voltage | 3 mA sink current | 0 | 0.4 | V |
| $t_r$(1) | Rise Time for both SDA and SCL | | $20 + 0.1C_b$(3)(2) | 300 | ns |
| $t_{of}$(1) | Output Fall Time from $V_{IHmin}$ to $V_{ILmax}$ | $10\ pF < C_b < 400\ pF$(3) | $20 + 0.1C_b$(3)(2) | 250 | ns |
| $t_{SP}$(1) | Spikes Suppressed by Input Filter | | 0 | 50(2) | ns |
| $I_i$ | Input Current each I/O Pin | $0.1V_{CC} < V_i < 0.9V_{CC}$ | -10 | 10 | µA |
| $C_i$(1) | Capacitance for each I/O Pin | | – | 10 | pF |
| $f_{SCL}$ | SCL Clock Frequency | $f_{CK}$(4) $> \max(16f_{SCL}, 250kHz)$(5) | 0 | 400 | kHz |
| Rp | Value of Pull-up resistor | $f_{SCL} \le 100kHz$ | $\frac{V_{CC} - 0{,}4V}{3mA}$ | $\frac{1000ns}{C_b}$ | Ω |
| | | $f_{SCL} > 100kHz$ | $\frac{V_{CC} - 0{,}4V}{3mA}$ | $\frac{300ns}{C_b}$ | Ω |
| $t_{HD;STA}$ | Hold Time (repeated) START Condition | $f_{SCL} \le 100kHz$ | 4.0 | – | µs |
| | | $f_{SCL} > 100kHz$ | 0.6 | – | µs |
| $t_{LOW}$ | Low Period of the SCL Clock | $f_{SCL} \le 100kHz$ | 4.7 | – | µs |
| | | $f_{SCL} > 100kHz$ | 1.3 | – | µs |
| $t_{HIGH}$ | High period of the SCL clock | $f_{SCL} \le 100kHz$ | 4.0 | – | µs |
| | | $f_{SCL} > 100kHz$ | 0.6 | – | µs |
| $t_{SU;STA}$ | Set-up time for a repeated START condition | $f_{SCL} \le 100kHz$ | 4.7 | – | µs |
| | | $f_{SCL} > 100kHz$ | 0.6 | – | µs |
| $t_{HD;DAT}$ | Data hold time | $f_{SCL} \le 100kHz$ | 0 | 3.45 | µs |
| | | $f_{SCL} > 100kHz$ | 0 | 0.9 | µs |
| $t_{SU;DAT}$ | Data setup time | $f_{SCL} \le 100kHz$ | 250 | – | ns |
| | | $f_{SCL} > 100kHz$ | 100 | – | ns |
| $t_{SU;STO}$ | Setup time for STOP condition | $f_{SCL} \le 100kHz$ | 4.0 | – | µs |
| | | $f_{SCL} > 100kHz$ | 0.6 | – | µs |
| $t_{BUF}$ | Bus free time between a STOP and START condition | $f_{SCL} \le 100kHz$ | 4.7 | – | µs |
| | | $f_{SCL} > 100kHz$ | 1.3 | – | µs |

Notes:
1. In the Atmel ATmega48PA/88PA/168PA, this parameter is characterized and not 100% tested.
2. Required only for $f_{SCL} > 100kHz$.
3. $C_b$ = capacitance of one bus line in pF.
4. $f_{CK}$ = CPU clock frequency
5. This requirement applies to all Atmel ATmega48PA/88PA/168PA 2-wire Serial Interface operation. Other devices connected to the 2-wire Serial Bus need only obey the general $f_{SCL}$ requirement.

**Figure 29-5.** Two-wire Serial Bus Timing



## 29.8 ADC Characteristics

**Table 29-9.** ADC Characteristics

| Symbol | Parameter | Condition | Min. | Typ | Max | Units |
|---|---|---|---|---|---|---|
| | Resolution | –40°C to 125°C,<br>2.70V to 5.50V<br>ADC clock = 200kHz | | 10 | | Bits |
| TUE | Absolute accuracy | $V_{CC}$ = 4V, $V_{REF}$ = 4V | | 2.2 | 3.5 | LSB |
| INL | Integral Non-Linearity | $V_{CC}$ = 4V, $V_{REF}$ = 4V | | 0.6 | 1.5 | LSB |
| DNL | Differential Non-Linearity | $V_{CC}$ = 4V, $V_{REF}$ = 4V | | 0.3 | 0.7 | LSB |
| | Gain Error | $V_{CC}$ = 4V, $V_{REF}$ = 4V | –4.0 | | 3.0 | LSB |
| | Offset Error | $V_{CC}$ = 4V, $V_{REF}$ = 4V | –3.5 | | 3.5 | LSB |
| | Clock Frequency | | 50 | | 200 | kHz |
| $AV_{CC}$(1) | Analog Supply Voltage | | $V_{CC}$ - 0.3 | | $V_{CC}$ + 0.3 | V |
| $V_{REF}$ | Reference Voltage | | 1.0 | | $AV_{CC}$ | V |
| $V_{IN}$ | Input Voltage | | GND | | $V_{REF}$ | V |
| | Input Bandwidth | | | 38.5 | | kHz |
| $V_{INT}$ | Internal Voltage Reference | | 1.0 | 1.1 | 1.2 | V |
| $R_{REF}$ | Reference Input Resistance | | 22 | 32 | 42 | kΩ |
| $R_{AIN}$ | Analog Input Resistance | | | 100 | | MΩ |

Note: 1. $AV_{CC}$ absolute min./max: 2.7V/5.5V

## 29.9 Parallel Programming Characteristics

**Table 29-10.** Parallel Programming Characteristics, $V_{CC}$ = 5V ±10%

| Symbol | Parameter | Min. | Typ | Max | Units |
|---|---|---|---|---|---|
| $V_{PP}$ | Programming Enable Voltage | 11.5 | | 12.5 | V |
| $I_{PP}$ | Programming Enable Current | | | 250 | μA |
| $t_{DVXH}$ | Data and Control Valid before XTAL1 High | 67 | | | ns |
| $t_{XLXH}$ | XTAL1 Low to XTAL1 High | 200 | | | ns |
| $t_{XHXL}$ | XTAL1 Pulse Width High | 150 | | | ns |
| $t_{XLDX}$ | Data and Control Hold after XTAL1 Low | 67 | | | ns |
| $t_{XLWL}$ | XTAL1 Low to $\overline{\text{WR}}$ Low | 0 | | | ns |
| $t_{XLPH}$ | XTAL1 Low to PAGEL high | 0 | | | ns |
| $t_{PLXH}$ | PAGEL low to XTAL1 high | 150 | | | ns |
| $t_{BVPH}$ | BS1 Valid before PAGEL High | 67 | | | ns |
| $t_{PHPL}$ | PAGEL Pulse Width High | 150 | | | ns |
| $t_{PLBX}$ | BS1 Hold after PAGEL Low | 67 | | | ns |
| $t_{WLBX}$ | BS2/1 Hold after $\overline{\text{WR}}$ Low | 67 | | | ns |
| $t_{PLWL}$ | PAGEL Low to $\overline{\text{WR}}$ Low | 67 | | | ns |
| $t_{BVWL}$ | BS1 Valid to $\overline{\text{WR}}$ Low | 67 | | | ns |
| $t_{WLWH}$ | $\overline{\text{WR}}$ Pulse Width Low | 150 | | | ns |
| $t_{WLRL}$ | $\overline{\text{WR}}$ Low to RDY/$\overline{\text{BSY}}$ Low | 0 | | 1 | μs |
| $t_{WLRH}$ | $\overline{\text{WR}}$ Low to RDY/$\overline{\text{BSY}}$ High[1] | 3.7 | | 4.5 | ms |
| $t_{WLRH\_CE}$ | $\overline{\text{WR}}$ Low to RDY/$\overline{\text{BSY}}$ High for Chip Erase[2] | 7.5 | | 9 | ms |
| $t_{XLOL}$ | XTAL1 Low to $\overline{\text{OE}}$ Low | 0 | | | ns |
| $t_{BVDV}$ | BS1 Valid to DATA valid | 0 | | 250 | ns |
| $t_{OLDV}$ | $\overline{\text{OE}}$ Low to DATA Valid | | | 250 | ns |
| $t_{OHDZ}$ | $\overline{\text{OE}}$ High to DATA Tri-stated | | | 250 | ns |

Notes: 1. $t_{WLRH}$ is valid for the Write Flash, Write EEPROM, Write Fuse bits and Write Lock bits commands.

2. $t_{WLRH\_CE}$ is valid for the Chip Erase command.

**Figure 29-6.** Parallel Programming Timing, Including some General Timing Requirements

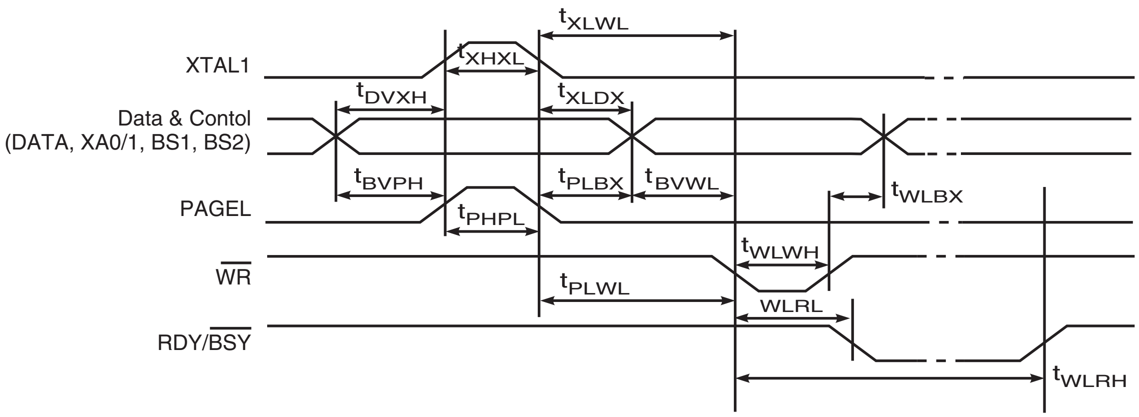
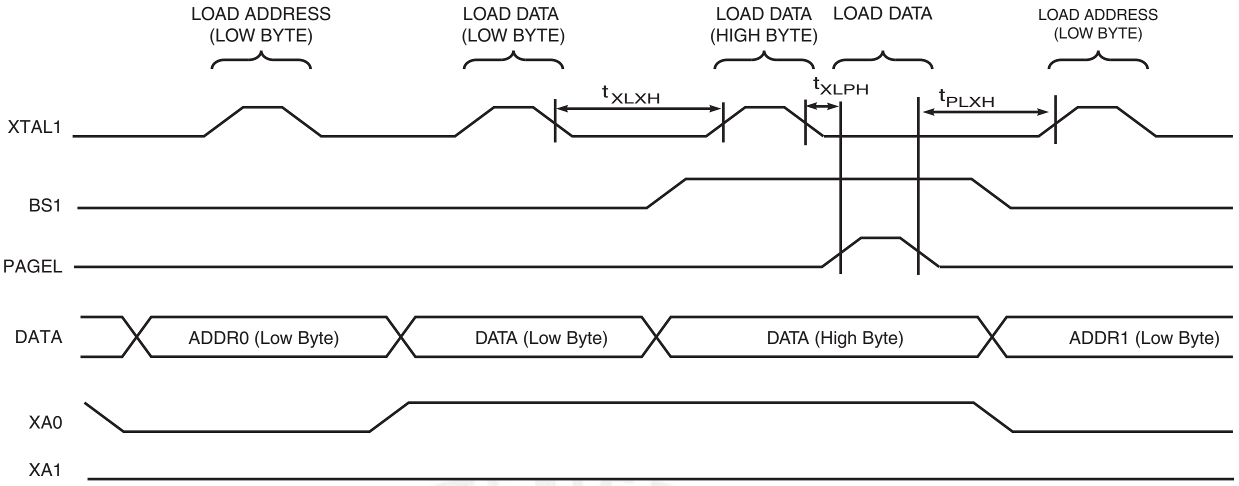
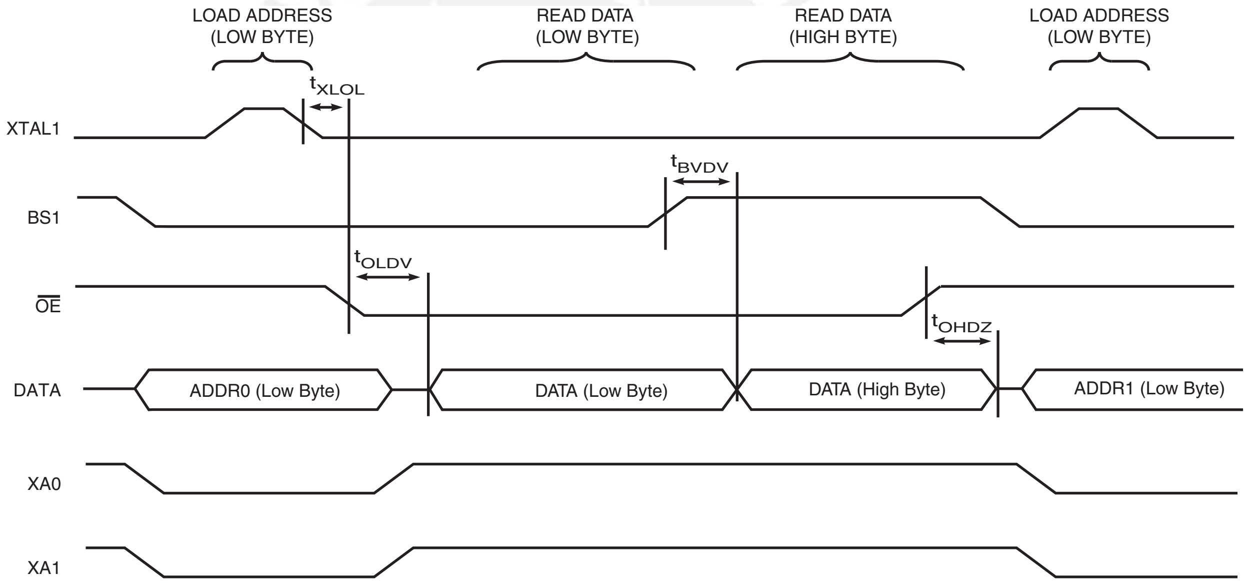


**Figure 29-7.** Parallel Programming Timing, Loading Sequence with Timing Requirements[1]



Note: 1. The timing requirements shown in Figure 29-6 (i.e., $t_{DVXH}$, $t_{XHXL}$, and $t_{XLDX}$) also apply to loading operation.

**Figure 29-8.** Parallel Programming Timing, Reading Sequence (within the Same Page) with Timing Requirements[1]



Note: 1. The timing requirements shown in Figure 29-6 (i.e., $t_{DVXH}$, $t_{XHXL}$, and $t_{XLDX}$) also apply to reading operation.

# 30. Typical Characteristics

The following charts show typical behavior. These figures are not tested during manufacturing. All current consumption measurements are performed with all I/O pins configured as inputs and with internal pull-ups enabled. A square wave generator with rail-to-rail output is used as clock source.

## 30.1 ATmega48PA Typical Characteristics

### 30.1.1 Active Supply Current

**Figure 30-1.** Active Supply Current versus Low Frequency (0.1-1.0MHz)

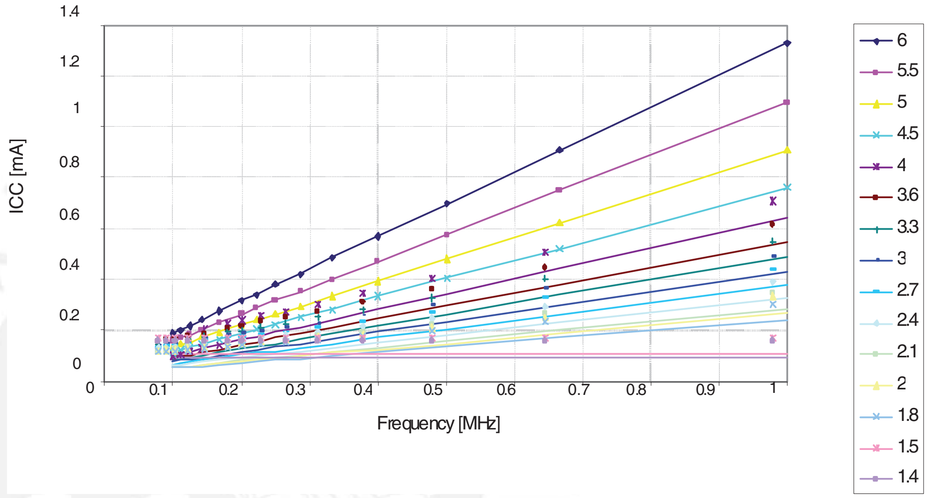
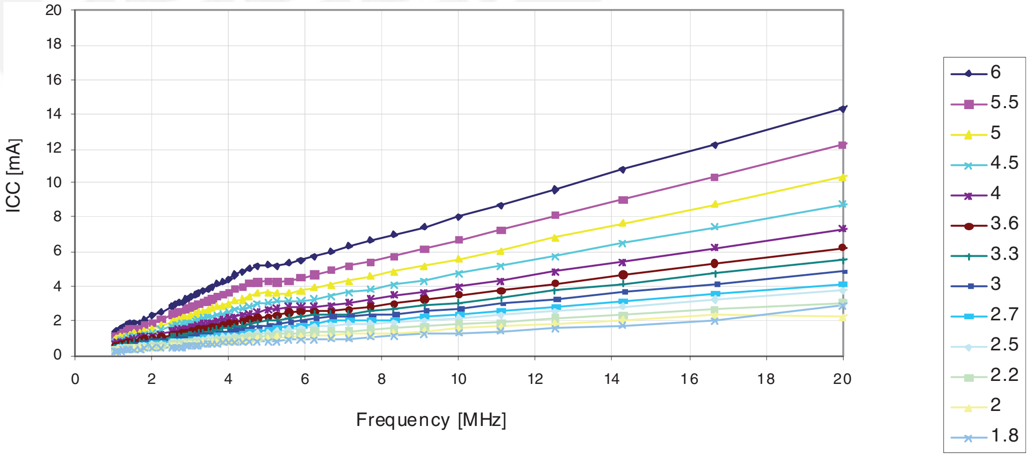


**Figure 30-2.** Active Supply Current versus Frequency (1-16MHz)



### 30.1.2 Idle Supply Current

**Figure 30-3.** Idle Supply Current versus Low Frequency (0.1-1.0MHz)

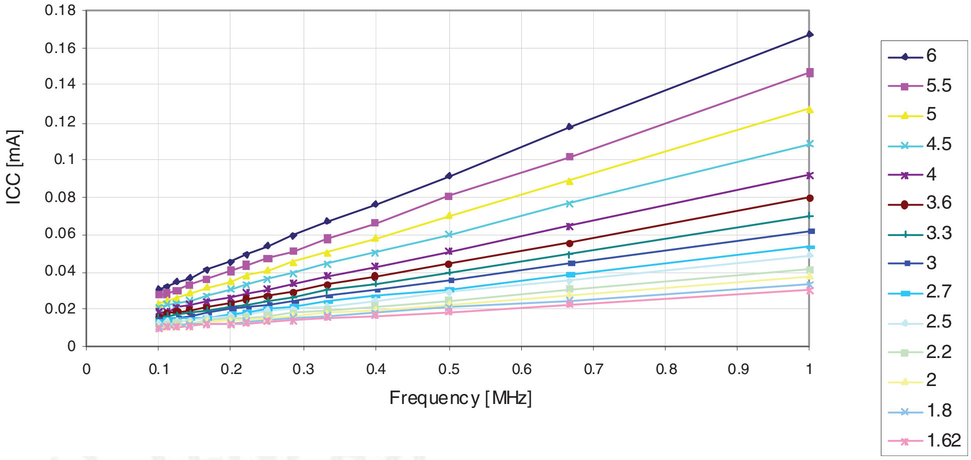
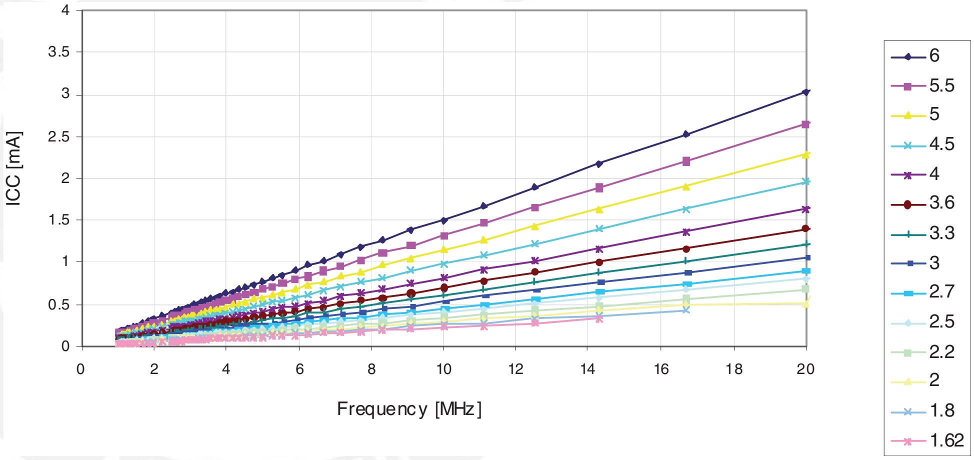


**Figure 30-4.** Idle Supply Current versus Frequency (1-16MHz)



### 30.1.3 Supply Current of IO Modules

The tables and formulas below can be used to calculate the additional current consumption for the different I/O modules in Active and Idle mode. The enabling or disabling of the I/O modules are controlled by the Power Reduction Register. See "Power Reduction Register" on page 42 for details.

**Table 30-1.** Additional Current Consumption for the Different I/O Modules (Absolute Values)

| PRR bit | Typical Numbers | | |
|---|---|---|---|
| | $V_{CC}$ = 2V, F = 1MHz | $V_{CC}$ = 3V, F = 4MHz | $V_{CC}$ = 5V, F = 8MHz |
| PRUSART0 | 2.9µA | 20.7µA | 97.4µA |
| PRTWI | 6.0µA | 44.8µA | 219.7µA |
| PRTIM2 | 5.0µA | 34.5µA | 141.3µA |
| PRTIM1 | 3.6µA | 24.4µA | 107.7µA |
| PRTIM0 | 1.4µA | 9.5µA | 38.4µA |
| PRSPI | 5.0µA | 38.0µA | 190.4µA |
| PRADC | 6.1µA | 47.4µA | 244.7µA |

**Table 30-2.** Additional Current Consumption (Percentage) in Active and Idle Mode

| PRR bit | Additional Current consumption compared to Active with external clock (see Figure 30-1 on page 324 and Figure 30-2 on page 324) | Additional Current consumption compared to Idle with external clock (see Figure 30-3 on page 325 and Figure 30-4 on page 325) |
|---|---|---|
| PRUSART0 | 1.8% | 11.4% |
| PRTWI | 3.9% | 20.6% |
| PRTIM2 | 2.9% | 15.7% |
| PRTIM1 | 2.1% | 11.2% |
| PRTIM0 | 0.8% | 4.2% |
| PRSPI | 3.3% | 17.6% |
| PRADC | 4.2% | 22.1% |

It is possible to calculate the typical current consumption based on the numbers from Table 30-2 for other $V_{CC}$ and frequency settings than listed in Table 30-1.

#### *30.1.3.1 Example*

Calculate the expected current consumption in idle mode with TIMER1, ADC, and SPI enabled at $V_{CC}$ = 2.0V and F = 1MHz. From Table 30-2, third column, we see that we need to add 11.2% for the TIMER1, 22.1% for the ADC, and 17.6% for the SPI module. Reading from Figure 30-3 on page 325, we find that the idle current consumption is ~0.028mA at $V_{CC}$ = 2.0V and F = 1MHz. The total current consumption in idle mode with TIMER1, ADC, and SPI enabled, gives:

$$ICC_{total} \approx 0.028\ \text{mA} \cdot (1 + 0.112 + 0.221 + 0.176) \approx 0.042\ \text{mA}$$

### 30.1.4 Power-down Supply Current

**Figure 30-5.** Power-Down Supply Current versus $V_{CC}$ (Watchdog Timer Disabled)

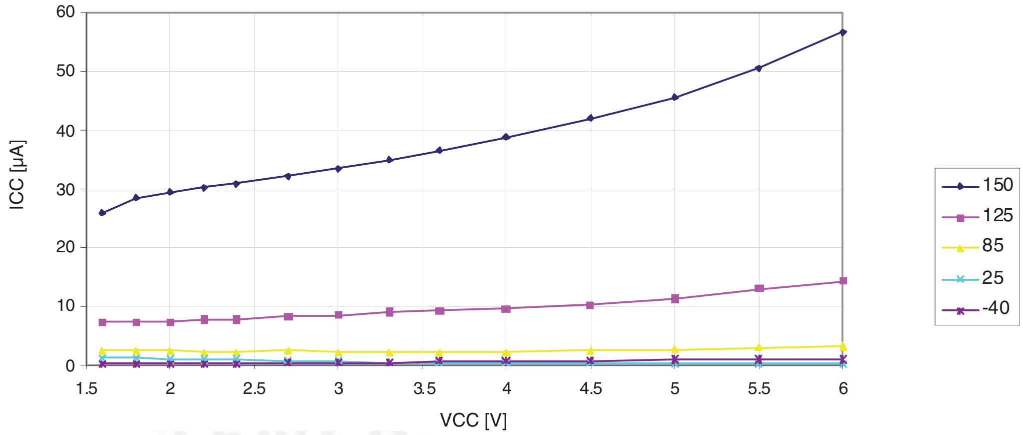
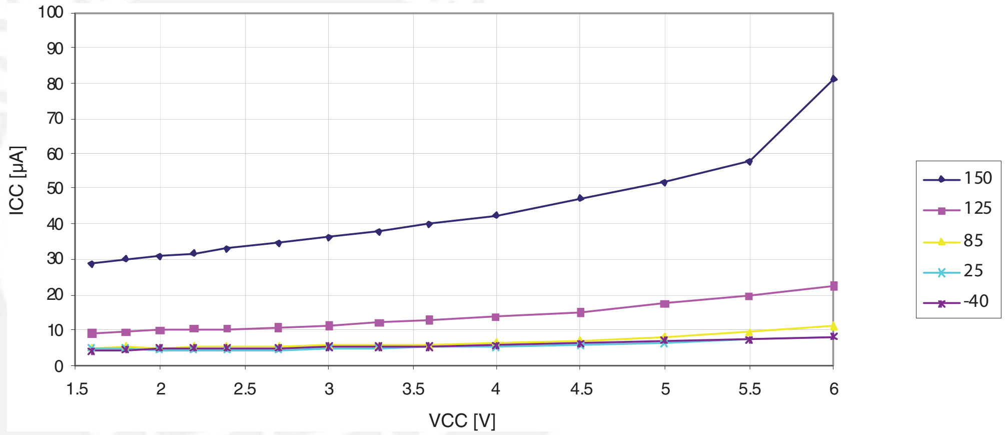


**Figure 30-6.** Power-Down Supply Current versus $V_{CC}$ (Watchdog Timer Enabled)



### 30.1.5 Pin Pull-Up

**Figure 30-7.** I/O Pin Pull-up Resistor Current versus Input Voltage ($V_{CC}$ = 5.0V)

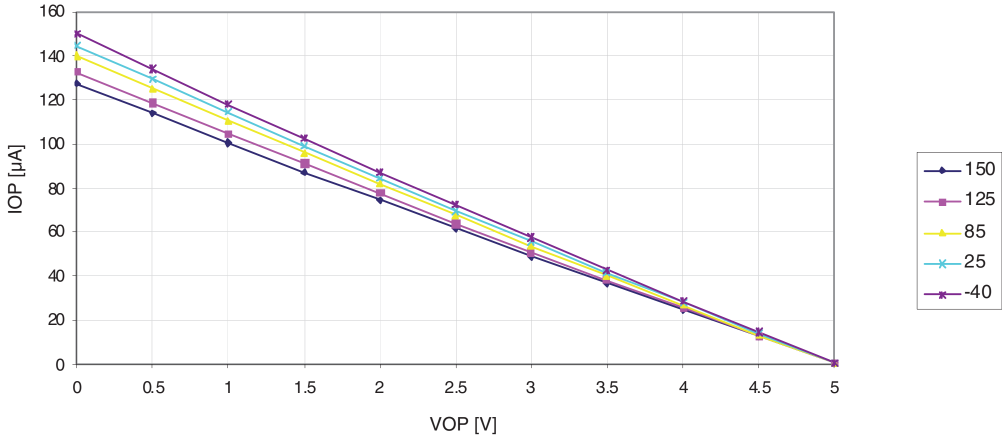
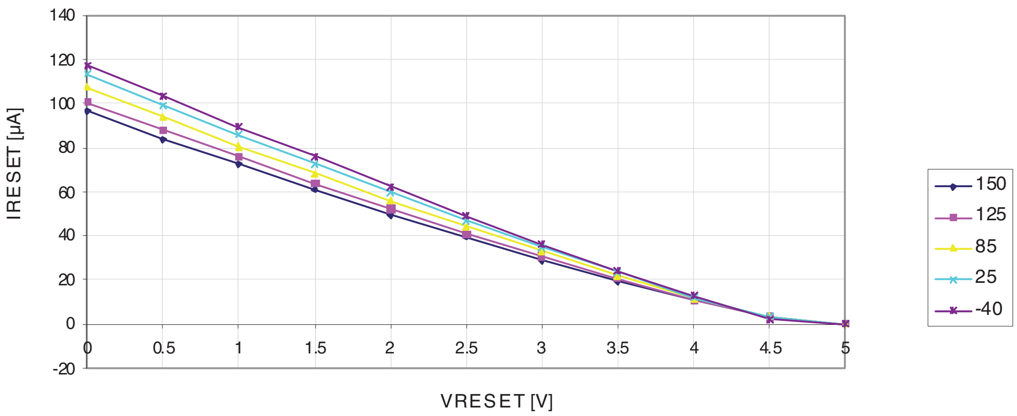


**Figure 30-8.** Reset Pull-up Resistor Current versus Reset Pin Voltage ($V_{CC}$ = 5V)



## 30.1.6 Pin Driver Strength

**Figure 30-9.** I/O Pin Output Voltage versus Sink Current ($V_{CC}$ = 3V)

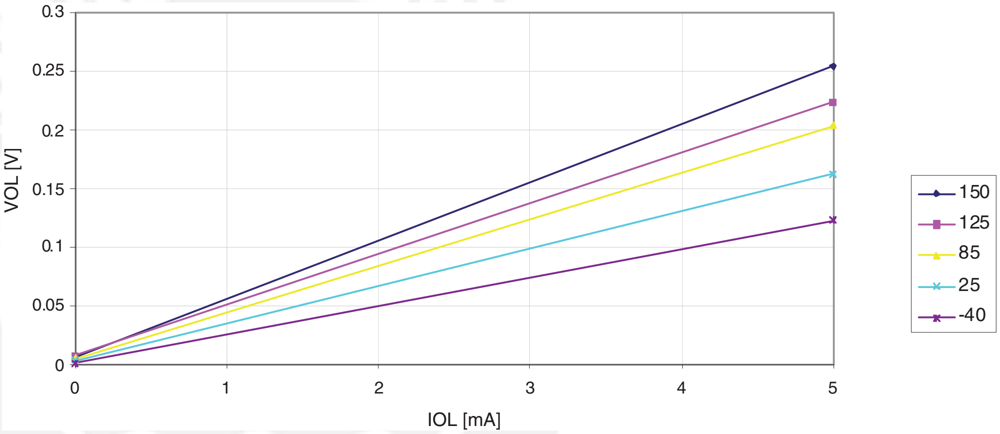


**Figure 30-10.** I/O Pin Output Voltage versus Sink Current ($V_{CC}$ = 5V)

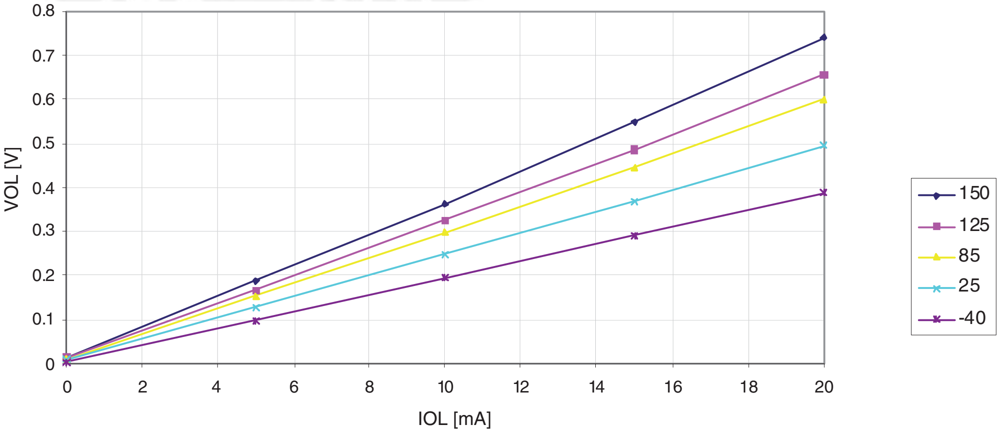


**Figure 30-11.** I/O Pin Output Voltage versus Source Current (Vcc = 3V)

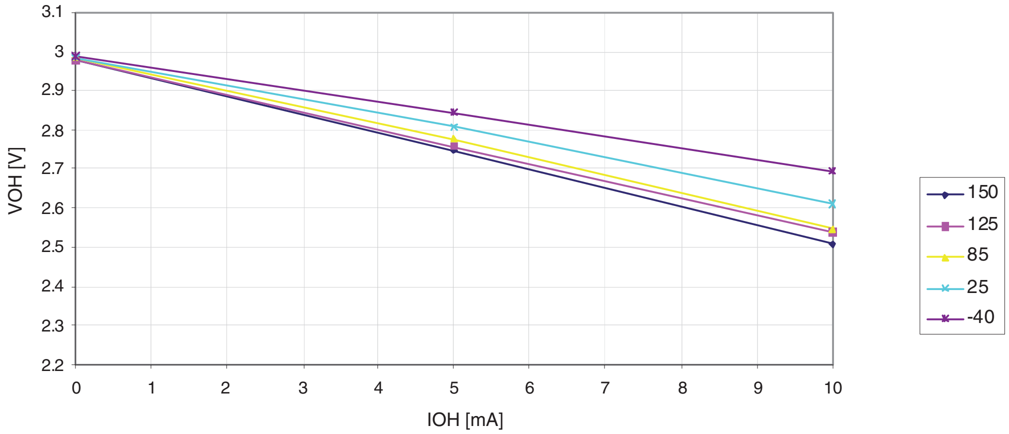
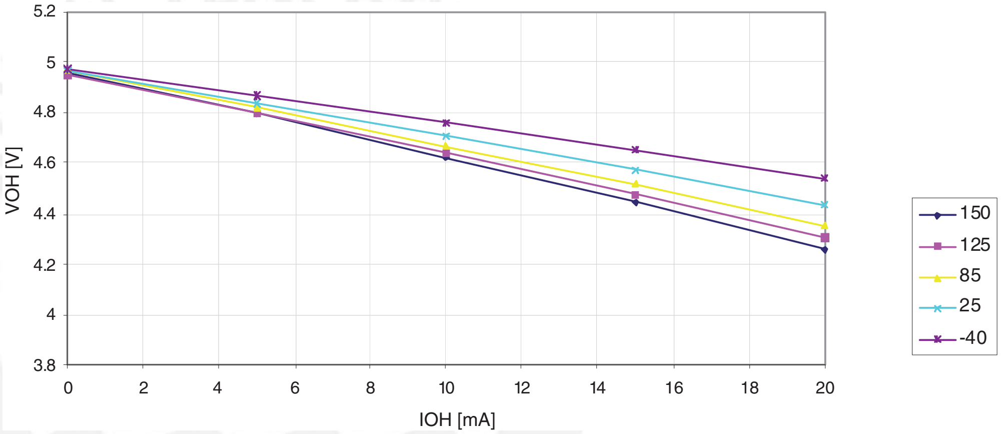


**Figure 30-12.** I/O Pin Output Voltage versus Source Current ($V_{CC}$ = 5V)



### 30.1.7 Pin Threshold and Hysteresis

**Figure 30-13.** I/O Pin Input Threshold Voltage versus $V_{CC}$ ($V_{IH}$, I/O Pin read as ‘1’)

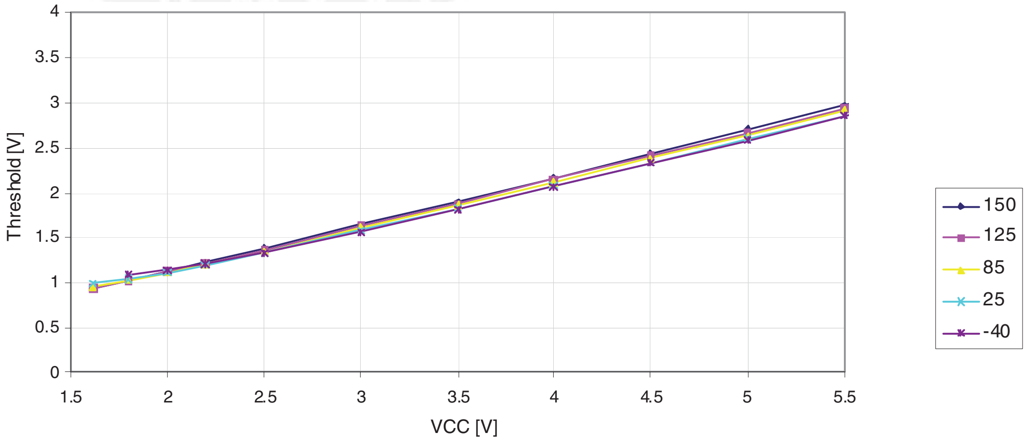


**Figure 30-14.** I/O Pin Input Threshold Voltage versus $V_{CC}$ ($V_{IL}$, I/O Pin read as '0')

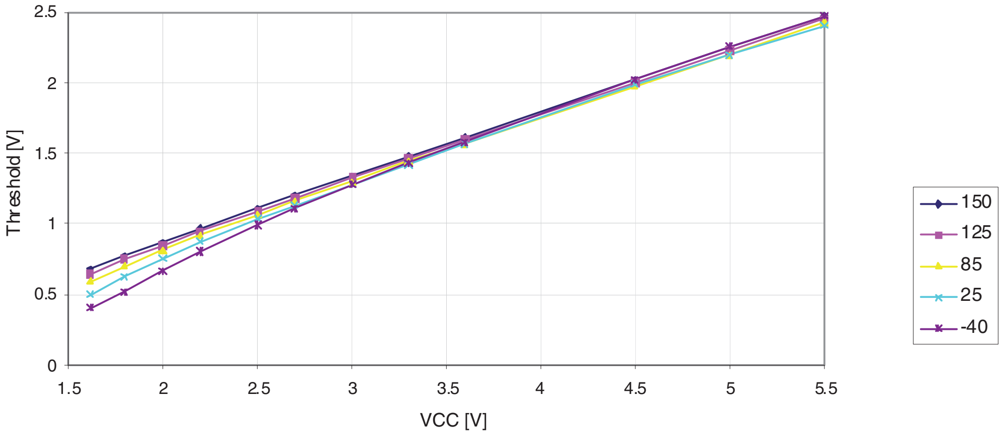


**Figure 30-15.** Reset Input Threshold Voltage versus $V_{CC}$ ($V_{IH}$, I/O Pin read as '1')

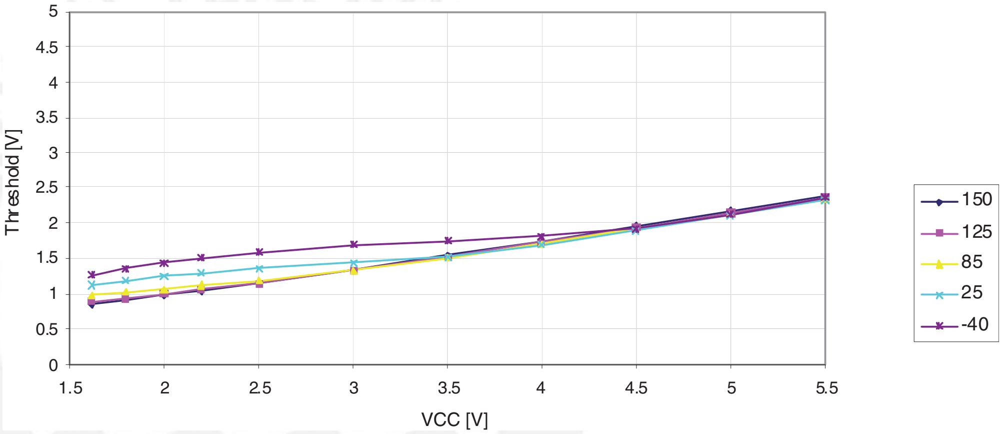
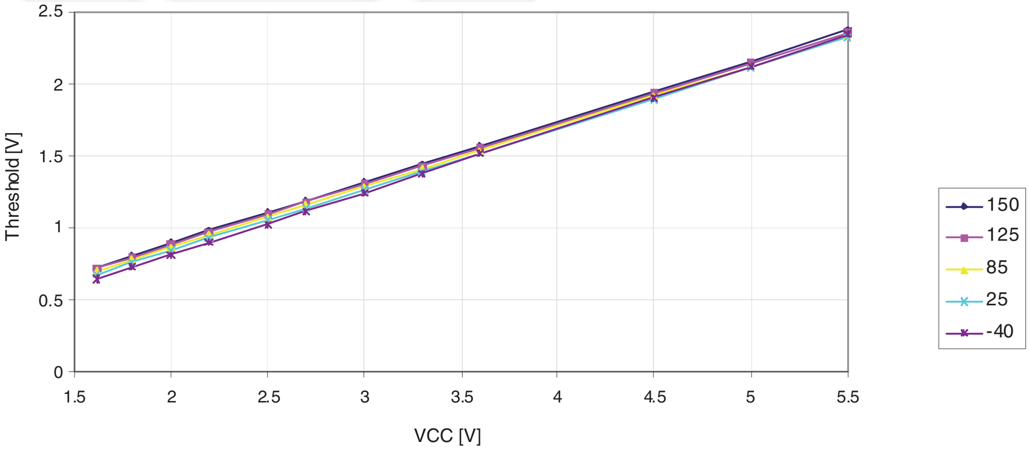


**Figure 30-16.** Reset Input Threshold Voltage versus $V_{CC}$ ($V_{IL}$, I/O Pin read as '0')



### 30.1.8 BOD Threshold

**Figure 30-17.** BOD Thresholds versus Temperature (BODLEVEL is 1.8V)

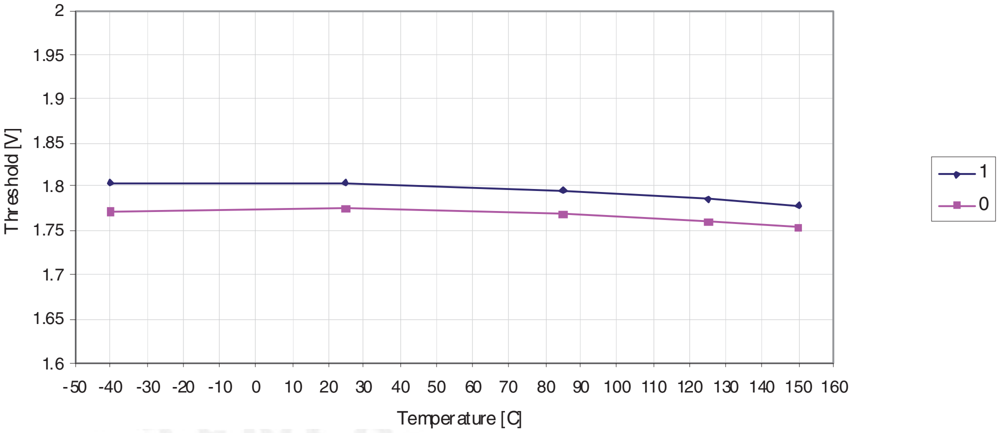


**Figure 30-18.** BOD Thresholds versus Temperature (BODLEVEL is 2.7V)

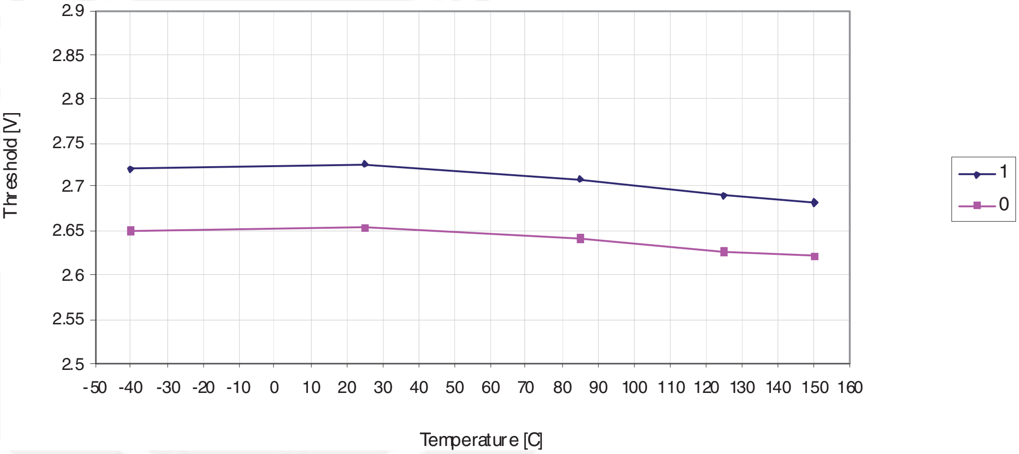


**Figure 30-19.** BOD Thresholds versus Temperature (BODLEVEL is 4.3V)

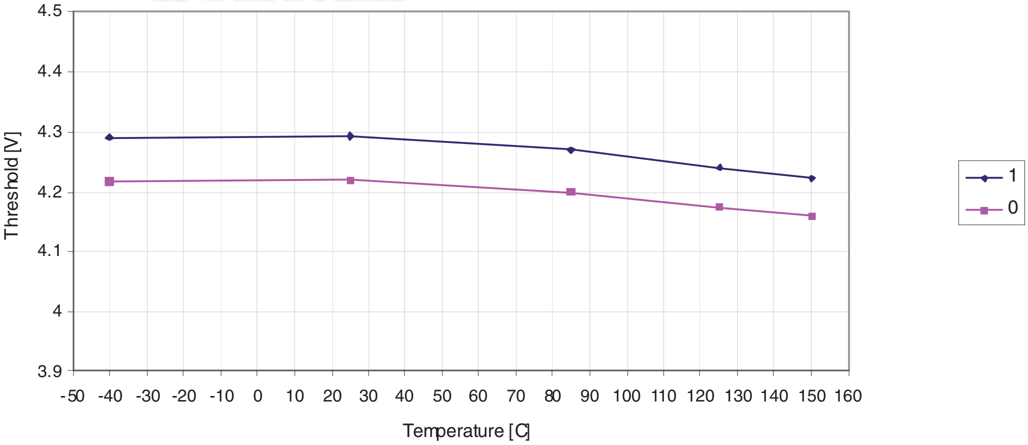
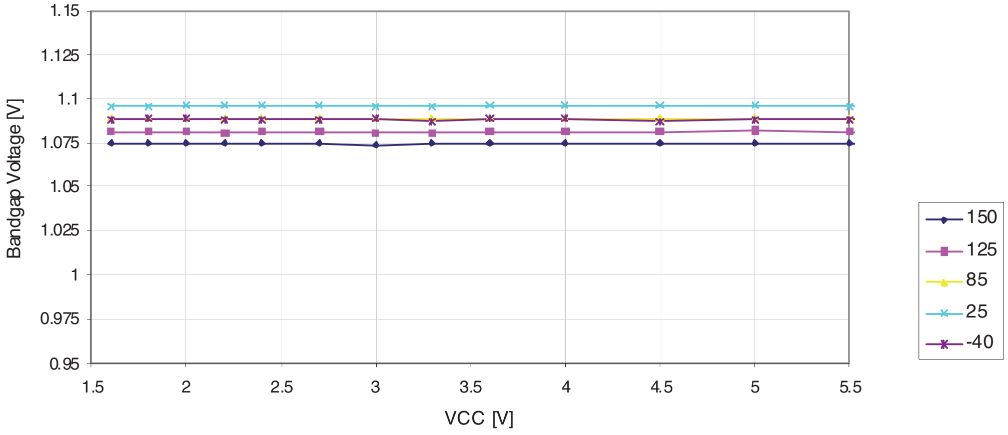


**Figure 30-20.** Bandgap Voltage versus $V_{CC}$



### 30.1.9 Internal Oscillator Speed

**Figure 30-21.** Watchdog Oscillator Frequency versus Temperature

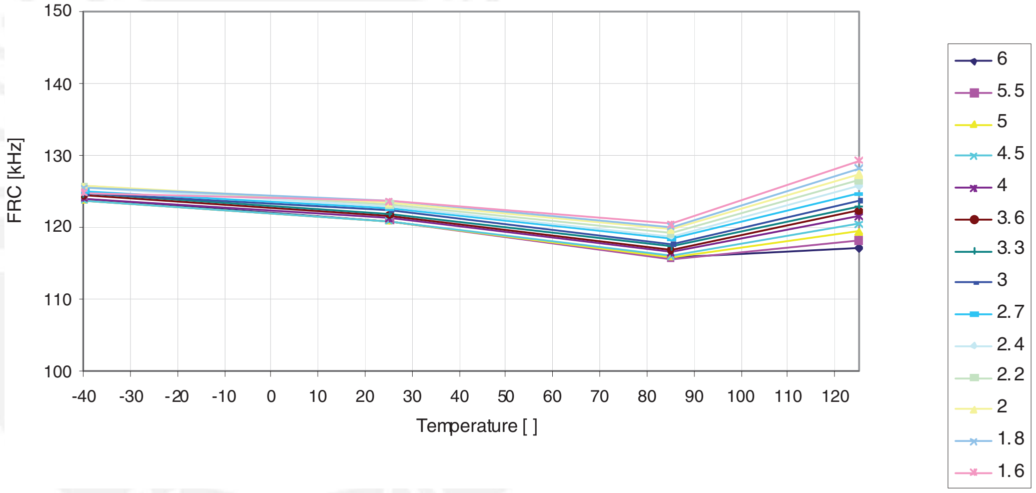


**Figure 30-22.** Watchdog Oscillator Frequency versus $V_{CC}$

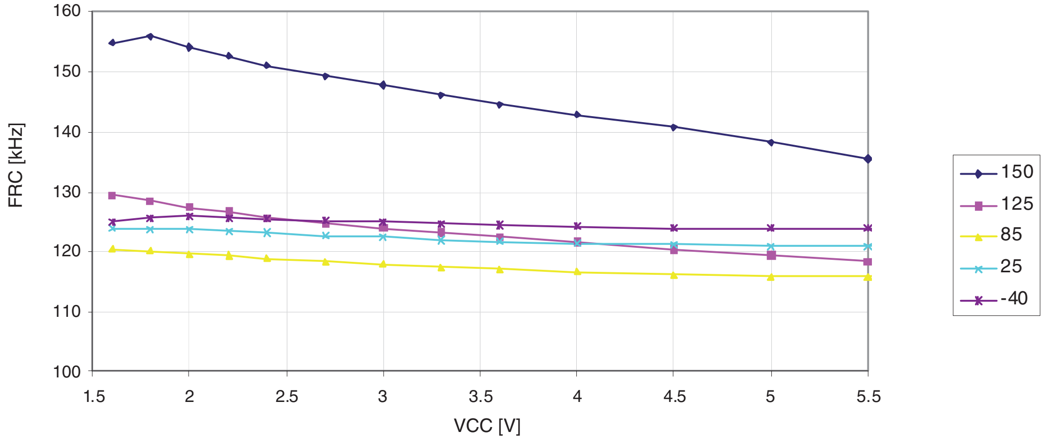


**Figure 30-23.** Calibrated 8MHz RC Oscillator Frequency versus $V_{CC}$

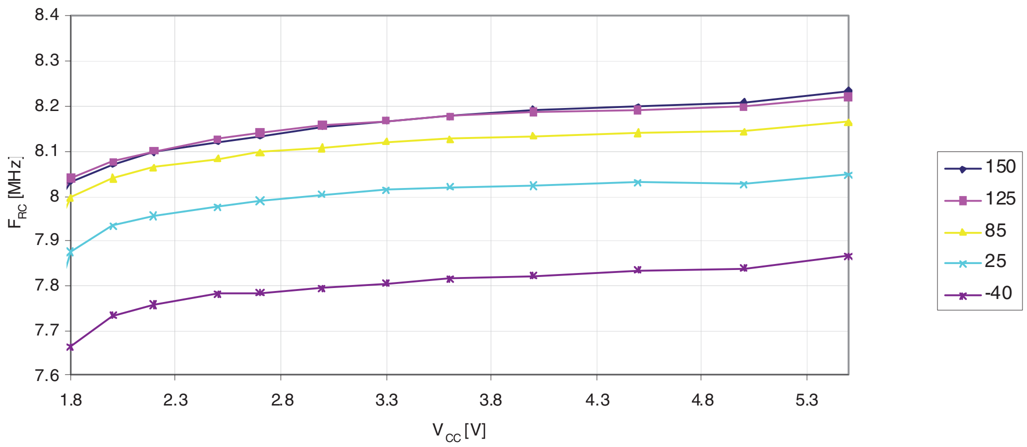


**Figure 30-24.** Calibrated 8MHz RC Oscillator Frequency versus Temperature

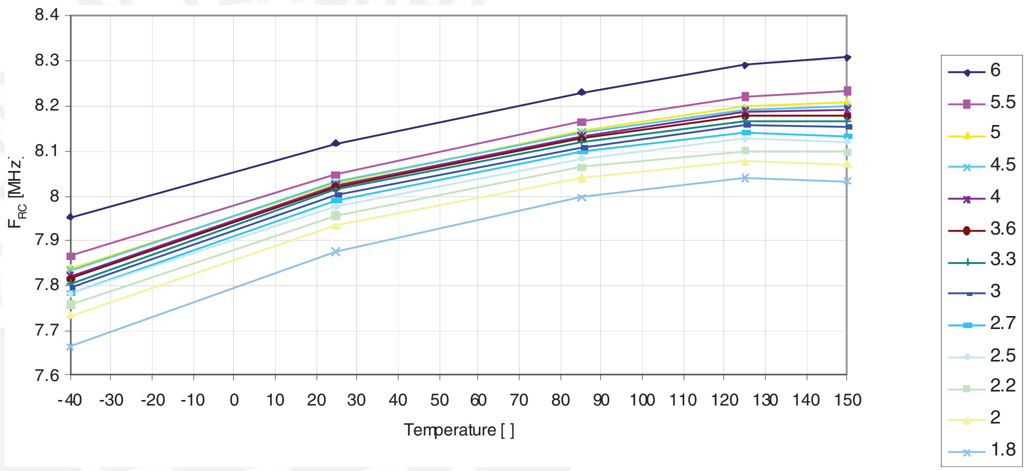
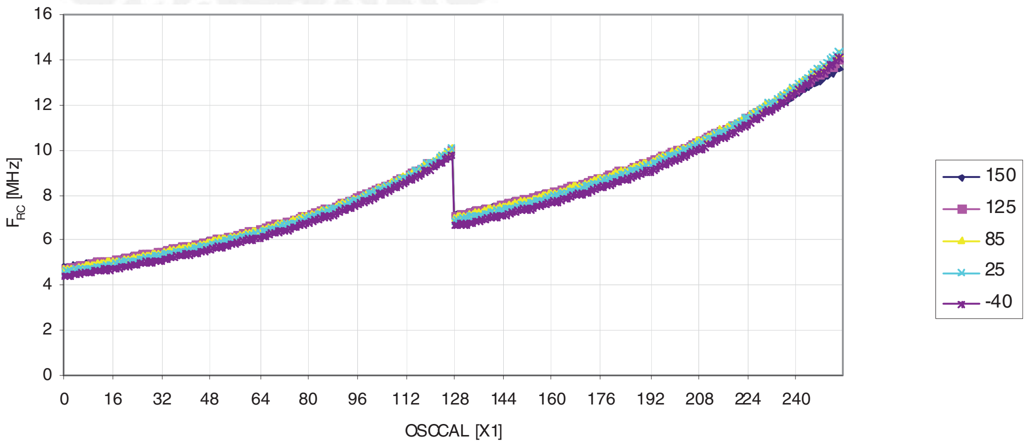
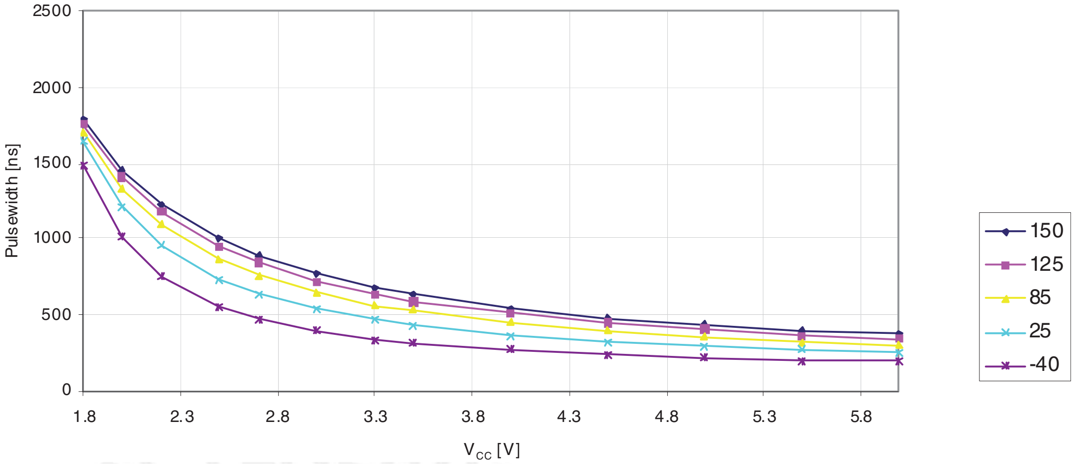


**Figure 30-25.** Calibrated 8MHz RC Oscillator Frequency versus OSCCAL Value



### 30.1.10 Reset Pulse width

Figure 30-26. Minimum Reset Pulse width versus $V_{CC}$



## 30.2 ATmega88PA Typical Characteristics

### 30.2.1 Active Supply Current

Figure 30-27. Active Supply Current versus Low Frequency (0.1-1.0MHz)

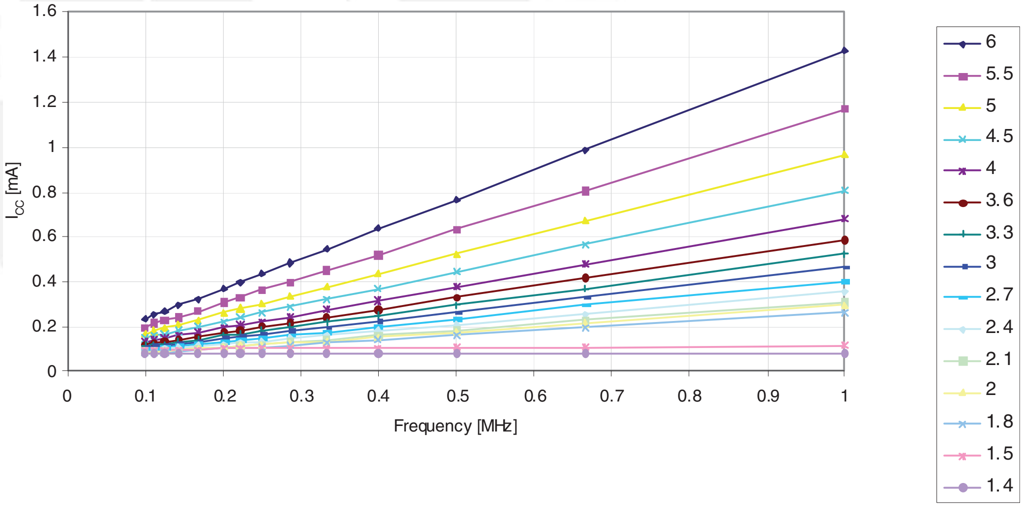
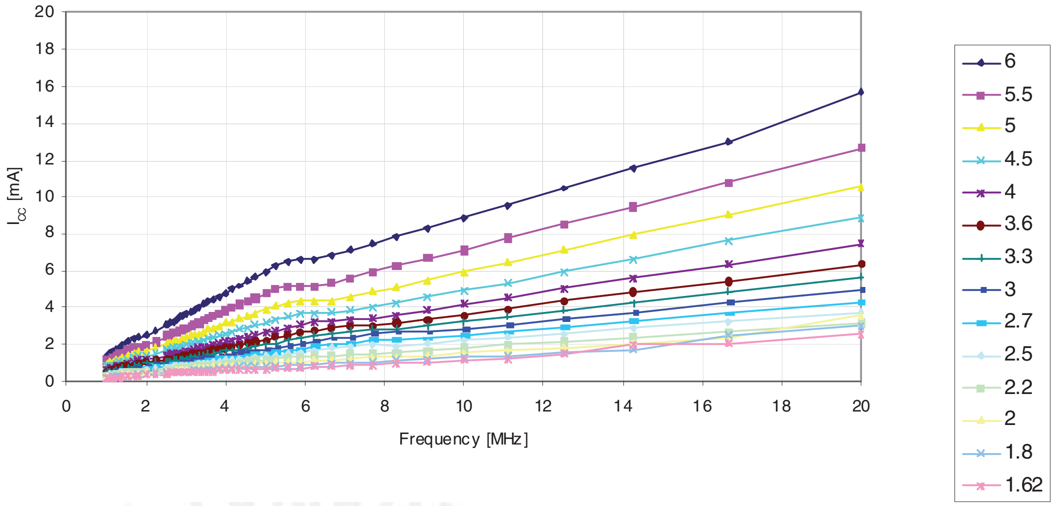


**Figure 30-28.** Active Supply Current versus Frequency (1-16MHz)



### 30.2.2 Idle Supply Current

**Figure 30-29.** Idle Supply Current versus Low Frequency (0.1-1.0MHz)

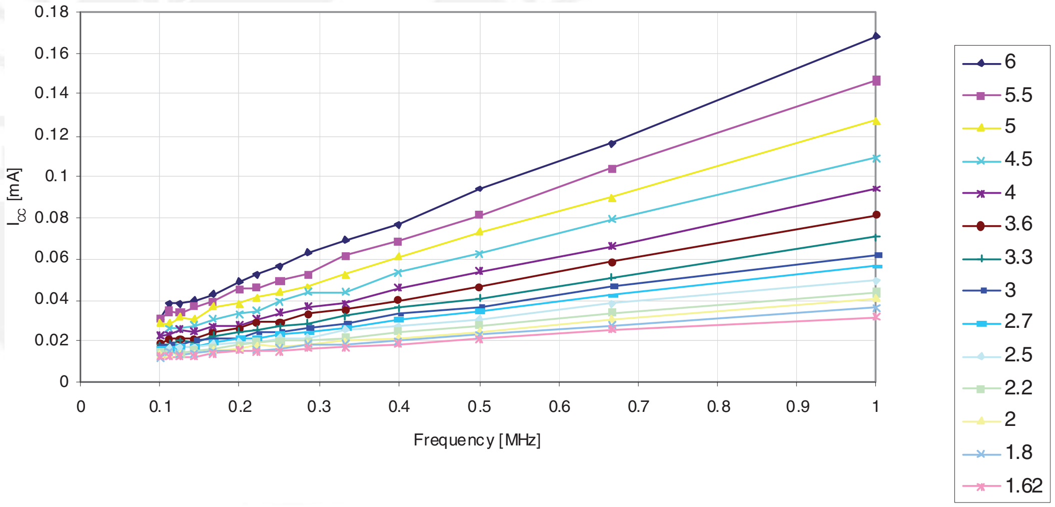
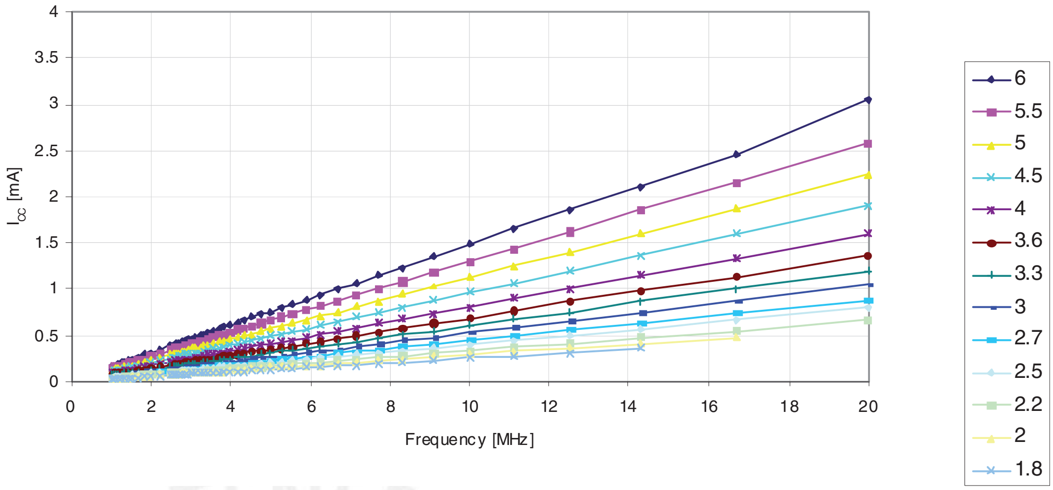


**Figure 30-30.** Idle Supply Current versus Frequency (1-16MHz)



### 30.2.3 Supply Current of IO Modules

The tables and formulas below can be used to calculate the additional current consumption for the different I/O modules in Active and Idle mode. The enabling or disabling of the I/O modules are controlled by the Power Reduction Register. See "Power Reduction Register" on page 42 for details.

**Table 30-3.** Additional Current Consumption for the Different I/O Modules (Absolute Values)

| PRR bit | Typical numbers | | |
|---|---|---|---|
| | $V_{CC}$ = 2V, F = 1MHz | $V_{CC}$ = 3V, F = 4MHz | $V_{CC}$ = 5V, F = 8MHz |
| PRUSART0 | 2.9µA | 20.7µA | 97.4µA |
| PRTWI | 6.0µA | 44.8µA | 219.7µA |
| PRTIM2 | 5.0µA | 34.5µA | 141.3µA |
| PRTIM1 | 3.6µA | 24.4µA | 107.7µA |
| PRTIM0 | 1.4µA | 9.5µA | 38.4µA |
| PRSPI | 5.0µA | 38.0µA | 190.4µA |
| PRADC | 6.1µA | 47.4µA | 244.7µA |

**Table 30-4.** Additional Current Consumption (percentage) in Active and Idle mode

| PRR bit | Additional Current consumption compared to Active with external clock (see Figure 30-1 on page 324 and Figure 30-2 on page 324) | Additional Current consumption compared to Idle with external clock (see Figure 30-3 on page 325 and Figure 30-4 on page 325) |
|---|---|---|
| PRUSART0 | 1.8% | 11.4% |
| PRTWI | 3.9% | 20.6% |
| PRTIM2 | 2.9% | 15.7% |
| PRTIM1 | 2.1% | 11.2% |
| PRTIM0 | 0.8% | 4.2% |
| PRSPI | 3.3% | 17.6% |
| PRADC | 4.2% | 22.1% |

It is possible to calculate the typical current consumption based on the numbers from Table 30-2 on page 326 for other $V_{CC}$ and frequency settings than listed in Table 30-1 on page 326.

#### 30.2.3.1 *Example*

Calculate the expected current consumption in idle mode with TIMER1, ADC, and SPI enabled at $V_{CC}$ = 2.0V and F = 1MHz. From Table 30-2 on page 326, third column, we see that we need to add 11.2% for the TIMER1, 22.1% for the ADC, and 17.6% for the SPI module. Reading from Figure 30-3 on page 325, we find that the idle current consumption is ~0.028mA at $V_{CC}$ = 2.0V and F = 1MHz. The total current consumption in idle mode with TIMER1, ADC, and SPI enabled, gives:

$$I_{CCtotal} \approx 0.028 \text{ mA} \cdot (1 + 0.112 + 0.221 + 0.176) \approx 0.042 \text{ mA}$$

### 30.2.4 Power-down Supply Current

**Figure 30-31.** Power-Down Supply Current versus $V_{CC}$ (Watchdog Timer Disabled)

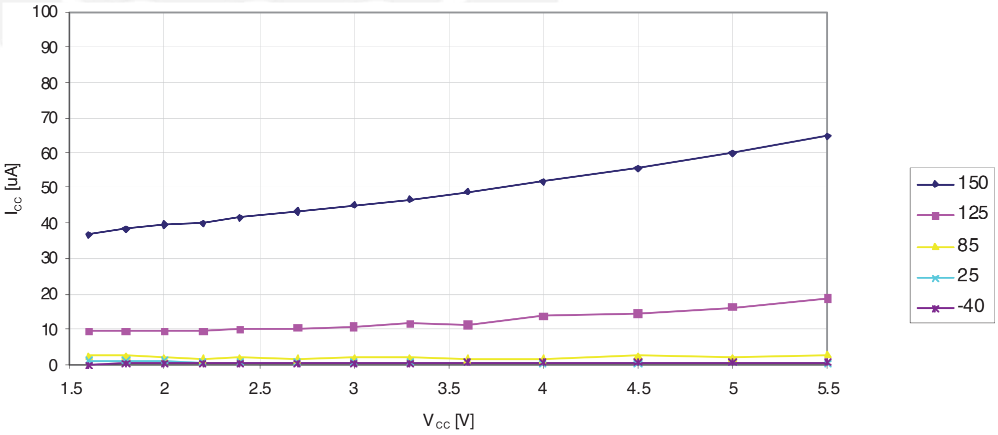
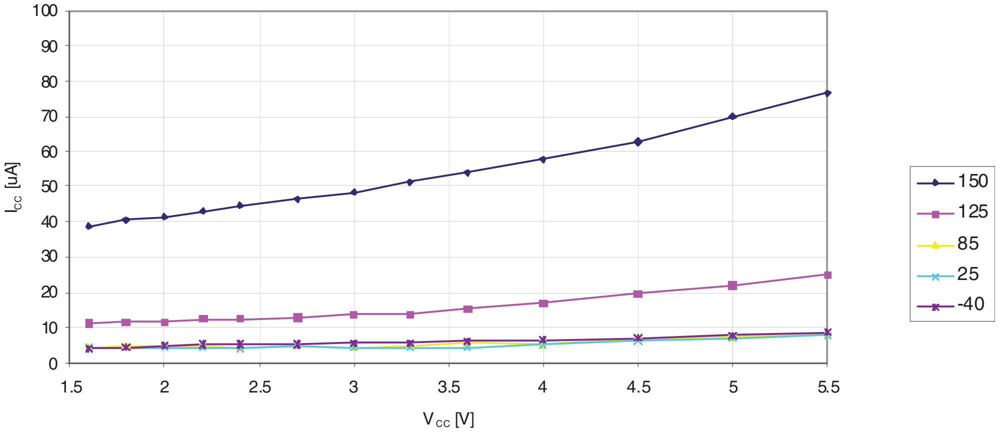


**Figure 30-32.** Power-Down Supply Current versus $V_{CC}$ (Watchdog Timer Enabled)



### 30.2.5 Pin Pull-Up

**Figure 30-33.** I/O Pin Pull-up Resistor Current versus Input Voltage ($V_{CC}$ = 5.0V)

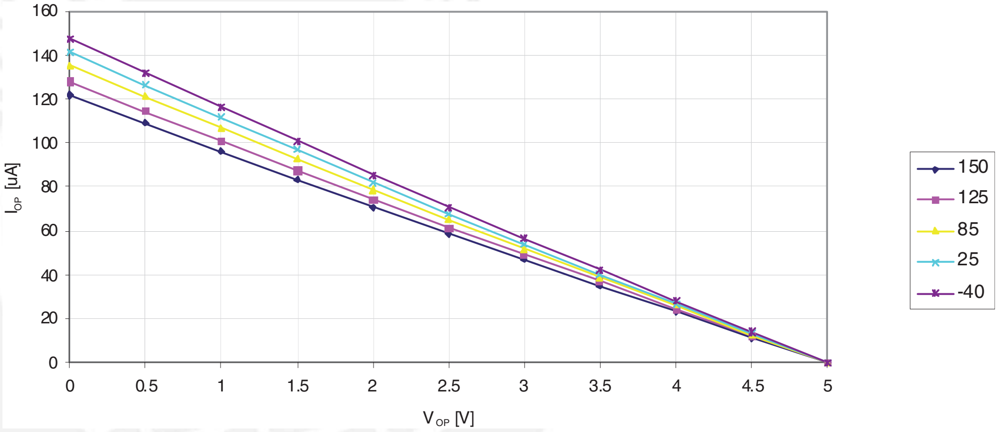
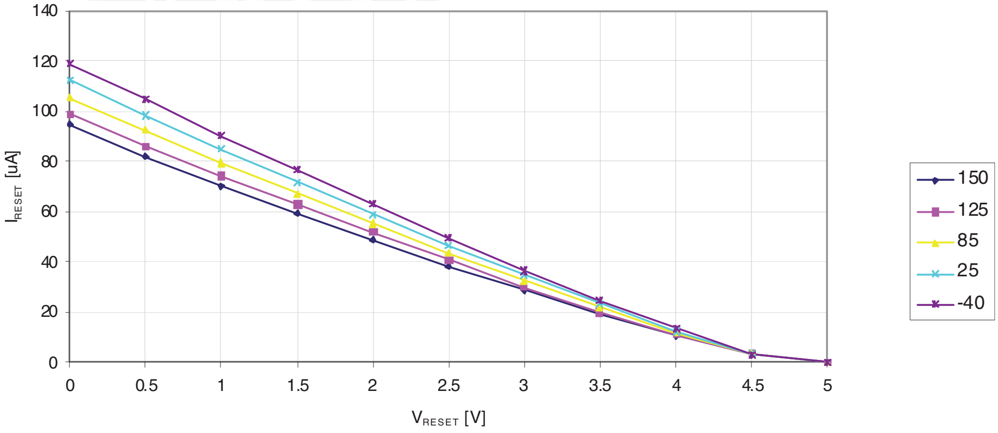


**Figure 30-34.** Reset Pull-up Resistor Current versus Reset Pin Voltage ($V_{CC}$ = 5V)



### 30.2.6 Pin Driver Strength

**Figure 30-35.** I/O Pin Output Voltage versus Sink Current ($V_{CC}$ = 3V)

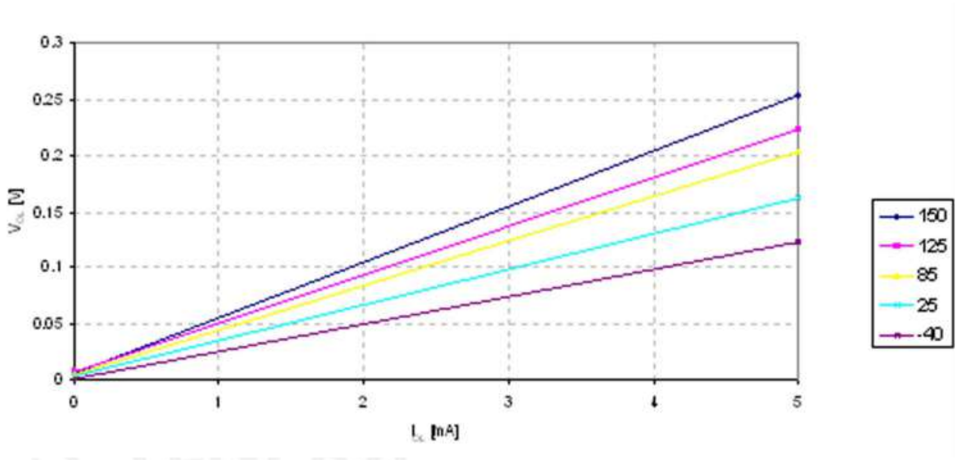


**Figure 30-36.** I/O Pin Output Voltage versus Sink Current ($V_{CC}$ = 5V)

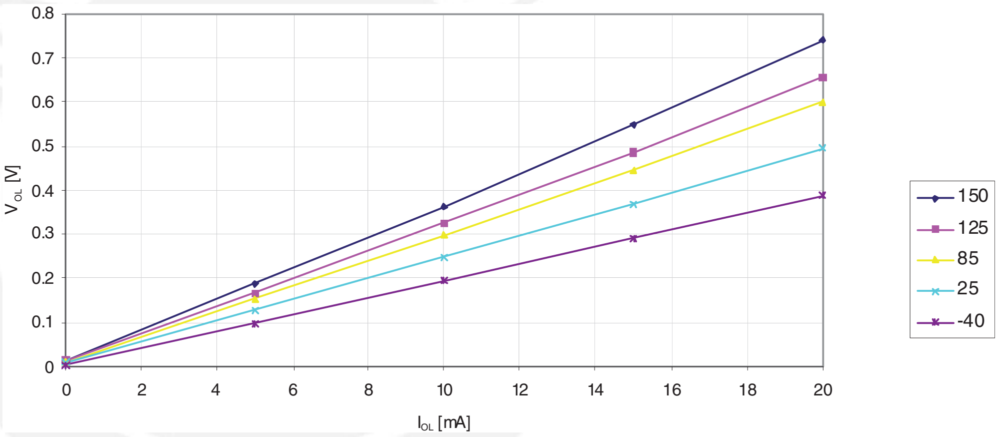


**Figure 30-37.** I/O Pin Output Voltage versus Source Current (Vcc = 3V)

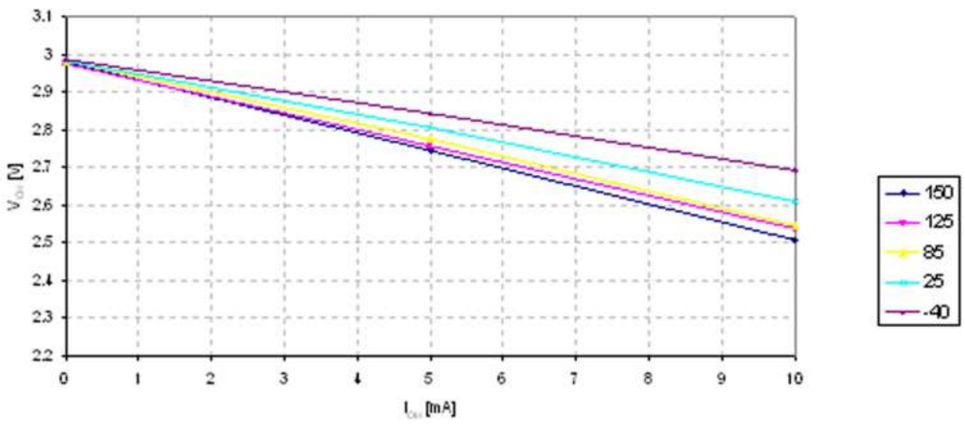
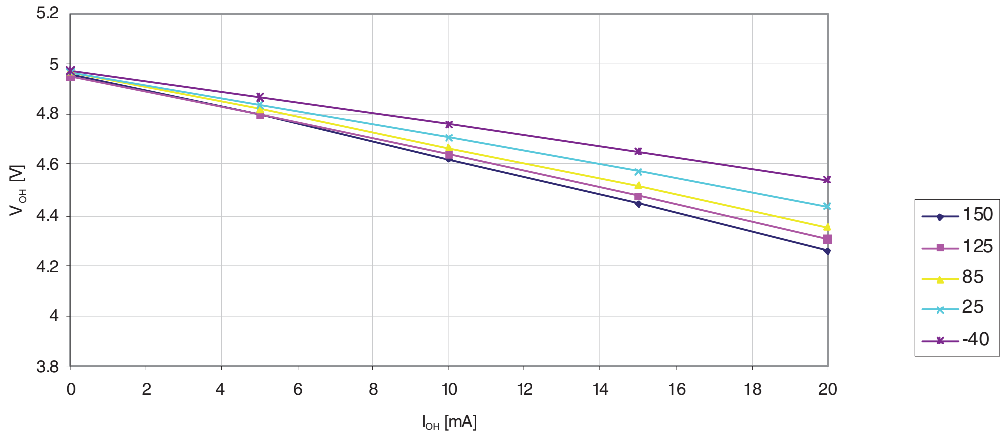


**Figure 30-38.** I/O Pin Output Voltage versus Source Current ($V_{CC}$ = 5V)



## 30.2.7 Pin Threshold and Hysteresis

**Figure 30-39.** I/O Pin Input Threshold Voltage versus $V_{CC}$ ($V_{IH}$, I/O Pin read as '1')

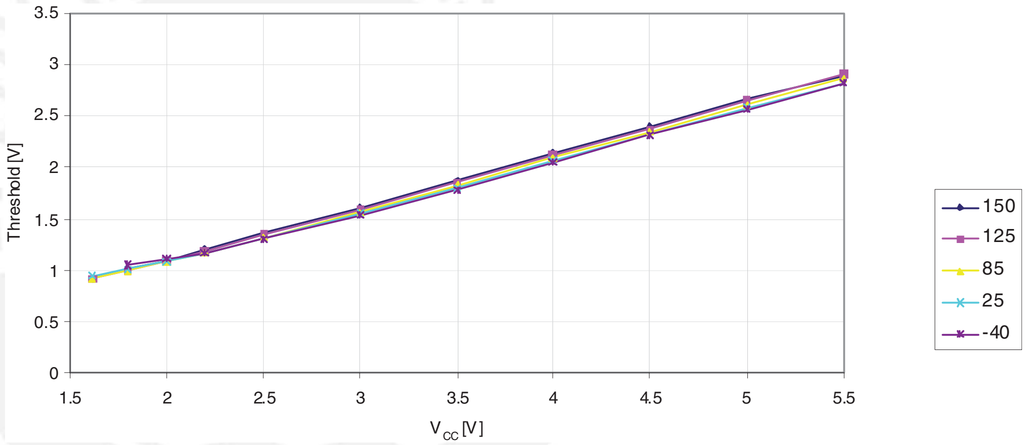


**Figure 30-40.** I/O Pin Input Threshold Voltage versus $V_{CC}$ ($V_{IL}$, I/O Pin read as '0')

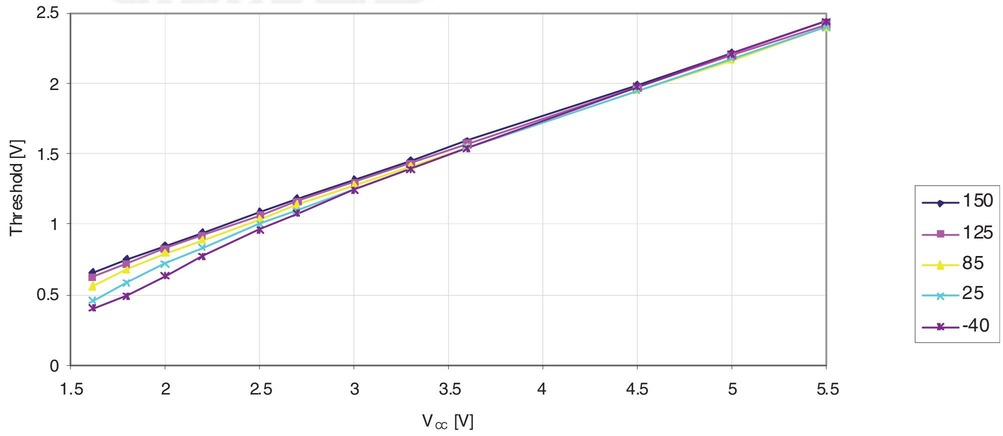


**Figure 30-41.** Reset Input Threshold Voltage versus $V_{CC}$ ($V_{IH}$, I/O Pin read as '1')

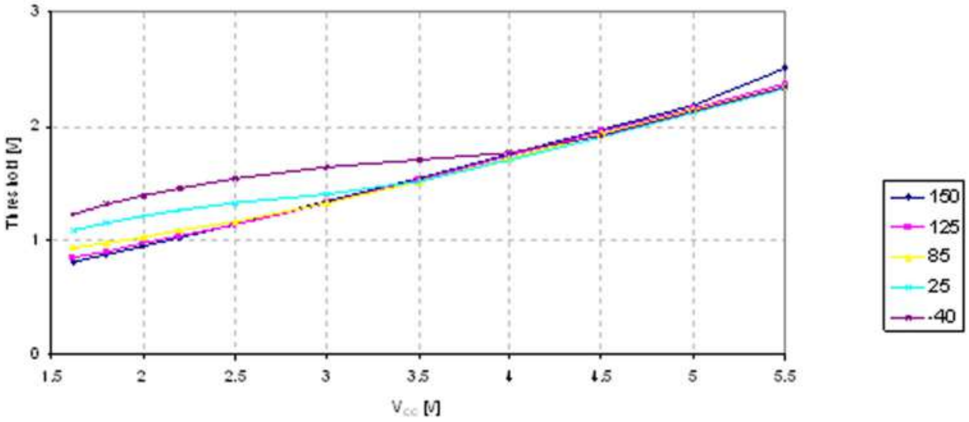
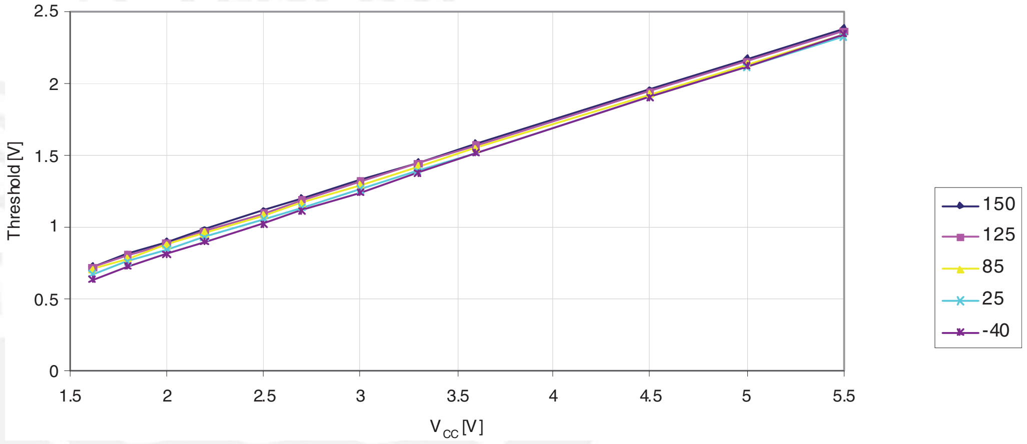


**Figure 30-42.** Reset Input Threshold Voltage versus $V_{CC}$ ($V_{IL}$, I/O Pin read as '0')



### 30.2.8 BOD Threshold

**Figure 30-43.** BOD Thresholds versus Temperature (BODLEVEL is 1.8V)

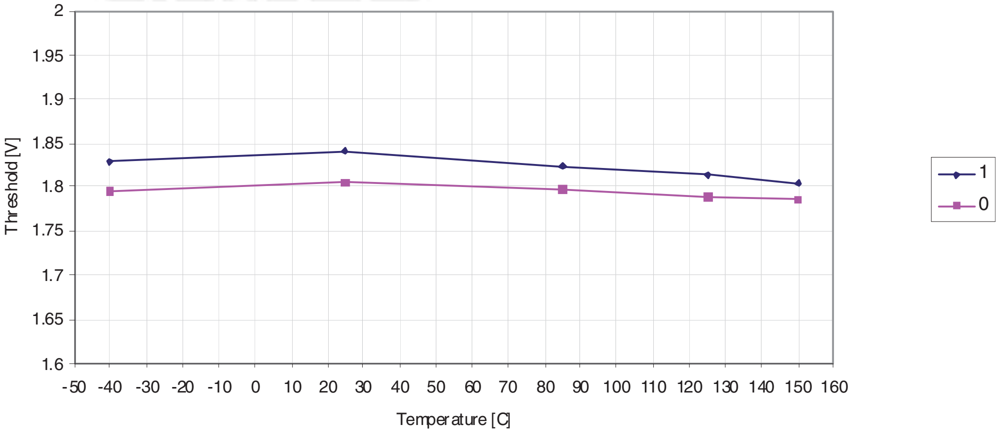


Figure 30-44. BOD Thresholds versus Temperature (BODLEVEL is 2.7V)

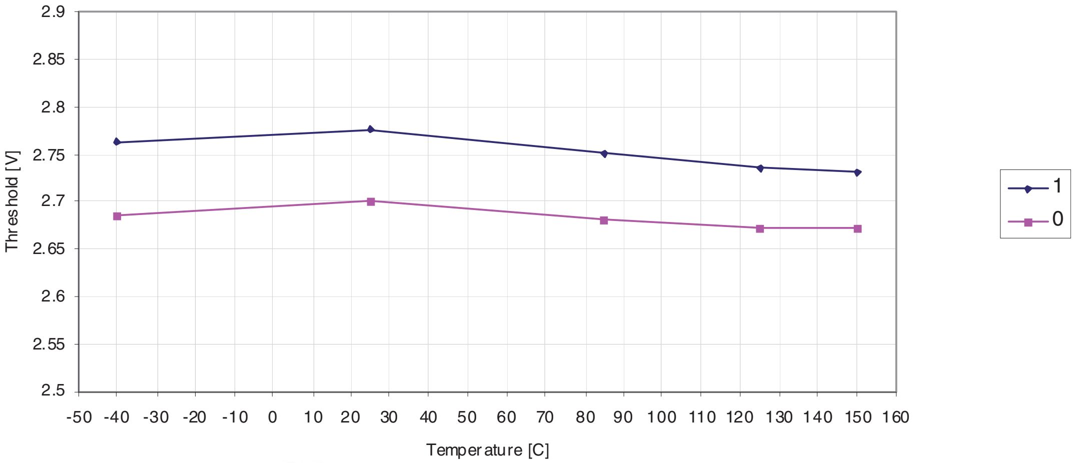


Figure 30-45. BOD Thresholds versus Temperature (BODLEVEL is 4.3V)

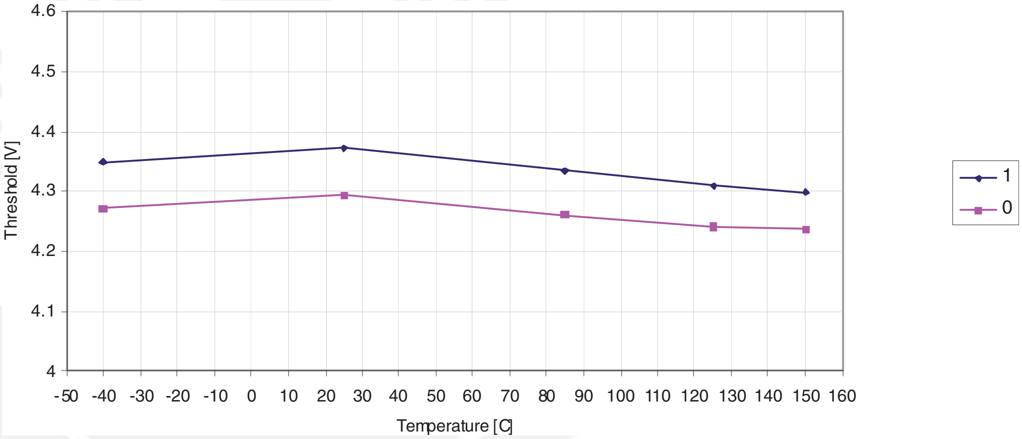
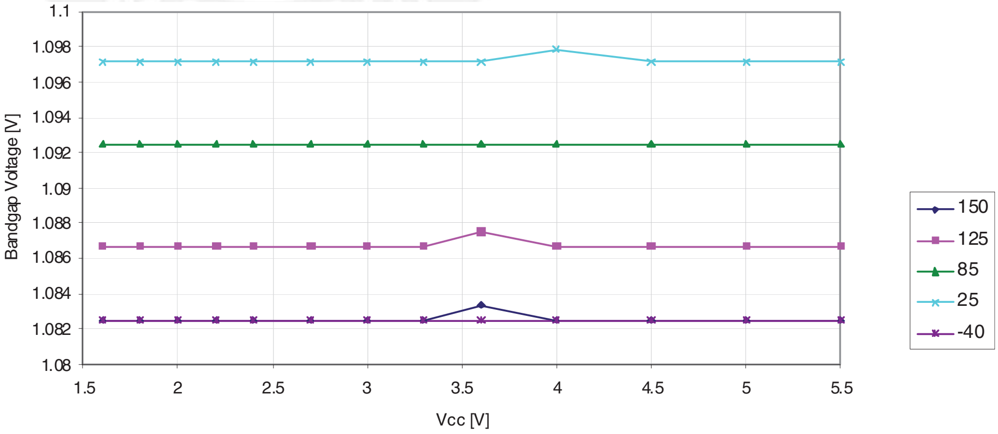


Figure 30-46. Bandgap Voltage versus $V_{CC}$



### 30.2.9 Internal Oscillator Speed

**Figure 30-47.** Watchdog Oscillator Frequency versus Temperature

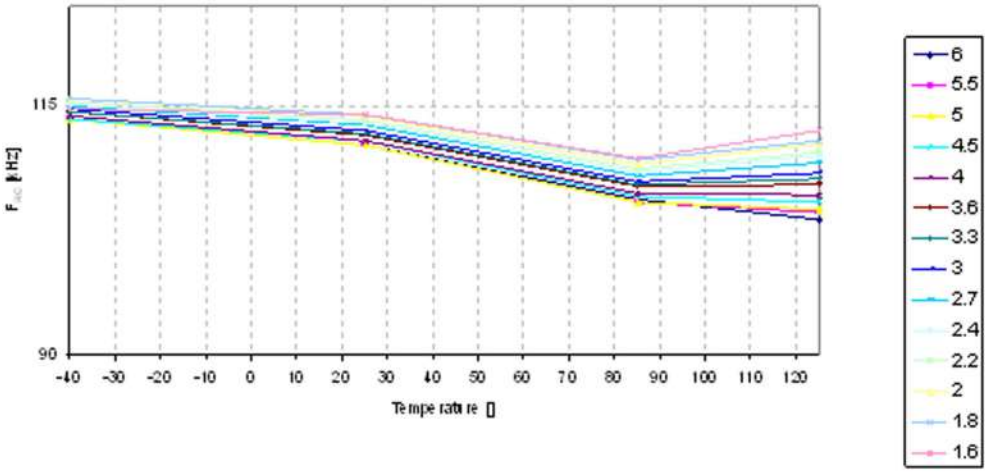


**Figure 30-48.** Watchdog Oscillator Frequency versus $V_{CC}$

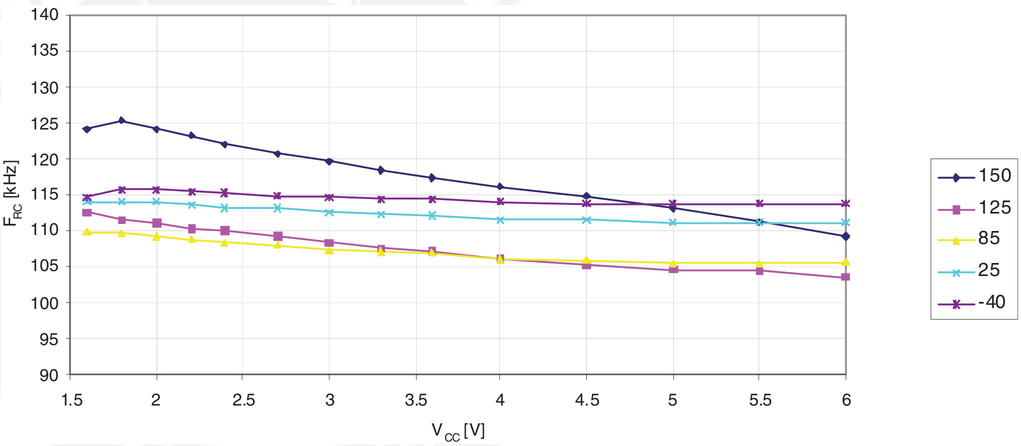


**Figure 30-49.** Calibrated 8MHz RC Oscillator Frequency versus $V_{CC}$

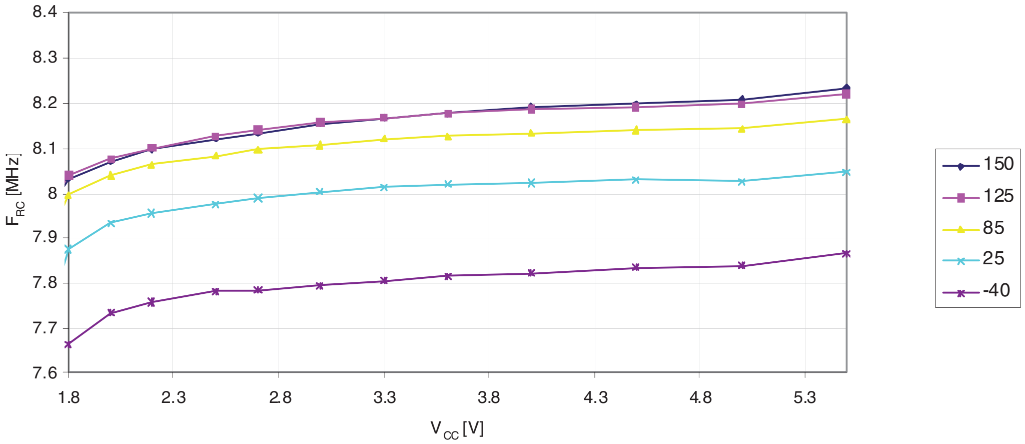


**Figure 30-50.** Calibrated 8MHz RC Oscillator Frequency versus Temperature

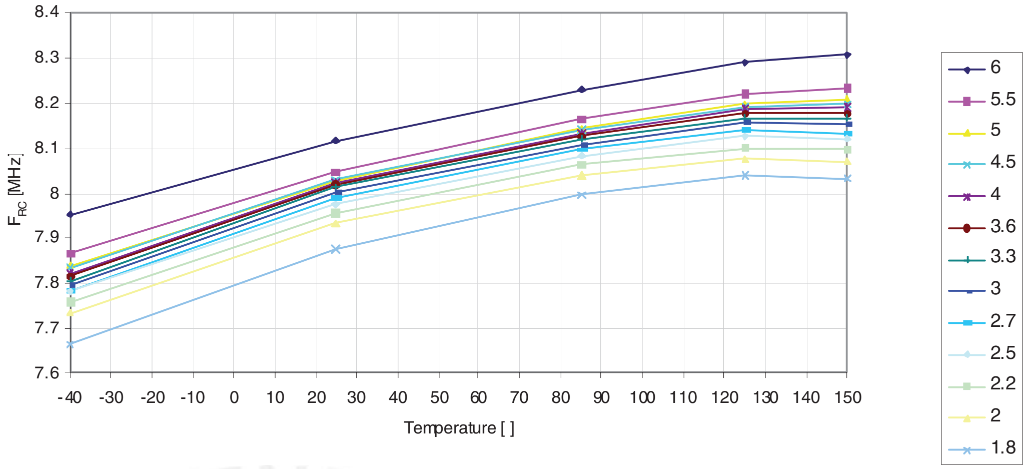
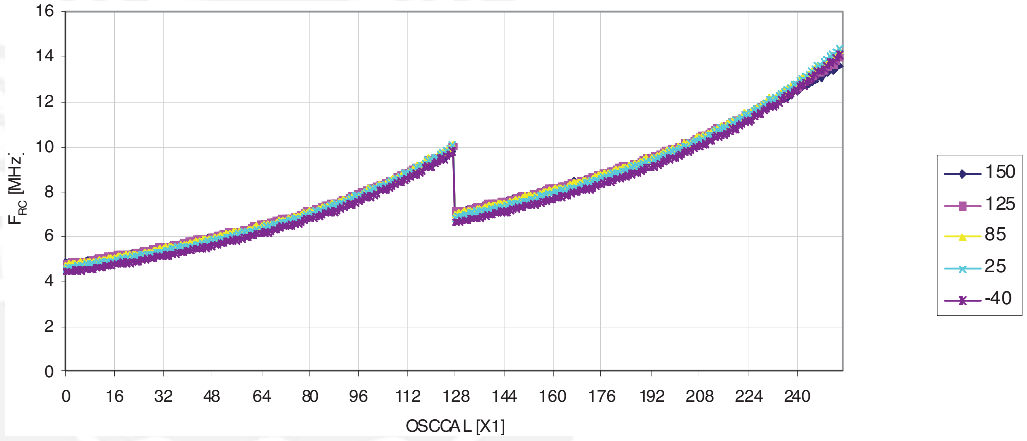
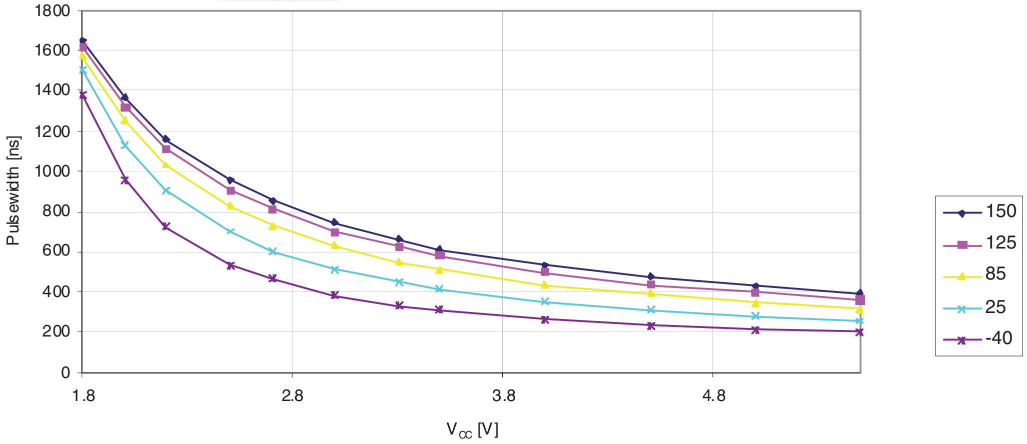


**Figure 30-51.** Calibrated 8MHz RC Oscillator Frequency versus OSCCAL Value



### 30.2.10 Reset Pulse width

**Figure 30-52.** Minimum Reset Pulse width versus $V_{CC}$



## 30.3 ATmega168PA Typical Characteristics

### 30.3.1 Active Supply Current

**Figure 30-53.** Active Supply Current versus Low Frequency (0.1-1.0MHz)

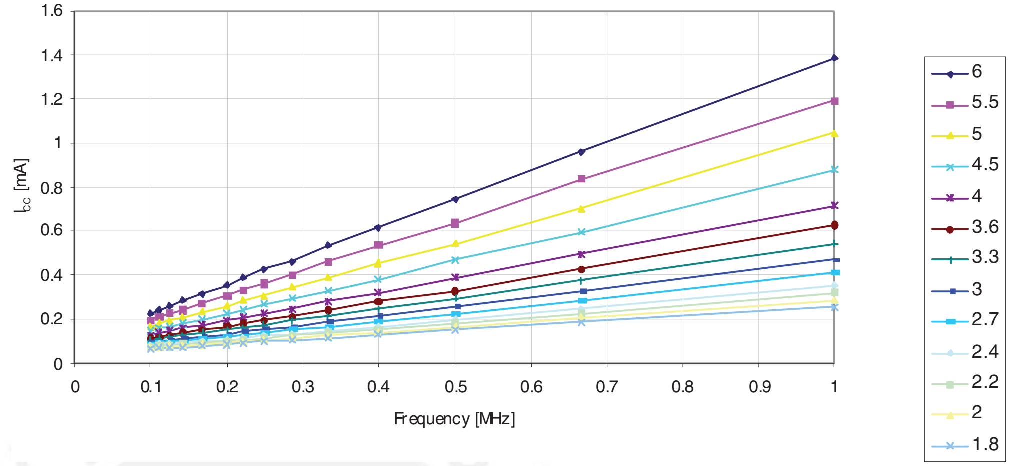
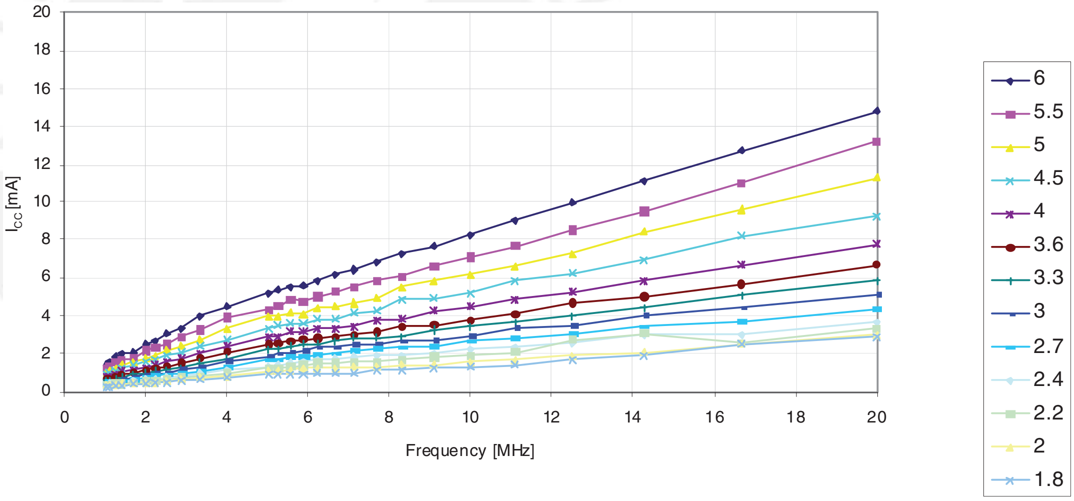


**Figure 30-54.** Active Supply Current versus Frequency (1-16MHz)



### 30.3.2 Idle Supply Current

**Figure 30-55.** Idle Supply Current versus Low Frequency (0.1-1.0MHz)

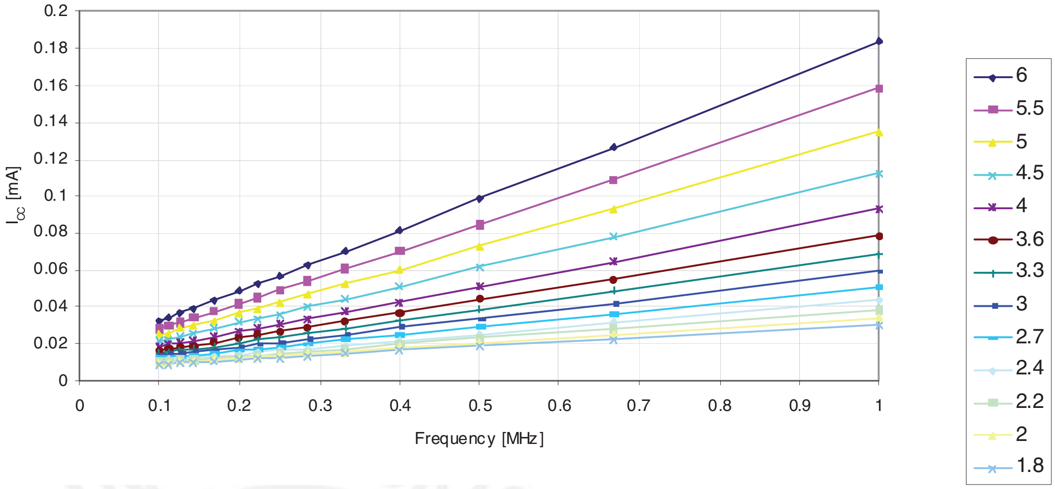
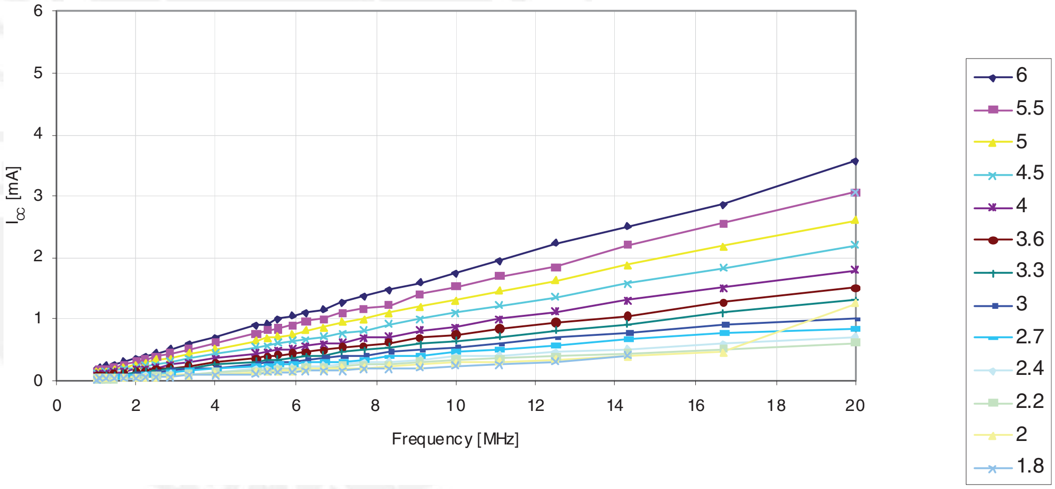


**Figure 30-56.** Idle Supply Current versus Frequency (1-16MHz)



### 30.3.3 Supply Current of IO Modules

The tables and formulas below can be used to calculate the additional current consumption for the different I/O modules in Active and Idle mode. The enabling or disabling of the I/O modules are controlled by the Power Reduction Register. See "Power Reduction Register" on page 42 for details.

**Table 30-5.** Additional Current Consumption for the Different I/O Modules (Absolute Values)

| PRR bit | Typical numbers | | |
|---|---|---|---|
| | $V_{CC}$ = 2V, F = 1MHz | $V_{CC}$ = 3V, F = 4MHz | $V_{CC}$ = 5V, F = 8MHz |
| PRUSART0 | 2.9µA | 20.7µA | 97.4µA |
| PRTWI | 6.0µA | 44.8µA | 219.7µA |
| PRTIM2 | 5.0µA | 34.5µA | 141.3µA |
| PRTIM1 | 3.6µA | 24.4µA | 107.7µA |
| PRTIM0 | 1.4µA | 9.5µA | 38.4µA |
| PRSPI | 5.0µA | 38.0µA | 190.4µA |
| PRADC | 6.1µA | 47.4µA | 244.7µA |

**Table 30-6.** Additional Current Consumption (Percentage) in Active and Idle Mode

| PRR bit | Additional Current consumption compared to Active with external clock (see Figure 30-1 on page 324 and Figure 30-2 on page 324) | Additional Current consumption compared to Idle with external clock (see Figure 30-3 on page 325 and Figure 30-4 on page 325) |
|---|---|---|
| PRUSART0 | 1.8% | 11.4% |
| PRTWI | 3.9% | 20.6% |
| PRTIM2 | 2.9% | 15.7% |
| PRTIM1 | 2.1% | 11.2% |
| PRTIM0 | 0.8% | 4.2% |
| PRSPI | 3.3% | 17.6% |
| PRADC | 4.2% | 22.1% |

It is possible to calculate the typical current consumption based on the numbers from Table 30-2 on page 326 for other $V_{CC}$ and frequency settings than listed in Table 30-1 on page 326.

#### *30.3.3.1 Example*

Calculate the expected current consumption in idle mode with TIMER1, ADC, and SPI enabled at $V_{CC}$ = 2.0V and F = 1MHz. From Table 30-2 on page 326, third column, we see that we need to add 11.2% for the TIMER1, 22.1% for the ADC, and 17.6% for the SPI module. Reading from Figure 30-3 on page 325, we find that the idle current consumption is ~0.028mA at $V_{CC}$ = 2.0V and F = 1MHz. The total current consumption in idle mode with TIMER1, ADC, and SPI enabled, gives:

$$I_{CC\text{total}} \approx 0.028\text{ mA} \cdot (1 + 0.112 + 0.221 + 0.176) \approx 0.042\text{ mA}$$

### 30.3.4 Power-down Supply Current

**Figure 30-57.** Power-Down Supply Current versus $V_{CC}$ (Watchdog Timer Disabled)

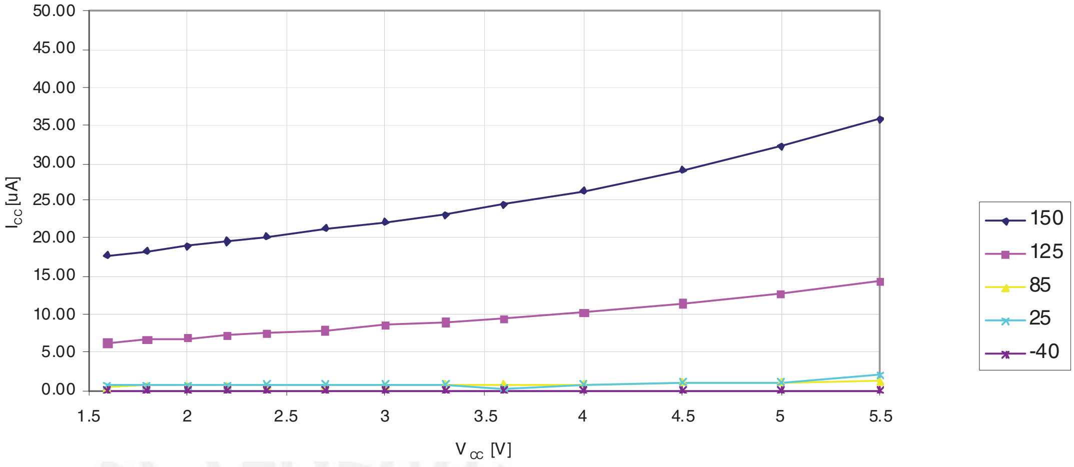
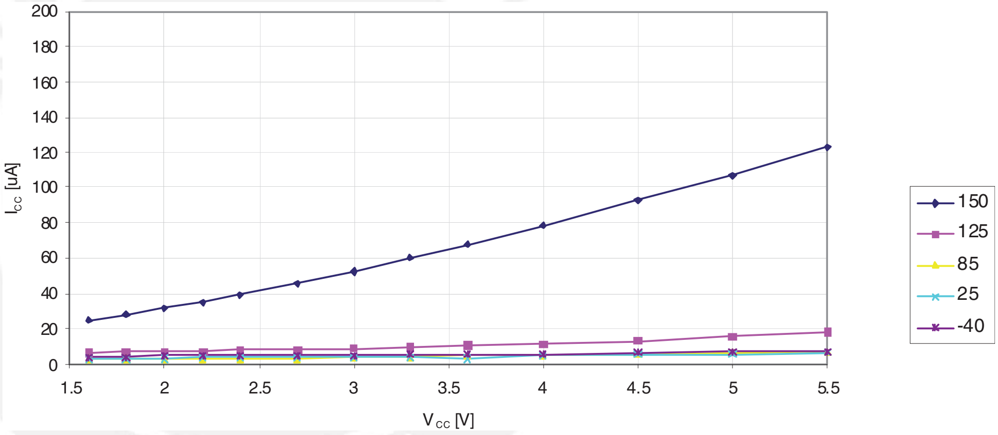


**Figure 30-58.** Power-Down Supply Current versus $V_{CC}$ (Watchdog Timer Enabled)



### 30.3.5 Pin Pull-Up

**Figure 30-59.** I/O Pin Pull-up Resistor Current versus Input Voltage ($V_{CC}$ = 5.0V)

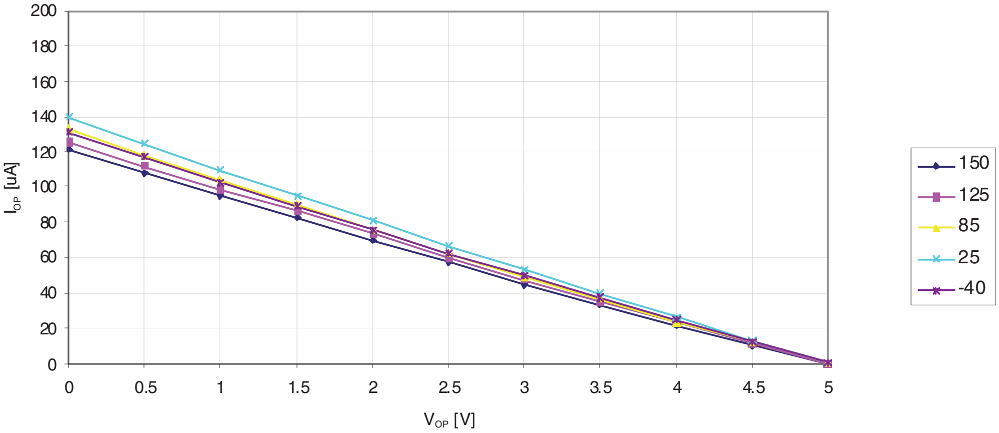
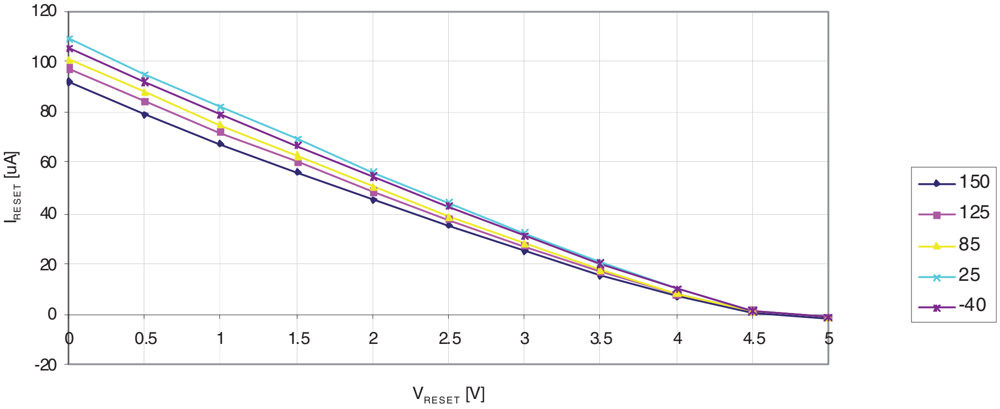


**Figure 30-60.** Reset Pull-up Resistor Current versus Reset Pin Voltage ($V_{CC}$ = 5V)



### 30.3.6 Pin Driver Strength

**Figure 30-61.** I/O Pin Output Voltage versus Sink Current ($V_{CC}$ = 3V)

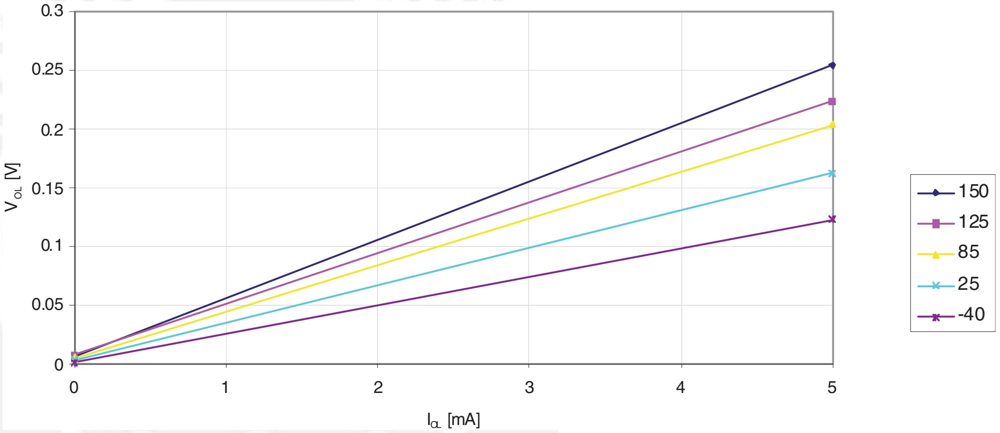


**Figure 30-62.** I/O Pin Output Voltage versus Sink Current ($V_{CC}$ = 5V)

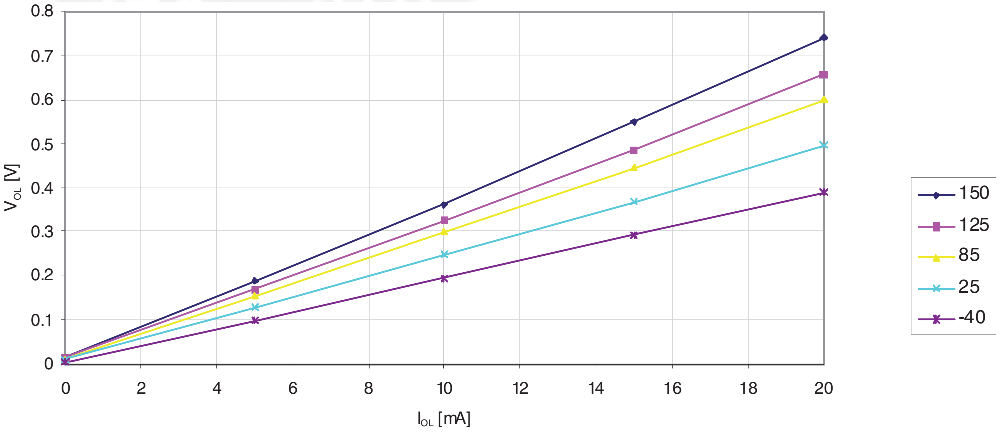


**Figure 30-63.** I/O Pin Output Voltage versus Source Current (Vcc = 3V)

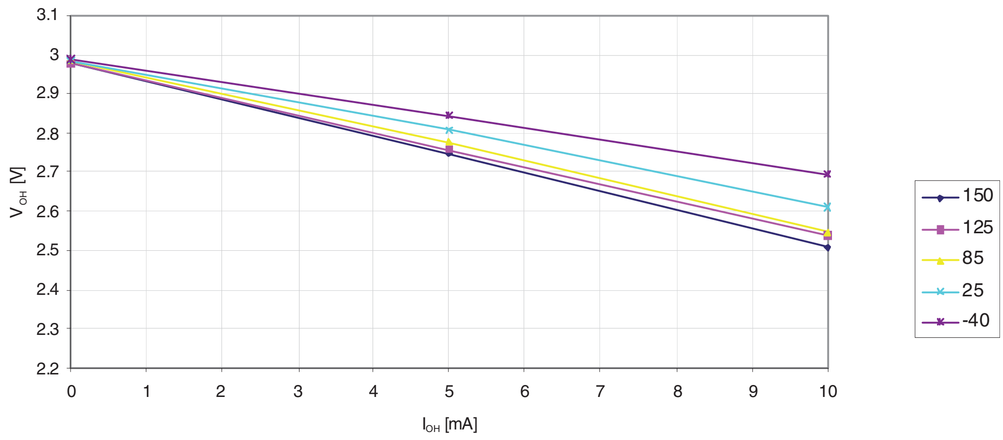
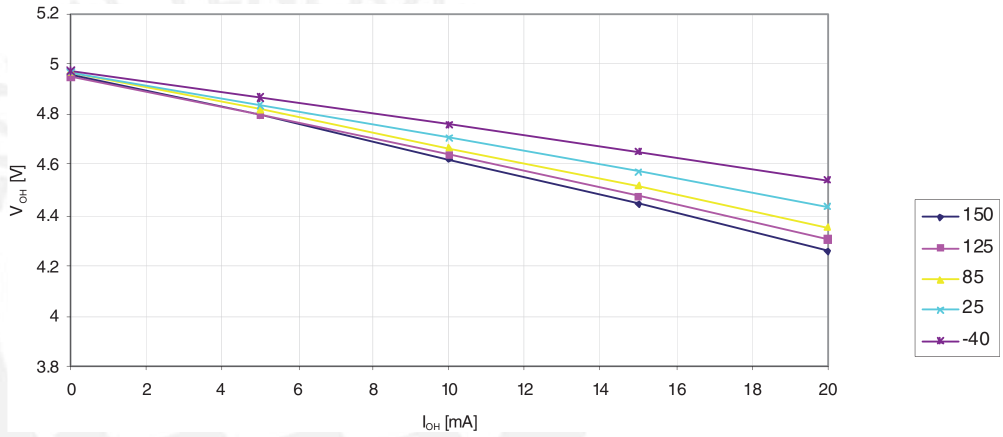


**Figure 30-64.** I/O Pin Output Voltage versus Source Current ($V_{CC}$ = 5V)



### 30.3.7 Pin Threshold and Hysteresis

**Figure 30-65.** I/O Pin Input Threshold Voltage versus $V_{CC}$ ($V_{IH}$, I/O Pin read as '1')

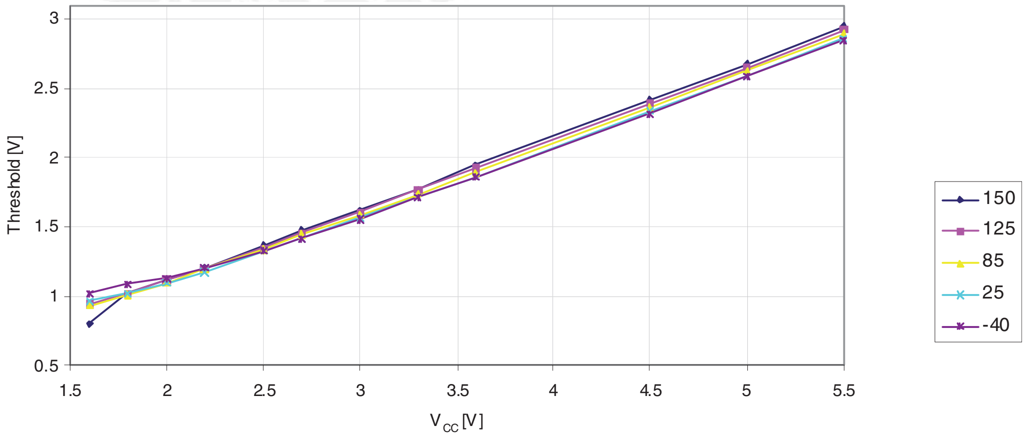


**Figure 30-66.** I/O Pin Input Threshold Voltage versus $V_{CC}$ ($V_{IL}$, I/O Pin read as ‘0’)

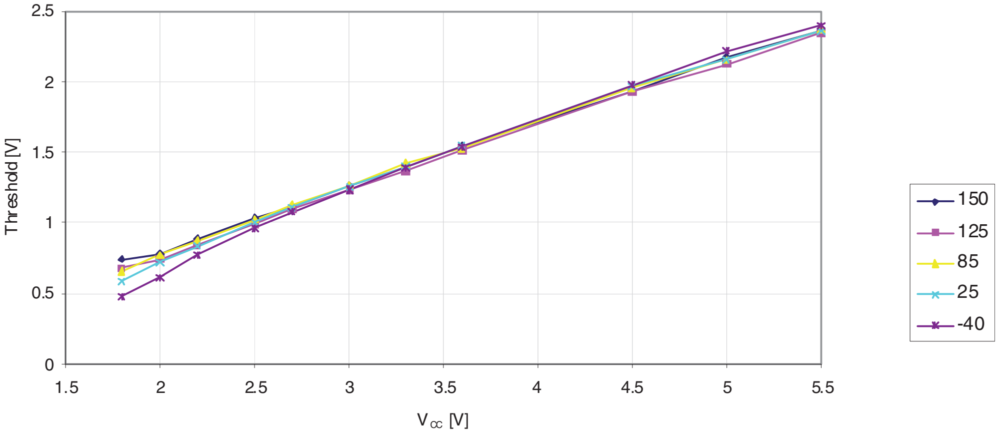


**Figure 30-67.** Reset Input Threshold Voltage versus $V_{CC}$ ($V_{IH}$, I/O Pin read as ‘1’)

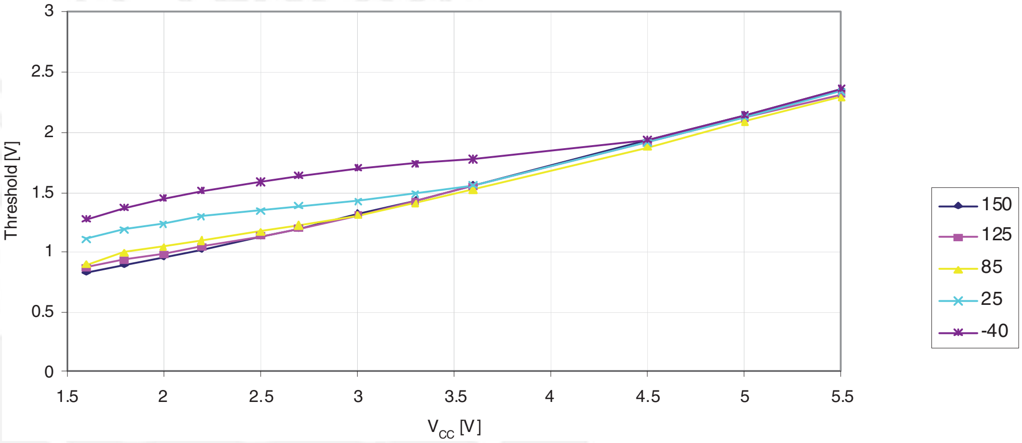
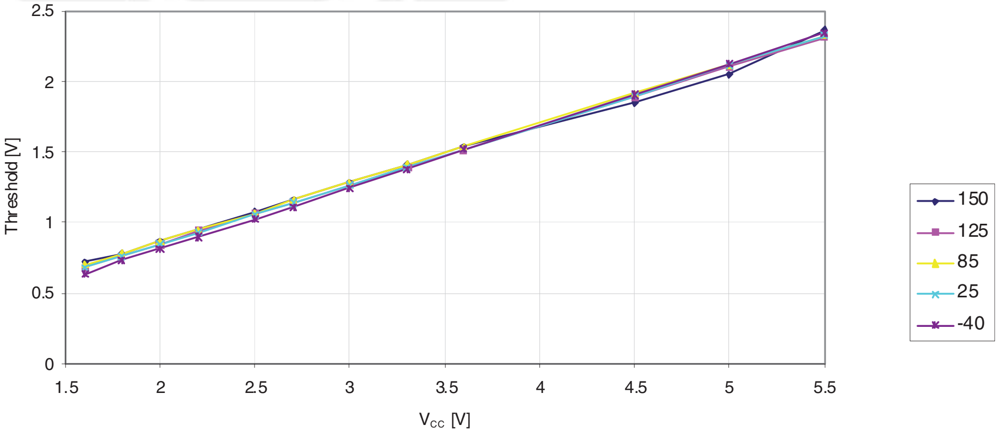


**Figure 30-68.** Reset Input Threshold Voltage versus $V_{CC}$ ($V_{IL}$, I/O Pin read as ‘0’)



### 30.3.8 BOD Threshold

**Figure 30-69.** BOD Thresholds versus Temperature (BODLEVEL is 1.8V)

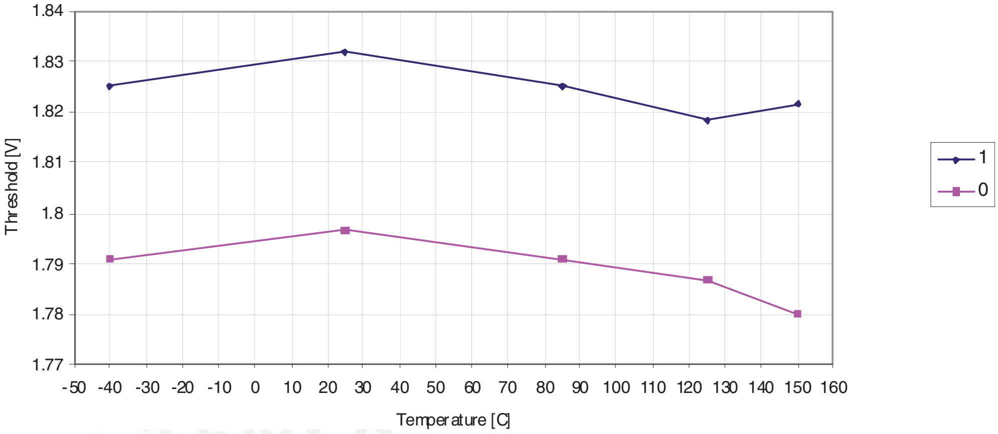


**Figure 30-70.** BOD Thresholds versus Temperature (BODLEVEL is 2.7V)

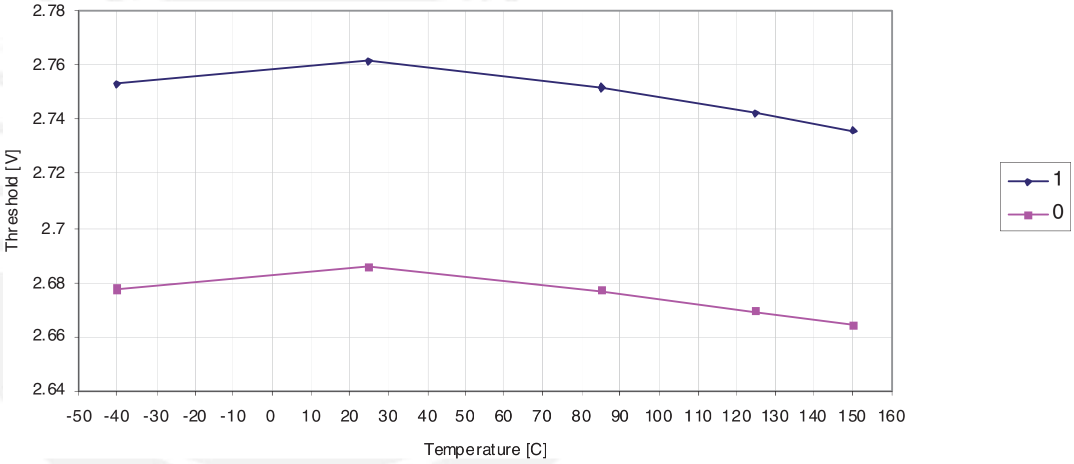


**Figure 30-71.** BOD Thresholds versus Temperature (BODLEVEL is 4.3V)

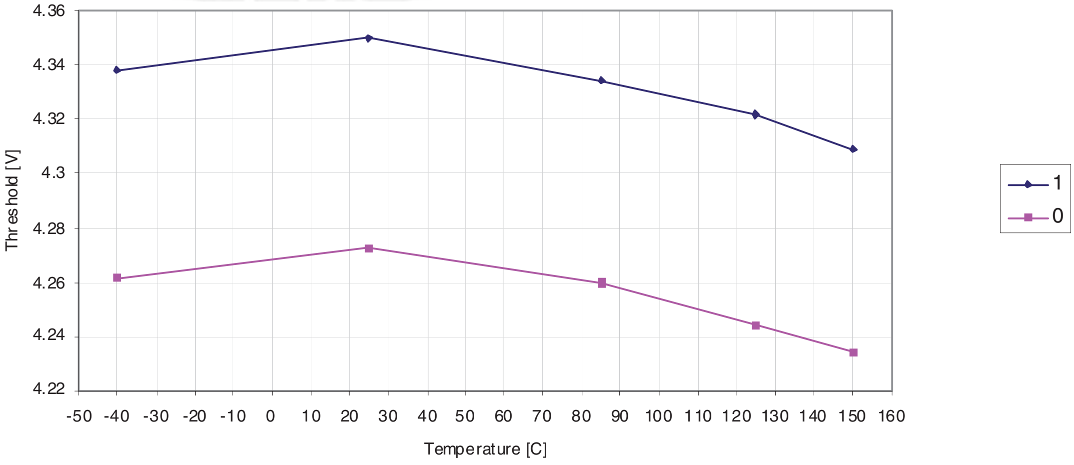
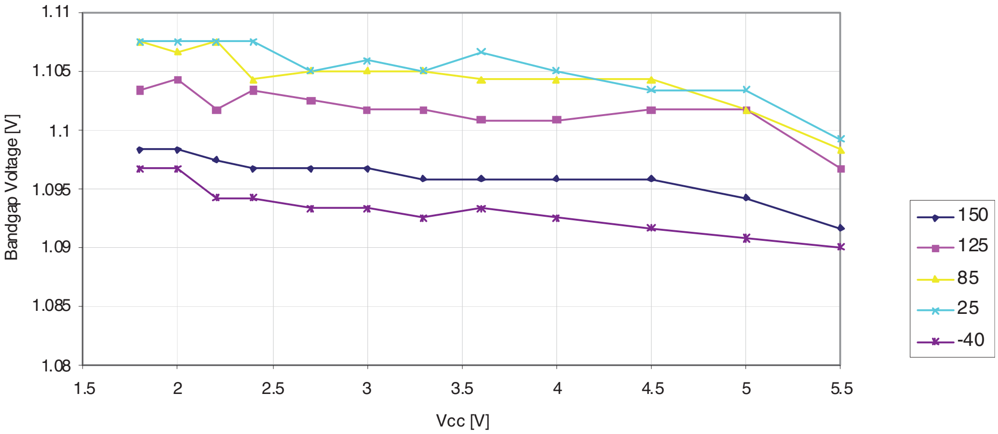


**Figure 30-72.** Bandgap Voltage versus $V_{CC}$



### 30.3.9 Internal Oscillator Speed

**Figure 30-73.** Watchdog Oscillator Frequency versus Temperature

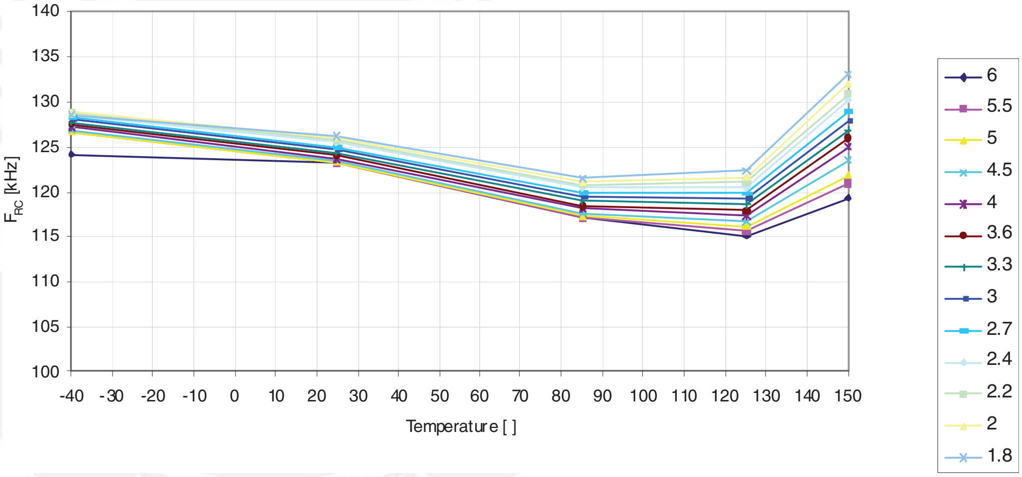


**Figure 30-74.** Watchdog Oscillator Frequency versus $V_{CC}$

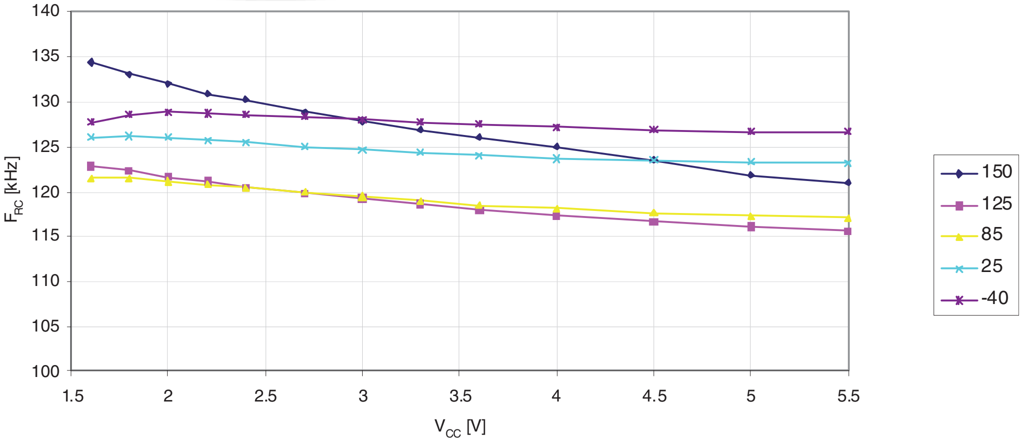


**Figure 30-75.** Calibrated 8MHz RC Oscillator Frequency versus $V_{CC}$

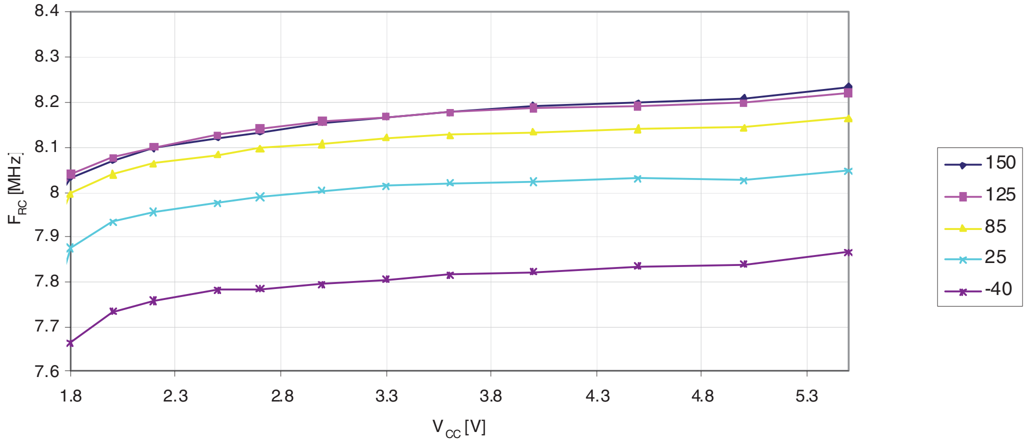


**Figure 30-76.** Calibrated 8MHz RC Oscillator Frequency versus Temperature

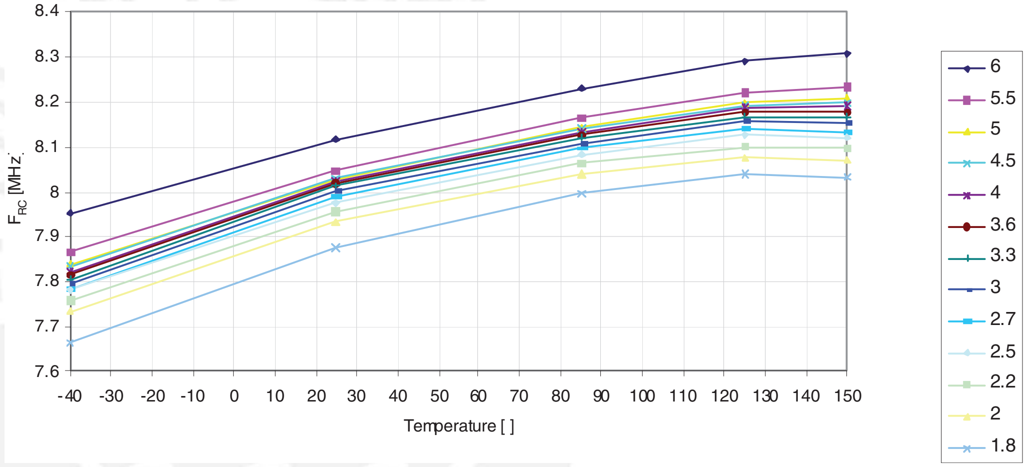
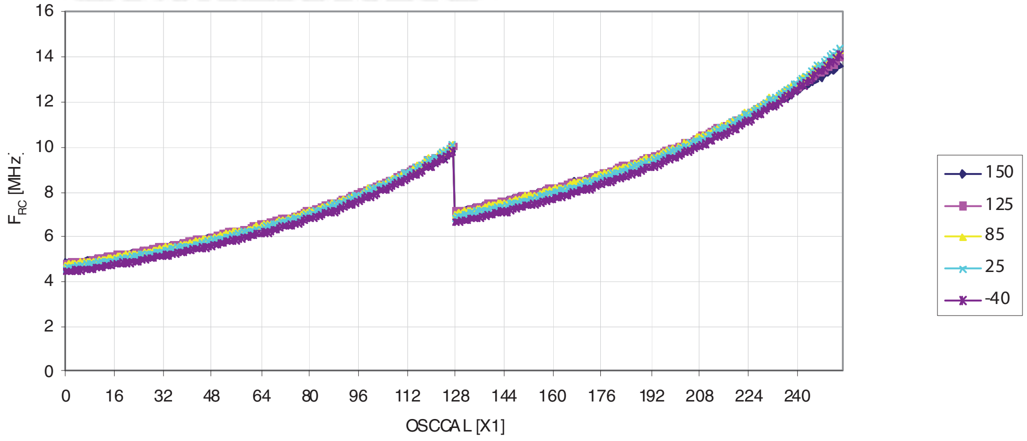
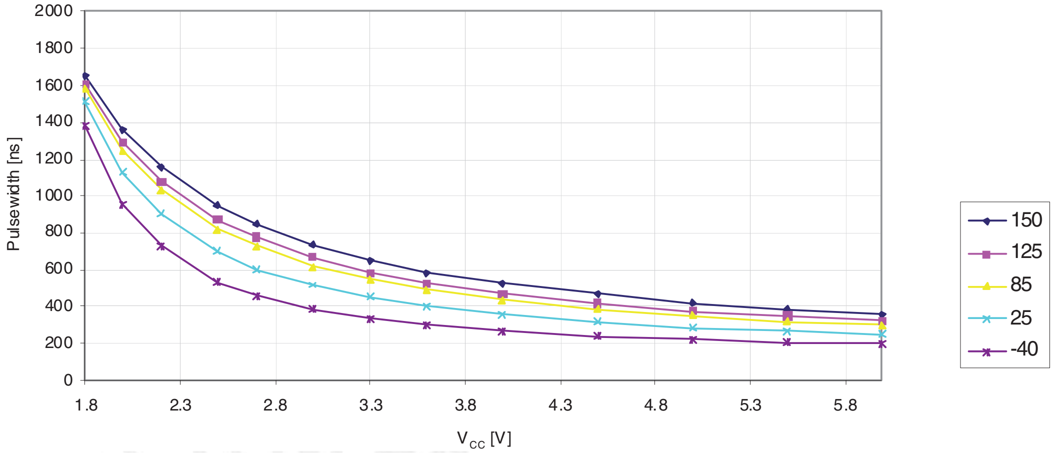


**Figure 30-77.** Calibrated 8MHz RC Oscillator Frequency versus OSCCAL Value



### 30.3.10 Reset Pulse width

**Figure 30-78.** Minimum Reset Pulse width versus $V_{CC}$



# 31. Register Summary

| Address | Name | Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 | Page |
|---|---|---|---|---|---|---|---|---|---|---|
| (0xFF) | Reserved | – | – | – | – | – | – | – | – | |
| (0xFE) | Reserved | – | – | – | – | – | – | – | – | |
| (0xFD) | Reserved | – | – | – | – | – | – | – | – | |
| (0xFC) | Reserved | – | – | – | – | – | – | – | – | |
| (0xFB) | Reserved | – | – | – | – | – | – | – | – | |
| (0xFA) | Reserved | – | – | – | – | – | – | – | – | |
| (0xF9) | Reserved | – | – | – | – | – | – | – | – | |
| (0xF8) | Reserved | – | – | – | – | – | – | – | – | |
| (0xF7) | Reserved | – | – | – | – | – | – | – | – | |
| (0xF6) | Reserved | – | – | – | – | – | – | – | – | |
| (0xF5) | Reserved | – | – | – | – | – | – | – | – | |
| (0xF4) | Reserved | – | – | – | – | – | – | – | – | |
| (0xF3) | Reserved | – | – | – | – | – | – | – | – | |
| (0xF2) | Reserved | – | – | – | – | – | – | – | – | |
| (0xF1) | Reserved | – | – | – | – | – | – | – | – | |
| (0xF0) | Reserved | – | – | – | – | – | – | – | – | |
| (0xEF) | Reserved | – | – | – | – | – | – | – | – | |
| (0xEE) | Reserved | – | – | – | – | – | – | – | – | |
| (0xED) | Reserved | – | – | – | – | – | – | – | – | |
| (0xEC) | Reserved | – | – | – | – | – | – | – | – | |
| (0xEB) | Reserved | – | – | – | – | – | – | – | – | |
| (0xEA) | Reserved | – | – | – | – | – | – | – | – | |
| (0xE9) | Reserved | – | – | – | – | – | – | – | – | |
| (0xE8) | Reserved | – | – | – | – | – | – | – | – | |
| (0xE7) | Reserved | – | – | – | – | – | – | – | – | |
| (0xE6) | Reserved | – | – | – | – | – | – | – | – | |
| (0xE5) | Reserved | – | – | – | – | – | – | – | – | |
| (0xE4) | Reserved | – | – | – | – | – | – | – | – | |
| (0xE3) | Reserved | – | – | – | – | – | – | – | – | |
| (0xE2) | Reserved | – | – | – | – | – | – | – | – | |
| (0xE1) | Reserved | – | – | – | – | – | – | – | – | |
| (0xE0) | Reserved | – | – | – | – | – | – | – | – | |
| (0xDF) | Reserved | – | – | – | – | – | – | – | – | |
| (0xDE) | Reserved | – | – | – | – | – | – | – | – | |
| (0xDD) | Reserved | – | – | – | – | – | – | – | – | |
| (0xDC) | Reserved | – | – | – | – | – | – | – | – | |
| (0xDB) | Reserved | – | – | – | – | – | – | – | – | |
| (0xDA) | Reserved | – | – | – | – | – | – | – | – | |
| (0xD9) | Reserved | – | – | – | – | – | – | – | – | |
| (0xD8) | Reserved | – | – | – | – | – | – | – | – | |
| (0xD7) | Reserved | – | – | – | – | – | – | – | – | |
| (0xD6) | Reserved | – | – | – | – | – | – | – | – | |
| (0xD5) | Reserved | – | – | – | – | – | – | – | – | |
| (0xD4) | Reserved | – | – | – | – | – | – | – | – | |
| (0xD3) | Reserved | – | – | – | – | – | – | – | – | |

Notes:
1. For compatibility with future devices, reserved bits should be written to zero if accessed. Reserved I/O memory addresses should never be written.
2. I/O Registers within the address range 0x00 - 0x1F are directly bit-accessible using the SBI and CBI instructions. In these registers, the value of single bits can be checked by using the SBIS and SBIC instructions.
3. Some of the Status Flags are cleared by writing a logical one to them. Note that, unlike most other AVRs, the CBI and SBI instructions will only operate on the specified bit, and can therefore be used on registers containing such Status Flags. The CBI and SBI instructions work with registers 0x00 to 0x1F only.
4. When using the I/O specific commands IN and OUT, the I/O addresses 0x00 - 0x3F must be used. When addressing I/O Registers as data space using LD and ST instructions, 0x20 must be added to these addresses. The Atmel ATmega48PA/88PA/168PA is a complex microcontroller with more peripheral units than can be supported within the 64 location reserved in Opcode for the IN and OUT instructions. For the Extended I/O space from 0x60 - 0xFF in SRAM, only the ST/STS/STD and LD/LDS/LDD instructions can be used.
5. Only valid for the Atmel ATmega48PA/88PA/168PA.
6. BODS and BODSE only available for picoPower devices ATmega48PA/88PA/168PA

## 31. Register Summary (Continued)

| Address | Name | Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 | Page |
|---|---|---|---|---|---|---|---|---|---|---|
| (0xD2) | Reserved | – | – | – | – | – | – | – | – | |
| (0xD1) | Reserved | – | – | – | – | – | – | – | – | |
| (0xD0) | Reserved | – | – | – | – | – | – | – | – | |
| (0xCF) | Reserved | – | – | – | – | – | – | – | – | |
| (0xCE) | Reserved | – | – | – | – | – | – | – | – | |
| (0xCD) | Reserved | – | – | – | – | – | – | – | – | |
| (0xCC) | Reserved | – | – | – | – | – | – | – | – | |
| (0xCB) | Reserved | – | – | – | – | – | – | – | – | |
| (0xCA) | Reserved | – | – | – | – | – | – | – | – | |
| (0xC9) | Reserved | – | – | – | – | – | – | – | – | |
| (0xC8) | Reserved | – | – | – | – | – | – | – | – | |
| (0xC7) | Reserved | – | – | – | – | – | – | – | – | |
| (0xC6) | UDR0 | USART I/O Data Register | | | | | | | | 196 |
| (0xC5) | UBRR0H | | | | | USART Baud Rate Register High | | | | 201 |
| (0xC4) | UBRR0L | USART Baud Rate Register Low | | | | | | | | 201 |
| (0xC3) | Reserved | – | – | – | – | – | – | – | – | |
| (0xC2) | UCSR0C | UMSEL01 | UMSEL00 | UPM01 | UPM00 | USBS0 | UCSZ01 /UDORD0 | UCSZ00 / UCPHA0 | UCPOL0 | 199/210 |
| (0xC1) | UCSR0B | RXCIE0 | TXCIE0 | UDRIE0 | RXEN0 | TXEN0 | UCSZ02 | RXB80 | TXB80 | 198 |
| (0xC0) | UCSR0A | RXC0 | TXC0 | UDRE0 | FE0 | DOR0 | UPE0 | U2X0 | MPCM0 | 197 |
| (0xBF) | Reserved | – | – | – | – | – | – | – | – | |
| (0xBE) | Reserved | – | – | – | – | – | – | – | – | |
| (0xBD) | TWAMR | TWAM6 | TWAM5 | TWAM4 | TWAM3 | TWAM2 | TWAM1 | TWAM0 | – | 243 |
| (0xBC) | TWCR | TWINT | TWEA | TWSTA | TWSTO | TWWC | TWEN | – | TWIE | 240 |
| (0xBB) | TWDR | 2-wire Serial Interface Data Register | | | | | | | | 242 |
| (0xBA) | TWAR | TWA6 | TWA5 | TWA4 | TWA3 | TWA2 | TWA1 | TWA0 | TWGCE | 243 |
| (0xB9) | TWSR | TWS7 | TWS6 | TWS5 | TWS4 | TWS3 | – | TWPS1 | TWPS0 | 241 |
| (0xB8) | TWBR | 2-wire Serial Interface Bit Rate Register | | | | | | | | 240 |
| (0xB7) | Reserved | – | | – | – | – | – | – | – | |
| (0xB6) | ASSR | – | EXCLK | AS2 | TCN2UB | OCR2AUB | OCR2BUB | TCR2AUB | TCR2BUB | 164 |
| (0xB5) | Reserved | – | – | – | – | – | – | – | – | |
| (0xB4) | OCR2B | Timer/Counter2 Output Compare Register B | | | | | | | | 162 |
| (0xB3) | OCR2A | Timer/Counter2 Output Compare Register A | | | | | | | | 162 |
| (0xB2) | TCNT2 | Timer/Counter2 (8-bit) | | | | | | | | 162 |
| (0xB1) | TCCR2B | FOC2A | FOC2B | – | – | WGM22 | CS22 | CS21 | CS20 | 161 |
| (0xB0) | TCCR2A | COM2A1 | COM2A0 | COM2B1 | COM2B0 | – | – | WGM21 | WGM20 | 158 |
| (0xAF) | Reserved | – | – | – | – | – | – | – | – | |
| (0xAE) | Reserved | – | – | – | – | – | – | – | – | |
| (0xAD) | Reserved | – | – | – | – | – | – | – | – | |
| (0xAC) | Reserved | – | – | – | – | – | – | – | – | |
| (0xAB) | Reserved | – | – | – | – | – | – | – | – | |
| (0xAA) | Reserved | – | – | – | – | – | – | – | – | |
| (0xA9) | Reserved | – | – | – | – | – | – | – | – | |
| (0xA8) | Reserved | – | – | – | – | – | – | – | – | |
| (0xA7) | Reserved | – | – | – | – | – | – | – | – | |
| (0xA6) | Reserved | – | – | – | – | – | – | – | – | |
| (0xA5) | Reserved | – | – | – | – | – | – | – | – | |

Notes:

1. For compatibility with future devices, reserved bits should be written to zero if accessed. Reserved I/O memory addresses should never be written.
2. I/O Registers within the address range 0x00 - 0x1F are directly bit-accessible using the SBI and CBI instructions. In these registers, the value of single bits can be checked by using the SBIS and SBIC instructions.
3. Some of the Status Flags are cleared by writing a logical one to them. Note that, unlike most other AVRs, the CBI and SBI instructions will only operate on the specified bit, and can therefore be used on registers containing such Status Flags. The CBI and SBI instructions work with registers 0x00 to 0x1F only.
4. When using the I/O specific commands IN and OUT, the I/O addresses 0x00 - 0x3F must be used. When addressing I/O Registers as data space using LD and ST instructions, 0x20 must be added to these addresses. The Atmel ATmega48PA/88PA/168PA is a complex microcontroller with more peripheral units than can be supported within the 64 location reserved in Opcode for the IN and OUT instructions. For the Extended I/O space from 0x60 - 0xFF in SRAM, only the ST/STS/STD and LD/LDS/LDD instructions can be used.
5. Only valid for the Atmel ATmega48PA/88PA/168PA.
6. BODS and BODSE only available for picoPower devices ATmega48PA/88PA/168PA

# 31. Register Summary (Continued)

| Address | Name | Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 | Page |
|---|---|---|---|---|---|---|---|---|---|---|
| (0xA4) | Reserved | – | – | – | – | – | – | – | – | |
| (0xA3) | Reserved | – | – | – | – | – | – | – | – | |
| (0xA2) | Reserved | – | – | – | – | – | – | – | – | |
| (0xA1) | Reserved | – | – | – | – | – | – | – | – | |
| (0xA0) | Reserved | – | – | – | – | – | – | – | – | |
| (0x9F) | Reserved | – | – | – | – | – | – | – | – | |
| (0x9E) | Reserved | – | – | – | – | – | – | – | – | |
| (0x9D) | Reserved | – | – | – | – | – | – | – | – | |
| (0x9C) | Reserved | – | – | – | – | – | – | – | – | |
| (0x9B) | Reserved | – | – | – | – | – | – | – | – | |
| (0x9A) | Reserved | – | – | – | – | – | – | – | – | |
| (0x99) | Reserved | – | – | – | – | – | – | – | – | |
| (0x98) | Reserved | – | – | – | – | – | – | – | – | |
| (0x97) | Reserved | – | – | – | – | – | – | – | – | |
| (0x96) | Reserved | – | – | – | – | – | – | – | – | |
| (0x95) | Reserved | – | – | – | – | – | – | – | – | |
| (0x94) | Reserved | – | – | – | – | – | – | – | – | |
| (0x93) | Reserved | – | – | – | – | – | – | – | – | |
| (0x92) | Reserved | – | – | – | – | – | – | – | – | |
| (0x91) | Reserved | – | – | – | – | – | – | – | – | |
| (0x90) | Reserved | – | – | – | – | – | – | – | – | |
| (0x8F) | Reserved | – | – | – | – | – | – | – | – | |
| (0x8E) | Reserved | – | – | – | – | – | – | – | – | |
| (0x8D) | Reserved | – | – | – | – | – | – | – | – | |
| (0x8C) | Reserved | – | – | – | – | – | – | – | – | |
| (0x8B) | OCR1BH | Timer/Counter1 - Output Compare Register B High Byte | | | | | | | | 137 |
| (0x8A) | OCR1BL | Timer/Counter1 - Output Compare Register B Low Byte | | | | | | | | 137 |
| (0x89) | OCR1AH | Timer/Counter1 - Output Compare Register A High Byte | | | | | | | | 137 |
| (0x88) | OCR1AL | Timer/Counter1 - Output Compare Register A Low Byte | | | | | | | | 137 |
| (0x87) | ICR1H | Timer/Counter1 - Input Capture Register High Byte | | | | | | | | 138 |
| (0x86) | ICR1L | Timer/Counter1 - Input Capture Register Low Byte | | | | | | | | 138 |
| (0x85) | TCNT1H | Timer/Counter1 - Counter Register High Byte | | | | | | | | 137 |
| (0x84) | TCNT1L | Timer/Counter1 - Counter Register Low Byte | | | | | | | | 137 |
| (0x83) | Reserved | – | – | – | – | – | – | – | – | |
| (0x82) | TCCR1C | FOC1A | FOC1B | – | – | – | – | – | – | 136 |
| (0x81) | TCCR1B | ICNC1 | ICES1 | – | WGM13 | WGM12 | CS12 | CS11 | CS10 | 135 |
| (0x80) | TCCR1A | COM1A1 | COM1A0 | COM1B1 | COM1B0 | – | – | WGM11 | WGM10 | 133 |
| (0x7F) | DIDR1 | – | – | – | – | – | – | AIN1D | AIN0D | 247 |
| (0x7E) | DIDR0 | – | – | ADC5D | ADC4D | ADC3D | ADC2D | ADC1D | ADC0D | 266 |
| (0x7D) | Reserved | – | – | – | – | – | – | – | – | |
| (0x7C) | ADMUX | REFS1 | REFS0 | ADLAR | – | MUX3 | MUX2 | MUX1 | MUX0 | 262 |
| (0x7B) | ADCSRB | – | ACME | – | – | – | ADTS2 | ADTS1 | ADTS0 | 265 |
| (0x7A) | ADCSRA | ADEN | ADSC | ADATE | ADIF | ADIE | ADPS2 | ADPS1 | ADPS0 | 263 |
| (0x79) | ADCH | ADC Data Register High byte | | | | | | | | 265 |
| (0x78) | ADCL | ADC Data Register Low byte | | | | | | | | 265 |
| (0x77) | Reserved | – | – | – | – | – | – | – | – | |

Notes:
1. For compatibility with future devices, reserved bits should be written to zero if accessed. Reserved I/O memory addresses should never be written.
2. I/O Registers within the address range 0x00 - 0x1F are directly bit-accessible using the SBI and CBI instructions. In these registers, the value of single bits can be checked by using the SBIS and SBIC instructions.
3. Some of the Status Flags are cleared by writing a logical one to them. Note that, unlike most other AVRs, the CBI and SBI instructions will only operate on the specified bit, and can therefore be used on registers containing such Status Flags. The CBI and SBI instructions work with registers 0x00 to 0x1F only.
4. When using the I/O specific commands IN and OUT, the I/O addresses 0x00 - 0x3F must be used. When addressing I/O Registers as data space using LD and ST instructions, 0x20 must be added to these addresses. The Atmel ATmega48PA/88PA/168PA is a complex microcontroller with more peripheral units than can be supported within the 64 location reserved in Opcode for the IN and OUT instructions. For the Extended I/O space from 0x60 - 0xFF in SRAM, only the ST/STS/STD and LD/LDS/LDD instructions can be used.
5. Only valid for the Atmel ATmega48PA/88PA/168PA.
6. BODS and BODSE only available for picoPower devices ATmega48PA/88PA/168PA

## 31. Register Summary (Continued)

| Address | Name | Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 | Page |
|---|---|---|---|---|---|---|---|---|---|---|
| (0x76) | Reserved | – | – | – | – | – | – | – | – | |
| (0x75) | Reserved | – | – | – | – | – | – | – | – | |
| (0x74) | Reserved | – | – | – | – | – | – | – | – | |
| (0x73) | Reserved | – | – | – | – | – | – | – | – | |
| (0x72) | Reserved | – | – | – | – | – | – | – | – | |
| (0x71) | Reserved | – | – | – | – | – | – | – | – | |
| (0x70) | TIMSK2 | – | – | – | – | – | OCIE2B | OCIE2A | TOIE2 | 163 |
| (0x6F) | TIMSK1 | – | – | ICIE1 | – | – | OCIE1B | OCIE1A | TOIE1 | 138 |
| (0x6E) | TIMSK0 | – | – | – | – | – | OCIE0B | OCIE0A | TOIE0 | 109 |
| (0x6D) | PCMSK2 | PCINT23 | PCINT22 | PCINT21 | PCINT20 | PCINT19 | PCINT18 | PCINT17 | PCINT16 | 72 |
| (0x6C) | PCMSK1 | – | PCINT14 | PCINT13 | PCINT12 | PCINT11 | PCINT10 | PCINT9 | PCINT8 | 72 |
| (0x6B) | PCMSK0 | PCINT7 | PCINT6 | PCINT5 | PCINT4 | PCINT3 | PCINT2 | PCINT1 | PCINT0 | 72 |
| (0x6A) | Reserved | – | – | – | – | – | – | – | – | |
| (0x69) | EICRA | – | – | – | – | ISC11 | ISC10 | ISC01 | ISC00 | 69 |
| (0x68) | PCICR | – | – | – | – | – | PCIE2 | PCIE1 | PCIE0 | |
| (0x67) | Reserved | – | – | – | – | – | – | – | – | |
| (0x66) | OSCCAL | Oscillator Calibration Register | | | | | | | | 37 |
| (0x65) | Reserved | – | – | – | – | – | – | – | – | |
| (0x64) | PRR | PRTWI | PRTIM2 | PRTIM0 | – | PRTIM1 | PRSPI | PRUSART0 | PRADC | 42 |
| (0x63) | Reserved | – | – | – | – | – | – | – | – | |
| (0x62) | Reserved | – | – | – | – | – | – | – | – | |
| (0x61) | CLKPR | CLKPCE | – | – | – | CLKPS3 | CLKPS2 | CLKPS1 | CLKPS0 | 37 |
| (0x60) | WDTCSR | WDIF | WDIE | WDP3 | WDCE | WDE | WDP2 | WDP1 | WDP0 | 56 |
| 0x3F (0x5F) | SREG | I | T | H | S | V | N | Z | C | 10 |
| 0x3E (0x5E) | SPH | – | – | – | – | – | (SP10) [5.] | SP9 | SP8 | 12 |
| 0x3D (0x5D) | SPL | SP7 | SP6 | SP5 | SP4 | SP3 | SP2 | SP1 | SP0 | 12 |
| 0x3C (0x5C) | Reserved | – | – | – | – | – | – | – | – | |
| 0x3B (0x5B) | Reserved | – | – | – | – | – | – | – | – | |
| 0x3A (0x5A) | Reserved | – | – | – | – | – | – | – | – | |
| 0x39 (0x59) | Reserved | – | – | – | – | – | – | – | – | |
| 0x38 (0x58) | Reserved | – | – | – | – | – | – | – | – | |
| 0x37 (0x57) | SPMCSR | SPMIE | (RWWSB)[5.] | – | (RWWSRE)[5.] | BLBSET | PGWRT | PGERS | SELFPRGEN | 292 |
| 0x36 (0x56) | Reserved | – | – | – | – | – | – | – | – | |
| 0x35 (0x55) | MCUCR | – | BODS[6] | BODSE[6] | PUD | – | – | IVSEL | IVCE | 45/66/91 |
| 0x34 (0x54) | MCUSR | – | – | – | – | WDRF | BORF | EXTRF | PORF | 55 |
| 0x33 (0x53) | SMCR | – | – | – | – | SM2 | SM1 | SM0 | SE | 40 |
| 0x32 (0x52) | Reserved | – | – | – | – | – | – | – | – | |
| 0x31 (0x51) | Reserved | – | – | – | – | – | – | – | – | |
| 0x30 (0x50) | ACSR | ACD | ACBG | ACO | ACI | ACIE | ACIC | ACIS1 | ACIS0 | 245 |
| 0x2F (0x4F) | Reserved | – | – | – | – | – | – | – | – | |
| 0x2E (0x4E) | SPDR | SPI Data Register | | | | | | | | 175 |
| 0x2D (0x4D) | SPSR | SPIF | WCOL | – | – | – | – | – | SPI2X | 174 |
| 0x2C (0x4C) | SPCR | SPIE | SPE | DORD | MSTR | CPOL | CPHA | SPR1 | SPR0 | 173 |
| 0x2B (0x4B) | GPIOR2 | General Purpose I/O Register 2 | | | | | | | | 25 |
| 0x2A (0x4A) | GPIOR1 | General Purpose I/O Register 1 | | | | | | | | 25 |
| 0x29 (0x49) | Reserved | – | – | – | – | – | – | – | – | |

Notes:

1. For compatibility with future devices, reserved bits should be written to zero if accessed. Reserved I/O memory addresses should never be written.
2. I/O Registers within the address range 0x00 - 0x1F are directly bit-accessible using the SBI and CBI instructions. In these registers, the value of single bits can be checked by using the SBIS and SBIC instructions.
3. Some of the Status Flags are cleared by writing a logical one to them. Note that, unlike most other AVRs, the CBI and SBI instructions will only operate on the specified bit, and can therefore be used on registers containing such Status Flags. The CBI and SBI instructions work with registers 0x00 to 0x1F only.
4. When using the I/O specific commands IN and OUT, the I/O addresses 0x00 - 0x3F must be used. When addressing I/O Registers as data space using LD and ST instructions, 0x20 must be added to these addresses. The Atmel ATmega48PA/88PA/168PA is a complex microcontroller with more peripheral units than can be supported within the 64 location reserved in Opcode for the IN and OUT instructions. For the Extended I/O space from 0x60 - 0xFF in SRAM, only the ST/STS/STD and LD/LDS/LDD instructions can be used.
5. Only valid for the Atmel ATmega48PA/88PA/168PA.
6. BODS and BODSE only available for picoPower devices ATmega48PA/88PA/168PA

## 31. Register Summary (Continued)

| Address | Name | Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 | Page |
|---|---|---|---|---|---|---|---|---|---|---|
| 0x28 (0x48) | OCR0B | Timer/Counter0 Output Compare Register B | | | | | | | | |
| 0x27 (0x47) | OCR0A | Timer/Counter0 Output Compare Register A | | | | | | | | |
| 0x26 (0x46) | TCNT0 | Timer/Counter0 (8-bit) | | | | | | | | |
| 0x25 (0x45) | TCCR0B | FOC0A | FOC0B | – | – | WGM02 | CS02 | CS01 | CS00 | |
| 0x24 (0x44) | TCCR0A | COM0A1 | COM0A0 | COM0B1 | COM0B0 | – | – | WGM01 | WGM00 | |
| 0x23 (0x43) | GTCCR | TSM | – | – | – | – | – | PSRASY | PSRSYNC | 142/165 |
| 0x22 (0x42) | EEARH | (EEPROM Address Register High Byte) [5.] | | | | | | | | 21 |
| 0x21 (0x41) | EEARL | EEPROM Address Register Low Byte | | | | | | | | 21 |
| 0x20 (0x40) | EEDR | EEPROM Data Register | | | | | | | | 21 |
| 0x1F (0x3F) | EECR | – | – | EEPM1 | EEPM0 | EERIE | EEMPE | EEPE | EERE | 21 |
| 0x1E (0x3E) | GPIOR0 | General Purpose I/O Register 0 | | | | | | | | 25 |
| 0x1D (0x3D) | EIMSK | – | – | – | – | – | – | INT1 | INT0 | 70 |
| 0x1C (0x3C) | EIFR | – | – | – | – | – | – | INTF1 | INTF0 | 70 |
| 0x1B (0x3B) | PCIFR | – | – | – | – | – | PCIF2 | PCIF1 | PCIF0 | |
| 0x1A (0x3A) | Reserved | – | – | – | – | – | – | – | – | |
| 0x19 (0x39) | Reserved | – | – | – | – | – | – | – | – | |
| 0x18 (0x38) | Reserved | – | – | – | – | – | – | – | – | |
| 0x17 (0x37) | TIFR2 | – | – | – | – | – | OCF2B | OCF2A | TOV2 | 163 |
| 0x16 (0x36) | TIFR1 | – | – | ICF1 | – | – | OCF1B | OCF1A | TOV1 | 139 |
| 0x15 (0x35) | TIFR0 | – | – | – | – | – | OCF0B | OCF0A | TOV0 | |
| 0x14 (0x34) | Reserved | – | – | – | – | – | – | – | – | |
| 0x13 (0x33) | Reserved | – | – | – | – | – | – | – | – | |
| 0x12 (0x32) | Reserved | – | – | – | – | – | – | – | – | |
| 0x11 (0x31) | Reserved | – | – | – | – | – | – | – | – | |
| 0x10 (0x30) | Reserved | – | – | – | – | – | – | – | – | |
| 0x0F (0x2F) | Reserved | – | – | – | – | – | – | – | – | |
| 0x0E (0x2E) | Reserved | – | – | – | – | – | – | – | – | |
| 0x0D (0x2D) | Reserved | – | – | – | – | – | – | – | – | |
| 0x0C (0x2C) | Reserved | – | – | – | – | – | – | – | – | |
| 0x0B (0x2B) | PORTD | PORTD7 | PORTD6 | PORTD5 | PORTD4 | PORTD3 | PORTD2 | PORTD1 | PORTD0 | 92 |
| 0x0A (0x2A) | DDRD | DDD7 | DDD6 | DDD5 | DDD4 | DDD3 | DDD2 | DDD1 | DDD0 | 92 |
| 0x09 (0x29) | PIND | PIND7 | PIND6 | PIND5 | PIND4 | PIND3 | PIND2 | PIND1 | PIND0 | 92 |
| 0x08 (0x28) | PORTC | – | PORTC6 | PORTC5 | PORTC4 | PORTC3 | PORTC2 | PORTC1 | PORTC0 | 91 |
| 0x07 (0x27) | DDRC | – | DDC6 | DDC5 | DDC4 | DDC3 | DDC2 | DDC1 | DDC0 | 91 |
| 0x06 (0x26) | PINC | – | PINC6 | PINC5 | PINC4 | PINC3 | PINC2 | PINC1 | PINC0 | 91 |
| 0x05 (0x25) | PORTB | PORTB7 | PORTB6 | PORTB5 | PORTB4 | PORTB3 | PORTB2 | PORTB1 | PORTB0 | 91 |
| 0x04 (0x24) | DDRB | DDB7 | DDB6 | DDB5 | DDB4 | DDB3 | DDB2 | DDB1 | DDB0 | 91 |
| 0x03 (0x23) | PINB | PINB7 | PINB6 | PINB5 | PINB4 | PINB3 | PINB2 | PINB1 | PINB0 | 91 |
| 0x02 (0x22) | Reserved | – | – | – | – | – | – | – | – | |
| 0x01 (0x21) | Reserved | – | – | – | – | – | – | – | – | |
| 0x0 (0x20) | Reserved | – | – | – | – | – | – | – | – | |

Notes:
1. For compatibility with future devices, reserved bits should be written to zero if accessed. Reserved I/O memory addresses should never be written.
2. I/O Registers within the address range 0x00 - 0x1F are directly bit-accessible using the SBI and CBI instructions. In these registers, the value of single bits can be checked by using the SBIS and SBIC instructions.
3. Some of the Status Flags are cleared by writing a logical one to them. Note that, unlike most other AVRs, the CBI and SBI instructions will only operate on the specified bit, and can therefore be used on registers containing such Status Flags. The CBI and SBI instructions work with registers 0x00 to 0x1F only.
4. When using the I/O specific commands IN and OUT, the I/O addresses 0x00 - 0x3F must be used. When addressing I/O Registers as data space using LD and ST instructions, 0x20 must be added to these addresses. The Atmel ATmega48PA/88PA/168PA is a complex microcontroller with more peripheral units than can be supported within the 64 location reserved in Opcode for the IN and OUT instructions. For the Extended I/O space from 0x60 - 0xFF in SRAM, only the ST/STS/STD and LD/LDS/LDD instructions can be used.
5. Only valid for the Atmel ATmega48PA/88PA/168PA.
6. BODS and BODSE only available for picoPower devices ATmega48PA/88PA/168PA

## 32. Instruction Set Summary

| Mnemonics | Operands | Description | Operation | Flags | #Clocks |
|---|---|---|---|---|---|
| **ARITHMETIC AND LOGIC INSTRUCTIONS** | | | | | |
| ADD | Rd, Rr | Add two Registers | Rd ←Rd + Rr | Z,C,N,V,H | 1 |
| ADC | Rd, Rr | Add with Carry two Registers | Rd ←Rd + Rr + C | Z,C,N,V,H | 1 |
| ADIW | Rdl,K | Add Immediate to Word | Rdh:Rdl ←Rdh:Rdl + K | Z,C,N,V,S | 2 |
| SUB | Rd, Rr | Subtract two Registers | Rd ←Rd - Rr | Z,C,N,V,H | 1 |
| SUBI | Rd, K | Subtract Constant from Register | Rd ←Rd - K | Z,C,N,V,H | 1 |
| SBC | Rd, Rr | Subtract with Carry two Registers | Rd ←Rd - Rr - C | Z,C,N,V,H | 1 |
| SBCI | Rd, K | Subtract with Carry Constant from Reg. | Rd ←Rd - K - C | Z,C,N,V,H | 1 |
| SBIW | Rdl,K | Subtract Immediate from Word | Rdh:Rdl ←Rdh:Rdl - K | Z,C,N,V,S | 2 |
| AND | Rd, Rr | Logical AND Registers | Rd ←Rd • Rr | Z,N,V | 1 |
| ANDI | Rd, K | Logical AND Register and Constant | Rd ←Rd • K | Z,N,V | 1 |
| OR | Rd, Rr | Logical OR Registers | Rd ←Rd v Rr | Z,N,V | 1 |
| ORI | Rd, K | Logical OR Register and Constant | Rd ←Rd v K | Z,N,V | 1 |
| EOR | Rd, Rr | Exclusive OR Registers | Rd ←Rd ⊕ Rr | Z,N,V | 1 |
| COM | Rd | One’s Complement | Rd ←0xFF −Rd | Z,C,N,V | 1 |
| NEG | Rd | Two’s Complement | Rd ←0x00 −Rd | Z,C,N,V,H | 1 |
| SBR | Rd,K | Set Bit(s) in Register | Rd ←Rd v K | Z,N,V | 1 |
| CBR | Rd,K | Clear Bit(s) in Register | Rd ←Rd • (0xFF - K) | Z,N,V | 1 |
| INC | Rd | Increment | Rd ←Rd + 1 | Z,N,V | 1 |
| DEC | Rd | Decrement | Rd ←Rd −1 | Z,N,V | 1 |
| TST | Rd | Test for Zero or Minus | Rd ←Rd • Rd | Z,N,V | 1 |
| CLR | Rd | Clear Register | Rd ←Rd ⊕ Rd | Z,N,V | 1 |
| SER | Rd | Set Register | Rd ←0xFF | None | 1 |
| MUL | Rd, Rr | Multiply Unsigned | R1:R0 ←Rd x Rr | Z,C | 2 |
| MULS | Rd, Rr | Multiply Signed | R1:R0 ←Rd x Rr | Z,C | 2 |
| MULSU | Rd, Rr | Multiply Signed with Unsigned | R1:R0 ←Rd x Rr | Z,C | 2 |
| FMUL | Rd, Rr | Fractional Multiply Unsigned | R1:R0 ←(Rd x Rr) << 1 | Z,C | 2 |
| FMULS | Rd, Rr | Fractional Multiply Signed | R1:R0 ←(Rd x Rr) << 1 | Z,C | 2 |
| FMULSU | Rd, Rr | Fractional Multiply Signed with Unsigned | R1:R0 ←(Rd x Rr) << 1 | Z,C | 2 |
| **BRANCH INSTRUCTIONS** | | | | | |
| RJMP | k | Relative Jump | PC ←PC + k + 1 | None | 2 |
| IJMP | | Indirect Jump to (Z) | PC ←Z | None | 2 |
| JMP[1] | k | Direct Jump | PC ←k | None | 3 |
| RCALL | k | Relative Subroutine Call | PC ←PC + k + 1 | None | 3 |
| ICALL | | Indirect Call to (Z) | PC ←Z | None | 3 |
| CALL[1] | k | Direct Subroutine Call | PC ←k | None | 4 |
| RET | | Subroutine Return | PC ←STACK | None | 4 |
| RETI | | Interrupt Return | PC ←STACK | I | 4 |
| CPSE | Rd,Rr | Compare, Skip if Equal | if (Rd = Rr) PC ←PC + 2 or 3 | None | 1/2/3 |
| CP | Rd,Rr | Compare | Rd −Rr | Z, N,V,C,H | 1 |
| CPC | Rd,Rr | Compare with Carry | Rd −Rr −C | Z, N,V,C,H | 1 |
| CPI | Rd,K | Compare Register with Immediate | Rd −K | Z, N,V,C,H | 1 |
| SBRC | Rr, b | Skip if Bit in Register Cleared | if (Rr(b)=0) PC ←PC + 2 or 3 | None | 1/2/3 |
| SBRS | Rr, b | Skip if Bit in Register is Set | if (Rr(b)=1) PC ←PC + 2 or 3 | None | 1/2/3 |
| SBIC | P, b | Skip if Bit in I/O Register Cleared | if (P(b)=0) PC ←PC + 2 or 3 | None | 1/2/3 |
| SBIS | P, b | Skip if Bit in I/O Register is Set | if (P(b)=1) PC ←PC + 2 or 3 | None | 1/2/3 |
| BRBS | s, k | Branch if Status Flag Set | if (SREG(s) = 1) then PC←PC+k + 1 | None | 1/2 |
| BRBC | s, k | Branch if Status Flag Cleared | if (SREG(s) = 0) then PC←PC+k + 1 | None | 1/2 |
| BREQ | k | Branch if Equal | if (Z = 1) then PC ←PC + k + 1 | None | 1/2 |
| BRNE | k | Branch if Not Equal | if (Z = 0) then PC ←PC + k + 1 | None | 1/2 |
| BRCS | k | Branch if Carry Set | if (C = 1) then PC ←PC + k + 1 | None | 1/2 |
| BRCC | k | Branch if Carry Cleared | if (C = 0) then PC ←PC + k + 1 | None | 1/2 |
| BRSH | k | Branch if Same or Higher | if (C = 0) then PC ←PC + k + 1 | None | 1/2 |
| BRLO | k | Branch if Lower | if (C = 1) then PC ←PC + k + 1 | None | 1/2 |
| BRMI | k | Branch if Minus | if (N = 1) then PC ←PC + k + 1 | None | 1/2 |
| BRPL | k | Branch if Plus | if (N = 0) then PC ←PC + k + 1 | None | 1/2 |
| BRGE | k | Branch if Greater or Equal, Signed | if (N ⊕ V= 0) then PC ←PC + k + 1 | None | 1/2 |
| BRLT | k | Branch if Less Than Zero, Signed | if (N ⊕ V= 1) then PC ←PC + k + 1 | None | 1/2 |
| BRHS | k | Branch if Half Carry Flag Set | if (H = 1) then PC ←PC + k + 1 | None | 1/2 |
| BRHC | k | Branch if Half Carry Flag Cleared | if (H = 0) then PC ←PC + k + 1 | None | 1/2 |
| BRTS | k | Branch if T Flag Set | if (T = 1) then PC ←PC + k + 1 | None | 1/2 |
| BRTC | k | Branch if T Flag Cleared | if (T = 0) then PC ←PC + k + 1 | None | 1/2 |

Note: 1. These instructions are only available in the Atmel ATmega168PA.

## 32. Instruction Set Summary (Continued)

| Mnemonics | Operands | Description | Operation | Flags | #Clocks |
|---|---|---|---|---|---|
| BRVS | k | Branch if Overflow Flag is Set | if (V = 1) then PC ←PC + k + 1 | None | 1/2 |
| BRVC | k | Branch if Overflow Flag is Cleared | if (V = 0) then PC ←PC + k + 1 | None | 1/2 |
| BRIE | k | Branch if Interrupt Enabled | if ( I = 1) then PC ←PC + k + 1 | None | 1/2 |
| BRID | k | Branch if Interrupt Disabled | if ( I = 0) then PC ←PC + k + 1 | None | 1/2 |
| **BIT AND BIT-TEST INSTRUCTIONS** | | | | | |
| SBI | P,b | Set Bit in I/O Register | I/O(P,b) ←1 | None | 2 |
| CBI | P,b | Clear Bit in I/O Register | I/O(P,b) ←0 | None | 2 |
| LSL | Rd | Logical Shift Left | Rd(n+1) ←Rd(n), Rd(0) ←0 | Z,C,N,V | 1 |
| LSR | Rd | Logical Shift Right | Rd(n) ←Rd(n+1), Rd(7) ←0 | Z,C,N,V | 1 |
| ROL | Rd | Rotate Left Through Carry | Rd(0)←C,Rd(n+1)←Rd(n),C←Rd(7) | Z,C,N,V | 1 |
| ROR | Rd | Rotate Right Through Carry | Rd(7)←C,Rd(n)←Rd(n+1),C←Rd(0) | Z,C,N,V | 1 |
| ASR | Rd | Arithmetic Shift Right | Rd(n) ←Rd(n+1), n=0...6 | Z,C,N,V | 1 |
| SWAP | Rd | Swap Nibbles | Rd(3...0)←Rd(7...4),Rd(7...4)←Rd(3...0) | None | 1 |
| BSET | s | Flag Set | SREG(s) ←1 | SREG(s) | 1 |
| BCLR | s | Flag Clear | SREG(s) ←0 | SREG(s) | 1 |
| BST | Rr, b | Bit Store from Register to T | T ←Rr(b) | T | 1 |
| BLD | Rd, b | Bit load from T to Register | Rd(b) ←T | None | 1 |
| SEC | | Set Carry | C ←1 | C | 1 |
| CLC | | Clear Carry | C ←0 | C | 1 |
| SEN | | Set Negative Flag | N ←1 | N | 1 |
| CLN | | Clear Negative Flag | N ←0 | N | 1 |
| SEZ | | Set Zero Flag | Z ←1 | Z | 1 |
| CLZ | | Clear Zero Flag | Z ←0 | Z | 1 |
| SEI | | Global Interrupt Enable | I ←1 | I | 1 |
| CLI | | Global Interrupt Disable | I ←0 | I | 1 |
| SES | | Set Signed Test Flag | S ←1 | S | 1 |
| CLS | | Clear Signed Test Flag | S ←0 | S | 1 |
| SEV | | Set Twos Complement Overflow. | V ←1 | V | 1 |
| CLV | | Clear Twos Complement Overflow | V ←0 | V | 1 |
| SET | | Set T in SREG | T ←1 | T | 1 |
| CLT | | Clear T in SREG | T ←0 | T | 1 |
| SEH | | Set Half Carry Flag in SREG | H ←1 | H | 1 |
| CLH | | Clear Half Carry Flag in SREG | H ←0 | H | 1 |
| **DATA TRANSFER INSTRUCTIONS** | | | | | |
| MOV | Rd, Rr | Move Between Registers | Rd ←Rr | None | 1 |
| MOVW | Rd, Rr | Copy Register Word | Rd+1:Rd ←Rr+1:Rr | None | 1 |
| LDI | Rd, K | Load Immediate | Rd ←K | None | 1 |
| LD | Rd, X | Load Indirect | Rd ←(X) | None | 2 |
| LD | Rd, X+ | Load Indirect and Post-Inc. | Rd ←(X), X ←X + 1 | None | 2 |
| LD | Rd, - X | Load Indirect and Pre-Dec. | X ←X - 1, Rd ←(X) | None | 2 |
| LD | Rd, Y | Load Indirect | Rd ←(Y) | None | 2 |
| LD | Rd, Y+ | Load Indirect and Post-Inc. | Rd ←(Y), Y ←Y + 1 | None | 2 |
| LD | Rd, - Y | Load Indirect and Pre-Dec. | Y ←Y - 1, Rd ←(Y) | None | 2 |
| LDD | Rd,Y+q | Load Indirect with Displacement | Rd ←(Y + q) | None | 2 |
| LD | Rd, Z | Load Indirect | Rd ←(Z) | None | 2 |
| LD | Rd, Z+ | Load Indirect and Post-Inc. | Rd ←(Z), Z ←Z+1 | None | 2 |
| LD | Rd, -Z | Load Indirect and Pre-Dec. | Z ←Z - 1, Rd ←(Z) | None | 2 |
| LDD | Rd, Z+q | Load Indirect with Displacement | Rd ←(Z + q) | None | 2 |
| LDS | Rd, k | Load Direct from SRAM | Rd ←(k) | None | 2 |
| ST | X, Rr | Store Indirect | (X) ←Rr | None | 2 |
| ST | X+, Rr | Store Indirect and Post-Inc. | (X) ←Rr, X ←X + 1 | None | 2 |
| ST | - X, Rr | Store Indirect and Pre-Dec. | X ←X - 1, (X) ←Rr | None | 2 |
| ST | Y, Rr | Store Indirect | (Y) ←Rr | None | 2 |
| ST | Y+, Rr | Store Indirect and Post-Inc. | (Y) ←Rr, Y ←Y + 1 | None | 2 |
| ST | - Y, Rr | Store Indirect and Pre-Dec. | Y ←Y - 1, (Y) ←Rr | None | 2 |
| STD | Y+q,Rr | Store Indirect with Displacement | (Y + q) ←Rr | None | 2 |
| ST | Z, Rr | Store Indirect | (Z) ←Rr | None | 2 |
| ST | Z+, Rr | Store Indirect and Post-Inc. | (Z) ←Rr, Z ←Z + 1 | None | 2 |
| ST | -Z, Rr | Store Indirect and Pre-Dec. | Z ←Z - 1, (Z) ←Rr | None | 2 |
| STD | Z+q,Rr | Store Indirect with Displacement | (Z + q) ←Rr | None | 2 |
| STS | k, Rr | Store Direct to SRAM | (k) ←Rr | None | 2 |
| LPM | | Load Program Memory | R0 ←(Z) | None | 3 |
| LPM | Rd, Z | Load Program Memory | Rd ←(Z) | None | 3 |

Note: 1. These instructions are only available in the Atmel ATmega168PA.

## 32. Instruction Set Summary (Continued)

| Mnemonics | Operands | Description | Operation | Flags | #Clocks |
|---|---|---|---|---|---|
| LPM | Rd, Z+ | Load Program Memory and Post-Inc | Rd ←(Z), Z ←Z+1 | None | 3 |
| SPM | | Store Program Memory | (Z) ←R1:R0 | None | - |
| IN | Rd, P | In Port | Rd ←P | None | 1 |
| OUT | P, Rr | Out Port | P ←Rr | None | 1 |
| PUSH | Rr | Push Register on Stack | STACK ←Rr | None | 2 |
| POP | Rd | Pop Register from Stack | Rd ←STACK | None | 2 |
| **MCU CONTROL INSTRUCTIONS** | | | | | |
| NOP | | No Operation | | None | 1 |
| SLEEP | | Sleep | (see specific descr. for Sleep function) | None | 1 |
| WDR | | Watchdog Reset | (see specific descr. for WDR/timer) | None | 1 |
| BREAK | | Break | For On-chip Debug Only | None | N/A |

Note: 1. These instructions are only available in the Atmel ATmega168PA.

# 33. Ordering Information

## 33.1 ATmega48PA/88PA/168PA

| Speed (MHz) | Power Supply (V) | Ordering Code | Package[1] | Operational Range |
|---|---|---|---|---|
| 16[2] | 2.7 - 5.5 | ATmega48PA-15AZ | MA | Automotive<br>(-40°C to 125°C) |
| | | ATmega48PA-15MZ | PN | |
| | | ATmega88PA-15AZ | MA | |
| | | ATmega88PA-15MZ | PN | |
| | | ATmega168PA-15AZ | MA | |
| | | ATmega168PA-15MZ | PN | |

Notes: 1. Pb-free packaging complies to the European Directive for Restriction of Hazardous Substances (RoHS directive).Also Halide free and fully Green.

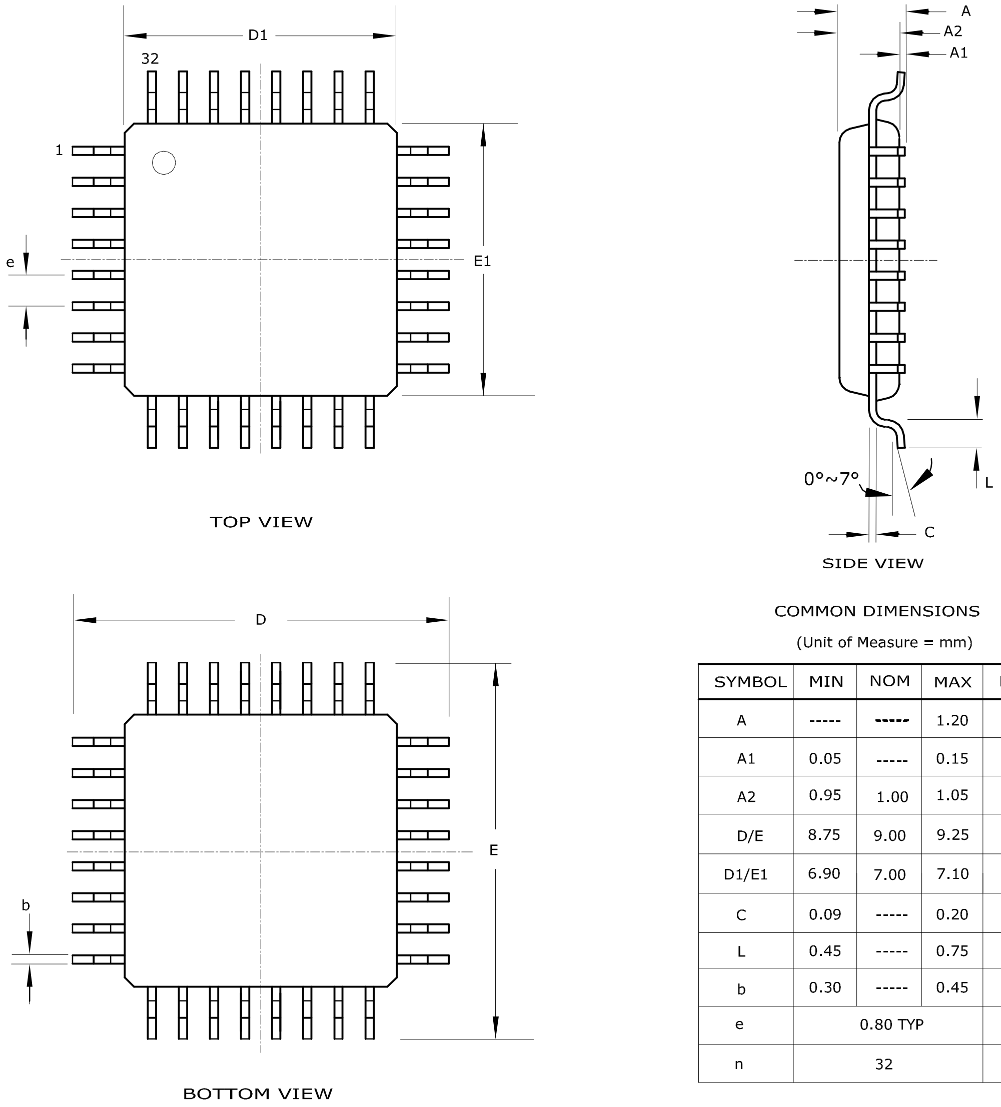
2. See "Speed Grades" on page 315.

| Package Type | |
|---|---|
| MA | MA, 32 - Lead, 7x7 mm Body Size, 1.0 mm Body Thickness 0.8 mm Lead Pitch, Thin Profile Plastic Quad Flat Package (TQFP) |
| PN | PN, 32-Lead, 5.0x5.0 mm Body, 0.50 mm, Quad Flat No Lead Package (QFN) |

# 34. Packaging Information

## 34.1 MA

DRAWINGS NOT SCALED



TOP VIEW

SIDE VIEW

BOTTOM VIEW

COMMON DIMENSIONS

(Unit of Measure = mm)

| SYMBOL | MIN | NOM | MAX | NOTE |
|---|---|---|---|---|
| A | ----- | ----- | 1.20 | |
| A1 | 0.05 | ----- | 0.15 | |
| A2 | 0.95 | 1.00 | 1.05 | |
| D/E | 8.75 | 9.00 | 9.25 | |
| D1/E1 | 6.90 | 7.00 | 7.10 | 2 |
| C | 0.09 | ----- | 0.20 | |
| L | 0.45 | ----- | 0.75 | |
| b | 0.30 | ----- | 0.45 | |
| e | 0.80 TYP | | | |
| n | 32 | | | |

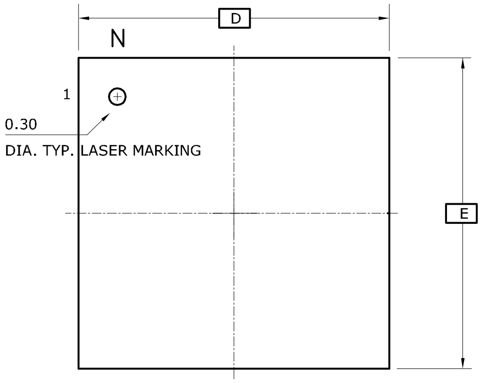
Notes :
1. This drawing is for general information only. Refer to JEDEC Drawing MS-026, Variation ABA.
2. Dimensions D1 and E1 do not include mold protrusion. Allowable protrusion is 0.25mm per side. Dimensions D1 and E1 are maximum plastic body size dimensions including mold mismatch.
3. Lead coplanarity is 0.10mm maximum.

02/29/2012

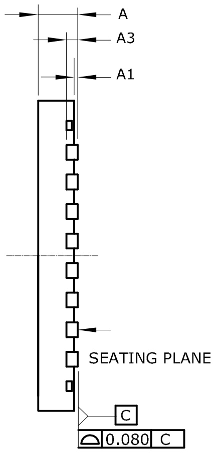
| | TITLE | GPC | DRAWING NO. | REV. |
|---|---|---|---|---|
| **Package Drawing Contact:**<br>**packagedrawings@atmel.com** | MA, 32 Lds - 0.80mm Pitch, 7x7x1.00mm Body size<br>Thin Profile Plastic Quad Flat Package (TQFP) | AUT | MA | C |

## 34.2 PN

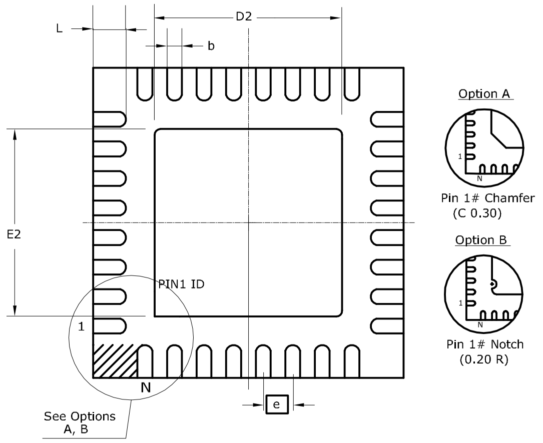
DRAWINGS NOT SCALED



TOP VIEW



SIDE VIEW



BOTTOM VIEW

COMMON DIMENSIONS

(Unit of Measure = mm)

| SYMBOL | MIN | NOM | MAX | NOTE |
|---|---|---|---|---|
| A | 0.80 | 0.85 | 0.90 | |
| A1 | 0.00 | ---- | 0.05 | |
| A3 | 0.20 REF | | | |
| D/E | 5.00 BSC | | | |
| D2/E2 | 3.00 | 3.10 | 3.20 | |
| L | 0.30 | 0.40 | 0.50 | |
| b | 0.18 | 0.25 | 0.30 | 2 |
| e | 0.50 BSC | | | |
| n | 32 | | | |

Notes :
1. This drawing is for general information only. Refer to JEDEC Drawing MO-220, Variation VHHD-2, for proper dimensions, tolerances, datums, etc.
2. Dimension b applies to metallized terminal and is measured between 0.15mm and 0.30mm from the terminal tip.
If the terminal has the optical radius on the other end of the terminal, the dimension should not be measured in that radius area.

01/31/2012

| | TITLE | GPC | DRAWING NO. | REV. |
|---|---|---|---|---|
| **Package Drawing Contact:**<br>**packagedrawings@atmel.com** | PN, 32 Leads , 0.50mm pitch, 5 x 5 mm<br>Very Thin quad Flat No Lead Package (VQFN) Sawn | ZMF | PN | I |

# 35. Errata

## 35.1 Errata ATmega48PA

The revision letter in this section refers to the revision of the ATmega48PA/88PA/168PA device.

### 35.1.1 Rev. D

- **Analog MUX can be turned off when setting ACME bit**

1. **Analog MUX can be turned off when setting ACME bit**
   If the ACME (Analog Comparator Multiplexer Enabled) bit in ADCSRB is set while MUX3 in ADMUX is '1' (ADMUX[3:0]=1xxx), all MUX'es are turned off until the ACME bit is cleared.

   **Problem Fix/Workaround**
   Clear the MUX3 bit before setting the ACME bit.

## 35.2 Errata Atmel ATmega88PA

The revision letter in this section refers to the revision of the Atmel ATmega88PA device.

### 35.2.1 Rev. F

- **Analog MUX can be turned off when setting ACME bit**

1. **Analog MUX can be turned off when setting ACME bit**
   If the ACME (Analog Comparator Multiplexer Enabled) bit in ADCSRB is set while MUX3 in ADMUX is '1' (ADMUX[3:0]=1xxx), all MUX'es are turned off until the ACME bit is cleared.

   **Problem Fix/Workaround**
   Clear the MUX3 bit before setting the ACME bit.

## 35.3 Errata Atmel ATmega168PA

The revision letter in this section refers to the revision of the Atmel ATmega168PA device.

### 35.3.1 Rev E

- **Analog MUX can be turned off when setting ACME bit**

1. **Analog MUX can be turned off when setting ACME bit**
   If the ACME (Analog Comparator Multiplexer Enabled) bit in ADCSRB is set while MUX3 in ADMUX is '1' (ADMUX[3:0]=1xxx), all MUX'es are turned off until the ACME bit is cleared.

   **Problem Fix/Workaround**
   Clear the MUX3 bit before setting the ACME bit.

# 36. Datasheet Revision History

Please note that the referring page numbers in this section are referred to this document. The referring revision in this section are referring to the document revision.

## 36.1 Rev. 9223D – 05/12

- Set datasheet from “Preliminary” to “Standard”

## 36.2 Rev. 9223C – 02/12

- Features on page 1 updated
- Figure 25-1 “The debugWIRE Setup” on page 267 updated
- Section 28.8 “Serial Downloading” on page 309 updated
- Section 29.2 “DC Characteristics” on pages 315 to 316 updated
- Section 29.3 “Speed Grades” on page 316 updated
- Section 29.4 “Clock Characteristics” on page 317 updated
- Section 29.5 “System and Reset Characteristics” on page 318 updated
- Section 29.8 “ADC Characteristics” on page 322 updated
- Section 30.1.5 “I/O Pin Output Voltage versus Sink Current ($V_{CC}$ = 1.8V)” on pages 329 to 330 updated
- Section 30.2.6 “Pin Driver Strength” on pages 340 to 341 updated
- Section 30.3.6 “Pin Driver Strength” on pages 350 to 351 updated
- Section 33 “Ordering Information” on page 365 updated

## 36.3 Rev. 9223B – 09/11

- ADC characteristics updated
- Temperature sensor updated

## 36.4 Rev. 9223A – 08/11

- Creation of the automotive version starting from industrial version based on the Atmel ATmega48PA/88PA/168PA datasheet 8271C-AVR-08/10. Temperature and voltage ranges reflecting Automotive requirements.

***Atmel Corporation***
2325 Orchard Parkway
San Jose, CA 95131
USA
**Tel:** (+1)(408) 441-0311
**Fax:** (+1)(408) 487-2600

***Atmel Asia Limited***
Unit 01-5 & 16, 19/F
BEA Tower, Millennium City 5
418 Kwun Tong Road
Kwun Tong, Kowloon
HONG KONG
**Tel:** (+852) 2245-6100
**Fax:** (+852) 2722-1369

***Atmel Munich GmbH***
Business Campus
Parkring 4
D-85748 Garching b. Munich
GERMANY
**Tel:** (+49) 89-31970-0
**Fax:** (+49) 89-3194621

***Atmel Japan***
9F, Tonetsu Shinkawa Bldg.
1-24-8 Shinkawa
Chuo-ku, Tokyo 104-0033
JAPAN
**Tel:** (+81) (3) 3523-3551
**Fax:** (+81) (3) 3523-7581

Disclaimer: The information in this document is provided in connection with Atmel products. No license, express or implied, by estoppel or otherwise, to any intellectual property right is granted by this document or in connection with the sale of Atmel products. EXCEPT AS SET FORTH IN THE ATMEL TERMS AND CONDITIONS OF SALES LOCATED ON THE ATMEL WEBSITE, ATMEL ASSUMES NO LIABILITY WHATSOEVER AND DISCLAIMS ANY EXPRESS, IMPLIED OR STATUTORY WARRANTY RELATING TO ITS PRODUCTS INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT. IN NO EVENT SHALL ATMEL BE LIABLE FOR ANY DIRECT, INDIRECT, CONSEQUENTIAL, PUNITIVE, SPECIAL OR INCIDENTAL DAMAGES (INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS AND PROFITS, BUSINESS INTERRUPTION, OR LOSS OF INFORMATION) ARISING OUT OF THE USE OR INABILITY TO USE THIS DOCUMENT, EVEN IF ATMEL HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. Atmel makes no representations or warranties with respect to the accuracy or completeness of the contents of this document and reserves the right to make changes to specifications and products descriptions at any time without notice. Atmel does not make any commitment to update the information contained herein. Unless specifically provided otherwise, Atmel products are not suitable for, and shall not be used in, automotive applications. Atmel products are not intended, authorized, or warranted for use as components in applications intended to support or sustain life.

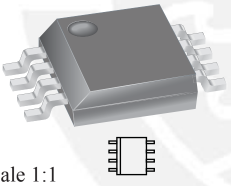
# *Fully Integrated, Hall Effect-Based Linear Current Sensor IC with 2.1 kVRMS Isolation and a Low-Resistance Current Conductor*

## Features and Benefits

- Low-noise analog signal path
- Device bandwidth is set via the new FILTER pin
- 5 µs output rise time in response to step input current
- 80 kHz bandwidth
- Total output error 1.5% at $T_A$= 25°C
- Small footprint, low-profile SOIC8 package
- 1.2 mΩ internal conductor resistance
- 2.1 kVRMS minimum isolation voltage from pins 1-4 to pins 5-8
- 5.0 V, single supply operation
- 66 to 185 mV/A output sensitivity
- Output voltage proportional to AC or DC currents
- Factory-trimmed for accuracy
- Extremely stable output offset voltage
- Nearly zero magnetic hysteresis
- Ratiometric output from supply voltage

TÜV America
Certificate Number:
U8V 06 05 54214 010

## Package: 8 Lead SOIC (suffix LC)



Approximate Scale 1:1

## Description

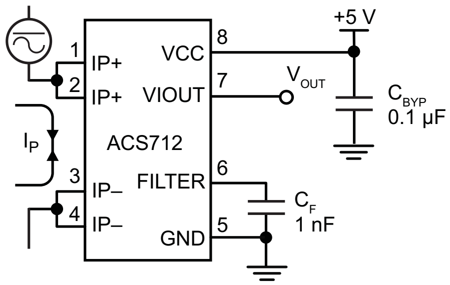
The Allegro® ACS712 provides economical and precise solutions for AC or DC current sensing in industrial, commercial, and communications systems. The device package allows for easy implementation by the customer. Typical applications include motor control, load detection and management, switch-mode power supplies, and overcurrent fault protection. The device is not intended for automotive applications.

The device consists of a precise, low-offset, linear Hall circuit with a copper conduction path located near the surface of the die. Applied current flowing through this copper conduction path generates a magnetic field which the Hall IC converts into a proportional voltage. Device accuracy is optimized through the close proximity of the magnetic signal to the Hall transducer. A precise, proportional voltage is provided by the low-offset, chopper-stabilized BiCMOS Hall IC, which is programmed for accuracy after packaging.

The output of the device has a positive slope (>$V_{IOUT(Q)}$) when an increasing current flows through the primary copper conduction path (from pins 1 and 2, to pins 3 and 4), which is the path used for current sampling. The internal resistance of this conductive path is 1.2 mΩ typical, providing low power loss. The thickness of the copper conductor allows survival of

*Continued on the next page…*

## Typical Application



Application 1. The ACS712 outputs an analog signal, $V_{OUT}$. that varies linearly with the uni- or bi-directional AC or DC primary sampled current, $I_P$, within the range specified. $C_F$ is recommended for noise management, with values that depend on the application.

### Description (continued)

the device at up to 5× overcurrent conditions. The terminals of the conductive path are electrically isolated from the signal leads (pins 5 through 8). This allows the ACS712 to be used in applications requiring electrical isolation without the use of opto-isolators or other costly isolation techniques.

The ACS712 is provided in a small, surface mount SOIC8 package. The leadframe is plated with 100% matte tin, which is compatible with standard lead (Pb) free printed circuit board assembly processes. Internally, the device is Pb-free, except for flip-chip high-temperature Pb-based solder balls, currently exempt from RoHS. The device is fully calibrated prior to shipment from the factory.

### Selection Guide

| Part Number | Packing* | $T_A$ (°C) | Optimized Range, $I_P$ (A) | Sensitivity, Sens (Typ) (mV/A) |
|---|---|---|---|---|
| ACS712ELCTR-05B-T | Tape and reel, 3000 pieces/reel | –40 to 85 | ±5 | 185 |
| ACS712ELCTR-20A-T | Tape and reel, 3000 pieces/reel | –40 to 85 | ±20 | 100 |
| ACS712ELCTR-30A-T | Tape and reel, 3000 pieces/reel | –40 to 85 | ±30 | 66 |

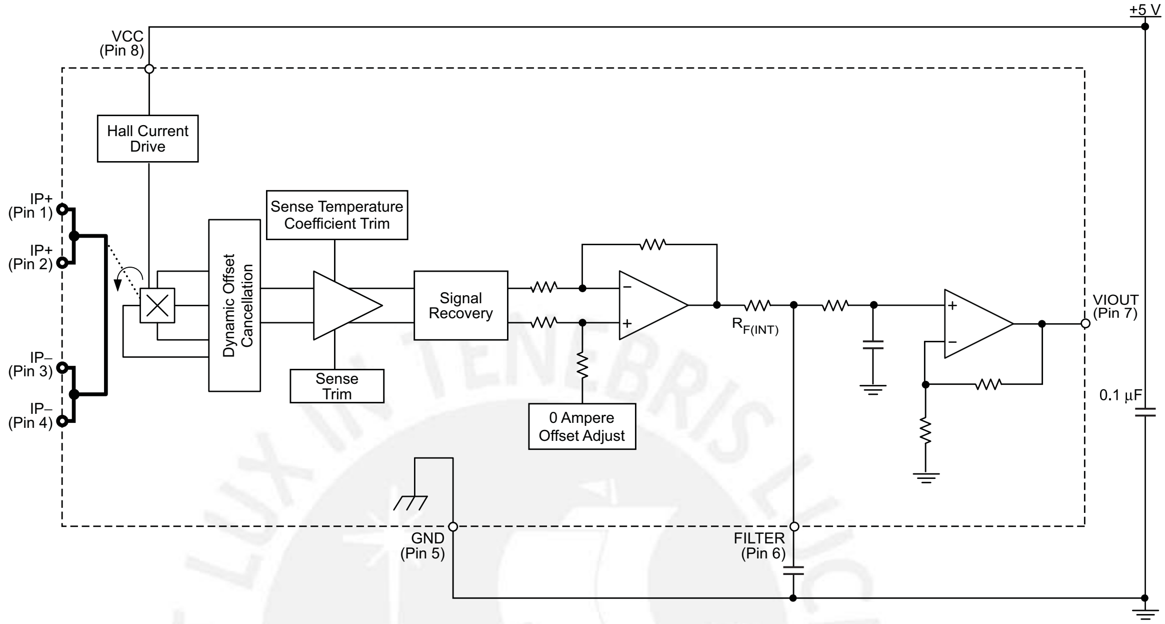
*Contact Allegro for additional packing options.

### Absolute Maximum Ratings

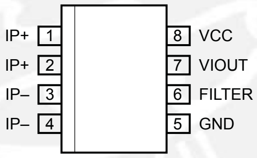
| Characteristic | Symbol | Notes | Rating | Units |
|---|---|---|---|---|
| Supply Voltage | $V_{CC}$ | | 8 | V |
| Reverse Supply Voltage | $V_{RCC}$ | | –0.1 | V |
| Output Voltage | $V_{IOUT}$ | | 8 | V |
| Reverse Output Voltage | $V_{RIOUT}$ | | –0.1 | V |
| Reinforced Isolation Voltage | $V_{ISO}$ | Pins 1-4 and 5-8; 60 Hz, 1 minute, $T_A$=25°C | 2100 | VAC |
| | | Maximum working voltage according to UL60950-1 | 184 | $V_{peak}$ |
| Basic Isolation Voltage | $V_{ISO(bsc)}$ | Pins 1-4 and 5-8; 60 Hz, 1 minute, $T_A$=25°C | 1500 | VAC |
| | | Maximum working voltage according to UL60950-1 | 354 | $V_{peak}$ |
| Output Current Source | $I_{IOUT(Source)}$ | | 3 | mA |
| Output Current Sink | $I_{IOUT(Sink)}$ | | 10 | mA |
| Overcurrent Transient Tolerance | $I_P$ | 1 pulse, 100 ms | 100 | A |
| Nominal Operating Ambient Temperature | $T_A$ | Range E | –40 to 85 | ºC |
| Maximum Junction Temperature | $T_J$(max) | | 165 | ºC |
| Storage Temperature | $T_{stg}$ | | –65 to 170 | ºC |

| Parameter | Specification |
|---|---|
| Fire and Electric Shock | CAN/CSA-C22.2 No. 60950-1-03<br>UL 60950-1:2003<br>EN 60950-1:2001 |

## Functional Block Diagram



## Pin-out Diagram



### Terminal List Table

| Number | Name | Description |
|---|---|---|
| 1 and 2 | IP+ | Terminals for current being sampled; fused internally |
| 3 and 4 | IP– | Terminals for current being sampled; fused internally |
| 5 | GND | Signal ground terminal |
| 6 | FILTER | Terminal for external capacitor that sets bandwidth |
| 7 | VIOUT | Analog output signal |
| 8 | VCC | Device power supply terminal |

**COMMON OPERATING CHARACTERISTICS**[1] over full range of $T_A$, $C_F$ = 1 nF, and $V_{CC}$ = 5 V, unless otherwise specified

| Characteristic | Symbol | Test Conditions | Min. | Typ. | Max. | Units |
|---|---|---|---|---|---|---|
| **ELECTRICAL CHARACTERISTICS** | | | | | | |
| Supply Voltage | $V_{CC}$ | | 4.5 | 5.0 | 5.5 | V |
| Supply Current | $I_{CC}$ | $V_{CC}$ = 5.0 V, output open | – | 10 | 13 | mA |
| Output Capacitance Load | $C_{LOAD}$ | VIOUT to GND | – | – | 10 | nF |
| Output Resistive Load | $R_{LOAD}$ | VIOUT to GND | 4.7 | – | – | kΩ |
| Primary Conductor Resistance | $R_{PRIMARY}$ | $T_A$ = 25°C | – | 1.2 | – | mΩ |
| Rise Time | $t_r$ | $I_P = I_P$(max), $T_A$ = 25°C, $C_{OUT}$ = open | – | 5 | – | μs |
| Frequency Bandwidth | f | –3 dB, $T_A$ = 25°C; $I_P$ is 10 A peak-to-peak | – | 80 | – | kHz |
| Nonlinearity | $E_{LIN}$ | Over full range of $I_P$ | – | 1.5 | – | % |
| Symmetry | $E_{SYM}$ | Over full range of $I_P$ | 98 | 100 | 102 | % |
| Zero Current Output Voltage | $V_{IOUT(Q)}$ | Bidirectional; $I_P$ = 0 A, $T_A$ = 25°C | – | $V_{CC}$ × 0.5 | – | V |
| Power-On Time | $t_{PO}$ | Output reaches 90% of steady-state level, $T_J$ = 25°C, 20 A present on leadframe | – | 35 | – | μs |
| Magnetic Coupling[2] | | | – | 12 | – | G/A |
| Internal Filter Resistance[3] | $R_{F(INT)}$ | | | 1.7 | | kΩ |

[1] Device may be operated at higher primary current levels, $I_P$, and ambient, $T_A$, and internal leadframe temperatures, $T_A$, provided that the Maximum Junction Temperature, $T_J$(max), is not exceeded.

[2] 1G = 0.1 mT.

[3] $R_{F(INT)}$ forms an RC circuit via the FILTER pin.

**COMMON THERMAL CHARACTERISTICS**[1]

| | | | Min. | Typ. | Max. | Units |
|---|---|---|---|---|---|---|
| Operating Internal Leadframe Temperature | $T_A$ | E range | –40 | – | 85 | °C |
| | | | | | Value | Units |
| Junction-to-Lead Thermal Resistance[2] | $R_{θJL}$ | Mounted on the Allegro ASEK 712 evaluation board | | | 5 | °C/W |
| Junction-to-Ambient Thermal Resistance | $R_{θJA}$ | Mounted on the Allegro 85-0322 evaluation board, includes the power consumed by the board | | | 23 | °C/W |

[1] Additional thermal information is available on the Allegro website.

[2] The Allegro evaluation board has 1500 mm$^2$ of 2 oz. copper on each side, connected to pins 1 and 2, and to pins 3 and 4, with thermal vias connecting the layers. Performance values include the power consumed by the PCB. Further details on the board are available from the Frequently Asked Questions document on our website. Further information about board design and thermal performance also can be found in the Applications Information section of this datasheet.

### x05B PERFORMANCE CHARACTERISTICS[1] $T_A$ = –40°C to 85°C, $C_F$ = 1 nF, and $V_{CC}$ = 5 V, unless otherwise specified

| Characteristic | Symbol | Test Conditions | Min. | Typ. | Max. | Units |
|---|---|---|---|---|---|---|
| Optimized Accuracy Range | $I_P$ | | –5 | – | 5 | A |
| Sensitivity | Sens | Over full range of $I_{P,}$ $T_A$ = 25°C | 180 | 185 | 190 | mV/A |
| Noise | $V_{NOISE(PP)}$ | Peak-to-peak, $T_A$ = 25°C, 185 mV/A programmed Sensitivity, $C_F$ = 47 nF, $C_{OUT}$ = open, 2 kHz bandwidth | – | 21 | – | mV |
| Zero Current Output Slope | $\Delta I_{OUT(Q)}$ | $T_A$ = –40°C to 25°C | – | –0.26 | – | mV/°C |
| | | $T_A$ = 25°C to 150°C | – | –0.08 | – | mV/°C |
| Sensitivity Slope | ΔSens | $T_A$ = –40°C to 25°C | – | 0.054 | – | mV/A/°C |
| | | $T_A$ = 25°C to 150°C | – | –0.008 | – | mV/A/°C |
| Total Output Error[2] | $E_{TOT}$ | $I_P$ =±5 A, $T_A$ = 25°C | – | ±1.5 | – | % |

[1] Device may be operated at higher primary current levels, $I_P$, and ambient temperatures, $T_A$, provided that the Maximum Junction Temperature, $T_{J(max)}$, is not exceeded.

[2] Percentage of $I_P$, with $I_P$ = 5 A. Output filtered.

### x20A PERFORMANCE CHARACTERISTICS[1] $T_A$ = –40°C to 85°C, $C_F$ = 1 nF, and $V_{CC}$ = 5 V, unless otherwise specified

| Characteristic | Symbol | Test Conditions | Min. | Typ. | Max. | Units |
|---|---|---|---|---|---|---|
| Optimized Accuracy Range | $I_P$ | | –20 | – | 20 | A |
| Sensitivity | Sens | Over full range of $I_{P,}$ $T_A$ = 25°C | 96 | 100 | 104 | mV/A |
| Noise | $V_{NOISE(PP)}$ | Peak-to-peak, $T_A$ = 25°C, 100 mV/A programmed Sensitivity, $C_F$ = 47 nF, $C_{OUT}$ = open, 2 kHz bandwidth | – | 11 | – | mV |
| Zero Current Output Slope | $\Delta I_{OUT(Q)}$ | $T_A$ = –40°C to 25°C | – | –0.34 | – | mV/°C |
| | | $T_A$ = 25°C to 150°C | – | –0.07 | – | mV/°C |
| Sensitivity Slope | ΔSens | $T_A$ = –40°C to 25°C | – | 0.017 | – | mV/A/°C |
| | | $T_A$ = 25°C to 150°C | – | –0.004 | – | mV/A/°C |
| Total Output Error[2] | $E_{TOT}$ | $I_P$ =±20 A, $T_A$ = 25°C | – | ±1.5 | – | % |

[1] Device may be operated at higher primary current levels, $I_P$, and ambient temperatures, $T_A$, provided that the Maximum Junction Temperature, $T_J$(max), is not exceeded.

[2] Percentage of $I_P$, with $I_P$ = 20 A. Output filtered.

### x30A PERFORMANCE CHARACTERISTICS[1] $T_A$ = –40°C to 85°C, $C_F$ = 1 nF, and $V_{CC}$ = 5 V, unless otherwise specified

| Characteristic | Symbol | Test Conditions | Min. | Typ. | Max. | Units |
|---|---|---|---|---|---|---|
| Optimized Accuracy Range | $I_P$ | | –30 | – | 30 | A |
| Sensitivity | Sens | Over full range of $I_P$, $T_A$ = 25°C | 63 | 66 | 69 | mV/A |
| Noise | $V_{NOISE(PP)}$ | Peak-to-peak, $T_A$ = 25°C, 66 mV/A programmed Sensitivity, $C_F$ = 47 nF, $C_{OUT}$ = open, 2 kHz bandwidth | – | 7 | – | mV |
| Zero Current Output Slope | $\Delta I_{OUT(Q)}$ | $T_A$ = –40°C to 25°C | – | –0.35 | – | mV/°C |
| | | $T_A$ = 25°C to 150°C | – | –0.08 | – | mV/°C |
| Sensitivity Slope | ΔSens | $T_A$ = –40°C to 25°C | – | 0.007 | – | mV/A/°C |
| | | $T_A$ = 25°C to 150°C | – | –0.002 | – | mV/A/°C |
| Total Output Error[2] | $E_{TOT}$ | $I_P$ = ±30 A, $T_A$ = 25°C | – | ±1.5 | – | % |

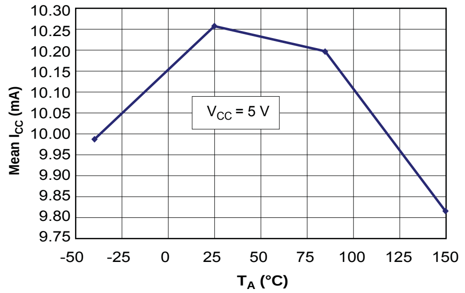
[1] Device may be operated at higher primary current levels, $I_P$, and ambient temperatures, $T_A$, provided that the Maximum Junction Temperature, $T_J$(max), is not exceeded.

[2] Percentage of $I_P$, with $I_P$ = 30 A. Output filtered.

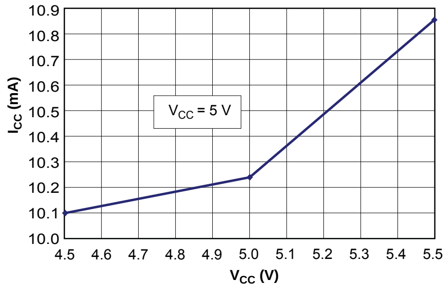
## Characteristic Performance

$I_P$ = 5 A, unless otherwise specified

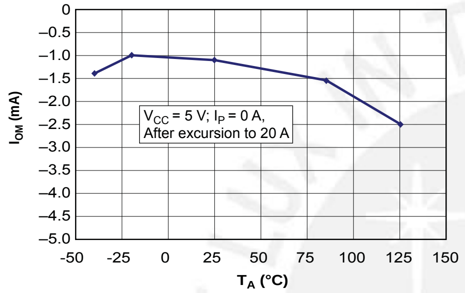
### Mean Supply Current versus Ambient Temperature



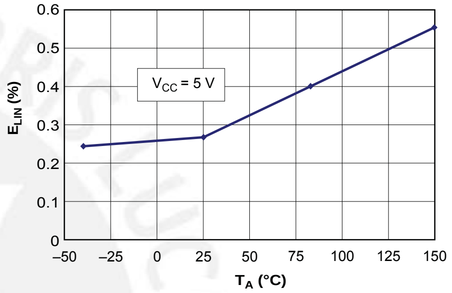
### Supply Current versus Supply Voltage



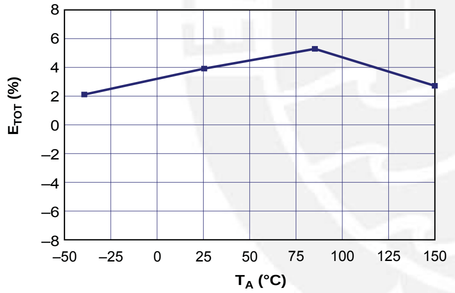
### Magnetic Offset versus Ambient Temperature



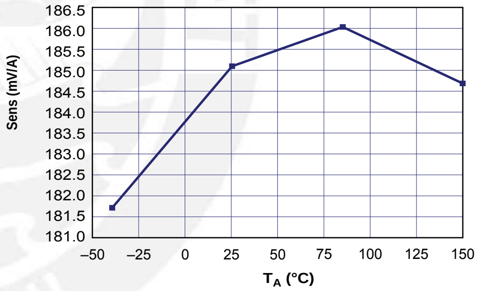
### Nonlinearity versus Ambient Temperature



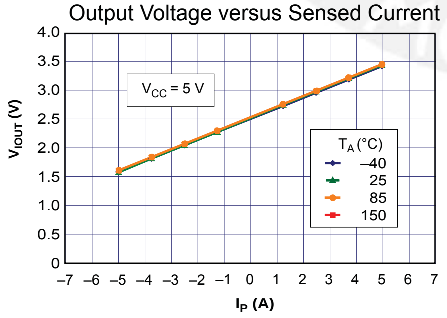
### Mean Total Output Error versus Ambient Temperature



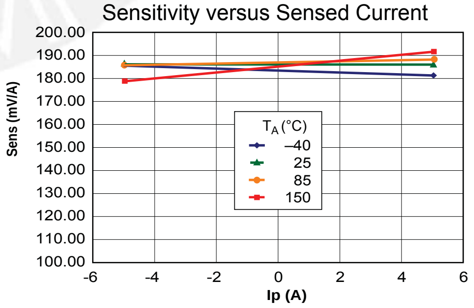
### Sensitivity versus Ambient Temperature



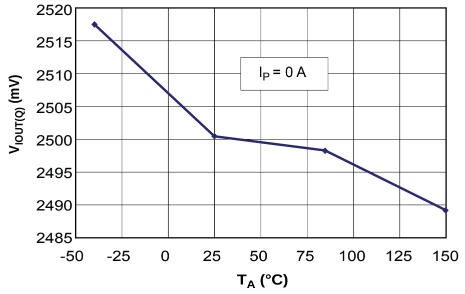
### Output Voltage versus Sensed Current



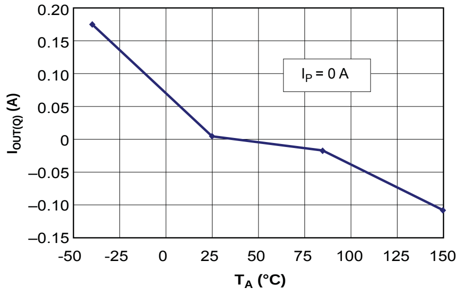
### Sensitivity versus Sensed Current



### 0 A Output Voltage versus Ambient Temperature



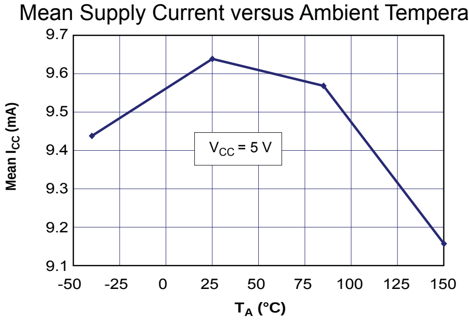
### 0 A Output Voltage Current versus Ambient Temperature



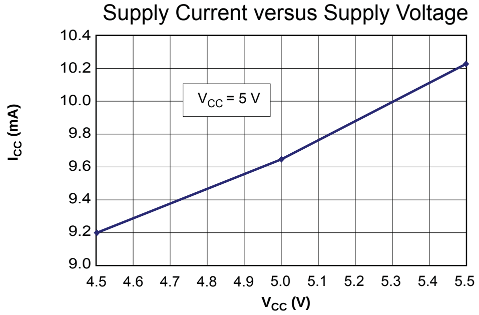
## Characteristic Performance

$I_P$ = 20 A, unless otherwise specified

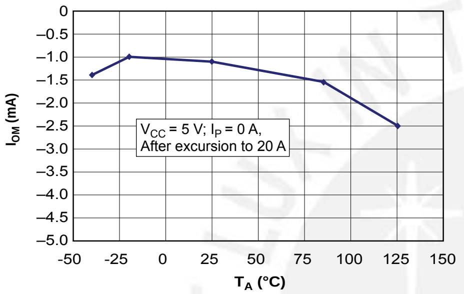
Mean Supply Current versus Ambient Temperature



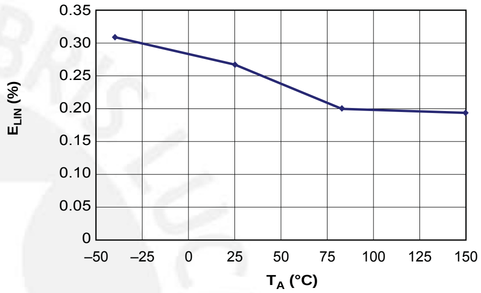
Supply Current versus Supply Voltage



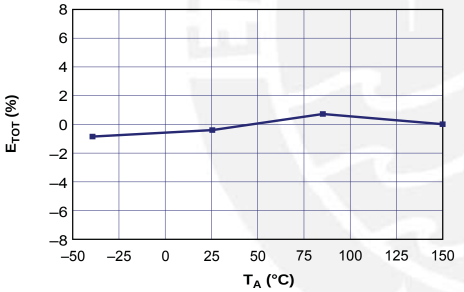
Magnetic Offset versus Ambient Temperature



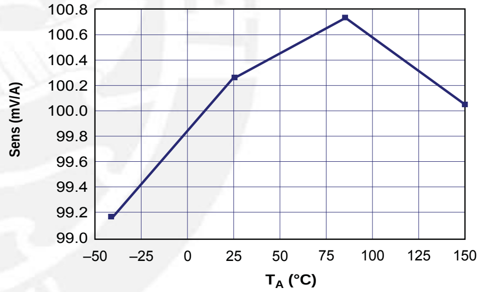
Nonlinearity versus Ambient Temperature



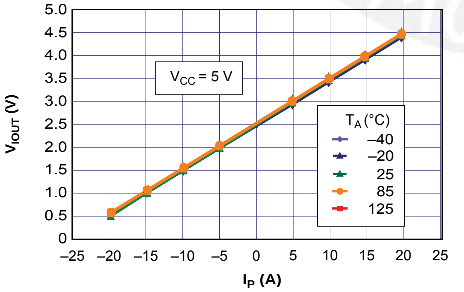
Mean Total Output Error versus Ambient Temperature



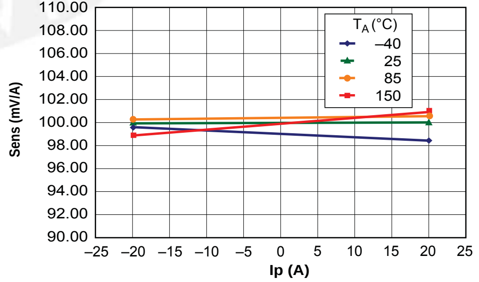
Sensitivity versus Ambient Temperature



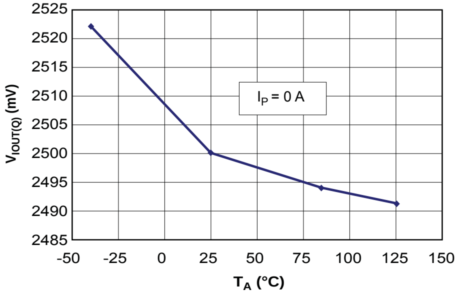
Output Voltage versus Sensed Current



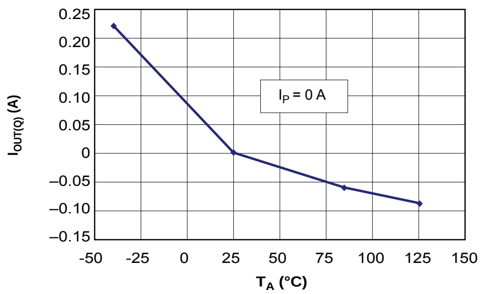
Sensitivity versus Sensed Current



0 A Output Voltage versus Ambient Temperature



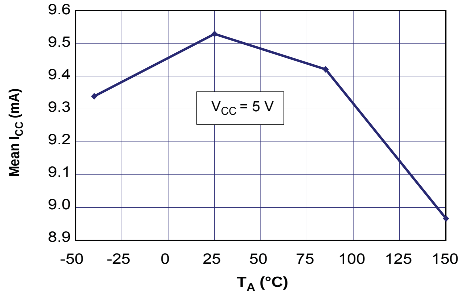
0 A Output Voltage Current versus Ambient Temperature



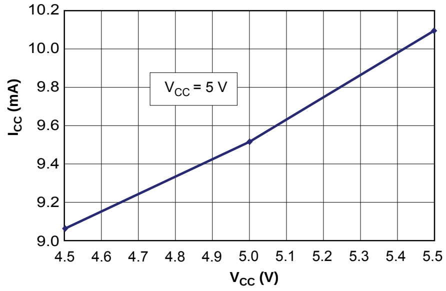
## Characteristic Performance

$I_P$ = 30 A, unless otherwise specified

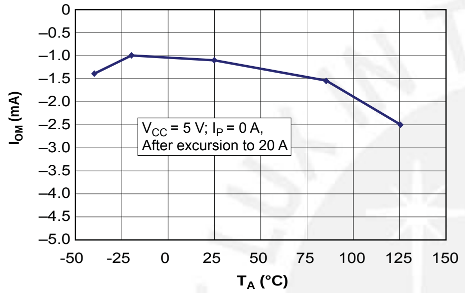
Mean Supply Current versus Ambient Temperature



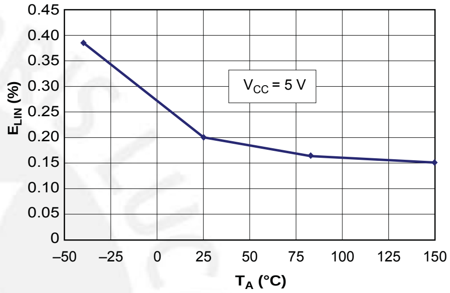
Supply Current versus Supply Voltage



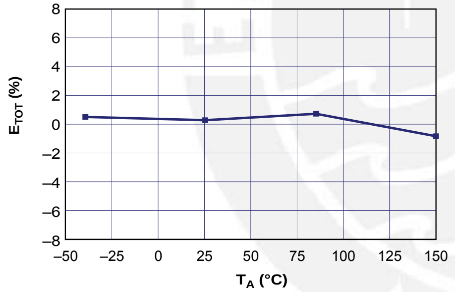
Magnetic Offset versus Ambient Temperature



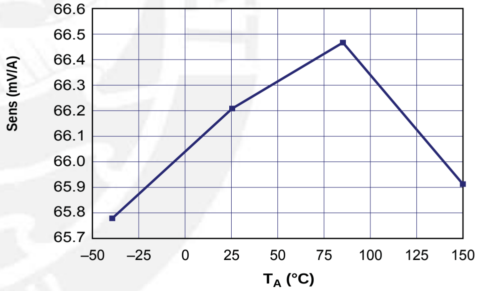
Nonlinearity versus Ambient Temperature



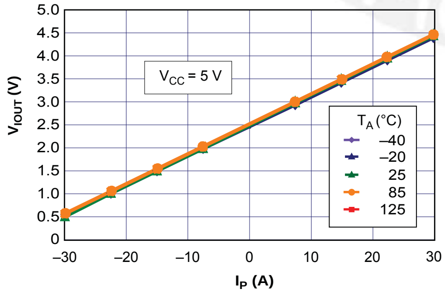
Mean Total Output Error versus Ambient Temperature



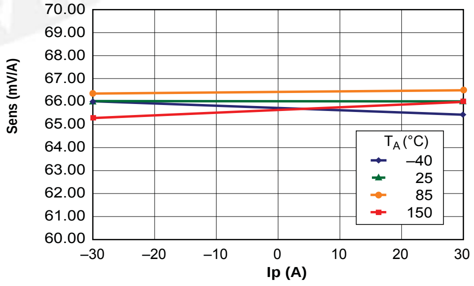
Sensitivity versus Ambient Temperature



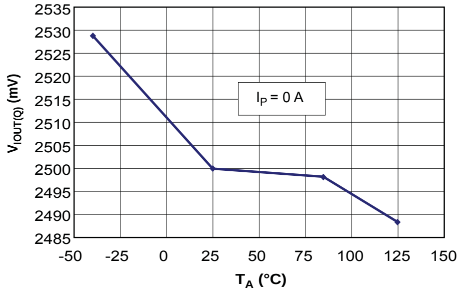
Output Voltage versus Sensed Current



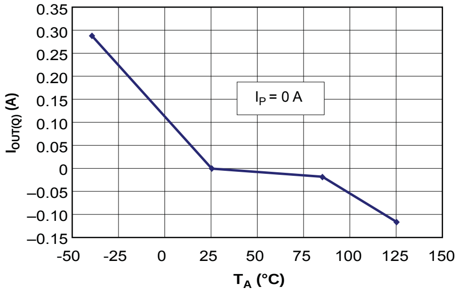
Sensitivity versus Sensed Current



0 A Output Voltage versus Ambient Temperature



0 A Output Voltage Current versus Ambient Temperature



## Definitions of Accuracy Characteristics

**Sensitivity (Sens).** The change in device output in response to a 1 A change through the primary conductor. The sensitivity is the product of the magnetic circuit sensitivity (G/A) and the linear IC amplifier gain (mV/G). The linear IC amplifier gain is programmed at the factory to optimize the sensitivity (mV/A) for the full-scale current of the device.

**Noise ($V_{NOISE}$).** The product of the linear IC amplifier gain (mV/G) and the noise floor for the Allegro Hall effect linear IC (≈1 G). The noise floor is derived from the thermal and shot noise observed in Hall elements. Dividing the noise (mV) by the sensitivity (mV/A) provides the smallest current that the device is able to resolve.

**Linearity ($E_{LIN}$).** The degree to which the voltage output from the IC varies in direct proportion to the primary current through its full-scale amplitude. Nonlinearity in the output can be attributed to the saturation of the flux concentrator approaching the full-scale current. The following equation is used to derive the linearity:

$$100\left\{1-\left[\frac{\Delta\text{ gain}\times\%\text{ sat }(V_{\text{IOUT\_full-scale amperes}}-V_{\text{IOUT(Q)}})}{2\,(V_{\text{IOUT\_half-scale amperes}}-V_{\text{IOUT(Q)}})}\right]\right\}$$

where $V_{\text{IOUT\_full-scale amperes}}$ = the output voltage (V) when the sampled current approximates full-scale $\pm I_P$ .

**Symmetry ($E_{SYM}$).** The degree to which the absolute voltage output from the IC varies in proportion to either a positive or negative full-scale primary current. The following formula is used to derive symmetry:

$$100\left(\frac{V_{\text{IOUT\_+ full-scale amperes}}-V_{\text{IOUT(Q)}}}{V_{\text{IOUT(Q)}}-V_{\text{IOUT\_–full-scale amperes}}}\right)$$

**Quiescent output voltage ($V_{IOUT(Q)}$).** The output of the device when the primary current is zero. For a unipolar supply voltage, it nominally remains at $V_{CC}/2$. Thus, $V_{CC}$ = 5 V translates into $V_{IOUT(Q)}$ = 2.5 V. Variation in $V_{IOUT(Q)}$ can be attributed to the resolution of the Allegro linear IC quiescent voltage trim and thermal drift.

**Electrical offset voltage ($V_{OE}$).** The deviation of the device output from its ideal quiescent value of $V_{CC}/2$ due to nonmagnetic causes. To convert this voltage to amperes, divide by the device sensitivity, Sens.

**Accuracy ($E_{TOT}$).** The accuracy represents the maximum deviation of the actual output from its ideal value. This is also known as the total output error. The accuracy is illustrated graphically in the output voltage versus current chart at right.

Accuracy is divided into four areas:

- **0 A at 25°C.** Accuracy at the zero current flow at 25°C, without the effects of temperature.
- **0 A over Δ temperature.** Accuracy at the zero current flow including temperature effects.
- **Full-scale current at 25°C.** Accuracy at the the full-scale current at 25°C, without the effects of temperature.
- **Full-scale current over Δ temperature.** Accuracy at the full-scale current flow including temperature effects.

**Ratiometry**. The ratiometric feature means that its 0 A output, $V_{IOUT(Q)}$, (nominally equal to $V_{CC}/2$) and sensitivity, Sens, are proportional to its supply voltage, $V_{CC}$. The following formula is used to derive the ratiometric change in 0 A output voltage, $\Delta V_{IOUT(Q)RAT}$ (%).

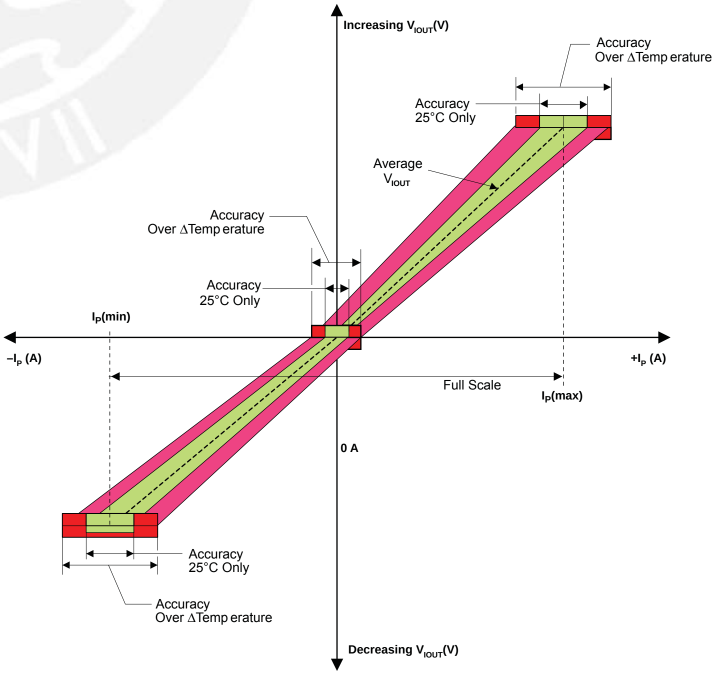
$$100\left(\frac{V_{\text{IOUT(Q)VCC}}\,/\,V_{\text{IOUT(Q)5V}}}{V_{\text{CC}}\,/\,5\text{ V}}\right)$$

The ratiometric change in sensitivity, $\Delta\text{Sens}_{RAT}$ (%), is defined as:

$$100\left(\frac{\mathit{Sens}_{\text{VCC}}\,/\,\mathit{Sens}_{\text{5V}}}{V_{\text{CC}}\,/\,5\text{ V}}\right)$$

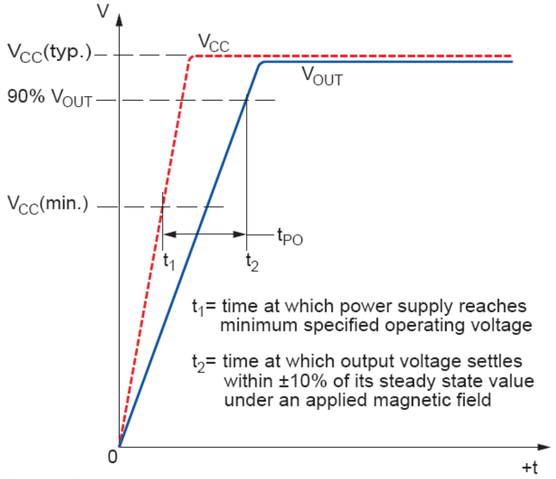
### Output Voltage versus Sampled Current

Accuracy at 0 A and at Full-Scale Current

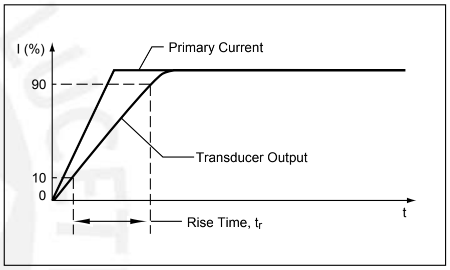


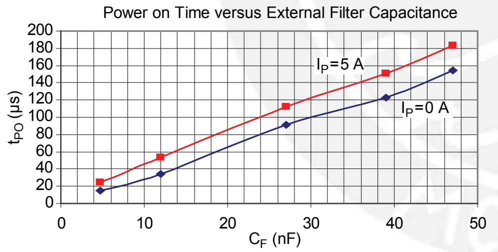
## Definitions of Dynamic Response Characteristics

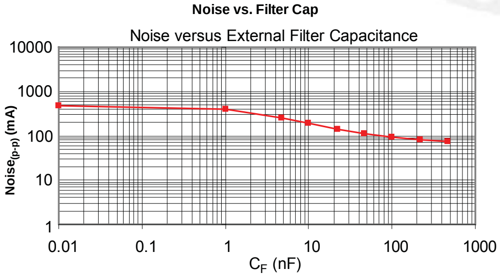
**Power-On Time ($t_{PO}$).** When the supply is ramped to its operating voltage, the device requires a finite time to power its internal components before responding to an input magnetic field. Power-On Time, $t_{PO}$ , is defined as the time it takes for the output voltage to settle within ±10% of its steady state value under an applied magnetic field, after the power supply has reached its minimum specified operating voltage, $V_{CC}$(min), as shown in the chart at right.

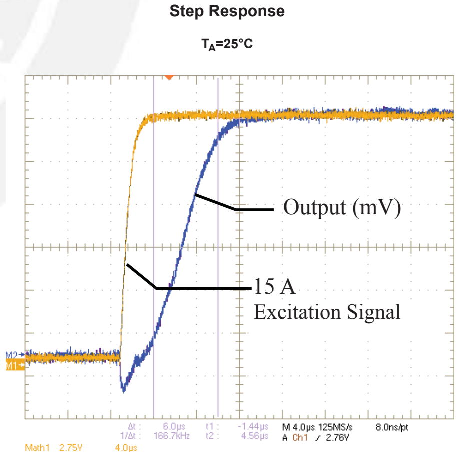

V
VCC(typ.)
VCC
VOUT
90% VOUT
VCC(min.)
tPO
t1
t2
0
+t
t1= time at which power supply reaches minimum specified operating voltage
t2= time at which output voltage settles within ±10% of its steady state value under an applied magnetic field

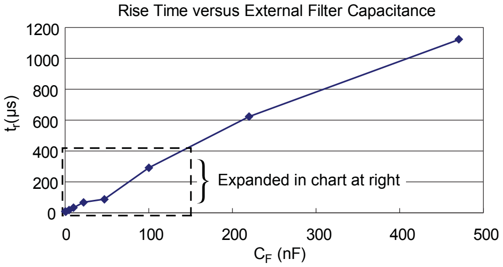
**Rise time ($t_r$).** The time interval between a) when the device reaches 10% of its full scale value, and b) when it reaches 90% of its full scale value. The rise time to a step response is used to derive the bandwidth of the device, in which $f(–3\text{ dB}) = 0.35 / t_r$. Both $t_r$ and $t_{RESPONSE}$ are detrimentally affected by eddy current losses observed in the conductive IC ground plane.



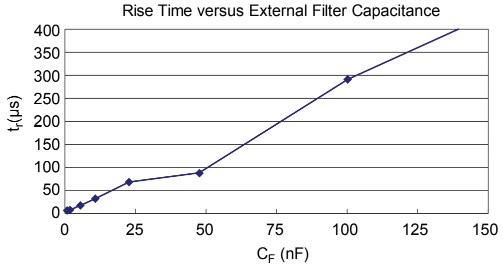








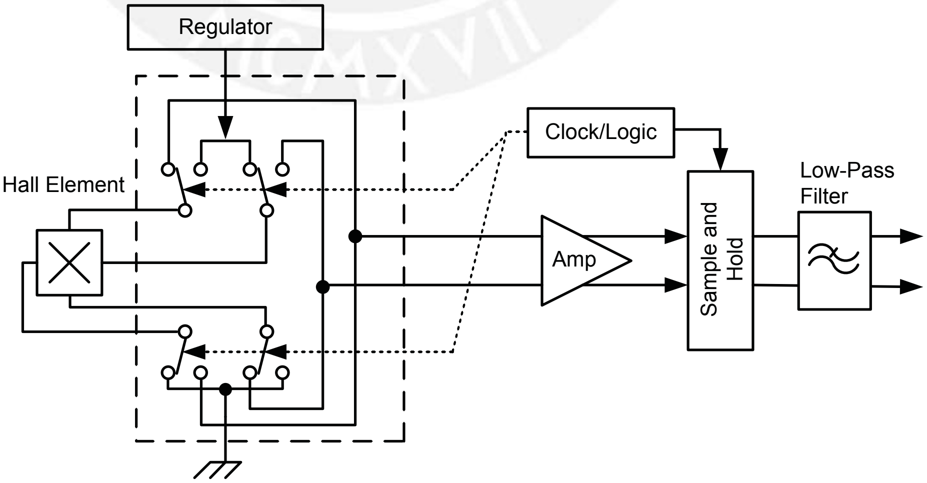
| $C_F$ (nF) | $t_r$ (μs) |
|---|---|
| 0 | 6.6 |
| 1 | 7.7 |
| 4.7 | 17.4 |
| 10 | 32.1 |
| 22 | 68.2 |
| 47 | 88.2 |
| 100 | 291.3 |
| 220 | 623.0 |
| 470 | 1120.0 |



## Chopper Stabilization Technique

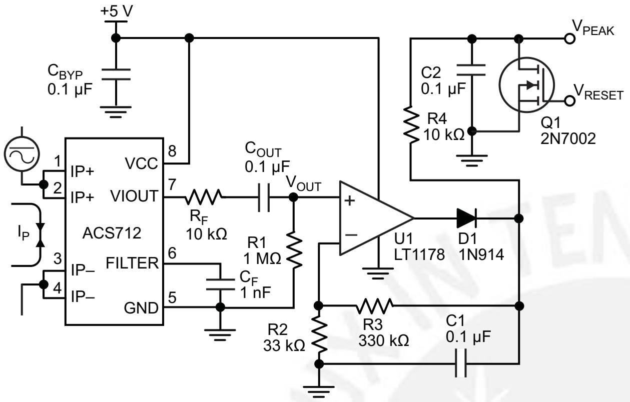
Chopper Stabilization is an innovative circuit technique that is used to minimize the offset voltage of a Hall element and an associated on-chip amplifier. Allegro patented a Chopper Stabilization technique that nearly eliminates Hall IC output drift induced by temperature or package stress effects. This offset reduction technique is based on a signal modulation-demodulation process. Modulation is used to separate the undesired DC offset signal from the magnetically induced signal in the frequency domain. Then, using a low-pass filter, the modulated DC offset is suppressed while the magnetically induced signal passes through the filter. As a result of this chopper stabilization approach, the output voltage from the Hall IC is desensitized to the effects of temperature and mechanical stress. This technique produces devices that have an extremely stable Electrical Offset Voltage, are immune to thermal stress, and have precise recoverability after temperature cycling.

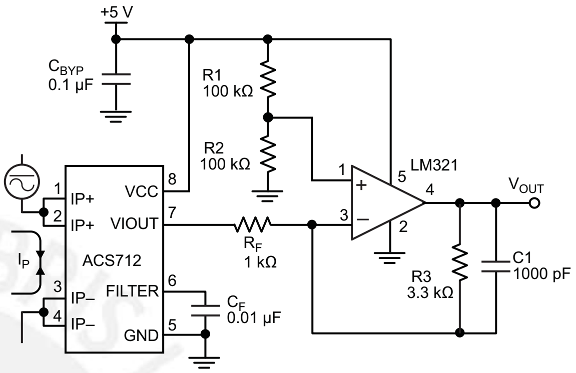
This technique is made possible through the use of a BiCMOS process that allows the use of low-offset and low-noise amplifiers in combination with high-density logic integration and sample and hold circuits.



Concept of Chopper Stabilization Technique

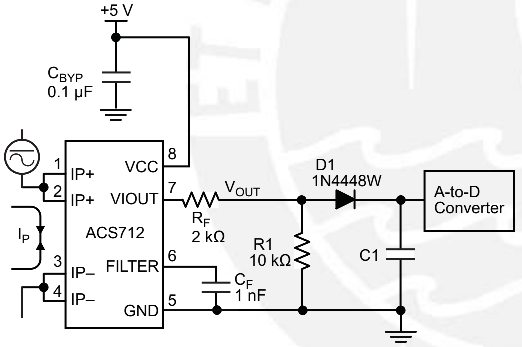
## Typical Applications

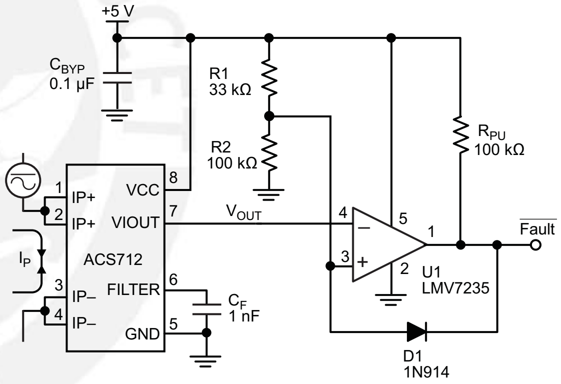




Application 3. This configuration increases gain to 610 mV/A (tested using the ACS712ELC-05A).

Application 2. Peak Detecting Circuit





Application 4. Rectified Output. 3.3 V scaling and rectification application for A-to-D converters. Replaces current transformer solutions with simpler ACS circuit. C1 is a function of the load resistance and filtering desired. R1 can be omitted if the full range is desired.

Application 5. 10 A Overcurrent Fault Latch. Fault threshold set by R1 and R2. This circuit latches an overcurrent fault and holds it until the 5 V rail is powered down.

## Improving Sensing System Accuracy Using the FILTER Pin

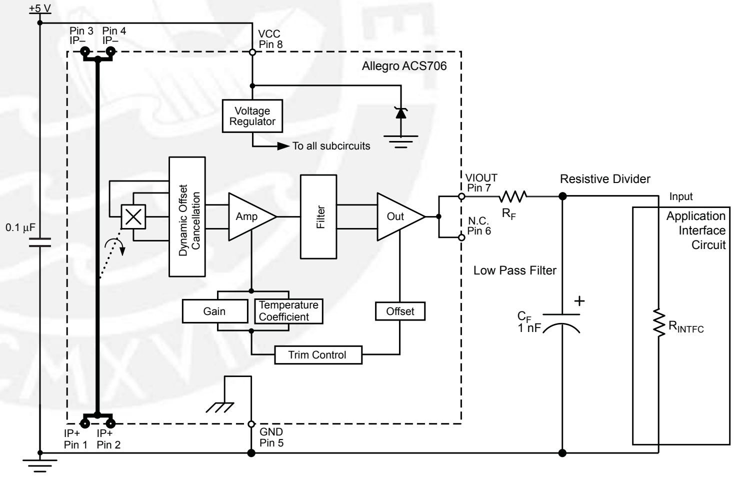
In low-frequency sensing applications, it is often advantageous to add a simple RC filter to the output of the device. Such a low-pass filter improves the signal-to-noise ratio, and therefore the resolution, of the device output signal. However, the addition of an RC filter to the output of a sensor IC can result in undesirable device output attenuation — even for DC signals.

Signal attenuation, $\Delta V_{ATT}$, is a result of the resistive divider effect between the resistance of the external filter, $R_F$ (see Application 6), and the input impedance and resistance of the customer interface circuit, $R_{INTFC}$. The transfer function of this resistive divider is given by:

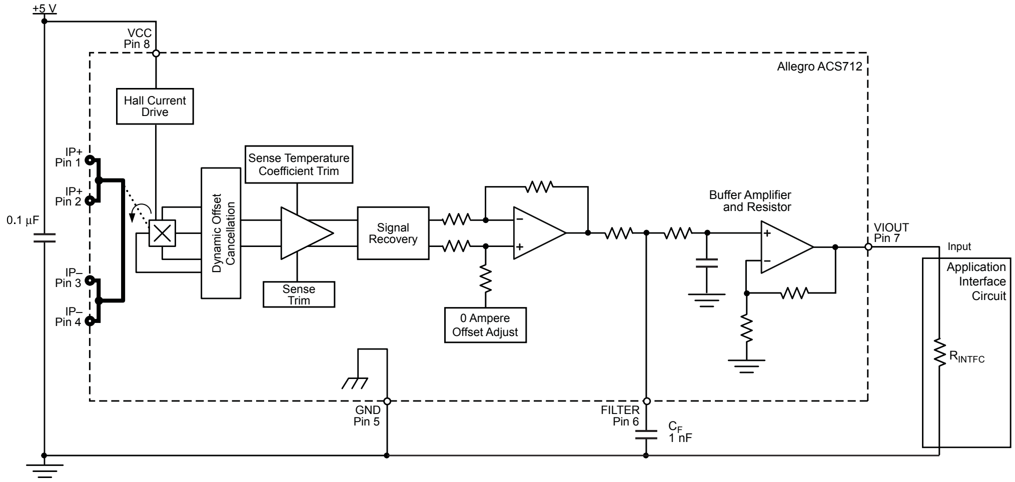
$$\Delta V_{ATT} = V_{IOUT}\left(\frac{R_{INTFC}}{R_F + R_{INTFC}}\right) .$$

Even if $R_F$ and $R_{INTFC}$ are designed to match, the two individual resistance values will most likely drift by different amounts over temperature. Therefore, signal attenuation will vary as a function of temperature. Note that, in many cases, the input impedance, $R_{INTFC}$, of a typical analog-to-digital converter (ADC) can be as low as 10 kΩ.

The ACS712 contains an internal resistor, a FILTER pin connection to the printed circuit board, and an internal buffer amplifier. With this circuit architecture, users can implement a simple RC filter via the addition of a capacitor, $C_F$ (see Application 7) from the FILTER pin to ground. The buffer amplifier inside of the ACS712 (located after the internal resistor and FILTER pin connection) eliminates the attenuation caused by the resistive divider effect described in the equation for $\Delta V_{ATT}$. Therefore, the ACS712 device is ideal for use in high-accuracy applications that cannot afford the signal attenuation associated with the use of an external RC low-pass filter.

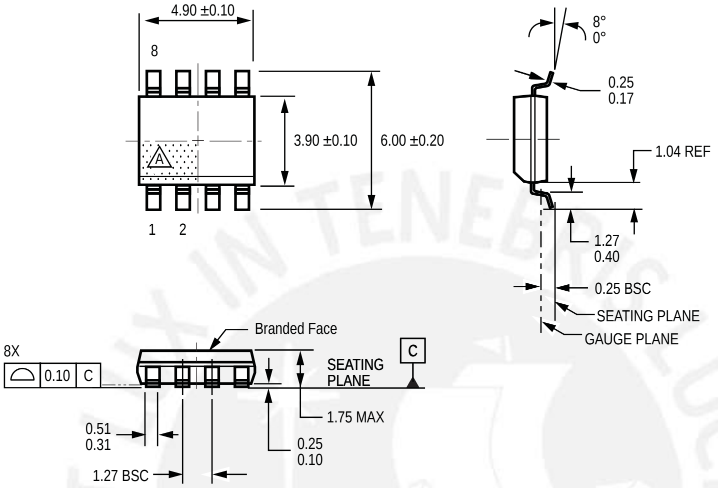


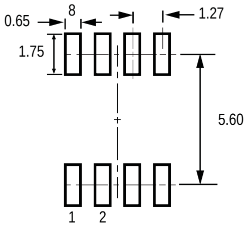
Application 6. When a low pass filter is constructed externally to a standard Hall effect device, a resistive divider may exist between the filter resistor, $R_F$, and the resistance of the customer interface circuit, $R_{INTFC}$. This resistive divider will cause excessive attenuation, as given by the transfer function for $\Delta V_{ATT}$.



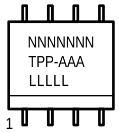
Application 7. Using the FILTER pin provided on the ACS712 eliminates the attenuation effects of the resistor divider between $R_F$ and $R_{INTFC}$, shown in Application 6.

## Package LC, 8-pin SOIC





△C PCB Layout Reference View



△B Standard Branding Reference View

N = Device part number
T = Device temperature range
P = Package Designator
A = Amperage
L = Lot number
Belly Brand = Country of Origin

For Reference Only; not for tooling use (reference MS-012AA)
Dimensions in millimeters
Dimensions exclusive of mold flash, gate burrs, and dambar protrusions
Exact case and lead configuration at supplier discretion within limits shown

△A Terminal #1 mark area

△B Branding scale and appearance at supplier discretion

△C △D Reference land pattern layout (reference IPC7351 SOIC127P600X175-8M); all pads a minimum of 0.20 mm from all adjacent pads; adjust as necessary to meet application process requirements and PCB layout tolerances

### Revision History

| Revision | Revision Date | Description of Revision |
|---|---|---|
| Rev. 14 | October 12, 2011 | Update branding specifications |
| | | |

The products described herein are protected by U.S. patents: 5,621,319; 7,598,601; and patent pending.

Allegro MicroSystems, Inc. reserves the right to make, from time to time, such departures from the detail specifications as may be required to permit improvements in the performance, reliability, or manufacturability of its products. Before placing an order, the user is cautioned to verify that the information being relied upon is current.

Allegro's products are not to be used in life support devices or systems, if a failure of an Allegro product can reasonably be expected to cause the failure of that life support device or system, or to affect the safety or effectiveness of that device or system.

The information included herein is believed to be accurate and reliable. However, Allegro MicroSystems, Inc. assumes no responsibility for its use; nor for any infringement of patents or other rights of third parties which may result from its use.

For the latest version of this document, visit our website:

**www.allegromicro.com**

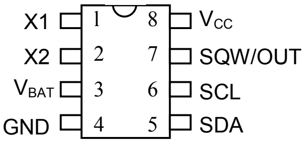
# DS1307
# 64 x 8 Serial Real-Time Clock

www.maxim-ic.com

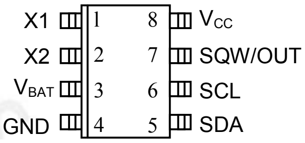
## FEATURES

- Real-time clock (RTC) counts seconds, minutes, hours, date of the month, month, day of the week, and year with leap-year compensation valid up to 2100
- 56-byte, battery-backed, nonvolatile (NV) RAM for data storage
- Two-wire serial interface
- Programmable squarewave output signal
- Automatic power-fail detect and switch circuitry
- Consumes less than 500nA in battery backup mode with oscillator running
- Optional industrial temperature range: -40°C to +85°C
- Available in 8-pin DIP or SOIC
- Underwriters Laboratory (UL) recognized

## PIN ASSIGNMENT



DS1307 8-Pin DIP (300-mil)



DS1307 8-Pin SOIC (150-mil)

## ORDERING INFORMATION

| | |
|---|---|
| DS1307 | 8-Pin DIP (300-mil) |
| DS1307Z | 8-Pin SOIC (150-mil) |
| DS1307N | 8-Pin DIP (Industrial) |
| DS1307ZN | 8-Pin SOIC (Industrial) |

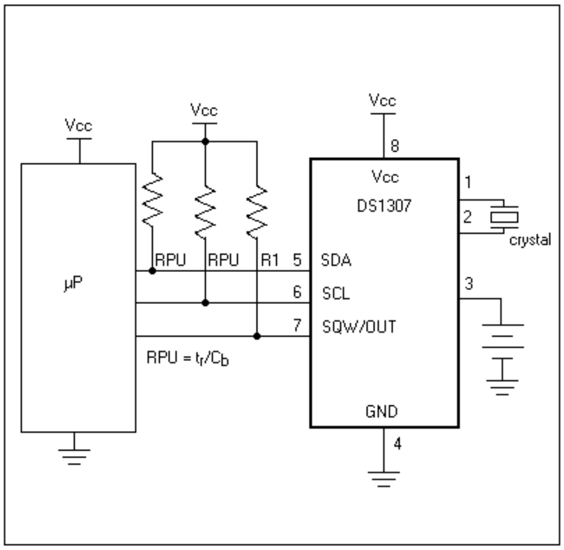
## PIN DESCRIPTION

| | |
|---|---|
| $V_{CC}$ | - Primary Power Supply |
| X1, X2 | - 32.768kHz Crystal Connection |
| $V_{BAT}$ | - +3V Battery Input |
| GND | - Ground |
| SDA | - Serial Data |
| SCL | - Serial Clock |
| SQW/OUT | - Square Wave/Output Driver |

## DESCRIPTION

The DS1307 Serial Real-Time Clock is a low-power, full binary-coded decimal (BCD) clock/calendar plus 56 bytes of NV SRAM. Address and data are transferred serially via a 2-wire, bi-directional bus. The clock/calendar provides seconds, minutes, hours, day, date, month, and year information. The end of the month date is automatically adjusted for months with fewer than 31 days, including corrections for leap year. The clock operates in either the 24-hour or 12-hour format with AM/PM indicator. The DS1307 has a built-in power sense circuit that detects power failures and automatically switches to the battery supply.

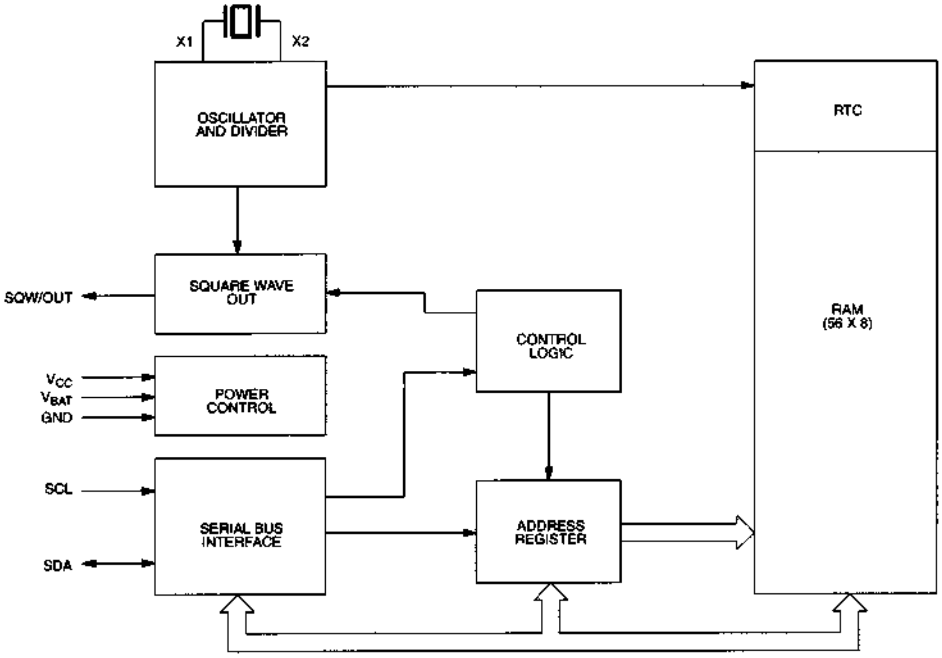
## TYPICAL OPERATING CIRCUIT



## OPERATION

The DS1307 operates as a slave device on the serial bus. Access is obtained by implementing a START condition and providing a device identification code followed by a register address. Subsequent registers can be accessed sequentially until a STOP condition is executed. When $V_{CC}$ falls below 1.25 x $V_{BAT}$ the device terminates an access in progress and resets the device address counter. Inputs to the device will not be recognized at this time to prevent erroneous data from being written to the device from an out of tolerance system. When $V_{CC}$ falls below $V_{BAT}$ the device switches into a low-current battery backup mode. Upon power-up, the device switches from battery to $V_{CC}$ when $V_{CC}$ is greater than $V_{BAT}$ + 0.2V and recognizes inputs when $V_{CC}$ is greater than 1.25 x $V_{BAT}$. The block diagram in Figure 1 shows the main elements of the serial RTC.

## **DS1307 BLOCK DIAGRAM** Figure 1



## SIGNAL DESCRIPTIONS

**$V_{CC}$, GND** – DC power is provided to the device on these pins. $V_{CC}$ is the +5V input. When 5V is applied within normal limits, the device is fully accessible and data can be written and read. When a 3V battery is connected to the device and $V_{CC}$ is below 1.25 x $V_{BAT}$, reads and writes are inhibited. However, the timekeeping function continues unaffected by the lower input voltage. As $V_{CC}$ falls below $V_{BAT}$ the RAM and timekeeper are switched over to the external power supply (nominal 3.0V DC) at $V_{BAT}$.

**$V_{BAT}$** – Battery input for any standard 3V lithium cell or other energy source. Battery voltage must be held between 2.0V and 3.5V for proper operation. The nominal write protect trip point voltage at which access to the RTC and user RAM is denied is set by the internal circuitry as 1.25 x $V_{BAT}$ nominal. A lithium battery with 48mAhr or greater will back up the DS1307 for more than 10 years in the absence of power at 25ºC. UL recognized to ensure against reverse charging current when used in conjunction with a lithium battery.

See "Conditions of Acceptability" at http://www.maxim-ic.com/TechSupport/QA/ntrl.htm**.**

**SCL (Serial Clock Input) –** SCL is used to synchronize data movement on the serial interface.

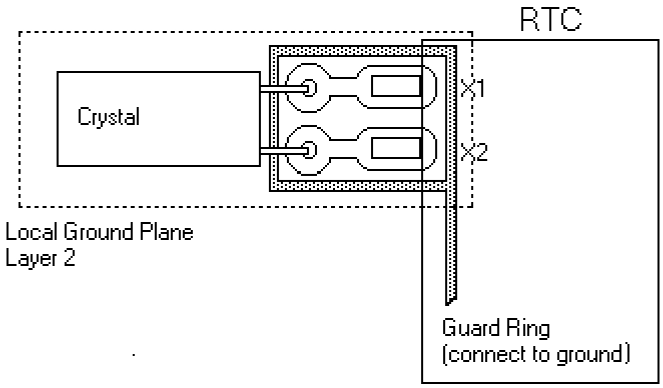
**SDA (Serial Data Input/Output) –** SDA is the input/output pin for the 2-wire serial interface. The SDA pin is open drain which requires an external pullup resistor.

**SQW/OUT (Square Wave/Output Driver) –** When enabled, the SQWE bit set to 1, the SQW/OUT pin outputs one of four square wave frequencies (1Hz, 4kHz, 8kHz, 32kHz). The SQW/OUT pin is open drain and requires an external pull-up resistor. SQW/OUT will operate with either Vcc or Vbat applied.

**X1, X2** – Connections for a standard 32.768kHz quartz crystal. The internal oscillator circuitry is designed for operation with a crystal having a specified load capacitance (CL) of 12.5pF.

For more information on crystal selection and crystal layout considerations, please consult Application Note 58, "Crystal Considerations with Dallas Real-Time Clocks." The DS1307 can also be driven by an external 32.768kHz oscillator. In this configuration, the X1 pin is connected to the external oscillator signal and the X2 pin is floated.

## RECOMMENDED LAYOUT FOR CRYSTAL



## CLOCK ACCURACY

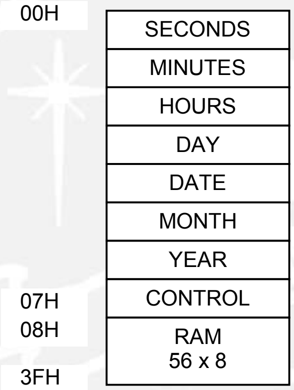
The accuracy of the clock is dependent upon the accuracy of the crystal and the accuracy of the match between the capacitive load of the oscillator circuit and the capacitive load for which the crystal was trimmed. Additional error will be added by crystal frequency drift caused by temperature shifts. External circuit noise coupled into the oscillator circuit may result in the clock running fast. See Application Note 58, “Crystal Considerations with Dallas Real-Time Clocks” for detailed information.

Please review Application Note 95, “Interfacing the DS1307 with a 8051-Compatible Microcontroller” for additional information.

## RTC AND RAM ADDRESS MAP

The address map for the RTC and RAM registers of the DS1307 is shown in Figure 2. The RTC registers are located in address locations 00h to 07h. The RAM registers are located in address locations 08h to 3Fh. During a multi-byte access, when the address pointer reaches 3Fh, the end of RAM space, it wraps around to location 00h, the beginning of the clock space.

## DS1307 ADDRESS MAP Figure 2



## CLOCK AND CALENDAR

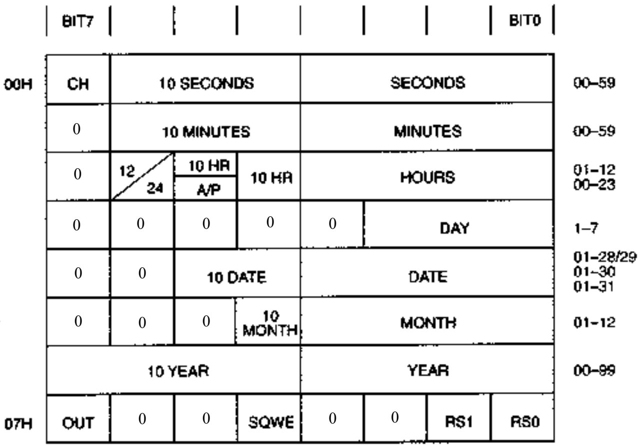
The time and calendar information is obtained by reading the appropriate register bytes. The RTC registers are illustrated in Figure 3. The time and calendar are set or initialized by writing the appropriate register bytes. The contents of the time and calendar registers are in the BCD format. Bit 7 of register 0 is the clock halt (CH) bit. When this bit is set to a 1, the oscillator is disabled. When cleared to a 0, the oscillator is enabled.

**Please note that the initial power-on state of all registers is not defined. Therefore, it is important to enable the oscillator (CH bit = 0) during initial configuration.**

The DS1307 can be run in either 12-hour or 24-hour mode. Bit 6 of the hours register is defined as the 12- or 24-hour mode select bit. When high, the 12-hour mode is selected. In the 12-hour mode, bit 5 is the AM/PM bit with logic high being PM. In the 24-hour mode, bit 5 is the second 10 hour bit (20-23 hours).

On a 2-wire START, the current time is transferred to a second set of registers. The time information is read from these secondary registers, while the clock may continue to run. This eliminates the need to re-read the registers in case of an update of the main registers during a read.

## **DS1307 TIMEKEEPER REGISTERS** Figure 3



|  | BIT7 |  |  |  |  |  |  | BIT0 |  |
|---|---|---|---|---|---|---|---|---|---|
| 00H | CH | 10 SECONDS |  |  | SECONDS |  |  |  | 00–59 |
|  | 0 | 10 MINUTES |  |  | MINUTES |  |  |  | 00–59 |
|  | 0 | 12/24 | 10 HR<br>A/P | 10 HR | HOURS |  |  |  | 01–12<br>00–23 |
|  | 0 | 0 | 0 | 0 | 0 | DAY |  |  | 1–7 |
|  | 0 | 0 | 10 DATE |  | DATE |  |  |  | 01–28/29<br>01–30<br>01–31 |
|  | 0 | 0 | 0 | 10 MONTH | MONTH |  |  |  | 01–12 |
|  | 10 YEAR |  |  |  | YEAR |  |  |  | 00–99 |
| 07H | OUT | 0 | 0 | SQWE | 0 | 0 | RS1 | RS0 |  |

## CONTROL REGISTER

The DS1307 control register is used to control the operation of the SQW/OUT pin.

| BIT 7 | BIT 6 | BIT 5 | BIT 4 | BIT 3 | BIT 2 | BIT 1 | BIT 0 |
|---|---|---|---|---|---|---|---|
| OUT | 0 | 0 | SQWE | 0 | 0 | RS1 | RS0 |

**OUT (Output control):** This bit controls the output level of the SQW/OUT pin when the square wave output is disabled. If SQWE = 0, the logic level on the SQW/OUT pin is 1 if OUT = 1 and is 0 if OUT = 0.

**SQWE (Square Wave Enable):** This bit, when set to a logic 1, will enable the oscillator output. The frequency of the square wave output depends upon the value of the RS0 and RS1 bits. With the square wave output set to 1Hz, the clock registers update on the falling edge of the square wave.

**RS (Rate Select):** These bits control the frequency of the square wave output when the square wave output has been enabled. Table 1 lists the square wave frequencies that can be selected with the RS bits.

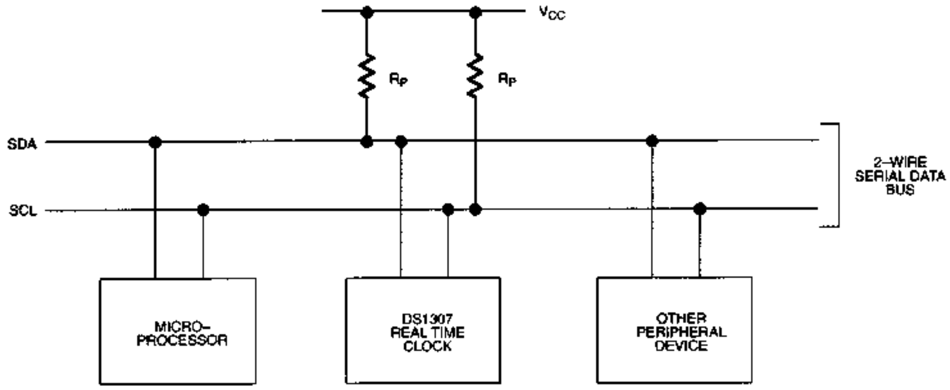
## **SQUAREWAVE OUTPUT FREQUENCY** Table 1

| RS1 | RS0 | SQW OUTPUT FREQUENCY |
|---|---|---|
| 0 | 0 | 1Hz |
| 0 | 1 | 4.096kHz |
| 1 | 0 | 8.192kHz |
| 1 | 1 | 32.768kHz |

## 2-WIRE SERIAL DATA BUS

The DS1307 supports a bi-directional, 2-wire bus and data transmission protocol. A device that sends data onto the bus is defined as a transmitter and a device receiving data as a receiver. The device that controls the message is called a master. The devices that are controlled by the master are referred to as slaves. The bus must be controlled by a master device that generates the serial clock (SCL), controls the bus access, and generates the START and STOP conditions. The DS1307 operates as a slave on the 2-wire bus. A typical bus configuration using this 2-wire protocol is show in Figure 4.

## TYPICAL 2-WIRE BUS CONFIGURATION Figure 4



Figures 5, 6, and 7 detail how data is transferred on the 2-wire bus.

- Data transfer may be initiated only when the bus is not busy.
- During data transfer, the data line must remain stable whenever the clock line is HIGH. Changes in the data line while the clock line is high will be interpreted as control signals.

Accordingly, the following bus conditions have been defined:

**Bus not busy:** Both data and clock lines remain HIGH.

**Start data transfer:** A change in the state of the data line, from HIGH to LOW, while the clock is HIGH, defines a START condition.

**Stop data transfer:** A change in the state of the data line, from LOW to HIGH, while the clock line is HIGH, defines the STOP condition.

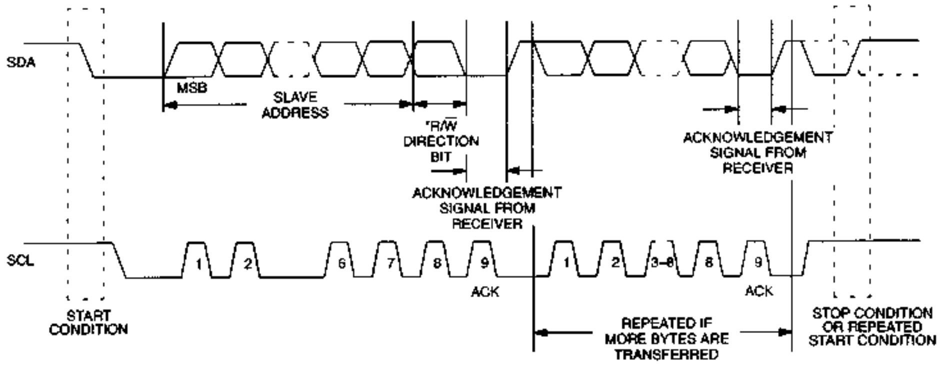
**Data valid:** The state of the data line represents valid data when, after a START condition, the data line is stable for the duration of the HIGH period of the clock signal. The data on the line must be changed during the LOW period of the clock signal. There is one clock pulse per bit of data.

Each data transfer is initiated with a START condition and terminated with a STOP condition. The number of data bytes transferred between START and STOP conditions is not limited, and is determined by the master device. The information is transferred byte-wise and each receiver acknowledges with a ninth bit. Within the 2-wire bus specifications a regular mode (100kHz clock rate) and a fast mode (400kHz clock rate) are defined. The DS1307 operates in the regular mode (100kHz) only.

**Acknowledge:** Each receiving device, when addressed, is obliged to generate an acknowledge after the reception of each byte. The master device must generate an extra clock pulse which is associated with this acknowledge bit.

A device that acknowledges must pull down the SDA line during the acknowledge clock pulse in such a way that the SDA line is stable LOW during the HIGH period of the acknowledge related clock pulse. Of course, setup and hold times must be taken into account. A master must signal an end of data to the slave by not generating an acknowledge bit on the last byte that has been clocked out of the slave. In this case, the slave must leave the data line HIGH to enable the master to generate the STOP condition.

## **DATA TRANSFER ON 2-WIRE SERIAL BUS** Figure 5


SDA
MSB
SLAVE ADDRESS
R/$\overline{W}$ DIRECTION BIT
ACKNOWLEDGEMENT SIGNAL FROM RECEIVER
ACKNOWLEDGEMENT SIGNAL FROM RECEIVER
SCL
1
2
6
7
8
9
ACK
1
2
3-8
8
9
ACK
START CONDITION
REPEATED IF MORE BYTES ARE TRANSFERRED
STOP CONDITION OR REPEATED START CONDITION

Depending upon the state of the R/$\overline{W}$ bit, two types of data transfer are possible:

1. **Data transfer from a master transmitter to a slave receiver.** The first byte transmitted by the master is the slave address. Next follows a number of data bytes. The slave returns an acknowledge bit after each received byte. Data is transferred with the most significant bit (MSB) first.

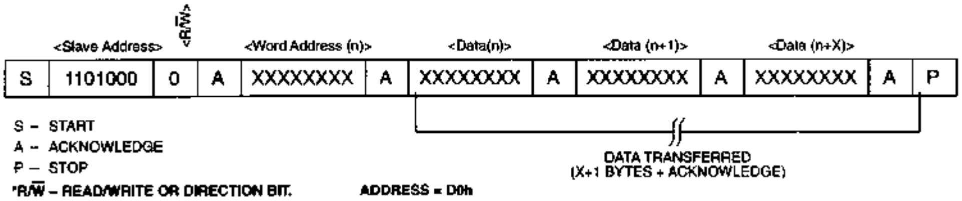
2. **Data transfer from a slave transmitter to a master receiver.** The first byte (the slave address) is transmitted by the master. The slave then returns an acknowledge bit. This is followed by the slave transmitting a number of data bytes. The master returns an acknowledge bit after all received bytes other than the last byte. At the end of the last received byte, a "not acknowledge" is returned.

The master device generates all of the serial clock pulses and the START and STOP conditions. A transfer is ended with a STOP condition or with a repeated START condition. Since a repeated START condition is also the beginning of the next serial transfer, the bus will not be released. Data is transferred with the most significant bit (MSB) first.

The DS1307 may operate in the following two modes:

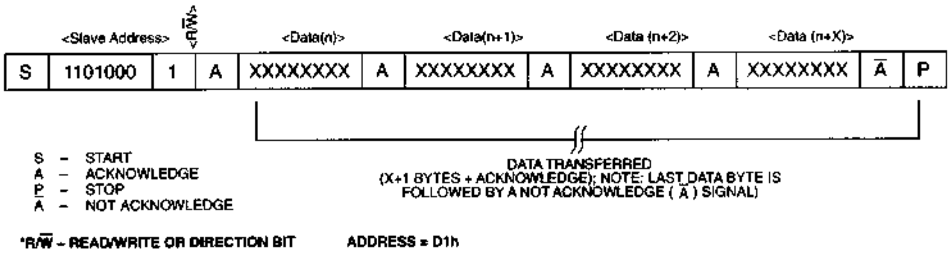
1. **Slave receiver mode (DS1307 write mode):** Serial data and clock are received through SDA and SCL. After each byte is received an acknowledge bit is transmitted. START and STOP conditions are recognized as the beginning and end of a serial transfer. Address recognition is performed by hardware after reception of the slave address and *direction bit (See Figure 6). The address byte is the first byte received after the start condition is generated by the master. The address byte contains the 7 bit DS1307 address, which is 1101000, followed by the *direction bit (R/$\overline{W}$) which, for a write, is a 0. After receiving and decoding the address byte the device outputs an acknowledge on the SDA line. After the DS1307 acknowledges the slave address + write bit, the master transmits a register address to the DS1307 This will set the register pointer on the DS1307. The master will then begin transmitting each byte of data with the DS1307 acknowledging each byte received. The master will generate a stop condition to terminate the data write.

## **DATA WRITE – SLAVE RECEIVER MODE** Figure 6



2. **Slave transmitter mode (DS1307 read mode):** The first byte is received and handled as in the slave receiver mode. However, in this mode, the *direction bit will indicate that the transfer direction is reversed. Serial data is transmitted on SDA by the DS1307 while the serial clock is input on SCL. START and STOP conditions are recognized as the beginning and end of a serial transfer (See Figure 7). The address byte is the first byte received after the start condition is generated by the master. The address byte contains the 7-bit DS1307 address, which is 1101000, followed by the *direction bit (R/$\overline{W}$) which, for a read, is a 1. After receiving and decoding the address byte the device inputs an acknowledge on the SDA line. The DS1307 then begins to transmit data starting with the register address pointed to by the register pointer. If the register pointer is not written to before the initiation of a read mode the first address that is read is the last one stored in the register pointer. The DS1307 must receive a "not acknowledge" to end a read.

## **DATA READ – SLAVE TRANSMITTER MODE** Figure 7



## ABSOLUTE MAXIMUM RATINGS*

| | |
|---|---|
| Voltage on Any Pin Relative to Ground | -0.5V to +7.0V |
| Storage Temperature | -55°C to +125°C |
| Soldering Temperature | 260°C for 10 seconds DIP<br>See JPC/JEDEC Standard J-STD-020A for<br>Surface Mount Devices |

* This is a stress rating only and functional operation of the device at these or any other conditions above those indicated in the operation sections of this specification is not implied. Exposure to absolute maximum rating conditions for extended periods of time may affect reliability.

| Range | Temperature | $V_{CC}$ |
|---|---|---|
| Commercial | 0°C to +70°C | 4.5V to 5.5V $V_{CC1}$ |
| Industrial | -40°C to +85°C | 4.5V to 5.5V $V_{CC1}$ |

## RECOMMENDED DC OPERATING CONDITIONS

(Over the operating range*)

| PARAMETER | SYMBOL | MIN | TYP | MAX | UNITS | NOTES |
|---|---|---|---|---|---|---|
| Supply Voltage | $V_{CC}$ | 4.5 | 5.0 | 5.5 | V | |
| Logic 1 | $V_{IH}$ | 2.2 | | $V_{CC}$ + 0.3 | V | |
| Logic 0 | $V_{IL}$ | -0.5 | | +0.8 | V | |
| $V_{BAT}$ Battery Voltage | $V_{BAT}$ | 2.0 | | 3.5 | V | |

*Unless otherwise specified.

## DC ELECTRICAL CHARACTERISTICS

(Over the operating range*)

| PARAMETER | SYMBOL | MIN | TYP | MAX | UNITS | NOTES |
|---|---|---|---|---|---|---|
| Input Leakage (SCL) | $I_{LI}$ | | | 1 | µA | |
| I/O Leakage (SDA & SQW/OUT) | $I_{LO}$ | | | 1 | µA | |
| Logic 0 Output ($I_{OL}$ = 5mA) | $V_{OL}$ | | | 0.4 | V | |
| Active Supply Current | $I_{CCA}$ | | | 1.5 | mA | 7 |
| Standby Current | $I_{CCS}$ | | | 200 | µA | 1 |
| Battery Current (OSC ON); SQW/OUT OFF | $I_{BAT1}$ | | 300 | 500 | nA | 2 |
| Battery Current (OSC ON); SQW/OUT ON (32kHz) | $I_{BAT2}$ | | 480 | 800 | nA | |
| Power-Fail Voltage | $V_{PF}$ | 1.216 x $V_{BAT}$ | 1.25 x $V_{BAT}$ | 1.284 x $V_{BAT}$ | V | 8 |

*Unless otherwise specified.

## AC ELECTRICAL CHARACTERISTICS

(Over the operating range*)

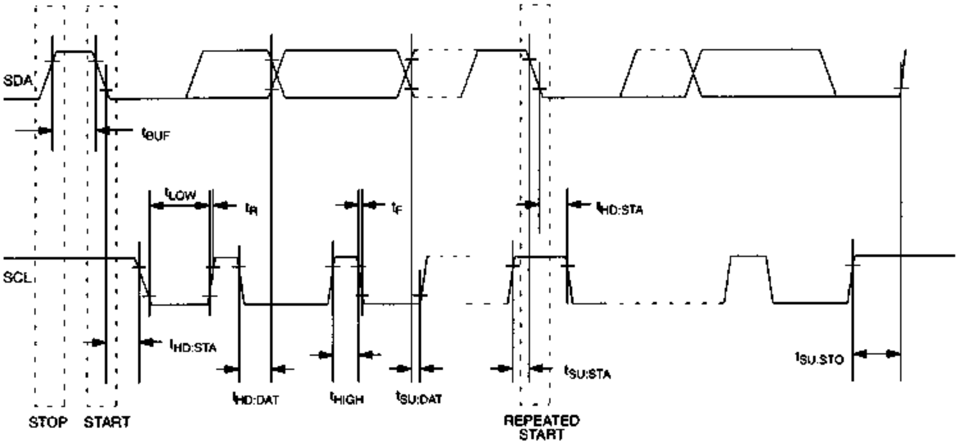
| PARAMETER | SYMBOL | MIN | TYP | MAX | UNITS | NOTES |
|---|---|---|---|---|---|---|
| SCL Clock Frequency | $f_{SCL}$ | 0 |  | 100 | kHz |  |
| Bus Free Time Between a STOP and START Condition | $t_{BUF}$ | 4.7 |  |  | µs |  |
| Hold Time (Repeated) START Condition | $t_{HD:STA}$ | 4.0 |  |  | µs | 3 |
| LOW Period of SCL Clock | $t_{LOW}$ | 4.7 |  |  | µs |  |
| HIGH Period of SCL Clock | $t_{HIGH}$ | 4.0 |  |  | µs |  |
| Set-up Time for a Repeated START Condition | $t_{SU:STA}$ | 4.7 |  |  | µs |  |
| Data Hold Time | $t_{HD:DAT}$ | 0 |  |  | µs | 4,5 |
| Data Set-up Time | $t_{SU:DAT}$ | 250 |  |  | ns |  |
| Rise Time of Both SDA and SCL Signals | $t_R$ |  |  | 1000 | ns |  |
| Fall Time of Both SDA and SCL Signals | $t_F$ |  |  | 300 | ns |  |
| Set-up Time for STOP Condition | $t_{SU:STO}$ | 4.7 |  |  | µs |  |
| Capacitive Load for each Bus Line | $C_B$ |  |  | 400 | pF | 6 |
| I/O Capacitance ($T_A$ = 25ºC) | $C_{I/O}$ |  | 10 |  | pF |  |
| Crystal Specified Load Capacitance ($T_A$ = 25ºC) |  |  | 12.5 |  | pF |  |

*Unless otherwise specified.

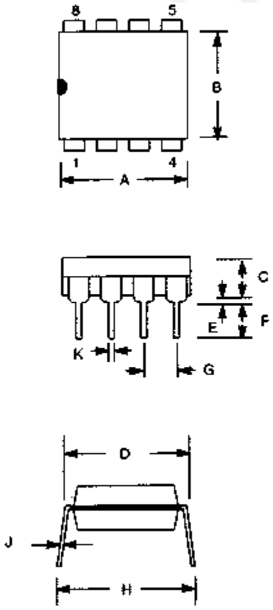
## NOTES:

1. $I_{CCS}$ specified with $V_{CC}$ = 5.0V and SDA, SCL = 5.0V.
2. $V_{CC}$ = 0V, $V_{BAT}$ = 3V.
3. After this period, the first clock pulse is generated.
4. A device must internally provide a hold time of at least 300ns for the SDA signal (referred to the $V_{IHMIN}$ of the SCL signal) in order to bridge the undefined region of the falling edge of SCL.
5. The maximum $t_{HD:DAT}$ has only to be met if the device does not stretch the LOW period ($t_{LOW}$) of the SCL signal.
6. $C_B$ – Total capacitance of one bus line in pF.
7. $I_{CCA}$ – SCL clocking at max frequency = 100kHz.
8. $V_{PF}$ measured at $V_{BAT}$ = 3.0V.

## TIMING DIAGRAM Figure 8

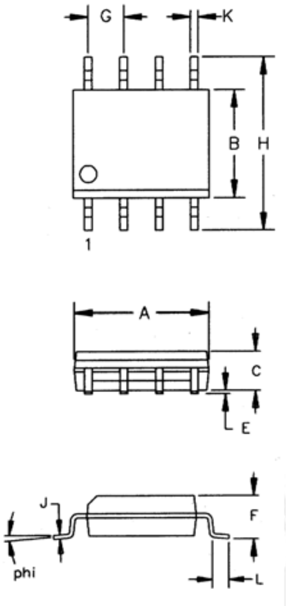


## DS1307 64 X 8 SERIAL REAL-TIME CLOCK 8-PIN DIP MECHANICAL DIMENSIONS



| PKG | 8-PIN | |
|---|---|---|
| DIM | MIN | MAX |
| A IN.<br>MM | 0.360<br>9.14 | 0.400<br>10.16 |
| B IN.<br>MM | 0.240<br>6.10 | 0.260<br>6.60 |
| C IN.<br>MM | 0.120<br>3.05 | 0.140<br>3.56 |
| D IN.<br>MM | 0.300<br>7.62 | 0.325<br>8.26 |
| E IN.<br>MM | 0.015<br>0.38 | 0.040<br>1.02 |
| F IN.<br>MM | 0.120<br>3.04 | 0.140<br>3.56 |
| G IN.<br>MM | 0.090<br>2.29 | 0.110<br>2.79 |
| H IN.<br>MM | 0.320<br>8.13 | 0.370<br>9.40 |
| J IN.<br>MM | 0.008<br>0.20 | 0.012<br>0.30 |
| K IN.<br>MM | 0.015<br>0.38 | 0.021<br>0.53 |

# DS1307Z 64 X 8 SERIAL REAL-TIME CLOCK 8-PIN SOIC (150-MIL) MECHANICAL DIMENSIONS



| PKG | 8-PIN (150 MIL) | |
|---|---|---|
| DIM | MIN | MAX |
| A IN.<br>MM | 0.188<br>4.78 | 0.196<br>4.98 |
| B IN.<br>MM | 0.150<br>3.81 | 0.158<br>4.01 |
| C IN.<br>MM | 0.048<br>1.22 | 0.062<br>1.57 |
| E IN.<br>MM | 0.004<br>0.10 | 0.010<br>0.25 |
| F IN.<br>MM | 0.053<br>1.35 | 0.069<br>1.75 |
| G IN.<br>MM | 0.050 BSC<br>1.27 BSC | |
| H IN.<br>MM | 0.230<br>5.84 | 0.244<br>6.20 |
| J IN.<br>MM | 0.007<br>0.18 | 0.011<br>0.28 |
| K IN.<br>MM | 0.012<br>0.30 | 0.020<br>0.51 |
| L IN.<br>MM | 0.016<br>0.41 | 0.050<br>1.27 |
| phi | 0° | 8° |

56-G2008-001

# XBee® / XBee-PRO® RF Modules

## Product Manual v1.xEx - 802.15.4 Protocol

For RF Module Part Numbers: XB24-A...-001, XBP24-A...-001

**IEEE® 802.15.4 RF Modules by Digi International**

Digi International Inc.
11001 Bren Road East
Minnetonka, MN 55343
877 912-3444 or 952 912-3444
http://www.digi.com

90000982_G
3/13/2012

| | | |
|---|---|---|
| Technical Support: | Phone: | (866) 765-9885 toll-free U.S.A. & Canada<br>(801) 765-9885 Worldwide<br>8:00 am - 5:00 pm [U.S. Mountain Time] |
| | Online Support: | http://www.digi.com/support/eservice/login.jsp |
| | Email: | rf-experts@digi.com |

# Contents

# 1. XBee®/XBee-PRO® RF Modules

The XBee and XBee-PRO RF Modules were engineered to meet IEEE 802.15.4 standards and support the unique needs of low-cost, low-power wireless sensor networks. The modules require minimal power and provide reliable delivery of  data between devices.

The modules operate within the ISM 2.4 GHz frequency band and are pin-for-pin compatible with each other.

## Key Features

### Long Range Data Integrity

**XBee**

- Indoor/Urban: up to 100' (30 m)
- Outdoor line-of-sight: up to 300' (90 m)
- Transmit Power: 1 mW (0 dBm)
- Receiver Sensitivity: -92 dBm

**XBee-PRO**

- Indoor/Urban: up to 300' (90 m), 200' (60 m) for International variant
- Outdoor line-of-sight: up to 1 mile (1600 m), 2500' (750 m) for International variant
- Transmit Power: 63mW (18dBm), 10mW (10dBm) for International variant
- Receiver Sensitivity: -100 dBm

RF Data Rate: 250,000 bps

### Advanced Networking & Security

Retries and Acknowledgements

DSSS (Direct Sequence Spread Spectrum)

Each direct sequence channels has over 65,000 unique network addresses available

Source/Destination Addressing

Unicast & Broadcast Communications

Point-to-point, point-to-multipoint and peer-to-peer topologies supported

Coordinator/End Device operations

Transparent and API Operations

128-bit Encryption

### Low Power

**XBee**

- TX Peak Current: 45 mA (@3.3 V)
- RX Current: 50 mA (@3.3 V)
- Power-down Current: < 10 µA

**XBee-PRO**

- TX Peak Current: 250mA (150mA for international variant)
- TX Peak Current (RPSMA module only): 340mA (180mA for international variant)
- RX Current: 55 mA (@3.3 V)
- Power-down Current: < 10 µA

### ADC and I/O line support

Analog-to-digital conversion, Digital I/O

I/O Line Passing

### Easy-to-Use

No configuration necessary for out-of box RF communications

Free X-CTU Software
(Testing and configuration software)

AT and API Command Modes for configuring module parameters

Extensive command set

Small form factor

## Worldwide Acceptance

**FCC Approval** (USA) Refer to Appendix A [p63] for FCC Requirements.
Systems that contain XBee®/XBee-PRO® RF Modules inherit Digi Certifications.

ISM (Industrial, Scientific & Medical) **2.4 GHz frequency band**

Manufactured under **ISO 9001:2000** registered standards

XBee®/XBee-PRO® RF Modules are optimized for use in the United States, Canada, Australia, Japan, and Europe. Contact Digi for complete list of government agency approvals.

## Specifications

Table 1-01. Specifications of the XBee®/XBee-PRO® RF Modules

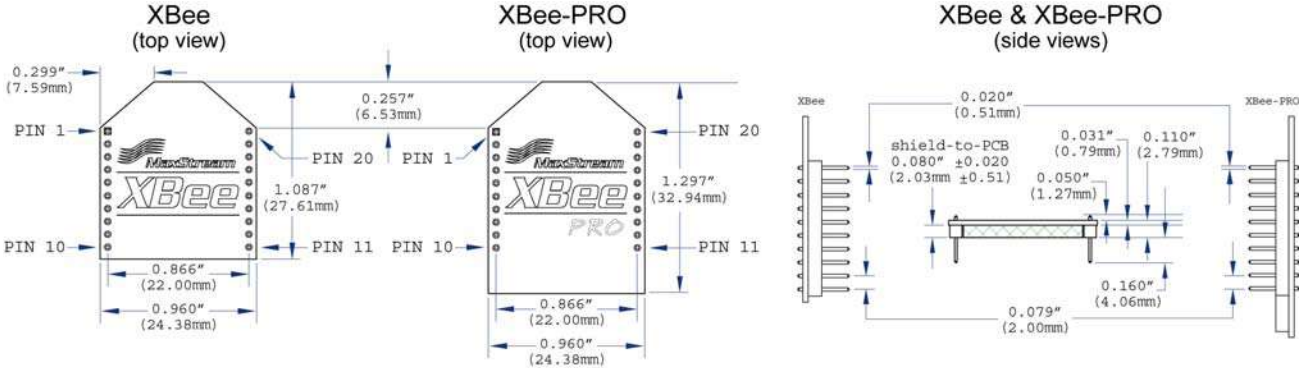
| Specification | XBee | XBee-PRO |
|---|---|---|
| Performance | | |
| Indoor/Urban Range | Up to 100 ft (30 m) | Up to 300 ft. (90 m), up to 200 ft (60 m) International variant |
| Outdoor RF line-of-sight Range | Up to 300 ft (90 m) | Up to 1 mile (1600 m), up to 2500 ft (750 m) international variant |
| Transmit Power Output (software selectable) | 1mW (0 dBm) | 63mW (18dBm)*<br>10mW (10 dBm) for International variant |
| RF Data Rate | 250,000 bps | 250,000 bps |
| Serial Interface Data Rate (software selectable) | 1200 bps - 250 kbps (non-standard baud rates also supported) | 1200 bps - 250 kbps (non-standard baud rates also supported) |
| Receiver Sensitivity | -92 dBm (1% packet error rate) | -100 dBm (1% packet error rate) |
| Power Requirements | | |
| Supply Voltage | 2.8 – 3.4 V | 2.8 – 3.4 V |
| Transmit Current (typical) | 45mA (@ 3.3 V) | 250mA (@3.3 V) (150mA for international variant) RPSMA module only: 340mA (@3.3 V) (180mA for international variant) |
| Idle / Receive Current (typical) | 50mA (@ 3.3 V) | 55mA (@ 3.3 V) |
| Power-down Current | < 10 µA | < 10 µA |
| General | | |
| Operating Frequency | ISM 2.4 GHz | ISM 2.4 GHz |
| Dimensions | 0.960” x 1.087” (2.438cm x 2.761cm) | 0.960” x 1.297” (2.438cm x 3.294cm) |
| Operating Temperature | -40 to 85º C (industrial) | -40 to 85º C (industrial) |
| Antenna Options | Integrated Whip Antenna, Embedded PCB Antenna, U.FL Connector, RPSMA connector | Integrated Whip Antenna, Embedded PCB Antenna, U.FL Connector, RPSMA connector |
| Networking & Security | | |
| Supported Network Topologies | Point-to-point, Point-to-multipoint & Peer-to-peer | |
| Number of Channels (software selectable) | 16 Direct Sequence Channels | 12 Direct Sequence Channels |
| Addressing Options | PAN ID, Channel and Addresses | PAN ID, Channel and Addresses |
| Agency Approvals | | |
| United States (FCC Part 15.247) | OUR-XBEE | OUR-XBEEPRO |
| Industry Canada (IC) | 4214A XBEE | 4214A XBEEPRO |
| Europe (CE) | ETSI | ETSI (Max. 10 dBm transmit power output)* |
| Japan | R201WW07215214 | R201WW08215111 (Max. 10 dBm transmit power output)*<br>Wire, chip, RPMSA, and U.FL versions are certified for Japan. PCB antenna version is not. |
| Australia | C-Tick | C-Tick |

* See Appendix A for region-specific certification requirements.

Antenna Options: The ranges specified are typical when using the integrated Whip (1.5 dBi) and Dipole (2.1 dBi) antennas. The PCB antenna option provides advantages in its form factor; however, it typically yields shorter range than the Whip and Dipole antenna options when transmitting outdoors.For more information, refer to the "XBee Antennas" Knowledgebase Article located on Digi's Support Web site

## Mechanical Drawings

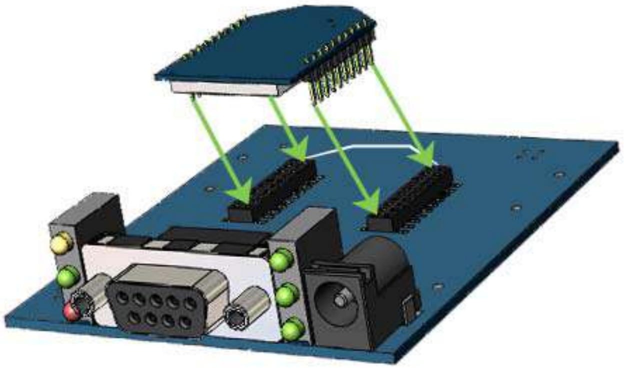
**Figure 1-01. Mechanical drawings of the XBee®/XBee-PRO® RF Modules (antenna options not shown)**
The XBee and XBee-PRO RF Modules are pin-for-pin compatible.



## Mounting Considerations

The XBee®/XBee-PRO® RF Module was designed to mount into a receptacle (socket) and therefore does not require any soldering when mounting it to a board. The XBee Development Kits contain RS-232 and USB interface boards which use two 20-pin receptacles to receive modules.

**Figure 1-02. XBee Module Mounting to an RS-232 Interface Board**.



The receptacles used on Digi development boards are manufactured by Century Interconnect. Several other manufacturers provide comparable mounting solutions; however, Digi currently uses the following receptacles:

- Through-hole single-row receptacles -
  Samtec P/N: MMS-110-01-L-SV (or equivalent)
- Surface-mount double-row receptacles -
  Century Interconnect P/N: CPRMSL20-D-0-1 (or equivalent)
- Surface-mount single-row receptacles -
  Samtec P/N: SMM-110-02-SM-S

Digi also recommends printing an outline of the module on the board to indicate the orientation the module should be mounted.

## Pin Signals

**Figure 1-03. XBee®/XBee-PRO® RF Module Pin Numbers**

(top sides shown - shields on bottom)

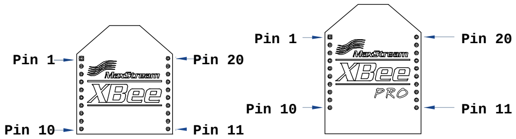


**Table 1-02. Pin Assignments for the XBee and XBee-PRO Modules**
(Low-asserted signals are distinguished with a horizontal line above signal name.)

| Pin # | Name | Direction | Description |
|---|---|---|---|
| 1 | VCC | - | Power supply |
| 2 | DOUT | Output | UART Data Out |
| 3 | DIN / $\overline{\text{CONFIG}}$ | Input | UART Data In |
| 4 | DO8* | Output | Digital Output 8 |
| 5 | $\overline{\text{RESET}}$ | Input | Module Reset (reset pulse must be at least 200 ns) |
| 6 | PWM0 / RSSI | Output | PWM Output 0 / RX Signal Strength Indicator |
| 7 | PWM1 | Output | PWM Output 1 |
| 8 | [reserved] | - | Do not connect |
| 9 | $\overline{\text{DTR}}$ / SLEEP_RQ / DI8 | Input | Pin Sleep Control Line or Digital Input 8 |
| 10 | GND | - | Ground |
| 11 | AD4 / DIO4 | Either | Analog Input 4 or Digital I/O 4 |
| 12 | $\overline{\text{CTS}}$ / DIO7 | Either | Clear-to-Send Flow Control or Digital I/O 7 |
| 13 | ON / $\overline{\text{SLEEP}}$ | Output | Module Status Indicator |
| 14 | VREF | Input | Voltage Reference for A/D Inputs |
| 15 | Associate / AD5 / DIO5 | Either | Associated Indicator, Analog Input 5 or Digital I/O 5 |
| 16 | $\overline{\text{RTS}}$ / DIO6 | Either | Request-to-Send Flow Control, or Digital I/O 6 |
| 17 | AD3 / DIO3 | Either | Analog Input 3 or Digital I/O 3 |
| 18 | AD2 / DIO2 | Either | Analog Input 2 or Digital I/O 2 |
| 19 | AD1 / DIO1 | Either | Analog Input 1 or Digital I/O 1 |
| 20 | AD0 / DIO0 | Either | Analog Input 0 or Digital I/O 0 |

* Function is not supported at the time of this release

**Notes:**

- Minimum connections: VCC, GND, DOUT & DIN
- Minimum connections for updating firmware: VCC, GND, DIN, DOUT, RTS & DTR
- Signal Direction is specified with respect to the module
- Module includes a 50k Ω pull-up resistor attached to $\overline{\text{RESET}}$
- Several of the input pull-ups can be configured using the PR command
- Unused pins should be left disconnected

## Design Notes

The XBee modules do not specifically require any external circuitry or specific connections for proper operation. However, there are some general design guidelines that are recommended for help in troubleshooting and building a robust design.

### Power Supply Design

Poor power supply can lead to poor radio performance, especially if the supply voltage is not kept within tolerance or is excessively noisy. To help reduce noise, we recommend placing a 1.0 µF and 8.2 pF capacitor as near as possible to pin 1 on the XBee. If using a switching regulator for the power supply, switching frequencies above 500 kHz are preferred. Power supply ripple should be limited to a maximum 100 mV peak to peak.

## Recommended Pin Connections

The only required pin connections are VCC, GND, DOUT and DIN. To support serial firmware updates, VCC, GND, DOUT, DIN, RTS, and DTR should be connected.

All unused pins should be left disconnected. All inputs on the radio can be pulled high with internal pull-up resistors using the PR software command. No specific treatment is needed for unused outputs.

Other pins may be connected to external circuitry for convenience of operation including the Associate LED pin (pin 15) and the commissioning button pin (pin 20). The Associate LED will flash differently depending on the state of the module, and a pushbutton attached to pin 20 can enable various deployment and troubleshooting functions without having to send UART commands.

If analog sampling is desired, VRef (pin 14) should be attached to a voltage reference.

## Board Layout

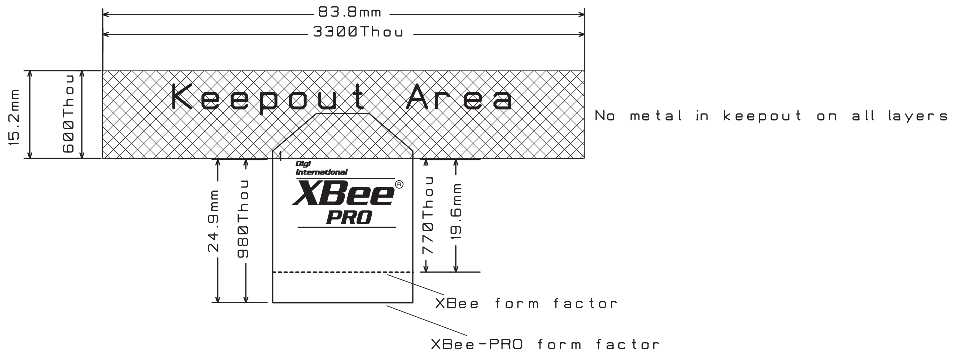
XBee modules are designed to be self sufficient and have minimal sensitivity to nearby processors, crystals or other PCB components. As with all PCB designs, Power and Ground traces should be thicker than signal traces and able to comfortably support the maximum current specifications. No other special PCB design considerations are required for integrating XBee radios except in the antenna section.

## Antenna Performance

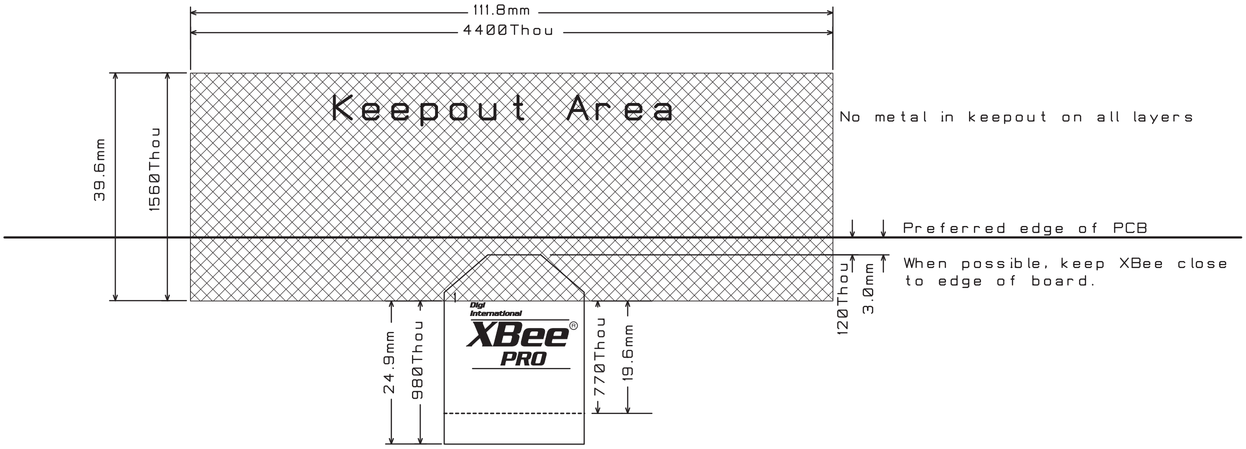
Antenna location is an important consideration for optimal performance. In general, antennas radiate and receive best perpendicular to the direction they point. Thus a vertical antenna's radiation pattern is strongest across the horizon. Metal objects near the antenna may impede the radiation pattern. Metal objects between the transmitter and receiver can block the radiation path or reduce the transmission distance, so antennas should be positioned away from them when possible. Some objects that are often overlooked are metal poles, metal studs or beams in structures, concrete (it is usually reinforced with metal rods), vehicles, elevators, ventilation ducts, refrigerators, microwave ovens, batteries, and tall electrolytic capacitors. If the XBee is to be placed inside a metal enclosure, an external antenna should be used.

XBee units with the Embedded PCB Antenna should not be placed inside a metal enclosure or have any ground planes or metal objects above or below the antenna. For best results, place the XBee at the edge of the host PCB on which it is mounted. Ensure that the ground, power and signal planes are vacant immediately below the antenna section. Digi recommends allowing a “keepout” area, which is shown in detail on the next page.

Minimum Keepout Area (All PCB Layers)



Recommended Keepout Area (All PCB Layers)



The antenna performance improves with a larger keepout area.

Notes:

1. Non-metal enclosures are recommended. For metal enclosures, an external antenna should be used.
2. Metal chassis or mounting structures in the keepout area should be at least 1 inch(2.54 cm) from antenna.
3. Maximize distance between antenna and metal objects that might be mounted in keepout area.
4. These keepout area guidelines do not apply for Wire Whip antennas or external RF connectors. Wire Whip antennas radiate best over the center of a ground plane.

# Electrical Characteristics

Table 1-03. DC Characteristics (VCC = 2.8 - 3.4 VDC)

| Symbol | Characteristic | Condition | Min | Typical | Max | Unit |
|---|---|---|---|---|---|---|
| $V_{IL}$ | Input Low Voltage | All Digital Inputs | - | - | 0.35 * VCC | V |
| $V_{IH}$ | Input High Voltage | All Digital Inputs | 0.7 * VCC | - | - | V |
| $V_{OL}$ | Output Low Voltage | $I_{OL}$ = 2 mA, VCC >= 2.7 V | - | - | 0.5 | V |
| $V_{OH}$ | Output High Voltage | $I_{OH}$ = -2 mA, VCC >= 2.7 V | VCC - 0.5 | - | - | V |
| $II_{IN}$ | Input Leakage Current | $V_{IN}$ = VCC or GND, all inputs, per pin | - | 0.025 | 1 | µA |
| $II_{OZ}$ | High Impedance Leakage Current | $V_{IN}$ = VCC or GND, all I/O High-Z, per pin | - | 0.025 | 1 | µA |
| TX | Transmit Current | VCC = 3.3 V | - | 45 (XBee)<br>215, 140 (PRO, Int) | - | mA |
| RX | Receive Current | VCC = 3.3 V | - | 50 (XBee)<br>55 (PRO) | - | mA |
| PWR-DWN | Power-down Current | SM parameter = 1 | - | < 10 | - | µA |

Table 1-04. ADC Characteristics (Operating)

| Symbol | Characteristic | Condition | Min | Typical | Max | Unit |
|---|---|---|---|---|---|---|
| $V_{REFH}$ | VREF - Analog-to-Digital converter reference range | | 2.08 | - | $V_{DDAD}$* | V |
| $I_{REF}$ | VREF - Reference Supply Current | Enabled | - | 200 | - | µA |
| | | Disabled or Sleep Mode | - | < 0.01 | 0.02 | µA |
| $V_{INDC}$ | Analog Input Voltage[1] | | $V_{SSAD}$ - 0.3 | - | $V_{DDAD}$ + 0.3 | V |

1. Maximum electrical operating range, not valid conversion range.

* $V_{DDAD}$ is connected to VCC.

Table 1-05. ADC Timing/Performance Characteristics[1]

| Symbol | Characteristic | Condition | Min | Typical | Max | Unit |
|---|---|---|---|---|---|---|
| $R_{AS}$ | Source Impedance at Input[2] | | - | - | 10 | kΩ |
| $V_{AIN}$ | Analog Input Voltage[3] | | $V_{REFL}$ | | $V_{REFH}$ | V |
| RES | Ideal Resolution (1 LSB)[4] | 2.08V ≤ $V_{DDAD}$ ≤ 3.6V | 2.031 | - | 3.516 | mV |
| DNL | Differential Non-linearity[5] | | - | ±0.5 | ±1.0 | LSB |
| INL | Integral Non-linearity[6] | | - | ±0.5 | ±1.0 | LSB |
| $E_{ZS}$ | Zero-scale Error[7] | | - | ±0.4 | ±1.0 | LSB |
| $F_{FS}$ | Full-scale Error[8] | | - | ±0.4 | ±1.0 | LSB |
| $E_{IL}$ | Input Leakage Error[9] | | - | ±0.05 | ±5.0 | LSB |
| $E_{TU}$ | Total Unadjusted Error[10] | | - | ±1.1 | ±2.5 | LSB |

1. All ACCURACY numbers are based on processor and system being in WAIT state (very little activity and no IO switching) and that adequate low-pass filtering is present on analog input pins (filter with 0.01 µF to 0.1 µF capacitor between analog input and VREFL). Failure to observe these guidelines may result in system or microcontroller noise causing accuracy errors which will vary based on board layout and the type and magnitude of the activity.

Data transmission and reception during data conversion may cause some degradation of these specifications, depending on the number and timing of packets. It is advisable to test the ADCs in your installation if best accuracy is required.

2. $R_{AS}$ is the real portion of the impedance of the network driving the analog input pin. Values greater than this amount may not fully charge the input circuitry of the ATD resulting in accuracy error.

3. Analog input must be between $V_{REFL}$ and $V_{REFH}$ for valid conversion. Values greater than $V_{REFH}$ will convert to $3FF.

4. The resolution is the ideal step size or 1LSB = $(V_{REFH}–V_{REFL})/1024$

5. Differential non-linearity is the difference between the current code width and the ideal code width (1LSB). The current code width is the difference in the transition voltages to and from the current code.

6. Integral non-linearity is the difference between the transition voltage to the current code and the adjusted ideal transition voltage for the current code. The adjusted ideal transition voltage is $(\text{Current Code}–1/2)*(1/((V_{REFH}+E_{FS})–(V_{REFL}+E_{ZS})))$.

7. Zero-scale error is the difference between the transition to the first valid code and the ideal transition to that code. The Ideal transition voltage to a given code is $(\text{Code}–1/2)*(1/(V_{REFH}–V_{REFL}))$.

8. Full-scale error is the difference between the transition to the last valid code and the ideal transition to that code. The ideal transition voltage to a given code is $(\text{Code}–1/2)*(1/(V_{REFH}–V_{REFL}))$.

9. Input leakage error is error due to input leakage across the real portion of the impedance of the network driving the analog pin. Reducing the impedance of the network reduces this error.

10. Total unadjusted error is the difference between the transition voltage to the current code and the ideal straight-line transfer function. This measure of error includes inherent quantization error (1/2LSB) and circuit error (differential, integral, zero-scale, and full-scale) error. The specified value of $E_{TU}$ assumes zero $E_{IL}$ (no leakage or zero real source impedance).

# 2. RF Module Operation

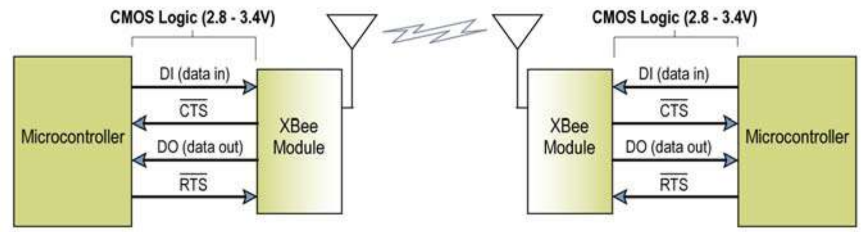
## Serial Communications

The XBee®/XBee-PRO® RF Modules interface to a host device through a logic-level asynchronous serial port. Through its serial port, the module can communicate with any logic and voltage compatible UART; or through a level translator to any serial device (For example: Through a Digi proprietary RS-232 or USB interface board).

### UART Data Flow

Devices that have a UART interface can connect directly to the pins of the RF module as shown in the figure below.

**Figure 2-01. System Data Flow Diagram in a UART-interfaced environment**
(Low-asserted signals distinguished with horizontal line over signal name.)

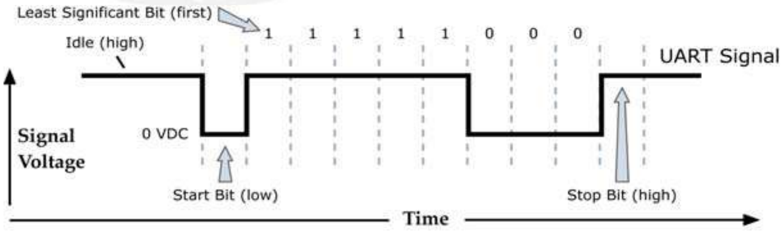


#### Serial Data

Data enters the module UART through the DI pin (pin 3) as an asynchronous serial signal. The signal should idle high when no data is being transmitted.

Each data byte consists of a start bit (low), 8 data bits (least significant bit first) and a stop bit (high). The following figure illustrates the serial bit pattern of data passing through the module.

**Figure 2-02. UART data packet 0x1F (decimal number "31") as transmitted through the RF module**
Example Data Format is 8-N-1 (bits - parity - # of stop bits)



Serial communications depend on the two UARTs (the microcontroller's and the RF module's) to be configured with compatible settings (baud rate, parity, start bits, stop bits, data bits).

The UART baud rate and parity settings on the XBee module can be configured with the BD and NB commands, respectively. See the command table in Chapter 3 for details.

## Transparent Operation

By default, XBee®/XBee-PRO® RF Modules operate in Transparent Mode. When operating in this mode, the modules act as a serial line replacement - all UART data received through the DI pin is queued up for RF transmission. When RF data is received, the data is sent out the DO pin.

### Serial-to-RF Packetization

Data is buffered in the DI buffer until one of the following causes the data to be packetized and transmitted:

1. No serial characters are received for the amount of time determined by the RO (Packetization Timeout) parameter. If RO = 0, packetization begins when a character is received.
2. The maximum number of characters that will fit in an RF packet (100) is received.
3. The Command Mode Sequence (GT + CC + GT) is received. Any character buffered in the DI buffer before the sequence is transmitted.

If the module cannot immediately transmit (for instance, if it is already receiving RF data), the serial data is stored in the DI Buffer. The data is packetized and sent at any RO timeout or when 100 bytes (maximum packet size) are received.

If the DI buffer becomes full, hardware or software flow control must be implemented in order to prevent overflow (loss of data between the host and module).

## API Operation

API (Application Programming Interface) Operation is an alternative to the default Transparent Operation. The frame-based API extends the level to which a host application can interact with the networking capabilities of the module.

When in API mode, all data entering and leaving the module is contained in frames that define operations or events within the module.

Transmit Data Frames (received through the DI pin (pin 3)) include:

- RF Transmit Data Frame
- Command Frame (equivalent to AT commands)

Receive Data Frames (sent out the DO pin (pin 2)) include:

- RF-received data frame
- Command response
- Event notifications such as reset, associate, disassociate, etc.

The API provides alternative means of configuring modules and routing data at the host application layer. A host application can send data frames to the module that contain address and payload information instead of using command mode to modify addresses. The module will send data frames to the application containing status packets; as well as source, RSSI and payload information from received data packets.

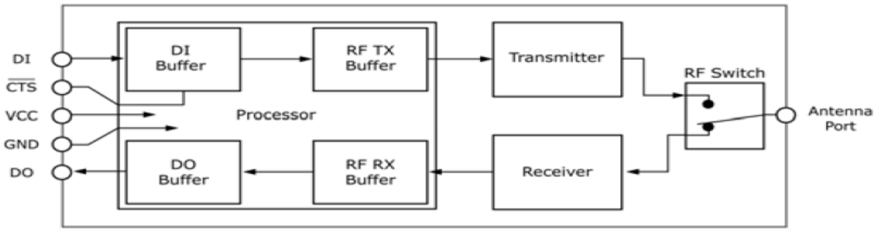
The API operation option facilitates many operations such as the examples cited below:

-> Transmitting data to multiple destinations without entering Command Mode
-> Receive success/failure status of each transmitted RF packet
-> Identify the source address of each received packet

To implement API operations, refer to API sections [p56].

## Flow Control

Figure 2-03. Internal Data Flow Diagram



### DI (Data In) Buffer

When serial data enters the RF module through the DI pin (pin 3), the data is stored in the DI Buffer until it can be processed.

**Hardware Flow Control (CTS).** When the DI buffer is 17 bytes away from being full; by default, the module de-asserts CTS (high) to signal to the host device to stop sending data [refer to D7 (DIO7 Configuration) parameter]. CTS is re-asserted after the DI Buffer has 34 bytes of memory available.

**How to eliminate the need for flow control:**

1. Send messages that are smaller than the DI buffer size (202 bytes).
2. Interface at a lower baud rate [BD (Interface Data Rate) parameter] than the throughput data rate.

**Case in which the DI Buffer may become full and possibly overflow:**

If the module is receiving a continuous stream of RF data, any serial data that arrives on the DI pin is placed in the DI Buffer. The data in the DI buffer will be transmitted over-the-air when the module is no longer receiving RF data in the network.

Refer to the RO (Packetization Timeout), BD (Interface Data Rate) and D7 (DIO7 Configuration) command descriptions for more information.

### DO (Data Out) Buffer

When RF data is received, the data enters the DO buffer and is sent out the serial port to a host device. Once the DO Buffer reaches capacity, any additional incoming RF data is lost.

**Hardware Flow Control (RTS).** If RTS is enabled for flow control (D6 (DIO6 Configuration) Parameter = 1), data will not be sent out the DO Buffer as long as RTS (pin 16) is de-asserted.

**Two cases in which the DO Buffer may become full and possibly overflow:**

1. If the RF data rate is set higher than the interface data rate of the module, the module will receive data from the transmitting module faster than it can send the data to the host.
2. If the host does not allow the module to transmit data out from the DO buffer because of being held off by hardware or software flow control.

Refer to the D6 (DIO6 Configuration) command description for more information.

## ADC and Digital I/O Line Support

The XBee®/XBee-PRO® RF Modules support ADC (Analog-to-digital conversion) and digital I/O line passing. The following pins support multiple functions:

**Table 2-01. Pin functions and their associated pin numbers and commands**
AD = Analog-to-Digital Converter, DIO = Digital Input/Output
Pin functions not applicable to this section are denoted within (parenthesis).

| Pin Function | Pin# | AT Command |
|---|---|---|
| AD0 / DIO0 | 20 | D0 |
| AD1 / DIO1 | 19 | D1 |
| AD2 / DIO2 | 18 | D2 |
| AD3 / DIO3 / (COORD_SEL) | 17 | D3 |
| AD4 / DIO4 | 11 | D4 |
| AD5 / DIO5 / (ASSOCIATE) | 15 | D5 |
| DIO6 / (RTS) | 16 | D6 |
| DIO7 / (CTS) | 12 | D7 |
| DI8 / (DTR) / (Sleep_RQ) | 9 | D8 |

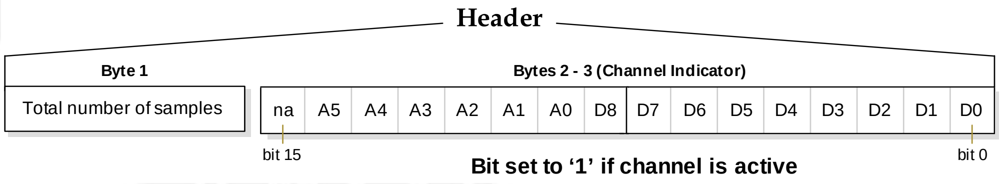
To enable ADC and DIO pin functions:

| | |
|---|---|
| For ADC Support: | Set ATDn = 2 |
| For Digital Input support: | Set ATDn = 3 |
| For Digital Output Low support: | Set ATDn = 4 |
| For Digital Output High support: | Set ATDn = 5 |

### I/O Data Format

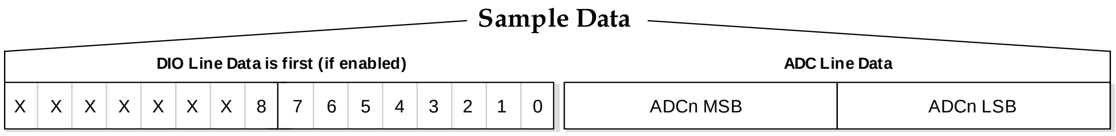
I/O data begins with a header. The first byte of the header defines the number of samples forthcoming. The last 2 bytes of the header (Channel Indicator) define which inputs are active. Each bit represents either a DIO line or ADC channel.

**Figure 2-04. Header**



Sample data follows the header and the channel indicator frame is used to determine how to read the sample data. If any of the DIO lines are enabled, the first 2 bytes are the DIO sample. The ADC data follows. ADC channel data is represented as an unsigned 10-bit value right-justified on a 16- bit boundary.

**Figure 2-05. Sample Data**



## API Support

I/O data is sent out the UART using an API frame. All other data can be sent and received using Transparent Operation [refer to p12] or API framing if API mode is enabled (AP > 0).

API Operations support two RX (Receive) frame identifiers for I/O data (set 16-bit address to 0xFFFE and the module will do 64-bit addressing):

- 0x82 for RX (Receive) Packet: 64-bit address I/O
- 0x83 for RX (Receive) Packet: 16-bit address I/O

The API command header is the same as shown in the “RX (Receive) Packet: 64-bit Address” and “RX (Receive) Packet: 16-bit Address” API types [refer to p62]. RX data follows the format described in the I/O Data Format section [p14].

**Applicable Commands:** AP (API Enable)

## Sleep Support

Automatic wakeup sampling can be suppressed by setting SO bit 1.When an RF module wakes, it will always do a sample based on any active ADC or DIO lines. This allows sampling based on the sleep cycle whether it be Cyclic Sleep (SM parameter = 4 or 5) or Pin Sleep (SM = 1 or 2). To gather more samples when awake, set the IR (Sample Rate) parameter.

For Cyclic Sleep modes: If the IR parameter is set, the module will stay awake until the IT (Samples before TX) parameter is met. The module will stay awake for ST (Time before Sleep) time.

**Applicable Commands:** IR (Sample Rate), IT (Samples before TX), SM (Sleep Mode), IC (DIO Change Detect), SO (Sleep Options)

## DIO Pin Change Detect

When “DIO Change Detect” is enabled (using the IC command), DIO lines 0-7 are monitored. When a change is detected on a DIO line, the following will occur:

1. An RF packet is sent with the updated DIO pin levels. This packet will not contain any ADC samples.
2. Any queued samples are transmitted before the change detect data. This may result in receiving a packet with less than IT (Samples before TX) samples.

Note: Change detect will not affect Pin Sleep wake-up. The D8 pin (DTR/Sleep_RQ/DI8) is the only line that will wake a module from Pin Sleep. If not all samples are collected, the module will still enter Sleep Mode after a change detect packet is sent.

**Applicable Commands**: IC (DIO Change Detect), IT (Samples before TX)

NOTE: Change detect is only supported when the Dx (DIOx Configuration) parameter equals 3,4 or 5.

## Sample Rate (Interval)

The Sample Rate (Interval) feature allows enabled ADC and DIO pins to be read periodically on modules that are not configured to operate in Sleep Mode. When one of the Sleep Modes is enabled and the IR (Sample Rate) parameter is set, the module will stay awake until IT (Samples before TX) samples have been collected.

Once a particular pin is enabled, the appropriate sample rate must be chosen. The maximum sample rate that can be achieved while using one A/D line is 1 sample/ms or 1 KHz (Note that the modem will not be able to keep up with transmission when IR & IT are equal to “1” and that configuring the modem to sample at rates greater than once every 20ms is not recommended).

**Applicable Commands**: IR (Sample Rate), IT (Samples before TX), SM (Sleep Mode)

## I/O Line Passing

Virtual wires can be set up between XBee®/XBee-PRO® Modules. When an RF data packet is received that contains I/O data, the receiving module can be setup to update any enabled outputs (PWM and DIO) based on the data it receives.

Note that I/O lines are mapped in pairs. For example: AD0 can only update PWM0 and DI5 can only update DO5. The default setup is for outputs not to be updated, which results in the I/O data being sent out the UART (refer to the IU (Enable I/O Output) command). To enable the outputs to be updated, the IA (I/O Input Address) parameter must be setup with the address of the module that has the appropriate inputs enabled. This effectively binds the outputs to a particular module's input. This does not affect the ability of the module to receive I/O line data from other modules - only its ability to update enabled outputs. The IA parameter can also be setup to accept I/O data for output changes from any module by setting the IA parameter to 0xFFFF.

When outputs are changed from their non-active state, the module can be setup to return the output level to it non-active state. The timers are set using the Tn (Dn Output Timer) and PT (PWM Output Timeout) commands. The timers are reset every time a valid I/O packet (passed IA check) is received. The IC (Change Detect) and IR (Sample Rate) parameters can be setup to keep the output set to their active output if the system needs more time than the timers can handle.

Note: DI8 cannot be used for I/O line passing.

**Applicable Commands:** IA (I/O Input Address), Tn (Dn Output Timeout), P0 (PWM0 Configuration), P1 (PWM1 Configuration), M0 (PWM0 Output Level), M1 (PWM1 Output Level), PT (PWM Output Timeout), RP (RSSSI PWM Timer)

## Configuration Example

As an example for a simple A/D link, a pair of RF modules could be set as follows:

| **Remote Configuration** | **Base Configuration** |
|---|---|
| DL = 0x1234 | DL = 0x5678 |
| MY = 0x5678 | MY = 0x1234 |
| D0 = 2 | P0 = 2 |
| D1 = 2 | P1 = 2 |
| IR = 0x14 | IU = 1 |
| IT = 5 | IA = 0x5678 (or 0xFFFF) |

These settings configure the remote module to sample AD0 and AD1 once each every 20 ms. It then buffers 5 samples each before sending them back to the base module. The base should then receive a 32-Byte transmission (20 Bytes data and 12 Bytes framing) every 100 ms.

## XBee®/XBee-PRO® Networks

The following terms will be used to explicate the network operations:

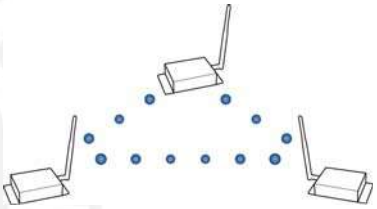
Table 2-02. Terms and definitions

| Term | Definition |
|---|---|
| PAN | Personal Area Network - A data communication network that includes one or more End Devices and optionally a Coordinator. |
| Coordinator | A Full-function device (FFD) that provides network synchronization by polling nodes [NonBeacon (w/ Coordinator) networks only] |
| End Device | *When in the same network as a Coordinator* - RF modules that rely on a Coordinator for synchronization and can be put into states of sleep for low-power applications. |
| Association | The establishment of membership between End Devices and a Coordinator. Association is only applicable in NonBeacon (w/Coordinator) networks. |

### Peer-to-Peer

By default, XBee®/XBee-PRO RF Modules are configured to operate within a Peer-to-Peer network topology and therefore are not dependent upon Master/Slave relationships. NonBeacon systems operate within a Peer-to-Peer network topology and therefore are not dependent upon Master/Slave relationships. This means that modules remain synchronized without use of master/server configurations and each module in the network shares both roles of master and slave. Digi's peer-to-peer architecture features fast synchronization times and fast cold start times. This default configuration accommodates a wide range of RF data applications.

Figure 2-06. Peer-to-Peer Architecture



A peer-to-peer network can be established by configuring each module to operate as an End Device (CE = 0), disabling End Device Association on all modules (A1 = 0) and setting ID and CH parameters to be identical across the network.

### NonBeacon (w/ Coordinator)

A device is configured as a Coordinator by setting the CE (Coordinator Enable) parameter to “1”. Coordinator power-up is governed by the A2 (Coordinator Association) parameter.

In a Coordinator system, the Coordinator can be configured to use direct or indirect transmissions. If the SP (Cyclic Sleep Period) parameter is set to “0”, the Coordinator will send data immediately. Otherwise, the SP parameter determines the length of time the Coordinator will retain the data before discarding it. Generally, SP (Cyclic Sleep Period) and ST (Time before Sleep) parameters should be set to match the SP and ST settings of the End Devices.

## Association

Association is the establishment of membership between End Devices and a Coordinator. The establishment of membership is useful in scenarios that require a central unit (Coordinator) to relay messages to or gather data from several remote units (End Devices), assign channels or assign PAN IDs.

An RF data network that consists of one Coordinator and one or more End Devices forms a PAN (Personal Area Network). Each device in a PAN has a PAN Identifier [ID (PAN ID) parameter]. PAN IDs must be unique to prevent miscommunication between PANs. The Coordinator PAN ID is set using the ID (PAN ID) and A2 (Coordinator Association) commands.

An End Device can associate to a Coordinator without knowing the address, PAN ID or channel of the Coordinator. The A1 (End Device Association) parameter bit fields determine the flexibility of an End Device during association. The A1 parameter can be used for an End Device to dynamically set its destination address, PAN ID and/or channel.

**For example**: If the PAN ID of a Coordinator is known, but the operating channel is not; the A1 command on the End Device should be set to enable the 'Auto_Associate' and 'Reassign_Channel' bits. Additionally, the ID parameter should be set to match the PAN ID of the associated Coordinator.

### Coordinator / End Device Setup and Operation

To configure a module to operate as a Coordinator, set the CE (Coordinator Enable) parameter to '1'. Set the CE parameter of End Devices to '0' (default). Coordinator and End Devices should contain matching firmware versions.

**NonBeacon (w/ Coordinator) Systems**

The Coordinator can be configured to use direct or indirect transmissions. If the SP (Cyclic Sleep Period) parameter is set to '0', the Coordinator will send data immediately. Otherwise, the SP parameter determines the length of time the Coordinator will retain the data before discarding it. Generally, SP (Cyclic Sleep Period) and ST (Time before Sleep) parameters should be set to match the SP and ST settings of the End Devices.

### Coordinator Start-up

Coordinator power-up is governed by the A2 (Coordinator Association) command. On power-up, the Coordinator undergoes the following sequence of events:

**1. Check A2 parameter- Reassign_PANID Flag**

**Set (bit 0 = 1)** - The Coordinator issues an Active Scan. The Active Scan selects one channel and transmits a request to the broadcast address (0xFFFF) and broadcast PAN ID (0xFFFF). It then listens on that channel for beacons from any Coordinator operating on that channel. The listen time on each channel is determined by the SD (Scan Duration) parameter value.

Once the time expires on that channel, the Active Scan selects another channel and again transmits the BeaconRequest as before. This process continues until all channels have been scanned, or until 5 PANs have been discovered. When the Active Scan is complete, the results include a list of PAN IDs and Channels that are being used by other PANs. This list is used to assign an unique PAN ID to the new Coordinator. The ID parameter will be retained if it is not found in the Active Scan results. Otherwise, the ID (PAN ID) parameter setting will be updated to a PAN ID that was not detected.

**Not Set (bit 0 = 0)** - The Coordinator retains its ID setting. No Active Scan is performed.

**2. Check A2 parameter - Reassign_Channel Flag (bit 1)**

**Set (bit 1 = 1)** - The Coordinator issues an Energy Scan. The Energy Scan selects one channel and scans for energy on that channel. The duration of the scan is specified by the SD (Scan Duration) parameter. Once the scan is completed on a channel, the Energy Scan selects the next channel and begins a new scan on that channel. This process continues until all channels have been scanned.

When the Energy Scan is complete, the results include the maximal energy values detected on each channel. This list is used to determine a channel where the least energy was detected. If an Active Scan was performed (Reassign_PANID Flag set), the channels used by the detected PANs are eliminated as possible channels. Thus, the results of the Energy Scan and the Active Scan (if performed) are used to find the best channel (channel with the least energy that is not used by any detected PAN). Once the best channel has been selected, the CH (Channel) parameter value is updated to that channel.

**Not Set (bit 1 = 0)** - The Coordinator retains its CH setting. An Energy Scan is not performed.

**3. Start Coordinator**

The Coordinator starts on the specified channel (CH parameter) and PAN ID (ID parameter). Note, these may be selected in steps 1 and/or 2 above. The Coordinator will only allow End Devices to associate to it if the A2 parameter “AllowAssociation” flag is set. Once the Coordinator has successfully started, the Associate LED will blink 1 time per second. (The LED is solid if the Coordinator has not started.)

**4. Coordinator Modifications**

Once a Coordinator has started:
Modifying the A2 (Reassign_Channel or Reassign_PANID bits), ID, CH or MY parameters will cause the Coordinator’s MAC to reset (The Coordinator RF module (including volatile RAM) is not reset). Changing the A2 AllowAssociation bit will not reset the Coordinator’s MAC. In a non-beaconing system, End Devices that associated to the Coordinator prior to a MAC reset will have knowledge of the new settings on the Coordinator. Thus, if the Coordinator were to change its ID, CH or MY settings, the End Devices would no longer be able to communicate with the non-beacon Coordinator. Once a Coordinator has started, the ID, CH, MY or A2 (Reassign_Channel or Reassign_PANID bits) should not be changed.

### End Device Start-up

End Device power-up is governed by the A1 (End Device Association) command. On power-up, the End Device undergoes the following sequence of events:

**1. Check A1 parameter - AutoAssociate Bit**

**Set (bit 2 = 1)** - End Device will attempt to associate to a Coordinator. (refer to steps 2-3).

**Not Set (bit 2 = 0)** - End Device will not attempt to associate to a Coordinator. The End Device will operate as specified by its ID, CH and MY parameters. Association is considered complete and the Associate LED will blink quickly (5 times per second). When the AutoAssociate bit is not set, the remaining steps (2-3) do not apply.

**2. Discover Coordinator (if Auto-Associate Bit Set)**

The End Device issues an Active Scan. The Active Scan selects one channel and transmits a BeaconRequest command to the broadcast address (0xFFFF) and broadcast PAN ID (0xFFFF). It then listens on that channel for beacons from any Coordinator operating on that channel. The listen time on each channel is determined by the SD parameter.

Once the time expires on that channel, the Active Scan selects another channel and again transmits the BeaconRequest command as before. This process continues until all channels have been scanned, or until 5 PANs have been discovered. When the Active Scan is complete, the results include a list of PAN IDs and Channels that are being used by detected PANs.

The End Device selects a Coordinator to associate with according to the A1 parameter “Reassign_PANID” and “Reassign_Channel” flags:

**Reassign_PANID Bit Set (bit 0 = 1)**- End Device can associate with a PAN with any ID value.

**Reassign_PANID Bit Not Set (bit 0 = 0)** - End Device will only associate with a PAN whose ID setting matches the ID setting of the End Device.

**Reassign_Channel Bit Set (bit 1 = 1)** - End Device can associate with a PAN with any CH value.

**Reassign_Channel Bit Not Set (bit 1 = 0)**- End Device will only associate with a PAN whose CH setting matches the CH setting of the End Device.

After applying these filters to the discovered Coordinators, if multiple candidate PANs exist, the End Device will select the PAN whose transmission link quality is the strongest. If no valid Coordinator is found, the End Device will either go to sleep (as dictated by its SM (Sleep Mode) parameter) or retry Association.

Note - An End Device will also disqualify Coordinators if they are not allowing association (A2 - AllowAssociation bit); or, if the Coordinator is not using the same NonBeacon scheme as the End Device. (They must both be programmed with NonBeacon code.)

**3. Associate to Valid Coordinator**

Once a valid Coordinator is found (step 2), the End Device sends an AssociationRequest message to the Coordinator. It then waits for an AssociationConfirmation to be sent from the Coordinator. Once the Confirmation is received, the End Device is Associated and the Associate LED will blink rapidly (2 times per second). The LED is solid if the End Device has not associated.

**4. End Device Changes once an End Device has associated**

Changing A1, ID or CH parameters will cause the End Device to disassociate and restart the Association procedure.

If the End Device fails to associate, the AI command can give some indication of the failure.

# XBee® / XBee-PRO® Addressing

Every RF data packet sent over-the-air contains a Source Address and Destination Address field in its header. The RF module conforms to the 802.15.4 specification and supports both short 16-bit addresses and long 64-bit addresses. A unique 64-bit IEEE source address is assigned at the factory and can be read with the SL (Serial Number Low) and SH (Serial Number High) commands. Short addressing must be configured manually. A module will use its unique 64-bit address as its Source Address if its MY (16-bit Source Address) value is “0xFFFF” or “0xFFFE”.

To send a packet to a specific module using 64-bit addressing: Set the Destination Address (DL + DH) of the sender to match the Source Address (SL + SH) of the intended destination module.

To send a packet to a specific module using 16-bit addressing: Set DL (Destination Address Low) parameter to equal the MY parameter of the intended destination module and set the DH (Destination Address High) parameter to '0'.

## Unicast Mode

By default, the RF module operates in Unicast Mode. Unicast Mode is the only mode that supports retries. While in this mode, receiving modules send an ACK (acknowledgement) of RF packet reception to the transmitter. If the transmitting module does not receive the ACK, it will re-send the packet up to three times or until the ACK is received.

**Short 16-bit addresses**. The module can be configured to use short 16-bit addresses as the Source Address by setting (MY < 0xFFFE). Setting the DH parameter (DH = 0) will configure the Destination Address to be a short 16-bit address (if DL < 0xFFFE). For two modules to communicate using short addressing, the Destination Address of the transmitter module must match the MY parameter of the receiver.

The following table shows a sample network configuration that would enable Unicast Mode communications using short 16-bit addresses.

Table 2-03. Sample Unicast Network Configuration (using 16-bit addressing)

| Parameter | RF Module 1 | RF Module 2 |
|---|---|---|
| MY (Source Address) | 0x01 | 0x02 |
| DH (Destination Address High) | 0 | 0 |
| DL (Destination Address Low) | 0x02 | 0x01 |

**Long 64-bit addresses**. The RF module’s serial number (SL parameter concatenated to the SH parameter) can be used as a 64-bit source address when the MY (16-bit Source Address) parameter is disabled. When the MY parameter is disabled (MY = 0xFFFF or 0xFFFE), the module’s source address is set to the 64-bit IEEE address stored in the SH and SL parameters.

When an End Device associates to a Coordinator, its MY parameter is set to 0xFFFE to enable 64-bit addressing. The 64-bit address of the module is stored as SH and SL parameters. To send a packet to a specific module, the Destination Address (DL + DH) on the sender must match the Source Address (SL + SH) of the desired receiver.

## Broadcast Mode

Any RF module within range will accept a packet that contains a broadcast address. When configured to operate in Broadcast Mode, receiving modules do not send ACKs (Acknowledgements) and transmitting modules do not automatically re-sent packets as is the case in Unicast Mode.

To send a broadcast packet to all modules regardless of 16-bit or 64-bit addressing, set the destination addresses of all the modules as shown below.

Sample Network Configuration (All modules in the network):

- DL (Destination Low Address) = 0x0000FFFF

If RR is set to 0, only one packet is broadcast. If RR > 0, (RR + 2) packets are sent in each broadcast. No acknowledgements are returned. See also the RR command description.

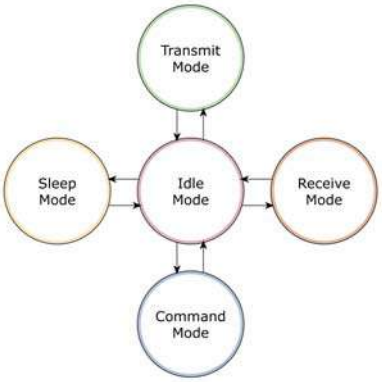
- DH (Destination High Address) = 0x00000000 (default value)

NOTE: When programming the module, parameters are entered in hexadecimal notation (without the “0x” prefix). Leading zeroes may be omitted.

# Modes of Operation

XBee®/XBee-PRO® RF Modules operate in five modes.

Figure 2-07. Modes of Operation



## Idle Mode

When not receiving or transmitting data, the RF module is in Idle Mode. The module shifts into the other modes of operation under the following conditions:

- Transmit Mode (Serial data is received in the DI Buffer)
- Receive Mode (Valid RF data is received through the antenna)
- Sleep Mode (Sleep Mode condition is met)
- Command Mode (Command Mode Sequence is issued)

## Transmit/Receive Modes

### RF Data Packets

Each transmitted data packet contains a Source Address and Destination Address field. The Source Address matches the address of the transmitting module as specified by the MY (Source Address) parameter (if MY >= 0xFFFE), the SH (Serial Number High) parameter or the SL (Serial Number Low) parameter. The <Destination Address> field is created from the DH (Destination Address High) and DL (Destination Address Low) parameter values. The Source Address and/or Destination Address fields will either contain a 16-bit short or long 64-bit long address.

The RF data packet structure follows the 802.15.4 specification.

[Refer to the XBee/XBee-PRO Addressing section for more information]

### Direct and Indirect Transmission

There are two methods to transmit data:

- Direct Transmission - data is transmitted immediately to the Destination Address
- Indirect Transmission - A packet is retained for a period of time and is only transmitted after the destination module (Source Address = Destination Address) requests the data.

Indirect Transmissions can only occur on a Coordinator. Thus, if all nodes in a network are End Devices, only Direct Transmissions will occur. Indirect Transmissions are useful to ensure packet delivery to a sleeping node. The Coordinator currently is able to retain up to 2 indirect messages.

### Direct Transmission

A Coordinator can be configured to use only Direct Transmission by setting the SP (Cyclic Sleep Period) parameter to "0". Also, a Coordinator using indirect transmissions will revert to direct transmission if it knows the destination module is awake.

To enable this behavior, the ST (Time before Sleep) value of the Coordinator must be set to match the ST value of the End Device. Once the End Device either transmits data to the Coordinator or polls the Coordinator for data, the Coordinator will use direct transmission for all subsequent data transmissions to that module address until ST time occurs with no activity (at which point it will revert to using indirect transmissions for that module address). "No activity" means no transmission or reception of messages with a specific address. Global messages will not reset the ST timer.

### Indirect Transmission

To configure Indirect Transmissions in a PAN (Personal Area Network), the SP (Cyclic Sleep Period) parameter value on the Coordinator must be set to match the longest sleep value of any End Device. The sleep period value on the Coordinator determines how long (time or number of beacons) the Coordinator will retain an indirect message before discarding it.

An End Device must poll the Coordinator once it wakes from Sleep to determine if the Coordinator has an indirect message for it. For Cyclic Sleep Modes, this is done automatically every time the module wakes (after SP time). For Pin Sleep Modes, the A1 (End Device Association) parameter value must be set to enable Coordinator polling on pin wake‐up. Alternatively, an End Device can use the FP (Force Poll) command to poll the Coordinator as needed.

## CCA (Clear Channel Assessment)

Prior to transmitting a packet, a CCA (Clear Channel Assessment) is performed on the channel to determine if the channel is available for transmission. The detected energy on the channel is compared with the CA (Clear Channel Assessment) parameter value. If the detected energy exceeds the CA parameter value, the packet is not transmitted.

Also, a delay is inserted before a transmission takes place. This delay is able to be set using the RN (Backoff Exponent) parameter. If RN is set to “0”, then there is no delay before the first CCA is performed. The RN parameter value is the equivalent of the “minBE” parameter in the 802.15.4 specification. The transmit sequence follows the 802.15.4 specification.

By default, the MM (MAC Mode) parameter = 0. On a CCA failure, the module will attempt to re‐send the packet up to two additional times.

When in Unicast packets with RR (Retries) = 0, the module will execute two CCA retries. Broadcast packets always get two CCA retries.

## Acknowledgement

If the transmission is not a broadcast message, the module will expect to receive an acknowledgement from the destination node. If an acknowledgement is not received, the packet will be resent up to 3 more times. If the acknowledgement is not received after all transmissions, an ACK failure is recorded.

## Sleep Mode

Sleep Modes enable the RF module to enter states of low-power consumption when not in use. In order to enter Sleep Mode, one of the following conditions must be met (in addition to the module having a non-zero SM parameter value):

- Sleep_RQ (pin 9) is asserted and the module is in a pin sleep mode (SM = 1, 2, or 5)
- The module is idle (no data transmission or reception) for the amount of time defined by the ST (Time before Sleep) parameter. [NOTE: ST is only active when SM = 4-5.]

Table 2-04. Sleep Mode Configurations

| Sleep Mode Setting | Transition into Sleep Mode | Transition out of Sleep Mode (wake) | Characteristics | Related Commands | Power Consumption |
|---|---|---|---|---|---|
| Pin Hibernate (SM = 1) | Assert (high) Sleep_RQ (pin 9) | De-assert (low) Sleep_RQ | Pin/Host-controlled / NonBeacon systems only / Lowest Power | (SM) | < 10 µA (@3.0 VCC) |
| Pin Doze (SM = 2) | Assert (high) Sleep_RQ (pin 9) | De-assert (low) Sleep_RQ | Pin/Host-controlled / NonBeacon systems only / Fastest wake-up | (SM) | < 50 µA |
| Cyclic Sleep (SM = 4) | Automatic transition to Sleep Mode as defined by the SM (Sleep Mode) and ST (Time before Sleep) parameters. | Transition occurs after the cyclic sleep time interval elapses. The time interval is defined by the SP (Cyclic Sleep Period) parameter. | RF module wakes in pre-determined time intervals to detect if RF data is present / When SM = 5 | (SM), SP, ST | < 50 µA when sleeping |
| Cyclic Sleep (SM = 5) | Automatic transition to Sleep Mode as defined by the SM (Sleep Mode) and ST (Time before Sleep) parameters or on a falling edge transition of the SLEEP_RQ pin. | Transition occurs after the cyclic sleep time interval elapses. The time interval is defined by the SP (Cyclic Sleep Period) parameter. | RF module wakes in pre-determined time intervals to detect if RF data is present. Module also wakes on a falling edge of SLEEP_RQ | (SM), SP, ST | < 50 µA when sleeping |

The SM command is central to setting Sleep Mode configurations. By default, Sleep Modes are disabled (SM = 0) and the module remains in Idle/Receive Mode. When in this state, the module is constantly ready to respond to serial or RF activity.

### Pin / Host-controlled Sleep Modes

The transient current when waking from pin sleep (SM = 1 or 2) does not exceed the idle current of the module. The current ramps up exponentially to its idle current.

**Pin Hibernate (SM = 1)**

- Pin/Host-controlled
- Typical power-down current: < 10 µA (@3.0 VCC)
- Typical wake-up time: 10.2 msec

Pin Hibernate Mode minimizes quiescent power (power consumed when in a state of rest or inactivity). This mode is voltage level-activated; when Sleep_RQ (pin 9) is asserted, the module will finish any transmit, receive or association activities, enter Idle Mode, and then enter a state of sleep. The module will not respond to either serial or RF activity while in pin sleep.

To wake a sleeping module operating in Pin Hibernate Mode, de-assert Sleep_RQ (pin 9). The module will wake when Sleep_RQ is de-asserted and is ready to transmit or receive when the CTS line is low. When waking the module, the pin must be de-asserted at least two 'byte times' after CTS goes low. This assures that there is time for the data to enter the DI buffer.

**Pin Doze (SM = 2)**

- Pin/Host-controlled
- Typical power-down current: < 50 µA
- Typical wake-up time: 2.6 msec

Pin Doze Mode functions as does Pin Hibernate Mode; however, Pin Doze features faster wake-up time and higher power consumption.

To wake a sleeping module operating in Pin Doze Mode, de-assert Sleep_RQ (pin 9). The module will wake when Sleep_RQ is de-asserted and is ready to transmit or receive when the CTS line is low. When waking the module, the pin must be de-asserted at least two 'byte times' after CTS goes low. This assures that there is time for the data to enter the DI buffer.

## Cyclic Sleep Modes

### Cyclic Sleep Remote (SM = 4)

- Typical Power-down Current: < 50 µA (when asleep)
- Typical wake-up time: 2.6 msec

The Cyclic Sleep Modes allow modules to periodically check for RF data. When the SM parameter is set to ‘4’, the module is configured to sleep, then wakes once a cycle to check for data from a module configured as a Cyclic Sleep Coordinator (SM = 0, CE = 1). The Cyclic Sleep Remote sends a poll request to the coordinator at a specific interval set by the SP (Cyclic Sleep Period) parameter. The coordinator will transmit any queued data addressed to that specific remote upon receiving the poll request.

If no data is queued for the remote, the coordinator will not transmit and the remote will return to sleep for another cycle. If queued data is transmitted back to the remote, it will stay awake to allow for back and forth communication until the ST (Time before Sleep) timer expires.

Also note that $\overline{\text{CTS}}$ will go low each time the remote wakes, allowing for communication initiated by the remote host if desired.

### Cyclic Sleep Remote with Pin Wake-up (SM = 5)

Use this mode to wake a sleeping remote module through either the RF interface or by the de-assertion of Sleep_RQ for event-driven communications. The cyclic sleep mode works as described above (Cyclic Sleep Remote) with the addition of a pin-controlled wake-up at the remote module. The Sleep_RQ pin is edge-triggered, not level-triggered. The module will wake when a low is detected then set $\overline{\text{CTS}}$ low as soon as it is ready to transmit or receive.

Any activity will reset the ST (Time before Sleep) timer so the module will go back to sleep only after there is no activity for the duration of the timer. Once the module wakes (pin-controlled), further pin activity is ignored. The module transitions back into sleep according to the ST time regardless of the state of the pin.

### [Cyclic Sleep Coordinator (SM = 6)]

- Typical current = Receive current
- Always awake

NOTE: The SM=6 parameter value exists solely for backwards compatibility with firmware version 1.x60. If backwards compatibility with the older firmware version is not required, always use the CE (Coordinator Enable) command to configure a module as a Coordinator.

This mode configures a module to wake cyclic sleeping remotes through RF interfacing. The Coordinator will accept a message addressed to a specific remote 16 or 64-bit address and hold it in a buffer until the remote wakes and sends a poll request. Messages not sent directly (buffered and requested) are called "Indirect messages". The Coordinator only queues one indirect message at a time. The Coordinator will hold the indirect message for a period 2.5 times the sleeping period indicated by the SP (Cyclic Sleep Period) parameter. The Coordinator's SP parameter should be set to match the value used by the remotes.

## Command Mode

To modify or read RF Module parameters, the module must first enter into Command Mode - a state in which incoming characters are interpreted as commands. Two Command Mode options are supported: AT Command Mode [refer to section below] and API Command Mode [p56].

### AT Command Mode

**To Enter AT Command Mode:**

Send the 3-character command sequence "+++" and observe guard times before and after the command characters. [Refer to the "Default AT Command Mode Sequence" below.]

Default AT Command Mode Sequence (for transition to Command Mode):

- No characters sent for one second [GT (Guard Times) parameter = 0x3E8]
- Input three plus characters ("+++") within one second [CC (Command Sequence Character) parameter = 0x2B.]
- No characters sent for one second [GT (Guard Times) parameter = 0x3E8]

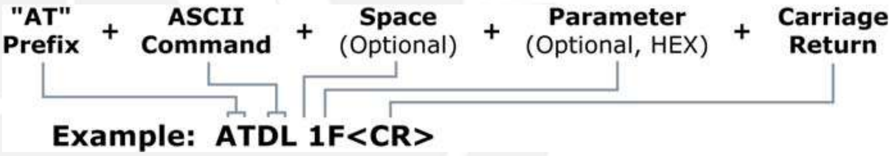
All of the parameter values in the sequence can be modified to reflect user preferences.

NOTE: Failure to enter AT Command Mode is most commonly due to baud rate mismatch. Ensure the 'Baud' setting on the "PC Settings" tab matches the interface data rate of the RF module. By default, the BD parameter = 3 (9600 bps).

**To Send AT Commands:**

Send AT commands and parameters using the syntax shown below.

Figure 2-08. Syntax for sending AT Commands



To read a parameter value stored in the RF module's register, omit the parameter field.

The preceding example would change the RF module Destination Address (Low) to "0x1F". To store the new value to non-volatile (long term) memory, subsequently send the WR (Write) command.

For modified parameter values to persist in the module's registry after a reset, changes must be saved to non-volatile memory using the WR (Write) Command. Otherwise, parameters are restored to previously saved values after the module is reset.

**System Response.** When a command is sent to the module, the module will parse and execute the command. Upon successful execution of a command, the module returns an "OK" message. If execution of a command results in an error, the module returns an "ERROR" message.

**To Exit AT Command Mode:**

1. Send the ATCN (Exit Command Mode) command (followed by a carriage return).

   [OR]

2. If no valid AT Commands are received within the time specified by CT (Command Mode Timeout) Command, the RF module automatically returns to Idle Mode.

For an example of programming the RF module using AT Commands and descriptions of each configurable parameter, refer to the RF Module Configuration chapter [p27].

# 3. RF Module Configuration

## Programming the RF Module

Refer to the Command Mode section [p26] for more information about entering Command Mode, sending AT commands and exiting Command Mode. For information regarding module programming using API Mode, refer to the API Operation sections [p56].

### Programming Examples

**Setup**

The programming examples in this section require the installation of Digi's X-CTU Software and a serial connection to a PC. (Digi stocks RS-232 and USB boards to facilitate interfacing with a PC.)

1. Install Digi's X-CTU Software to a PC by double-clicking the "setup_X-CTU.exe" file. (The file is located on the Digi CD and www.digi.com/xctu.)
2. Mount the RF module to an interface board, then connect the module assembly to a PC.
3. Launch the X-CTU Software and select the 'PC Settings' tab. Verify the baud and parity settings of the Com Port match those of the RF module.

NOTE: Failure to enter AT Command Mode is most commonly due to baud rate mismatch. Ensure the 'Baud' setting on the 'PC Settings' tab matches the interface data rate of the RF module. By default, the BD parameter = 3 (which corresponds to 9600 bps).

**Sample Configuration: Modify RF Module Destination Address**

Example: Utilize the X-CTU "Terminal" tab to change the RF module's DL (Destination Address Low) parameter and save the new address to non-volatile memory.

After establishing a serial connection between the RF module and a PC [refer to the 'Setup' section above], select the "Terminal" tab of the X-CTU Software and enter the following command lines ('CR' stands for carriage return):

Method 1 (One line per command)

| Send AT Command | System Response |
|---|---|
| +++ | OK <CR> (Enter into Command Mode) |
| ATDL <Enter> | {current value} <CR> (Read Destination Address Low) |
| ATDL1A0D <Enter> | OK <CR> (Modify Destination Address Low) |
| ATWR <Enter> | OK <CR> (Write to non-volatile memory) |
| ATCN <Enter> | OK <CR> (Exit Command Mode) |

Method 2 (Multiple commands on one line)

| Send AT Command | System Response |
|---|---|
| +++ | OK <CR> (Enter into Command Mode) |
| ATDL <Enter> | {current value} <CR> (Read Destination Address Low) |
| ATDL1A0D,WR,CN <Enter> | OK<CR> OK<CR> OK<CR> |

**Sample Configuration: Restore RF Module Defaults**

Example: Utilize the X-CTU "Modem Configuration" tab to restore default parameter values.

After establishing a connection between the module and a PC [refer to the 'Setup' section above], select the "Modem Configuration" tab of the X-CTU Software.

1. Select the 'Read' button.
2. Select the 'Restore' button.

## Remote Configuration Commands

The API firmware has provisions to send configuration commands to remote devices using the Remote Command Request API frame (see API Operation). This API frame can be used to send commands to a remote module to read or set command parameters.

The API firmware has provisions to send configuration commands (set or read) to a remote module using the Remote Command Request API frame (see API Operations). Remote commands can be issued to read or set command parameters on a remote device.

### Sending a Remote Command

To send a remote command, the Remote Command Request frame should be populated with values for the 64 bit and 16 bit addresses. If 64 bit addressing is desired then the 16 bit address field should be filled with 0xFFFE. If any value other than 0xFFFE is used in the 16 bit address field then the 64 bit address field will be ignored and 16 bit addressing will be used. If a command response is desired, the Frame ID should be set to a non-zero value.

### Applying Changes on Remote

When remote commands are used to change command parameter settings on a remote device, parameter changes do not take effect until the changes are applied. For example, changing the BD parameter will not change the actual serial interface rate on the remote until the changes are applied. Changes can be applied using remote commands in one of three ways:

Set the apply changes option bit in the API frame

Issue an AC command to the remote device

Issue a WR + FR command to the remote device to save changes and reset the device.

### Remote Command Responses

If the remote device receives a remote command request transmission, and the API frame ID is non-zero, the remote will send a remote command response transmission back to the device that sent the remote command. When a remote command response transmission is received, a device sends a remote command response API frame out its UART. The remote command response indicates the status of the command (success, or reason for failure), and in the case of a command query, it will include the register value.

The device that sends a remote command will not receive a remote command response frame if:

The destination device could not be reached

The frame ID in the remote command request is set to 0.

## Command Reference Tables

XBee®/XBee-PRO® RF Modules expect numerical values in hexadecimal. Hexadecimal values are designated by a “0x” prefix. Decimal equivalents are designated by a “d” suffix. Commands are contained within the following command categories (listed in the order that their tables appear):

- Special
- Networking & Security
- RF Interfacing
- Sleep (Low Power)
- Serial Interfacing
- I/O Settings
- Diagnostics
- AT Command Options

All modules within a PAN should operate using the same firmware version.

### Special

Table 3-01. XBee-PRO Commands - Special

| AT Command | Command Category | Name and Description | Parameter Range | Default |
|---|---|---|---|---|
| WR | Special | Write. Write parameter values to non-volatile memory so that parameter modifications persist through subsequent power-up or reset.<br>Note: Once WR is issued, no additional characters should be sent to the module until after the response "OK\r" is received. | - | - |
| RE | Special | Restore Defaults. Restore module parameters to factory defaults. | - | - |
| FR ( v1.x80*) | Special | Software Reset. Responds immediately with an OK then performs a hard reset ~100ms later. | - | - |

* Firmware version in which the command was first introduced (firmware versions are numbered in hexadecimal notation.)

### Networking & Security

Table 3-02. XBee®/XBee-PRO® Commands - Networking & Security (Sub-categories designated within {brackets})

| AT Command | Command Category | Name and Description | Parameter Range | Default |
|---|---|---|---|---|
| CH | Networking {Addressing} | Channel. Set/Read the channel number used for transmitting and receiving data between RF modules (uses 802.15.4 protocol channel numbers). | 0x0B - 0x1A (XBee)<br>0x0C - 0x17 (XBee-PRO) | 0x0C (12d) |
| ID | Networking {Addressing} | PAN ID. Set/Read the PAN (Personal Area Network) ID.<br>Use 0xFFFF to broadcast messages to all PANs. | 0 - 0xFFFF | 0x3332 (13106d) |
| DH | Networking {Addressing} | Destination Address High. Set/Read the upper 32 bits of the 64-bit destination address. When combined with DL, it defines the destination address used for transmission. To transmit using a 16-bit address, set DH parameter to zero and DL less than 0xFFFF. 0x000000000000FFFF is the broadcast address for the PAN. | 0 - 0xFFFFFFFF | 0 |
| DL | Networking {Addressing} | Destination Address Low. Set/Read the lower 32 bits of the 64-bit destination address. When combined with DH, DL defines the destination address used for transmission. To transmit using a 16-bit address, set DH parameter to zero and DL less than 0xFFFF. 0x000000000000FFFF is the broadcast address for the PAN. | 0 - 0xFFFFFFFF | 0 |
| MY | Networking {Addressing} | 16-bit Source Address. Set/Read the RF module 16-bit source address. Set MY = 0xFFFF to disable reception of packets with 16-bit addresses. 64-bit source address (serial number) and broadcast address (0x000000000000FFFF) is always enabled. | 0 - 0xFFFF | 0 |
| SH | Networking {Addressing} | Serial Number High. Read high 32 bits of the RF module's unique IEEE 64-bit address. 64-bit source address is always enabled. | 0 - 0xFFFFFFFF [read-only] | Factory-set |
| SL | Networking {Addressing} | Serial Number Low. Read low 32 bits of the RF module's unique IEEE 64-bit address. 64-bit source address is always enabled. | 0 - 0xFFFFFFFF [read-only] | Factory-set |
| RR ( v1.xA0*) | Networking {Addressing} | XBee Retries. Set/Read the maximum number of retries the module will execute in addition to the 3 retries provided by the 802.15.4 MAC. For each XBee retry, the 802.15.4 MAC can execute up to 3 retries. | 0 - 6 | 0 |
| RN | Networking {Addressing} | Random Delay Slots. Set/Read the minimum value of the back-off exponent in the CSMA-CA algorithm that is used for collision avoidance. If RN = 0, collision avoidance is disabled during the first iteration of the algorithm (802.15.4 - macMinBE). | 0 - 3 [exponent] | 0 |
| MM ( v1.x80*) | Networking {Addressing} | MAC Mode. MAC Mode. Set/Read MAC Mode value. MAC Mode enables/disables the use of a Digi header in the 802.15.4 RF packet. When Modes 1 or 3 are enabled (MM=1,3), duplicate packet detection is enabled as well as certain AT commands. Please see the detailed MM description on page 47 for additional information. | 0 - 3<br>0 = Digi Mode<br>1 = 802.15.4 (no ACKs)<br>2 = 802.15.4 (with ACKs)<br>3 = Digi Mode (no ACKs) | 0 |
| NI ( v1.x80*) | Networking {Identification} | Node Identifier. Stores a string identifier. The register only accepts printable ASCII data. A string can not start with a space. Carriage return ends command. Command will automatically end when maximum bytes for the string have been entered. This string is returned as part of the ND (Node Discover) command. This identifier is also used with the DN (Destination Node) command. | 20-character ASCII string | - |
| ND ( v1.x80*) | Networking {Identification} | Node Discover. Discovers and reports all RF modules found. The following information is reported for each module discovered (the example cites use of Transparent operation (AT command format) - refer to the long ND command description regarding differences between Transparent and API operation).<br>MY<CR><br>SH<CR><br>SL<CR><br>DB<CR><br>NI<CR><CR><br>The amount of time the module allows for responses is determined by the NT parameter. In Transparent operation, command completion is designated by a <CR> (carriage return). ND also accepts a Node Identifier as a parameter. In this case, only a module matching the supplied identifier will respond. If ND self-response is enabled (NO=1) the module initiating the node discover will also output a response for itself. | optional 20-character NI value | |
| NT ( v1.xA0*) | Networking {Identification} | Node Discover Time. Set/Read the amount of time a node will wait for responses from other nodes when using the ND (Node Discover) command. | 0x01 - 0xFC [x 100 ms] | 0x19 |

**Table 3-02. XBee®/XBee-PRO® Commands - Networking & Security** (Sub-categories designated within {brackets})

| AT Command | Command Category | Name and Description | Parameter Range | Default |
|---|---|---|---|---|
| NO (v1xC5) | Networking {Identification} | Node Discover Options. Enables node discover self-response on the module. | 0-1 | 0 |
| DN ( v1.x80*) | Networking {Identification} | Destination Node. Resolves an NI (Node Identifier) string to a physical address. The following events occur upon successful command execution:<br>1. DL and DH are set to the address of the module with the matching Node Identifier.<br>2. "OK" is returned.<br>3. RF module automatically exits AT Command Mode<br>If there is no response from a module within 200 msec or a parameter is not specified (left blank), the command is terminated and an "ERROR" message is returned. | 20-character ASCII string | - |
| CE ( v1.x80*) | Networking {Association} | Coordinator Enable. Set/Read the coordinator setting. | 0 - 1<br>0 = End Device<br>1 = Coordinator | 0 |
| SC ( v1.x80*) | Networking {Association} | Scan Channels. Set/Read list of channels to scan for all Active and Energy Scans as a bitfield. This affects scans initiated in command mode (AS, ED) and during End Device Association and Coordinator startup:<br>bit 0 - 0x0B bit 4 - 0x0F bit 8 - 0x13 bit12 - 0x17<br>bit 1 - 0x0C bit 5 - 0x10 bit 9 - 0x14 bit13 - 0x18<br>bit 2 - 0x0D bit 6 - 0x11 bit 10 - 0x15 bit14 - 0x19<br>bit 3 - 0x0E bit 7 - 0x12 bit 11 - 0x16 bit 15 - 0x1A | 0 - 0xFFFF [bitfield]<br>(bits 0, 14, 15 not allowed on the XBee-PRO) | 0x1FFE<br>(all XBee-PRO Channels) |
| SD ( v1.x80*) | Networking {Association} | Scan Duration. Set/Read the scan duration exponent.<br>*End Device* - Duration of Active Scan during Association.<br>*Coordinator* - If 'ReassignPANID' option is set on Coordinator [refer to A2 parameter], SD determines the length of time the Coordinator will scan channels to locate existing PANs. If 'ReassignChannel' option is set, SD determines how long the Coordinator will perform an Energy Scan to determine which channel it will operate on.<br>'Scan Time' is measured as (# of channels to scan] * (2 ^ SD) * 15.36ms). The number of channels to scan is set by the SC command. The XBee can scan up to 16 channels (SC = 0xFFFF). The XBee PRO can scan up to 13 channels (SC = 0x3FFE).<br>Example: The values below show results for a 13 channel scan:<br>If SD = 0, time = 0.18 sec SD = 8, time = 47.19 sec<br>SD = 2, time = 0.74 sec SD = 10, time = 3.15 min<br>SD = 4, time = 2.95 sec SD = 12, time = 12.58 min<br>SD = 6, time = 11.80 sec SD = 14, time = 50.33 min | 0-0x0F [exponent] | 4 |
| A1 ( v1.x80*) | Networking {Association} | End Device Association. Set/Read End Device association options.<br>bit 0 - ReassignPanID<br>0 - Will only associate with Coordinator operating on PAN ID that matches module ID<br>1 - May associate with Coordinator operating on any PAN ID<br>bit 1 - ReassignChannel<br>0 - Will only associate with Coordinator operating on matching CH Channel setting<br>1 - May associate with Coordinator operating on any Channel<br>bit 2 - AutoAssociate<br>0 - Device will not attempt Association<br>1 - Device attempts Association until success<br>Note: This bit is used only for Non-Beacon systems. End Devices in Beacon-enabled system must always associate to a Coordinator<br>bit 3 - PollCoordOnPinWake<br>0 - Pin Wake will not poll the Coordinator for indirect (pending) data<br>1 - Pin Wake will send Poll Request to Coordinator to extract any pending data<br>bits 4 - 7 are reserved | 0 - 0x0F [bitfield] | 0 |
| A2 ( v1.x80*) | Networking {Association} | Coordinator Association. Set/Read Coordinator association options.<br>bit 0 - ReassignPanID<br>0 - Coordinator will not perform Active Scan to locate available PAN ID. It will operate on ID (PAN ID).<br>1 - Coordinator will perform Active Scan to determine an available ID (PAN ID). If a PAN ID conflict is found, the ID parameter will change.<br>bit 1 - ReassignChannel -<br>0 - Coordinator will not perform Energy Scan to determine free channel. It will operate on the channel determined by the CH parameter.<br>1 - Coordinator will perform Energy Scan to find a free channel, then operate on that channel.<br>bit 2 - AllowAssociation -<br>0 - Coordinator will not allow any devices to associate to it.<br>1 - Coordinator will allow devices to associate to it.<br>bits 3 - 7 are reserved | 0 - 7 [bitfield] | 0 |

**Table 3-02. XBee®/XBee-PRO® Commands - Networking & Security** (Sub-categories designated within {brackets})

| AT Command | Command Category | Name and Description | Parameter Range | Default |
|---|---|---|---|---|
| AI ( v1.x80*) | Networking {Association} | Association Indication. Read errors with the last association request:<br>0x00 - Successful Completion - Coordinator successfully started or End Device association complete<br>0x01 - Active Scan Timeout<br>0x02 - Active Scan found no PANs<br>0x03 - Active Scan found PAN, but the CoordinatorAllowAssociation bit is not set<br>0x04 - Active Scan found PAN, but Coordinator and End Device are not configured to support beacons<br>0x05 - Active Scan found PAN, but the Coordinator ID parameter does not match the ID parameter of the End Device<br>0x06 - Active Scan found PAN, but the Coordinator CH parameter does not match the CH parameter of the End Device<br>0x07 - Energy Scan Timeout<br>0x08 - Coordinator start request failed<br>0x09 - Coordinator could not start due to invalid parameter<br>0x0A - Coordinator Realignment is in progress<br>0x0B - Association Request not sent<br>0x0C - Association Request timed out - no reply was received<br>0x0D - Association Request had an Invalid Parameter<br>0x0E - Association Request Channel Access Failure. Request was not transmitted - CCA failure<br>0x0F - Remote Coordinator did not send an ACK after Association Request was sent<br>0x10 - Remote Coordinator did not reply to the Association Request, but an ACK was received after sending the request<br>0x11 - [reserved]<br>0x12 - Sync-Loss - Lost synchronization with a Beaconing Coordinator<br>0x13 - Disassociated - No longer associated to Coordinator<br>0xFF - RF Module is attempting to associate | 0 - 0x13 [read-only] | - |
| DA ( v1.x80*) | Networking {Association} | Force Disassociation. End Device will immediately disassociate from a Coordinator (if associated) and reattempt to associate. | - | - |
| FP ( v1.x80*) | Networking {Association} | Force Poll. Request indirect messages being held by a coordinator. | - | - |
| AS ( v1.x80*) | Networking {Association} | Active Scan. Send Beacon Request to Broadcast Address (0xFFFF) and Broadcast PAN (0xFFFF) on every channel. The parameter determines the time the radio will listen for Beacons on each channel. A PanDescriptor is created and returned for every Beacon received from the scan. Each PanDescriptor contains the following information:<br>CoordAddress (SH, SL)<CR><br>CoordPanID (ID)<CR><br>CoordAddrMode <CR><br>0x02 = 16-bit Short Address<br>0x03 = 64-bit Long Address<br>Channel (CH parameter) <CR><br>SecurityUse<CR><br>ACLEntry<CR><br>SecurityFailure<CR><br>SuperFrameSpec<CR> (2 bytes):<br>bit 15 - Association Permitted (MSB)<br>bit 14 - PAN Coordinator<br>bit 13 - Reserved<br>bit 12 - Battery Life Extension<br>bits 8-11 - Final CAP Slot<br>bits 4-7 - Superframe Order<br>bits 0-3 - Beacon Order<br>GtsPermit<CR><br>RSSI<CR> (RSSI is returned as -dBm)<br>TimeStamp<CR> (3 bytes)<br><CR><br>A carriage return <CR> is sent at the end of the AS command. The Active Scan is capable of returning up to 5 PanDescriptors in a scan. The actual scan time on each channel is measured as Time = [(2 ^SD PARAM) * 15.36] ms. Note the total scan time is this time multiplied by the number of channels to be scanned (16 for the XBee and 13 for the XBee-PRO). Also refer to SD command description. | 0 - 6 | - |
| ED ( v1.x80*) | Networking {Association} | Energy Scan. Send an Energy Detect Scan. This parameter determines the length of scan on each channel. The maximal energy on each channel is returned & each value is followed by a carriage return. An additional carriage return is sent at the end of the command. The values returned represent the detected energy level in units of -dBm. The actual scan time on each channel is measured as Time = [(2 ^ED) * 15.36] ms. Note the total scan time is this time multiplied by the number of channels to be scanned (refer to SD parameter). | 0 - 6 | - |
| EE ( v1.xA0*) | Networking {Security} | AES Encryption Enable. Disable/Enable 128-bit AES encryption support. Use in conjunction with the KY command. | 0 - 1 | 0 (disabled) |
| KY ( v1.xA0*) | Networking {Security} | AES Encryption Key. Set the 128-bit AES (Advanced Encryption Standard) key for encrypting/decrypting data. The KY register cannot be read. | 0 - (any 16-Byte value) | - |

* Firmware version in which the command was first introduced (firmware versions are numbered in hexadecimal notation.)

### RF Interfacing

Table 3-03. XBee/XBee-PRO Commands - RF Interfacing

| AT Command | Command Category | Name and Description | Parameter Range | Default |
|---|---|---|---|---|
| PL | RF Interfacing | **Power Level.** Select/Read the power level at which the RF module transmits conducted power. | 0 - 4 (XBee / XBee-PRO)<br>0 = -10 / 10 dBm<br>1 = -6 / 12 dBm<br>2 = -4 / 14 dBm<br>3 = -2 / 16 dBm<br>4 = 0 / 18 dBm<br>XBee-PRO International variant:<br>PL=4: 10 dBm<br>PL=3: 8 dBm<br>PL=2: 2 dBm<br>PL=1: -3 dBm<br>PL=0: -3 dBm | 4 |
| CA (v1.x80*) | RF Interfacing | **CCA Threshold.** Set/read the CCA (Clear Channel Assessment) threshold. Prior to transmitting a packet, a CCA is performed to detect energy on the channel. If the detected energy is above the CCA Threshold, the module will not transmit the packet. | 0x24 - 0x50 [-dBm] | 0x2C<br>(-44d dBm) |

* Firmware version in which the command was first introduced (firmware versions are numbered in hexadecimal notation.)

### Sleep (Low Power)

Table 3-04. XBee®/XBee-PRO® Commands - Sleep (Low Power)

| AT Command | Command Category | Name and Description | Parameter Range | Default |
|---|---|---|---|---|
| SM | Sleep<br>(Low Power) | **Sleep Mode.** Set/Read Sleep Mode configurations. | 0 - 5<br>0 = No Sleep<br>1 = Pin Hibernate<br>2 = Pin Doze<br>3 = Reserved<br>4 = Cyclic sleep remote<br>5 = Cyclic sleep remote w/ pin wake-up<br>6 = [Sleep Coordinator] for backwards compatibility w/ v1.x6 only; otherwise, use CE command. | 0 |
| SO | Sleep (Low Power) | Sleep Options Set/Read the sleep mode options.<br>Bit 0 - Poll wakeup disable<br>0 - Normal operations. A module configured for cyclic sleep will poll for data on waking.<br>1 - Disable wakeup poll. A module configured for cyclic sleep will not poll for data on waking.<br>Bit 1 - ADC/DIO wakeup sampling disable.<br>0 - Normal operations. A module configured in a sleep mode with ADC/DIO sampling enabled will automatically perform a sampling on wakeup.<br>1 - Suppress sample on wakeup. A module configured in a sleep mode with ADC/DIO sampling enabled will not automatically sample on wakeup. | 0-4 | 0 |
| ST | Sleep<br>(Low Power) | **Time before Sleep.** <NonBeacon firmware> Set/Read time period of inactivity (no serial or RF data is sent or received) before activating Sleep Mode. ST parameter is only valid with Cyclic Sleep settings (SM = 4 - 5).<br>Coordinator and End Device ST values must be equal.<br>Also note, the GT parameter value must always be less than the ST value. (If GT > ST, the configuration will render the module unable to enter into command mode.) If the ST parameter is modified, also modify the GT parameter accordingly. | 1 - 0xFFFF [x 1 ms] | 0x1388<br>(5000d) |
| SP | Sleep<br>(Low Power) | **Cyclic Sleep Period.** <NonBeacon firmware> Set/Read sleep period for cyclic sleeping remotes. Coordinator and End Device SP values should always be equal. To send Direct Messages, set SP = 0.<br>*End Device* - SP determines the sleep period for cyclic sleeping remotes. Maximum sleep period is 268 seconds (0x68B0).<br>*Coordinator* - If non-zero, SP determines the time to hold an indirect message before discarding it. A Coordinator will discard indirect messages after a period of (2.5 * SP). | 0 - 0x68B0 [x 10 ms] | 0 |
| DP (1.x80*) | Sleep<br>(Low Power) | **Disassociated Cyclic Sleep Period.** <NonBeacon firmware><br>*End Device* - Set/Read time period of sleep for cyclic sleeping remotes that are configured for Association but are not associated to a Coordinator. (i.e. If a device is configured to associate, configured as a Cyclic Sleep remote, but does not find a Coordinator, it will sleep for DP time before reattempting association.) Maximum sleep period is 268 seconds (0x68B0). DP should be > 0 for NonBeacon systems. | 1 - 0x68B0 [x 10 ms] | 0x3E8<br>(1000d) |

* Firmware version in which the command was first introduced (firmware versions are numbered in hexadecimal notation.)

### Serial Interfacing

**Table 3-05. XBee-PRO Commands - Serial Interfacing**

| AT Command | Command Category | Name and Description | Parameter Range | Default |
|---|---|---|---|---|
| BD | Serial Interfacing | **Interface Data Rate**. Set/Read the serial interface data rate for communications between the RF module serial port and host.<br>Request non-standard baud rates with values above 0x80 using a terminal window. Read the BD register to find actual baud rate achieved. | 0 - 7 (standard baud rates)<br>0 = 1200 bps<br>1 = 2400<br>2 = 4800<br>3 = 9600<br>4 = 19200<br>5 = 38400<br>6 = 57600<br>7 = 115200<br>0x80 - 0x3D090<br>(non-standard baud rates up to 250 Kbps) | 3 |
| RO | Serial Interfacing | **Packetization Timeout**. Set/Read number of character times of inter-character delay required before transmission. Set to zero to transmit characters as they arrive instead of buffering them into one RF packet. | 0 - 0xFF [x character times] | 3 |
| AP (v1.x80*) | Serial Interfacing | **API Enable**. Disable/Enable API Mode. | 0 - 2<br>0 =Disabled<br>1 = API enabled<br>2 = API enabled (w/escaped control characters) | 0 |
| NB | Serial Interfacing | **Parity**. Set/Read parity settings. | 0 - 4<br>0 = 8-bit no parity<br>1 = 8-bit even<br>2 = 8-bit odd<br>3 = 8-bit mark<br>4 = 8-bit space | 0 |
| PR (v1.x80*) | Serial Interfacing | **Pull-up Resistor Enable**. Set/Read bitfield to configure internal pull-up resistor status for I/O lines<br>Bitfield Map:<br>bit 0 - AD4/DIO4 (pin11)<br>bit 1 - AD3 / DIO3 (pin17)<br>bit 2 - AD2/DIO2 (pin18)<br>bit 3 - AD1/DIO1 (pin19)<br>bit 4 - AD0 / DIO0 (pin20)<br>bit 5 - RTS / AD6 / DIO6 (pin16)<br>bit 6 - DTR / SLEEP_RQ / DI8 (pin9)<br>bit 7 - DIN/CONFIG (pin3)<br>Bit set to “1” specifies pull-up enabled; “0” specifies no pull-up | 0 - 0xFF | 0xFF |

* Firmware version in which the command was first introduced (firmware versions are numbered in hexadecimal notation.)

### I/O Settings

**Table 3-06. XBee-PRO Commands - I/O Settings** (sub-category designated within {brackets})

| AT Command | Command Category | Name and Description | Parameter Range | Default |
|---|---|---|---|---|
| D8 | I/O Settings | **DI8 Configuration**. Select/Read options for the DI8 line (pin 9) of the RF module. | 0 - 1<br>0 = Disabled<br>3 = DI<br>(1,2,4 & 5 n/a) | 0 |
| D7 (v1.x80*) | I/O Settings | **DIO7 Configuration**. Select/Read settings for the DIO7 line (pin 12) of the RF module. Options include CTS flow control and I/O line settings. | 0 - 1<br>0 = Disabled<br>1 = CTS Flow Control<br>2 = (n/a)<br>3 = DI<br>4 = DO low<br>5 = DO high<br>6 = RS485 Tx Enable Low<br>7 = RS485 Tx Enable High | 1 |
| D6 (v1.x80*) | I/O Settings | **DIO6 Configuration**. Select/Read settings for the DIO6 line (pin 16) of the RF module. Options include RTS flow control and I/O line settings. | 0 - 1<br>0 = Disabled<br>1 = RTS flow control<br>2 = (n/a)<br>3 = DI<br>4 = DO low<br>5 = DO high | 0 |

Table 3-06. **XBee-PRO Commands - I/O Settings** (sub-category designated within {brackets})

| AT Command | Command Category | Name and Description | Parameter Range | Default |
|---|---|---|---|---|
| D5 (v1.x80*) | I/O Settings | DIO5 Configuration. Configure settings for the DIO5 line (pin 15) of the RF module. Options include Associated LED indicator (blinks when associated) and I/O line settings. | 0 - 1<br>0 = Disabled<br>1 = Associated indicator<br>2 = ADC<br>3 = DI<br>4 = DO low<br>5 = DO high | 1 |
| D0 - D4 (v1.xA0*) | I/O Settings | (DIO4 -DIO4) Configuration. Select/Read settings for the following lines: AD0/DIO0 (pin 20), AD1/DIO1 (pin 19), AD2/DIO2 (pin 18), AD3/DIO3 (pin 17), AD4/DIO4 (pin 11). Options include: Analog-to-digital converter, Digital Input and Digital Output. | 0 - 1<br>0 = Disabled<br>1 = (n/a)<br>2 = ADC<br>3 = DI<br>4 = DO low<br>5 = DO high | 0 |
| IU (v1.xA0*) | I/O Settings | I/O Output Enable. Disables/Enables I/O data received to be sent out UART. The data is sent using an API frame regardless of the current AP parameter value. | 0 - 1<br>0 = Disabled<br>1 = Enabled | 1 |
| IT (v1.xA0*) | I/O Settings | Samples before TX. Set/Read the number of samples to collect before transmitting data. Maximum number of samples is dependent upon the number of enabled inputs. | 1 - 0xFF | 1 |
| IS (v1.xA0*) | I/O Settings | Force Sample. Force a read of all enabled inputs (DI or ADC). Data is returned through the UART. If no inputs are defined (DI or ADC), this command will return error. | 8-bit bitmap (each bit represents the level of an I/O line setup as an output) | - |
| IO (v1.xA0*) | I/O Settings | Digital Output Level. Set digital output level to allow DIO lines that are setup as outputs to be changed through Command Mode. | - | - |
| IC (v1.xA0*) | I/O Settings | DIO Change Detect. Set/Read bitfield values for change detect monitoring. Each bit enables monitoring of DIO0 - DIO7 for changes. If detected, data is transmitted with DIO data only. Any samples queued waiting for transmission will be sent first. | 0 - 0xFF [bitfield] | 0 (disabled) |
| IR (v1.xA0*) | I/O Settings | Sample Rate. Set/Read sample rate. When set, this parameter causes the module to sample all enabled inputs at a specified interval. | 0 - 0xFFFF [x 1 msec] | 0 |
| IA (v1.xA0*) | I/O Settings {I/O Line Passing} | I/O Input Address. Set/Read addresses of module to which outputs are bound. Setting all bytes to 0xFF will not allow any received I/O packet to change outputs. Setting address to 0xFFFF will allow any received I/O packet to change outputs. | 0 - 0xFFFFFFFFFFFFFFFF | 0xFFFFFFF FFFFFFFFF |
| T0 - T7 (v1.xA0*) | I/O Settings {I/O Line Passing} | (D0 - D7) Output Timeout. Set/Read Output timeout values for lines that correspond with the D0 - D7 parameters. When output is set (due to I/O line passing) to a non-default level, a timer is started which when expired will set the output to it default level. The timer is reset when a valid I/O packet is received. | 0 - 0xFF [x 100 ms] | 0xFF |
| P0 | I/O Settings {I/O Line Passing} | PWM0 Configuration. Select/Read function for PWM0 pin. | 0 - 2<br>0 = Disabled<br>1 = RSSI<br>2 = PWM Output | 1 |
| P1 (v1.xA0*) | I/O Settings {I/O Line Passing} | PWM1 Configuration. Select/Read function for PWM1 pin. | 0 - 2<br>0 = Disabled<br>1 = RSSI<br>2 = PWM Output | 0 |
| M0 (v1.xA0*) | I/O Settings {I/O Line Passing} | PWM0 Output Level. Set/Read the PWM0 output level. | 0 - 0x03FF | - |
| M1 (v1.xA0*) | I/O Settings {I/O Line Passing} | PWM1 Output Level. Set/Read the PWM1 output level. | 0 - 0x03FF | - |
| PT (v1.xA0*) | I/O Settings {I/O Line Passing} | PWM Output Timeout. Set/Read output timeout value for both PWM outputs. When PWM is set to a non-zero value: Due to I/O line passing, a time is started which when expired will set the PWM output to zero. The timer is reset when a valid I/O packet is received.] | 0 - 0xFF [x 100 ms] | 0xFF |
| RP | I/O Settings {I/O Line Passing} | RSSI PWM Timer. Set/Read PWM timer register. Set the duration of PWM (pulse width modulation) signal output on the RSSI pin. The signal duty cycle is updated with each received packet and is shut off when the timer expires.] | 0 - 0xFF [x 100 ms] | 0x28 (40d) |

* Firmware version in which the command was first introduced (firmware versions are numbered in hexadecimal notation.)

### Diagnostics

Table 3-07. **XBee®/XBee-PRO® Commands - Diagnostics**

| AT Command | Command Category | Name and Description | Parameter Range | Default |
|---|---|---|---|---|
| VR | Diagnostics | Firmware Version. Read firmware version of the RF module. | 0 - 0xFFFF [read-only] | Factory-set |
| VL (v1.x80*) | Diagnostics | Firmware Version - Verbose. Read detailed version information (including application build date, MAC, PHY and bootloader versions). The VL command has been deprecated in version 10C9. It is not supported in firmware versions after 10C8 | - | - |

Table 3-07. XBee®/XBee-PRO® Commands - Diagnostics

| AT Command | Command Category | Name and Description | Parameter Range | Default |
|---|---|---|---|---|
| HV (v1.x80*) | Diagnostics | Hardware Version. Read hardware version of the RF module. | 0 - 0xFFFF [read-only] | Factory-set |
| DB | Diagnostics | Received Signal Strength. Read signal level [in dB] of last good packet received (RSSI). Absolute value is reported. (For example: 0x58 = -88 dBm) Reported value is accurate between -40 dBm and RX sensitivity. | 0x17-0x5C (XBee)<br>0x24-0x64 (XBee-PRO)<br>[read-only] | - |
| EC (v1.x80*) | Diagnostics | CCA Failures. Reset/Read count of CCA (Clear Channel Assessment) failures. This parameter value increments when the module does not transmit a packet because it detected energy above the CCA threshold level set with CA command. This count saturates at its maximum value. Set count to “0” to reset count. | 0 - 0xFFFF | - |
| EA (v1.x80*) | Diagnostics | ACK Failures. Reset/Read count of acknowledgment failures. This parameter value increments when the module expires its transmission retries without receiving an ACK on a packet transmission. This count saturates at its maximum value. Set the parameter to “0” to reset count. | 0 - 0xFFFF | - |
| ED (v1.x80*) | Diagnostics | Energy Scan. Send ‘Energy Detect Scan’. ED parameter determines the length of scan on each channel. The maximal energy on each channel is returned and each value is followed by a carriage return. Values returned represent detected energy levels in units of -dBm. Actual scan time on each channel is measured as Time = [(2 ^ SD) * 15.36] ms. Total scan time is this time multiplied by the number of channels to be scanned. | 0 - 6 | - |

* Firmware version in which the command was first introduced (firmware versions are numbered in hexadecimal notation.)

### AT Command Options

Table 3-08. XBee®/XBee-PRO® Commands - AT Command Options

| AT Command | Command Category | Name and Description | Parameter Range | Default |
|---|---|---|---|---|
| CT | AT Command Mode Options | Command Mode Timeout. Set/Read the period of inactivity (no valid commands received) after which the RF module automatically exits AT Command Mode and returns to Idle Mode. | 2 - 0xFFFF [x 100 ms] | 0x64 (100d) |
| CN | AT Command Mode Options | Exit Command Mode. Explicitly exit the module from AT Command Mode. | -- | -- |
| AC (v1.xA0*) | AT Command Mode Options | Apply Changes. Explicitly apply changes to queued parameter value(s) and re-initialize module. | -- | -- |
| GT | AT Command Mode Options | Guard Times. Set required period of silence before and after the Command Sequence Characters of the AT Command Mode Sequence (GT+ CC + GT). The period of silence is used to prevent inadvertent entrance into AT Command Mode. | 2 - 0x0CE4 [x 1 ms] | 0x3E8<br>(1000d) |
| CC | AT Command Mode Options | Command Sequence Character. Set/Read the ASCII character value to be used between Guard Times of the AT Command Mode Sequence (GT+CC+GT). The AT Command Mode Sequence enters the RF module into AT Command Mode. | 0 - 0xFF | 0x2B<br>(‘+’ ASCII) |

* Firmware version in which the command was first introduced (firmware versions are numbered in hexadecimal notation.)

## Command Descriptions

Command descriptions in this section are listed alphabetically. Command categories are designated within "< >" symbols that follow each command title. XBee®/XBee-PRO® RF Modules expect parameter values in hexadecimal (designated by the "0x" prefix).

All modules operating within the same network should contain the same firmware version.

### A1 (End Device Association) Command

<Networking {Association}> The A1 command is used to set and read association options for an End Device.

Use the table below to determine End Device behavior in relation to the A1 parameter.

AT Command: ATA1

Parameter Range: 0 – 0x0F [bitfield]

Default Parameter Value: 0

Related Commands: ID (PAN ID), NI (Node Identifier), CH (Channel), CE (Coordinator Enable), A2 (Coordinator Association)

Minimum Firmware Version Required: v1.x80

| Bit number | End Device Association Option |
|---|---|
| 0 - ReassignPanID | 0 - Will only associate with Coordinator operating on PAN ID that matches Node Identifier |
| | 1 - May associate with Coordinator operating on any PAN ID |
| 1 - ReassignChannel | 0 - Will only associate with Coordinator operating on Channel that matches CH setting |
| | 1 - May associate with Coordinator operating on any Channel |
| 2 - AutoAssociate | 0 - Device will not attempt Association |
| | 1 - Device attempts Association until success<br>Note: This bit is used only for Non-Beacon systems. End Devices in a Beaconing system must always associate to a Coordinator |
| 3 - PollCoordOnPinWake | 0 - Pin Wake will not poll the Coordinator for pending (indirect) Data |
| | 1 - Pin Wake will send Poll Request to Coordinator to extract any pending data |
| 4 - 7 | [reserved] |

### A2 (Coordinator Association) Command

<Networking {Association}> The A2 command is used to set and read association options of the Coordinator.

Use the table below to determine Coordinator behavior in relation to the A2 parameter.

AT Command: ATA2

Parameter Range: 0 – 7 [bitfield]

Default Parameter Value: 0

Related Commands: ID (PAN ID), NI (Node Identifier), CH (Channel), CE (Coordinator Enable), A1 (End Device Association), AS Active Scan), ED (Energy Scan)

Minimum Firmware Version Required: v1.x80

| Bit number | End Device Association Option |
|---|---|
| 0 - ReassignPanID | 0 - Coordinator will not perform Active Scan to locate available PAN ID. It will operate on ID (PAN ID). |
| | 1 - Coordinator will perform Active Scan to determine an available ID (PAN ID). If a PAN ID conflict is found, the ID parameter will change. |
| 1 - ReassignChannel | 0 - Coordinator will not perform Energy Scan to determine free channel. It will operate on the channel determined by the CH parameter. |
| | 1 - Coordinator will perform Energy Scan to find a free channel, then operate on that channel. |
| 2 - AllowAssociate | 0 - Coordinator will not allow any devices to associate to it. |
| | 1 - Coordinator will allow devices to associate to it. |
| 3 - 7 | [reserved] |

The binary equivalent of the default value (0x06) is 00000110. 'Bit 0' is the last digit of the sequence.

### AC (Apply Changes) Command

<AT Command Mode Options> The AC command is used to explicitly apply changes to module parameter values. ‘Applying changes’ means that the module is re-initialized based on changes made to its parameter values. Once changes are applied, the module immediately operates according to the new parameter values.

AT Command: ATAC

Minimum Firmware Version Required: v1.xA0

This behavior is in contrast to issuing the WR (Write) command. The WR command saves parameter values to non-volatile memory, but the module still operates according to previously saved values until the module is re-booted or the CN (Exit AT Command Mode) command is issued.

Refer to the “AT Command – Queue Parameter Value” API type for more information.

### AI (Association Indication) Command

<Networking {Association}> The AI command is used to indicate occurrences of errors during the last association request.

Use the table below to determine meaning of the returned values.

AT Command: ATAI

Parameter Range: 0 – 0x13 [read-only]

Related Commands: AS (Active Scan), ID (PAN ID), CH (Channel), ED (Energy Scan), A1 (End Device Association), A2 (Coordinator Association), CE (Coordinator Enable)

Minimum Firmware Version Required: v1.x80

| Returned Value (Hex) | Association Indication |
|---|---|
| 0x00 | Successful Completion - Coordinator successfully started or End Device association complete |
| 0x01 | Active Scan Timeout |
| 0x02 | Active Scan found no PANs |
| 0x03 | Active Scan found PAN, but the Coordinator Allow Association bit is not set |
| 0x04 | Active Scan found PAN, but Coordinator and End Device are not configured to support beacons |
| 0x05 | Active Scan found PAN, but Coordinator ID (PAN ID) value does not match the ID of the End Device |
| 0x06 | Active Scan found PAN, but Coordinator CH (Channel) value does not match the CH of the End Device |
| 0x07 | Energy Scan Timeout |
| 0x08 | Coordinator start request failed |
| 0x09 | Coordinator could not start due to Invalid Parameter |
| 0x0A | Coordinator Realignment is in progress |
| 0x0B | Association Request not sent |
| 0x0C | Association Request timed out - no reply was received |
| 0x0D | Association Request had an Invalid Parameter |
| 0x0E | Association Request Channel Access Failure - Request was not transmitted - CCA failure |
| 0x0F | Remote Coordinator did not send an ACK after Association Request was sent |
| 0x10 | Remote Coordinator did not reply to the Association Request, but an ACK was received after sending the request |
| 0x11 | [reserved] |
| 0x12 | Sync-Loss - Lost synchronization with a Beaconing Coordinator |
| 0x13 | Disassociated - No longer associated to Coordinator |
| 0xFF | RF Module is attempting to associate |

## AP (API Enable) Command

<Serial Interfacing> The AP command is used to enable the RF module to operate using a frame-based API instead of using the default Transparent (UART) mode.

| AT Command: ATAP | |
|---|---|
| Parameter Range:0 – 2 | |
| Parameter | Configuration |
| 0 | Disabled<br>(Transparent operation) |
| 1 | API enabled |
| 2 | API enabled<br>(with escaped characters) |
| Default Parameter Value:0 | |
| Minimum Firmware Version Required: v1.x80 | |

Refer to the API Operation section when API operation is enabled (AP = 1 or 2).

## AS (Active Scan) Command

<Network {Association}> The AS command is used to send a Beacon Request to a Broadcast (0xFFFF) and Broadcast PAN (0xFFFF) on every channel. The parameter determines the amount of time the RF module will listen for Beacons on each channel. A 'PanDescriptor' is created and returned for every Beacon received from the scan. Each PanDescriptor contains the following information:

| AT Command: ATAS |
|---|
| Parameter Range: 0 – 6 |
| Related Command: SD (Scan Duration), DL (Destination Low Address), DH (Destination High Address), ID (PAN ID), CH (Channel) |
| Minimum Firmware Version Required: v1.x80 |

CoordAddress (SH + SL parameters)<CR> (NOTE: If MY on the coordinator is set less than 0xFFFF, the MY value is displayed)
CoordPanID (ID parameter)<CR>
CoordAddrMode <CR>
  0x02 = 16-bit Short Address
  0x03 = 64-bit Long Address
Channel (CH parameter) <CR>
SecurityUse<CR>
ACLEntry<CR>
SecurityFailure<CR>
SuperFrameSpec<CR> (2 bytes):
  bit 15 - Association Permitted (MSB)
  bit 14 - PAN Coordinator
  bit 13 - Reserved
  bit 12 - Battery Life Extension
  bits 8-11 - Final CAP Slot
  bits 4-7 - Superframe Order
  bits 0-3 - Beacon Order
GtsPermit<CR>
RSSI<CR> (- RSSI is returned as -dBm)
TimeStamp<CR> (3 bytes)
<CR> (A carriage return <CR> is sent at the end of the AS command.

The Active Scan is capable of returning up to 5 PanDescriptors in a scan. The actual scan time on each channel is measured as Time = [(2 ^ (SD Parameter)) * 15.36] ms. Total scan time is this time multiplied by the number of channels to be scanned (16 for the XBee, 12 for the XBee-PRO).

NOTE: Refer the scan table in the SD description to determine scan times. If using API Mode, no <CR>'s are returned in the response. Refer to the API Mode Operation section.

### BD (Interface Data Rate) Command

<Serial Interfacing> The BD command is used to set and read the serial interface data rate used between the RF module and host. This parameter determines the rate at which serial data is sent to the module from the host. Modified interface data rates do not take effect until the CN (Exit AT Command Mode) command is issued and the system returns the 'OK' response.

When parameters 0-7 are sent to the module, the respective interface data rates are used (as shown in the table on the right).

The RF data rate is not affected by the BD parameter. If the interface data rate is set higher than the RF data rate, a flow control configuration may need to be implemented.

AT Command: ATBD

Parameter Range:0 – 7 (standard rates)
0x80-0x3D090 (non-standard rates up to 250 Kbps)

| Parameter | Configuration (bps) |
|---|---|
| 0 | 1200 |
| 1 | 2400 |
| 2 | 4800 |
| 3 | 9600 |
| 4 | 19200 |
| 5 | 38400 |
| 6 | 57600 |
| 7 | 115200 |

Default Parameter Value:3

**Non-standard Interface Data Rates:**

Any value above 0x07 will be interpreted as an actual baud rate. When a value above 0x07 is sent, the closest interface data rate represented by the number is stored in the BD register. For example, a rate of 19200 bps can be set by sending the following command line "ATBD4B00". NOTE: When using Digi's X-CTU Software, non-standard interface data rates can only be set and read using the X-CTU 'Terminal' tab. Non-standard rates are not accessible through the 'Modem Configuration' tab.

When the BD command is sent with a non-standard interface data rate, the UART will adjust to accommodate the requested interface rate. In most cases, the clock resolution will cause the stored BD parameter to vary from the parameter that was sent (refer to the table below). Reading the BD command (send "ATBD" command without an associated parameter value) will return the value actually stored in the module's BD register.

**Parameters Sent Versus Parameters Stored**

| BD Parameter Sent (HEX) | Interface Data Rate (bps) | BD Parameter Stored (HEX) |
|---|---|---|
| 0 | 1200 | 0 |
| 4 | 19,200 | 4 |
| 7 | 115,200* | 7 |
| 12C | 300 | 12B |
| 1C200 | 115,200 | 1B207 |

* The 115,200 baud rate setting is actually at 111,111 baud (-3.5% target UART speed).

### CA (CCA Threshold) Command

<RF Interfacing> CA command is used to set and read CCA (Clear Channel Assessment) thresholds.

Prior to transmitting a packet, a CCA is performed to detect energy on the transmit channel. If the detected energy is above the CCA Threshold, the RF module will not transmit the packet.

AT Command: ATCA

Parameter Range: 0 – 0x50 [-dBm]

Default Parameter Value: 0x2C
(-44 decimal dBm)

Minimum Firmware Version Required: v1.x80

### CC (Command Sequence Character) Command

<AT Command Mode Options> The CC command is used to set and read the ASCII character used between guard times of the AT Command Mode Sequence (GT + CC + GT). This sequence enters the RF module into AT Command Mode so that data entering the module from the host is recognized as commands instead of payload.

| |
|---|
| AT Command: ATCC |
| Parameter Range: 0 - 0xFF |
| Default Parameter Value: 0x2B (ASCII "+") |
| Related Command: GT (Guard Times) |

The AT Command Sequence is explained further in the AT Command Mode section.

### CE (Coordinator Enable) Command

<Networking {Association} The CE command is used to set and read the behavior (End Device vs. Coordinator) of the RF module.

| | |
|---|---|
| AT Command: ATCE | |
| Parameter Range:0 - 1 | |
| Parameter | Configuration |
| 0 | End Device |
| 1 | Coordinator |
| Default Parameter Value:0 | |
| Minimum Firmware Version Required: v1.x80 | |

### CH (Channel) Command

<Networking {Addressing}> The CH command is used to set/read the operating channel on which RF connections are made between RF modules. The channel is one of three addressing options available to the module. The other options are the PAN ID (ID command) and destination addresses (DL & DH commands).

| |
|---|
| AT Command: ATCH |
| Parameter Range: 0x0B - 0x1A (XBee)<br>0x0C - 0x17 (XBee-PRO) |
| Default Parameter Value: 0x0C (12 decimal) |
| Related Commands: ID (PAN ID), DL (Destination Address Low, DH (Destination Address High) |

In order for modules to communicate with each other, the modules must share the same channel number. Different channels can be used to prevent modules in one network from listening to transmissions of another. Adjacent channel rejection is 23 dB.

The module uses channel numbers of the 802.15.4 standard.
Center Frequency = 2.405 + (CH - 11d) * 5 MHz (d = decimal)

Refer to the XBee/XBee-PRO Addressing section for more information.

### CN (Exit Command Mode) Command

<AT Command Mode Options> The CN command is used to explicitly exit the RF module from AT Command Mode.

| |
|---|
| AT Command: ATCN |

### CT (Command Mode Timeout) Command

<AT Command Mode Options> The CT command is used to set and read the amount of inactive time that elapses before the RF module automatically exits from AT Command Mode and returns to Idle Mode.

Use the CN (Exit Command Mode) command to exit AT Command Mode manually.

| |
|---|
| AT Command: ATCT |
| Parameter Range:2 - 0xFFFF<br>[x 100 milliseconds] |
| Default Parameter Value: 0x64 (100 decimal (which equals 10 decimal seconds)) |
| Number of bytes returned: 2 |
| Related Command: CN (Exit Command Mode) |

### D0 - D4 (DIOn Configuration) Commands

<I/O Settings> The D0, D1, D2, D3 and D4 commands are used to select/read the behavior of their respective AD/DIO lines (pins 20, 19, 18, 17 and 11 respectively).

Options include:

- Analog-to-digital converter
- Digital input
- Digital output

AT Commands:
ATD0, ATD1, ATD2, ATD3, ATD4

Parameter Range:0 – 5

| Parameter | Configuration |
|---|---|
| 0 | Disabled |
| 1 | n/a |
| 2 | ADC |
| 3 | DI |
| 4 | DO low |
| 5 | DO high |

Default Parameter Value:0

Minimum Firmware Version Required: 1.x.A0

### D5 (DIO5 Configuration) Command

<I/O Settings> The D5 command is used to select/read the behavior of the DIO5 line (pin 15).

Options include:

- Associated Indicator (LED blinks when the module is associated)
- Analog-to-digital converter
- Digital input
- Digital output

AT Command: ATD5

Parameter Range:0 – 5

| Parameter | Configuration |
|---|---|
| 0 | Disabled |
| 1 | Associated Indicator |
| 2 | ADC |
| 3 | DI |
| 4 | DO low |
| 5 | DO high |

Default Parameter Value:1

Parameters 2-5 supported as of firmware version 1.xA0

### D6 (DIO6 Configuration) Command

<I/O Settings> The D6 command is used to select/read the behavior of the DIO6 line (pin 16).
Options include:

- RTS flow control
- Analog-to-digital converter
- Digital input
- Digital output

AT Command: ATD6

Parameter Range:0 – 5

| Parameter | Configuration |
|---|---|
| 0 | Disabled |
| 1 | RTS Flow Control |
| 2 | n/a |
| 3 | DI |
| 4 | DO low |
| 5 | DO high |

Default Parameter Value:0

Parameters 3-5 supported as of firmware version 1.xA0

### D7 (DIO7 Configuration) Command

<I/O Settings> The D7 command is used to select/read the behavior of the DIO7 line (pin 12). Options include:

- CTS flow control
- Analog-to-digital converter
- Digital input
- Digital output
- RS485 TX Enable (this output is 3V CMOS level, and is useful in a 3V CMOS to RS485 conversion circuit)

AT Command: ATD7

Parameter Range:0 – 5

| Parameter | Configuration |
|---|---|
| 0 | Disabled |
| 1 | CTS Flow Control |
| 2 | n/a |
| 3 | DI |
| 4 | DO low |
| 5 | DO high |
| 6 | RS485 TX Enable Low |
| 7 | RS485 TX Enable High |

Default Parameter Value:1

Parameters 3-7 supported as of firmware version 1.x.A0

### D8 (DI8 Configuration) Command

<I/O Settings> The D8 command is used to select/read the behavior of the DI8 line (pin 9). This command enables configuring the pin to function as a digital input. This line is also used with Pin Sleep.

AT Command: ATD8

Parameter Range:0 – 5
(1, 2, 4 & 5 n/a)

| Parameter | Configuration |
|---|---|
| 0 | Disabled |
| 3 | DI |

Default Parameter Value:0

Minimum Firmware Version Required: 1.xA0

### DA (Force Disassociation) Command

<(Special)> The DA command is used to immediately disassociate an End Device from a Coordinator and reattempt to associate.

AT Command: ATDA

Minimum Firmware Version Required: v1.x80

### DB (Received Signal Strength) Command

<Diagnostics> DB parameter is used to read the received signal strength (in dBm) of the last RF packet received. Reported values are accurate between -40 dBm and the RF module's receiver sensitivity.

AT Command: ATDB

Parameter Range [read-only]:
0x17-0x5C (XBee), 0x24-0x64 (XBee-PRO)

Absolute values are reported. For example: 0x58 = -88 dBm (decimal). If no packets have been received (since last reset, power cycle or sleep event), “0” will be reported.

### DH (Destination Address High) Command

<Networking {Addressing}> The DH command is used to set and read the upper 32 bits of the RF module's 64-bit destination address. When combined with the DL (Destination Address Low) parameter, it defines the destination address used for transmission.

An module will only communicate with other modules having the same channel (CH parameter), PAN ID (ID parameter) and destination address (DH + DL parameters).

AT Command: ATDH

Parameter Range: 0 – 0xFFFFFFFF

Default Parameter Value: 0

Related Commands: DL (Destination Address Low), CH (Channel), ID (PAN VID), MY (Source Address)

To transmit using a 16-bit address, set the DH parameter to zero and the DL parameter less than 0xFFFF. 0x000000000000FFFF (DL concatenated to DH) is the broadcast address for the PAN.

Refer to the XBee/XBee-PRO Addressing section for more information.

### DL (Destination Address Low) Command

| |
|---|
| AT Command: ATDL |
| Parameter Range: 0 - 0xFFFFFFFF |
| Default Parameter Value: 0 |
| Related Commands: DH (Destination Address High), CH (Channel), ID (PAN VID), MY (Source Address) |

<Networking {Addressing}> The DL command is used to set and read the lower 32 bits of the RF module's 64-bit destination address. When combined with the DH (Destination Address High) parameter, it defines the destination address used for transmission.

A module will only communicate with other modules having the same channel (CH parameter), PAN ID (ID parameter) and destination address (DH + DL parameters).

To transmit using a 16-bit address, set the DH parameter to zero and the DL parameter less than 0xFFFF. 0x000000000000FFFF (DL concatenated to DH) is the broadcast address for the PAN.

Refer to the XBee/XBee-PRO Addressing section for more information.

### DN (Destination Node) Command

| |
|---|
| AT Command: ATDN |
| Parameter Range: 20-character ASCII String |
| Minimum Firmware Version Required: v1.x80 |

<Networking {Identification}> The DN command is used to resolve a NI (Node Identifier) string to a physical address. The following events occur upon successful command execution:

1. DL and DH are set to the address of the module with the matching NI (Node Identifier).
2. ‘OK’ is returned.
3. RF module automatically exits AT Command Mode.

If there is no response from a modem within 200 msec or a parameter is not specified (left blank), the command is terminated and an ‘ERROR’ message is returned.

### DP (Disassociation Cyclic Sleep Period) Command

| |
|---|
| AT Command: ATDP |
| Parameter Range: 1 – 0x68B0<br>[x 10 milliseconds] |
| Default Parameter Value:0x3E8<br>(1000 decimal) |
| Related Commands: SM (Sleep Mode), SP (Cyclic Sleep Period), ST (Time before Sleep) |
| Minimum Firmware Version Required: v1.x80 |

<Sleep Mode (Low Power)>

**NonBeacon Firmware**

*End Device* - The DP command is used to set and read the time period of sleep for cyclic sleeping remotes that are configured for Association but are not associated to a Coordinator. (i.e. If a device is configured to associate, configured as a Cyclic Sleep remote, but does not find a Coordinator; it will sleep for DP time before reattempting association.) Maximum sleep period is 268 seconds (0x68B0). DP should be > 0 for NonBeacon systems.

### EA (ACK Failures) Command

| |
|---|
| AT Command: ATEA |
| Parameter Range:0 – 0xFFFF |
| Minimum Firmware Version Required: v1.x80 |

<Diagnostics> The EA command is used to reset and read the count of ACK (acknowledgement) failures. This parameter value increments when the module expires its transmission retries without receiving an ACK on a packet transmission. This count saturates at its maximum value.

Set the parameter to “0” to reset count.

### EC (CCA Failures) Command

<Diagnostics> The EC command is used to read and reset the count of CCA (Clear Channel Assessment) failures. This parameter value increments when the RF module does not transmit a packet due to the detection of energy that is above the CCA threshold level (set with CA command). This count saturates at its maximum value.

Set the EC parameter to "0" to reset count.

| AT Command: ATEC |
|---|
| Parameter Range:0 – 0xFFFF |
| Related Command: CA (CCA Threshold) |
| Minimum Firmware Version Required: v1.x80 |

### ED (Energy Scan) Command

<Networking {Association}> The ED command is used to send an "Energy Detect Scan". This parameter determines the length of scan on each channel. The maximal energy on each channel is returned and each value is followed by a carriage return. An additional carriage return is sent at the end of the command.

| AT Command: ATED |
|---|
| Parameter Range:0 – 6 |
| Related Command: SD (Scan Duration), SC (Scan Channel) |
| Minimum Firmware Version Required: v1.x80 |

The values returned represent the detected energy level in units of -dBm. The actual scan time on each channel is measured as Time = [(2 ^ ED PARAM) * 15.36] ms.

Note: Total scan time is this time multiplied by the number of channels to be scanned. Also refer to the SD (Scan Duration) table. Use the SC (Scan Channel) command to choose which channels to scan.

### EE (AES Encryption Enable) Command

<Networking {Security}> The EE command is used to set/read the parameter that disables/enables 128-bit AES encryption.

The XBee®/XBee-PRO® firmware uses the 802.15.4 Default Security protocol and uses AES encryption with a 128-bit key. AES encryption dictates that all modules in the network use the same key and the maximum RF packet size is 95 Bytes.

When encryption is enabled, the module will always use its 64-bit long address as the source address for RF packets. This does not affect how the MY (Source Address), DH (Destination Address High) and DL (Destination Address Low) parameters work

| AT Command: ATEE | |
|---|---|
| Parameter Range:0 – 1 | |
| Parameter | Configuration |
| 0 | Disabled |
| 1 | Enabled |
| Default Parameter Value:0 | |
| Related Commands: KY (Encryption Key), AP (API Enable), MM (MAC Mode) | |
| Minimum Firmware Version Required: v1.xA0 | |

If MM (MAC Mode) > 0 and AP (API Enable) parameter > 0:
With encryption enabled and a 16-bit short address set, receiving modules will only be able to issue RX (Receive) 64-bit indicators. This is not an issue when MM = 0.

If a module with a non-matching key detects RF data, but has an incorrect key: When encryption is enabled, non-encrypted RF packets received will be rejected and will not be sent out the UART.

Transparent Operation --> All RF packets are sent encrypted if the key is set.

API Operation --> Receive frames use an option bit to indicate that the packet was encrypted.

### FP (Force Poll) Command

<Networking (Association)> The FP command is used to request indirect messages being held by a Coordinator.

| AT Command: ATFP |
|---|
| Minimum Firmware Version Required: v1.x80 |

### FR (Software Reset) Command

<Special> The FR command is used to force a software reset on the RF module. The reset simulates powering off and then on again the module.

AT Command: ATFR

Minimum Firmware Version Required: v1.x80

### GT (Guard Times) Command

<AT Command Mode Options> GT Command is used to set the DI (data in from host) time-of-silence that surrounds the AT command sequence character (CC Command) of the AT Command Mode sequence (GT + CC + GT).

The DI time-of-silence is used to prevent inadvertent entrance into AT Command Mode.

Refer to the Command Mode section for more information regarding the AT Command Mode Sequence.

AT Command: ATGT

Parameter Range:2 – 0x0CE4
[x 1 millisecond]

Default Parameter Value:0x3E8
(1000 decimal)

Related Command: CC (Command Sequence Character)

### HV (Hardware Version) Command

<Diagnostics> The HV command is used to read the hardware version of the RF module.

AT Command: ATHV

Parameter Range:0 – 0xFFFF [Read-only]

Minimum Firmware Version Required: v1.x80

### IA (I/O Input Address) Command

<I/O Settings {I/O Line Passing}> The IA command is used to bind a module output to a specific address. Outputs will only change if received from this address. The IA command can be used to set/read both 16 and 64-bit addresses.

Setting all bytes to 0xFF will not allow the reception of any I/O packet to change outputs. Setting the IA address to 0xFFFF will cause the module to accept all I/O packets.

AT Command: ATIA

Parameter Range:0 – 0xFFFFFFFFFFFFFFFF

Default Parameter Value:0xFFFFFFFFFFFFFFFF (will not allow any received I/O packet to change outputs)

Minimum Firmware Version Required: v1.xA0

### IC (DIO Change Detect) Command

<I/O Settings> Set/Read bitfield values for change detect monitoring. Each bit enables monitoring of DIO0 - DIO7 for changes.

If detected, data is transmitted with DIO data only. Any samples queued waiting for transmission will be sent first.

AT Command: ATIC

Parameter Range:0 – 0xFF [bitfield]

Default Parameter Value:0 (disabled)

Minimum Firmware Version Required: 1.xA0

Refer to the “ADC and Digital I/O Line Support” sections of the “RF Module Operations” chapter for more information.

### ID (Pan ID) Command

<Networking {Addressing}> The ID command is used to set and read the PAN (Personal Area Network) ID of the RF module. Only modules with matching PAN IDs can communicate with each other. Unique PAN IDs enable control of which RF packets are received by a module.

AT Command: ATID

Parameter Range: 0 – 0xFFFF

Default Parameter Value:0x3332
(13106 decimal)

Setting the ID parameter to 0xFFFF indicates a global transmission for all PANs. It does not indicate a global receive.

### IO (Digital Output Level) Command

<I/O Settings> The IO command is used to set digital output levels. This allows DIO lines setup as outputs to be changed through Command Mode.

| |
|---|
| AT Command: ATIO |
| Parameter Range: 8-bit bitmap<br>(where each bit represents the level of an I/O line that is setup as an output.) |
| Minimum Firmware Version Required: v1.xA0 |

### IR (Sample Rate) Command

<I/O Settings> The IR command is used to set/ read the sample rate. When set, the module will sample all enabled DIO/ADC lines at a specified interval. This command allows periodic reads of the ADC and DIO lines in a non-Sleep Mode setup. A sample rate which requires transmissions at a rate greater than once every 20ms is not recommended.

| |
|---|
| AT Command: ATIR |
| Parameter Range: 0 - 0xFFFF [x 1 msec]<br>(cannot guarantee 1 ms timing when IT=1) |
| Default Parameter Value:0 |
| Related Command: IT (Samples before TX) |
| Minimum Firmware Version Required: v1.xA0 |

Example: When IR = 0x14, the sample rate is 20 ms (or 50 Hz).

### IS (Force Sample) Command

<I/O Settings> The IS command is used to force a read of all enabled DIO/ADC lines. The data is returned through the UART.

When operating in Transparent Mode (AP=0), the data is retuned in the following format:

| |
|---|
| AT Command: ATIS |
| Minimum Firmware Version Required: v1.xA0 |

```
All bytes are converted to ASCII:
    number of samples<CR>
    channel mask<CR>
    DIO data<CR> (If DIO lines are enabled<CR>
    ADC channel Data<cr> <-This will repeat for every enabled ADC channel<CR>
    <CR>  (end of data noted by extra <CR>)
```

When operating in API mode (AP > 0), the command will immediately return an 'OK' response. The data will follow in the normal API format for DIO data.

### IT (Samples before TX) Command

<I/O Settings> The IT command is used to set/ read the number of DIO and ADC samples to collect before transmitting data.

One ADC sample is considered complete when all enabled ADC channels have been read. The module can buffer up to 93 Bytes of sample data. Since the module uses a 10-bit A/D converter, each sample uses two Bytes. This leads to a maximum buffer size of 46 samples or IT=0x2E.

| |
|---|
| AT Command: ATIT |
| Parameter Range: 1 - 0xFF |
| Default Parameter Value:1 |
| Minimum Firmware Version Required: v1.xA0 |

When Sleep Modes are enabled and IR (Sample Rate) is set, the module will remain awake until IT samples have been collected.

### IU (I/O Output Enable) Command

<I/O Settings> The IU command is used to disable/enable I/O UART output. When enabled (IU = 1), received I/O line data packets are sent out the UART. The data is sent using an API frame regardless of the current AP parameter value.

| | |
|---|---|
| AT Command: ATIU | |
| Parameter Range:0 – 1 | |
| Parameter | Configuration |
| 0 | Disabled –<br>Received I/O line data packets will be NOT sent out UART. |
| 1 | Enabled –<br>Received I/O line data will be sent out UART |
| Default Parameter Value:1 | |
| Minimum Firmware Version Required: 1.xA0 | |

### KY (AES Encryption Key) Command

<Networking {Security}> The KY command is used to set the 128-bit AES (Advanced Encryption Standard) key for encrypting/decrypting data. Once set, the key cannot be read out of the module by any means.

| |
|---|
| AT Command: ATKY |
| Parameter Range:0 – (any 16-Byte value) |
| Default Parameter Value:0 |
| Related Command: EE (Encryption Enable) |
| Minimum Firmware Version Required: v1.xA0 |

The entire payload of the packet is encrypted using the key and the CRC is computed across the ciphertext. When encryption is enabled, each packet carries an additional 16 Bytes to convey the random CBC Initialization Vector (IV) to the receiver(s). The KY value may be “0” or any 128-bit value. Any other value, including entering KY by itself with no parameters, is invalid. All ATKY entries (valid or not) are received with a returned 'OK'.

A module with the wrong key (or no key) will receive encrypted data, but the data driven out the serial port will be meaningless. A module with a key and encryption enabled will receive data sent from a module without a key and the correct unencrypted data output will be sent out the serial port. Because CBC mode is utilized, repetitive data appears differently in different transmissions due to the randomly-generated IV.

When queried, the system will return an ‘OK’ message and the value of the key will not be returned.

### M0 (PWM0 Output Level) Command

<I/O Settings> The M0 command is used to set the output level of the PWM0 line (pin 6).

| |
|---|
| AT Command: ATM0 |
| Parameter Range:0 – 0x03FF [steps] |
| Default Parameter Value:0 |
| Related Commands: P0 (PWM0 Enable), AC (Apply Changes), CN (Exit Command Mode) |
| Minimum Firmware Version Required: v1.xA0 |

Before setting the line as an output:

1. Enable PWM0 output (P0 = 2)
2. Apply settings (use CN or AC)

The PWM period is 64 µsec and there are 0x03FF (1023 decimal) steps within this period. When M0 = 0 (0% PWM), 0x01FF (50% PWM), 0x03FF (100% PWM), etc.

### M1 (PWM1 Output Level) Command

<I/O Settings> The M1 command is used to set the output level of the PWM1 line (pin 7).

| |
|---|
| AT Command: ATM1 |
| Parameter Range:0 – 0x03FF |
| Default Parameter Value:0 |
| Related Commands: P1 (PWM1 Enable), AC (Apply Changes), CN (Exit Command Mode) |
| Minimum Firmware Version Required: v1.xA0 |

Before setting the line as an output:

1. Enable PWM1 output (P1 = 2)
2. Apply settings (use CN or AC)

### MM (MAC Mode) Command

<Networking {Addressing}> The MM command is used to set and read the MAC Mode value. The MM command disables/enables the use of a Digi header contained in the 802.15.4 RF packet. By default (MM = 0), Digi Mode is enabled and the module adds an extra header to the data portion of the 802.15.4 packet. This enables the following features:

- ND and DN command support
- Duplicate packet detection when using ACKs
- "RR command
- "DIO/AIO sampling support

The MM command allows users to turn off the use of the extra header. Modes 1 and 2 are strict 802.15.4 modes. If the Digi header is disabled, ND and DN parameters are also disabled.

Note: When MM=1 or 3, MAC and CCA failure retries are not supported.

AT Command: ATMM

Parameter Range:0 – 3

| Parameter | Configuration |
|---|---|
| 0 | Digi Mode (802.15.4 + Digi header) |
| 1 | 802.15.4 (no ACKs) |
| 2 | 802.15.4 (with ACKs) |
| 3 | Digi Mode (no ACKs) |

Default Parameter Value:0

Related Commands: ND (Node Discover), DN (Destination Node)

Minimum Firmware Version Required: v1.x80

### MY (16-bit Source Address) Command

<Networking {Addressing}> The MY command is used to set and read the 16-bit source address of the RF module.

By setting MY to 0xFFFF, the reception of RF packets having a 16-bit address is disabled. The 64-bit address is the module's serial number and is always enabled.

AT Command: ATMY

Parameter Range: 0 – 0xFFFF

Default Parameter Value: 0

Related Commands: DH (Destination Address High), DL (Destination Address Low), CH (Channel), ID (PAN ID)

### NB (Parity) Command

<Serial Interfacing> The NB command is used to select/read the parity settings of the RF module for UART communications.

**Note**: the module does not actually calculate and check the parity; it only interfaces with devices at the configured parity and stop bit settings.

AT Command: ATNB

Parameter Range: 0 – 4

| Parameter | Configuration |
|---|---|
| 0 | 8-bit no parity |
| 1 | 8-bit even |
| 2 | 8-bit odd |
| 3 | 8-bit mark |
| 4 | 8-bit space |

Default Parameter Value: 0

Number of bytes returned: 1

### ND (Node Discover) Command

| |
|---|
| AT Command: ATND |
| Range: optional 20-character NI value |
| Related Commands: CH (Channel), ID (Pan ID), MY (Source Address), SH (Serial Number High), SL (Serial Number Low), NI (Node Identifier), NT (Node Discover Time) |
| Minimum Firmware Version Required: v1.x80 |

<Networking {Identification}> The ND command is used to discover and report all modules on its current operating channel (CH parameter) and PAN ID (ID parameter). ND also accepts an NI (Node Identifier) value as a parameter. In this case, only a module matching the supplied identifier will respond.

ND uses a 64-bit long address when sending and responding to an ND request. The ND command causes a module to transmit a globally addressed ND command packet. The amount of time allowed for responses is determined by the NT (Node Discover Time) parameter.

In AT Command mode, command completion is designated by a carriage return (0x0D). Since two carriage returns end a command response, the application will receive three carriage returns at the end of the command. If no responses are received, the application should only receive one carriage return. When in API mode, the application should receive a frame (with no data) and status (set to 'OK') at the end of the command. When the ND command packet is received, the remote sets up a random time delay (up to 2.2 sec) before replying as follows:

Node Discover Response (AT command mode format - Transparent operation):
MY (Source Address) value<CR>
SH (Serial Number High) value<CR>
SL (Serial Number Low) value<CR>
DB (Received Signal Strength) value<CR>
NI (Node Identifier) value<CR>
<CR> (This is part of the response and not the end of command indicator.)

Node Discover Response (API format - data is binary (except for NI)):
2 bytes for MY (Source Address) value
4 bytes for SH (Serial Number High) value
4 bytes for SL (Serial Number Low) value
1 byte for DB (Received Signal Strength) value
NULL-terminated string for NI (Node Identifier) value (max 20 bytes w/out NULL terminator)

### NI (Node Identifier) Command

| |
|---|
| AT Command: ATNI |
| Parameter Range: 20-character ASCII string |
| Related Commands: ND (Node Discover), DN (Destination Node) |
| Minimum Firmware Version Required: v1.x80 |

<Networking {Identification}> The NI command is used to set and read a string for identifying a particular node.

Rules:

- Register only accepts printable ASCII data.
- A string can not start with a space.
- A carriage return ends command
- Command will automatically end when maximum bytes for the string have been entered.

This string is returned as part of the ND (Node Discover) command. This identifier is also used with the DN (Destination Node) command.

### NO (Node Discover Options) Command

| |
|---|
| AT Command: ATNO |
| Parameter Range: "0-1 |
| Related Commands: ND (Node Discover), DN (Destination Node) |
| Minimum Firmware Version Required: v1.xC5 |

<Networking {Identification}> The NO command is used to suppress/include a self-response to Node Discover commands. When NO=1 a module doing a Node Discover will include a response entry for itself.

### NT (Node Discover Time) Command

<Networking {Identification}> The NT command is used to set the amount of time a base node will wait for responses from other nodes when using the ND (Node Discover) command. The NT value is transmitted with the ND command.

Remote nodes will set up a random hold-off time based on this time. The remotes will adjust this time down by 250 ms to give each node the ability to respond before the base ends the command. Once the ND command has ended, any response received on the base will be discarded.

| |
|---|
| AT Command: ATNT |
| Parameter Range: 0x01 – 0xFC<br>[x 100 msec] |
| Default: 0x19 (2.5 decimal seconds) |
| Related Commands: ND (Node Discover) |
| Minimum Firmware Version Required: 1.xA0 |

### P0 (PWM0 Configuration) Command

<I/O Setting {I/O Line Passing}> The P0 command is used to select/read the function for PWM0 (Pulse Width Modulation output 0). This command enables the option of translating incoming data to a PWM so that the output can be translated back into analog form.

With the IA (I/O Input Address) parameter correctly set, AD0 values can automatically be passed to PWM0.

AT Command: ATP0
The second character in the command is the number zero (“0”), not the letter “O”.

Parameter Range: 0 – 2

| Parameter | Configuration |
|---|---|
| 0 | Disabled |
| 1 | RSSI |
| 2 | PWM0 Output |

Default Parameter Value: 1

### P1 (PWM1 Configuration) Command

<I/O Setting {I/O Line Passing}> The P1 command is used to select/read the function for PWM1 (Pulse Width Modulation output 1). This command enables the option of translating incoming data to a PWM so that the output can be translated back into analog form.

With the IA (I/O Input Address) parameter correctly set, AD1 values can automatically be passed to PWM1.

AT Command: ATP1

Parameter Range: 0 – 2

| Parameter | Configuration |
|---|---|
| 0 | Disabled |
| 1 | RSSI |
| 2 | PWM1 Output |

Default Parameter Value: 0

Minimum Firmware Version Required: v1.xA0

### PL (Power Level) Command

<RF Interfacing> The PL command is used to select and read the power level at which the RF module transmits conducted power.

When operating in Europe, XBee-PRO 802.15.4 modules must operate at or below a transmit power output level of 10dBm. Customers have 2 choices for transmitting at or below 10dBm:

- Order the standard XBee-PRO module and change the PL command to "0" (10dBm),
- Order the Japan variant of the XBee-PRO module, which has a maximum transmit output power of 10dBm.

AT Command: ATPL

Parameter Range: 0 – 4

| Parameter | XBee | XBee-PRO | XBee-PRO Japan variant |
|---|---|---|---|
| 0 | -10 dBm | 10 dBm | PL=4: 10 dBm |
| 1 | -6 dBm | 12 dBm | PL=3: 8 dBm |
| 2 | -4 dBm | 14 dBm | PL=2: 2 dBm |
| 3 | -2 dBm | 16 dBm | PL=1: -3 dBm |
| 4 | 0 dBm | 18 dBm | PL=0: -3 dBm |

Default Parameter Value: 4

### PR (Pull-up Resistor) Command

<Serial Interfacing> The PR command is used to set and read the bit field that is used to configure internal the pull-up resistor status for I/O lines. “1” specifies the pull-up resistor is enabled. “0” specifies no pull up.

| |
|---|
| AT Command: ATPR |
| Parameter Range: 0 – 0xFF |
| Default Parameter Value: 0xFF<br>(all pull–up resistors are enabled) |
| Minimum Firmware Version Required: v1.x80 |

bit 0 - AD4/DIO4 (pin 11)
bit 1 - AD3/DIO3 (pin 17)
bit 2 - AD2/DIO2 (pin 18)
bit 3 - AD1/DIO1 (pin 19)
bit 4 - AD0/DIO0 (pin 20)
bit 5 - AD6/DIO6 (pin 16)
bit 6 - DI8 (pin 9)
bit 7 - DIN/CONFIG (pin 3)

For example: Sending the command “ATPR 6F” will turn bits 0, 1, 2, 3, 5 and 6 ON; and bits 4 & 7 will be turned OFF. (The binary equivalent of “0x6F” is “01101111”. Note that ‘bit 0’ is the last digit in the bitfield.

### PT (PWM Output Timeout) Command

<I/O Settings {I/O Line Passing}> The PT command is used to set/read the output timeout value for both PWM outputs.

When PWM is set to a non-zero value: Due to I/O line passing, a time is started which when expired will set the PWM output to zero. The timer is reset when a valid I/O packet is received.

| |
|---|
| AT Command: ATPT |
| Parameter Range: 0 – 0xFF [x 100 msec] |
| Default Parameter Value: 0xFF |
| Minimum Firmware Version Required: 1.xA0 |

### RE (Restore Defaults) Command

<(Special)> The RE command is used to restore all configurable parameters to their factory default settings. The RE command does not write restored values to non-volatile (persistent) memory. Issue the WR (Write) command subsequent to issuing the RE command to save restored parameter values to non-volatile memory.

| |
|---|
| AT Command: ATRE |

### RN (Random Delay Slots) Command

<Networking & Security> The RN command is used to set and read the minimum value of the back-off exponent in the CSMA-CA algorithm. The CSMA-CA algorithm was engineered for collision avoidance (random delays are inserted to prevent data loss caused by data collisions).

| |
|---|
| AT Command: ATRN |
| Parameter Range: 0 – 3 [exponent] |
| Default Parameter Value: 0 |

If RN = 0, collision avoidance is disabled during the first iteration of the algorithm (802.15.4 - macMinBE).

CSMA-CA stands for "Carrier Sense Multiple Access - Collision Avoidance". Unlike CSMA-CD (reacts to network transmissions after collisions have been detected), CSMA-CA acts to prevent data collisions before they occur. As soon as a module receives a packet that is to be transmitted, it checks if the channel is clear (no other module is transmitting). If the channel is clear, the packet is sent over-the-air. If the channel is not clear, the module waits for a randomly selected period of time, then checks again to see if the channel is clear. After a time, the process ends and the data is lost.

### RO (Packetization Timeout) Command

| |
|---|
| AT Command: ATRO |
| Parameter Range:0 – 0xFF<br>[x character times] |
| Default Parameter Value: 3 |

<Serial Interfacing> RO command is used to set and read the number of character times of inter-character delay required before transmission.

RF transmission commences when data is detected in the DI (data in from host) buffer and RO character times of silence are detected on the UART receive lines (after receiving at least 1 byte).

RF transmission will also commence after 100 Bytes (maximum packet size) are received in the DI buffer.

Set the RO parameter to '0' to transmit characters as they arrive instead of buffering them into one RF packet.

### RP (RSSI PWM Timer) Command

| |
|---|
| AT Command: ATRP |
| Parameter Range:0 – 0xFF<br>[x 100 msec] |
| Default Parameter Value: 0x28 (40 decimal) |

<I/O Settings {I/O Line Passing}> The RP command is used to enable PWM (Pulse Width Modulation) output on the RF module. The output is calibrated to show the level a received RF signal is above the sensitivity level of the module. The PWM pulses vary from 24 to 100%. Zero percent means PWM output is inactive. One to 24% percent means the received RF signal is at or below the published sensitivity level of the module. The following table shows levels above sensitivity and PWM values.

The total period of the PWM output is 64 µs. Because there are 445 steps in the PWM output, the minimum step size is 144 ns.

**PWM Percentages**

| dB above Sensitivity | PWM percentage<br>(high period / total period) |
|---|---|
| 10 | 41% |
| 20 | 58% |
| 30 | 75% |

A non-zero value defines the time that the PWM output will be active with the RSSI value of the last received RF packet. After the set time when no RF packets are received, the PWM output will be set low (0 percent PWM) until another RF packet is received. The PWM output will also be set low at power-up until the first RF packet is received. A parameter value of 0xFF permanently enables the PWM output and it will always reflect the value of the last received RF packet.

### RR (XBee Retries) Command

| |
|---|
| AT Command: ATRR |
| Parameter Range: 0 – 6 |
| Default: 0 |
| Minimum Firmware Version Required: 1.xA0 |

<Networking {Addressing}> The RR command is used to set/read the maximum number of retries the module will execute in addition to the 3 retries provided by the 802.15.4 MAC. For each XBee retry, the 802.15.4 MAC can execute up to 3 retries.

The following applies when the DL parameter is set to 0xFFFF: If RR is set to zero (RR = 0), only one packet is broadcast. If RR is set to a value greater than zero (RR > 0), (RR + 2) packets are sent on each broadcast. No acknowledgements are returned on a broadcast.

This value does not need to be set on all modules for retries to work. If retries are enabled, the transmitting module will set a bit in the Digi RF Packet header which requests the receiving module to send an ACK (acknowledgement). If the transmitting module does not receive an ACK within 200 msec, it will re-send the packet within a random period up to 48 msec. Each XBee retry can potentially result in the MAC sending the packet 4 times (1 try plus 3 retries). Note that retries are not attempted for packets that are purged when transmitting with a Cyclic Sleep Coordinator.

### SC (Scan Channels) Command

<Networking {Association}> The SC command is used to set and read the list of channels to scan for all Active and Energy Scans as a bit field.

This affects scans initiated in command mode [AS (Active Scan) and ED (Energy Scan) commands] and during End Device Association and Coordinator startup.

AT Command: ATSC

Parameter Range: 1-0xFFFF [Bitfield]
(bits 0, 14, 15 are not allowed when using the XBee-PRO)

Default Parameter Value: 0x1FFE (all XBee-PRO channels)

Related Commands: ED (Energy Scan), SD (Scan Duration)

Minimum Firmware Version Required: v1.x80

| | | | |
|---|---|---|---|
| bit 0 - 0x0B | bit 4 - 0x0F | bit 8 - 0x13 | bit 12 - 0x17 |
| bit 1 - 0x0C | bit 5 - 0x10 | bit 9 - 0x14 | bit 13 - 0x18 |
| bit 2 - 0x0D | bit 6 - 0x11 | bit 10 - 0x15 | bit 14 - 0x19 |
| bit 3 - 0x0E | bit 7 - 0x12 | bit 11 - 0x16 | bit 15 - 0x1A |

### SD (Scan Duration) Command

<Networking {Association}> The SD command is used to set and read the exponent value that determines the duration (in time) of a scan.

***End Device*** (Duration of Active Scan during Association) - In a Beacon system, set SD = BE of the Coordinator. SD must be set at least to the highest BE parameter of any Beaconing Coordinator with which an End Device or Coordinator wish to discover.

AT Command: ATSD

Parameter Range: 0 - 0x0F

Default Parameter Value: 4

Related Commands: ED (Energy Scan), SC (Scan Channel)

Minimum Firmware Version Required: v1.x80

***Coordinator*** - If the 'ReassignPANID' option is set on the Coordinator [refer to A2 parameter], the SD parameter determines the length of time the Coordinator will scan channels to locate existing PANs. If the 'ReassignChannel' option is set, SD determines how long the Coordinator will perform an Energy Scan to determine which channel it will operate on.

Scan Time is measured as ((# of Channels to Scan) * (2 ^ SD) * 15.36ms). The number of channels to scan is set by the SC command. The XBee RF Module can scan up to 16 channels (SC = 0xFFFF). The XBee PRO RF Module can scan up to 12 channels (SC = 0x1FFE).

**Examples: Values below show results for a 12-channel scan**

| | |
|---|---|
| If SD = 0, time = 0.18 sec | SD = 8, time = 47.19 sec |
| SD = 2, time = 0.74 sec | SD = 10, time = 3.15 min |
| SD = 4, time = 2.95 sec | SD = 12, time = 12.58 min |
| SD = 6, time = 11.80 sec | SD = 14, time = 50.33 min |

### SH (Serial Number High) Command

<Diagnostics> The SH command is used to read the high 32 bits of the RF module's unique IEEE 64-bit address.

The module serial number is set at the factory and is read-only.

AT Command: ATSH

Parameter Range: 0 - 0xFFFFFFFF [read-only]

Related Commands: SL (Serial Number Low), MY (Source Address)

### SL (Serial Number Low) Command

<Diagnostics> The SL command is used to read the low 32 bits of the RF module's unique IEEE 64-bit address.

The module serial number is set at the factory and is read-only.

AT Command: ATSL

Parameter Range: 0 - 0xFFFFFFFF [read-only]

Related Commands: SH (Serial Number High), MY (Source Address)

## SM (Sleep Mode) Command

<Sleep Mode (Low Power)> The SM command is used to set and read Sleep Mode settings. By default, Sleep Modes are disabled (SM = 0) and the RF module remains in Idle/ Receive Mode. When in this state, the module is constantly ready to respond to either serial or RF activity.

* The Sleep Coordinator option (SM=6) only exists for backwards compatibility with firmware version 1.x06 only. In all other cases, use the CE command to enable a Coordinator.

AT Command: ATSM

Parameter Range: 0 – 6

| Parameter | Configuration |
|---|---|
| 0 | Disabled |
| 1 | Pin Hibernate |
| 2 | Pin Doze |
| 3 | (reserved) |
| 4 | Cyclic Sleep Remote |
| 5 | Cyclic Sleep Remote<br>(with Pin Wake-up) |
| 6 | Sleep Coordinator* |

Default Parameter Value: 0

## SO (Sleep Mode Command)

Sleep (Low Power) Sleep Options Set/Read the sleep mode options.

Bit 0 - Poll wakeup disable

- 0 - Normal operations. A module configured for cyclic sleep will poll for data on waking.
- 1 - Disable wakeup poll. A module configured for cyclic sleep will not poll for data on waking.

Bit 1 - ADC/DIO wakeup sampling disable.

- 0 - Normal operations. A module configured in a sleep mode with ADC/DIO sampling enabled will automatically perform a sampling on wakeup.
- 1 - Suppress sample on wakeup. A module configured in a sleep mode with ADC/DIO sampling enabled will not automatically sample on wakeup.

| AT Command: ATSO | |
|---|---|
| Parameter Range: | 0-4 |
| Default Parameter Value: | |
| Related Commands: SM (Sleep Mode), ST (Time before Sleep), DP (Disassociation Cyclic Sleep Period, BE (Beacon Order) | |

## SP (Cyclic Sleep Period) Command

<Sleep Mode (Low Power)> The SP command is used to set and read the duration of time in which a remote RF module sleeps. After the cyclic sleep period is over, the module wakes and checks for data. If data is not present, the module goes back to sleep. The maximum sleep period is 268 seconds (SP = 0x68B0).

The SP parameter is only valid if the module is configured to operate in Cyclic Sleep (SM = 4-6). Coordinator and End Device SP values should always be equal.

To send Direct Messages, set SP = 0.

| AT Command: ATSP | |
|---|---|
| Parameter Range: | NonBeacon Firmware:<br>0-0x68B0 [x 10 milliseconds] |
| Default Parameter Value: | |
| Related Commands: SM (Sleep Mode), ST (Time before Sleep), DP (Disassociation Cyclic Sleep Period, BE (Beacon Order) | |

### NonBeacon Firmware

*End Device* - SP determines the sleep period for cyclic sleeping remotes. Maximum sleep period is 268 seconds (0x68B0).

*Coordinator* - If non-zero, SP determines the time to hold an indirect message before discarding it. A Coordinator will discard indirect messages after a period of (2.5 * SP).

### ST (Time before Sleep) Command

<Sleep Mode (Low Power)> The ST command is used to set and read the period of inactivity (no serial or RF data is sent or received) before activating Sleep Mode.

| | |
|---|---|
| AT Command: ATST | |
| Parameter Range: | NonBeacon Firmware:<br>1 – 0xFFFF [x 1 millisecond] |
| Default Parameter Value: | |
| Related Commands: SM (Sleep Mode), ST (Time before Sleep) | |

**NonBeacon Firmware**

Set/Read time period of inactivity (no serial or RF data is sent or received) before activating Sleep Mode. ST parameter is only valid with Cyclic Sleep settings (SM = 4 - 5).

Coordinator and End Device ST values must be equal.

### T0 - T7 ((D0-D7) Output Timeout) Command

<I/O Settings {I/O Line Passing}> The T0, T1, T2, T3, T4, T5, T6 and T7 commands are used to set/read output timeout values for the lines that correspond with the D0 - D7 parameters. When output is set (due to I/O line passing) to a non-default level, a timer is started which when expired, will set the output to its default level. The timer is reset when a valid I/O packet is received. The Tn parameter defines the permissible amount of time to stay in a non-default (active) state. If Tn = 0, Output Timeout is disabled (output levels are held indefinitely).

| |
|---|
| AT Commands: ATT0 – ATT7 |
| Parameter Range:0 – 0xFF [x 100 msec] |
| Default Parameter Value:0xFF |
| Minimum Firmware Version Required: v1.xA0 |

### VL (Firmware Version - Verbose)

<Diagnostics> The VL command is used to read detailed version information about the RF module. The information includes:
application build date; MAC, PHY and bootloader versions; and build dates. This command was removed from firmware 1xC9 and later versions.

| |
|---|
| AT Command: ATVL |
| Parameter Range:0 – 0xFF<br>[x 100 milliseconds] |
| Default Parameter Value: 0x28 (40 decimal) |
| Minimum Firmware Version Required: v1.x80 – v1.xC8 |

### VR (Firmware Version) Command

<Diagnostics> The VR command is used to read which firmware version is stored in the module.

| |
|---|
| AT Command: ATVR |
| Parameter Range: 0 – 0xFFFF [read only] |

XBee version numbers will have four significant digits. The reported number will show three or four numbers and is stated in hexadecimal notation. A version can be reported as "ABC" or "ABCD". Digits ABC are the main release number and D is the revision number from the main release. "D" is not required and if it is not present, a zero is assumed for D. "B" is a variant designator. The following variants exist:

- "0" = Non-Beacon Enabled 802.15.4 Code
- "1" = Beacon Enabled 802.15.4 Code

### WR (Write) Command

<(Special)> The WR command is used to write configurable parameters to the RF module's non-volatile memory. Parameter values remain in the module's memory until overwritten by subsequent use of the WR Command.

| |
|---|
| AT Command: ATWR |

If changes are made without writing them to non-volatile memory, the module reverts back to previously saved parameters the next time the module is powered-on.

NOTE: Once the WR command is sent to the module, no additional characters should be sent until after the “OK/r” response is received.

# API Operation

By default, XBee®/XBee-PRO® RF Modules act as a serial line replacement (Transparent Operation) - all UART data received through the DI pin is queued up for RF transmission. When the module receives an RF packet, the data is sent out the DO pin with no additional information.

Inherent to Transparent Operation are the following behaviors:

- If module parameter registers are to be set or queried, a special operation is required for transitioning the module into Command Mode.
- In point-to-multipoint systems, the application must send extra information so that the receiving module(s) can distinguish between data coming from different remotes.

As an alternative to the default Transparent Operation, API (Application Programming Interface) Operations are available. API operation requires that communication with the module be done through a structured interface (data is communicated in frames in a defined order). The API specifies how commands, command responses and module status messages are sent and received from the module using a UART Data Frame.

## API Frame Specifications

Two API modes are supported and both can be enabled using the AP (API Enable) command. Use the following AP parameter values to configure the module to operate in a particular mode:

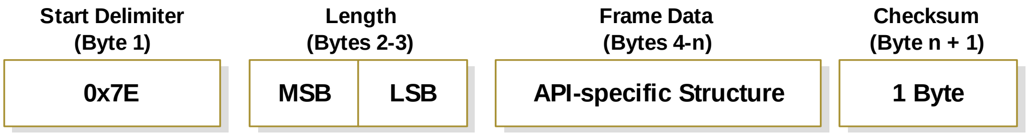
- AP = 0 (default): Transparent Operation (UART Serial line replacement)
  API modes are disabled.
- AP = 1: API Operation
- AP = 2: API Operation (with escaped characters)

Any data received prior to the start delimiter is silently discarded. If the frame is not received correctly or if the checksum fails, the data is silently discarded.

### API Operation (AP parameter = 1)

When this API mode is enabled (AP = 1), the UART data frame structure is defined as follows:

**Figure 3-09. UART Data Frame Structure:**

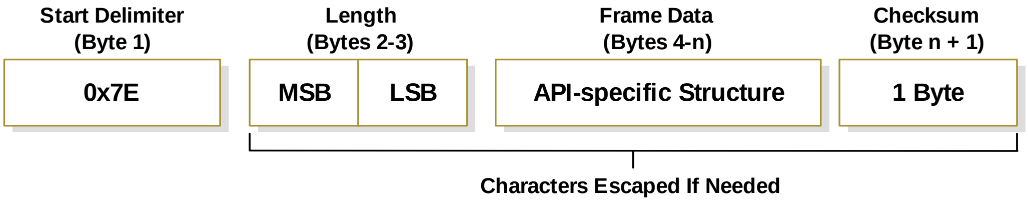


MSB = Most Significant Byte, LSB = Least Significant Byte

### API Operation - with Escape Characters (AP parameter = 2)

When this API mode is enabled (AP = 2), the UART data frame structure is defined as follows:

**Figure 3-10. UART Data Frame Structure - with escape control characters:**



MSB = Most Significant Byte, LSB = Least Significant Byte

**Escape characters**. When sending or receiving a UART data frame, specific data values must be escaped (flagged) so they do not interfere with the UART or UART data frame operation. To escape an interfering data byte, insert 0x7D and follow it with the byte to be escaped XOR'd with 0x20.

**Data bytes that need to be escaped:**

- 0x7E – Frame Delimiter
- 0x7D – Escape
- 0x11 – XON
- 0x13 – XOFF

**Example -** Raw UART Data Frame (before escaping interfering bytes):
0x7E 0x00 0x02 0x23 0x11 0xCB

0x11 needs to be escaped which results in the following frame:
0x7E 0x00 0x02 0x23 0x7D 0x31 0xCB

Note: In the above example, the length of the raw data (excluding the checksum) is 0x0002 and the checksum of the non-escaped data (excluding frame delimiter and length) is calculated as: 0xFF - (0x23 + 0x11) = (0xFF - 0x34) = 0xCB.

### Checksum

To test data integrity, a checksum is calculated and verified on non-escaped data.

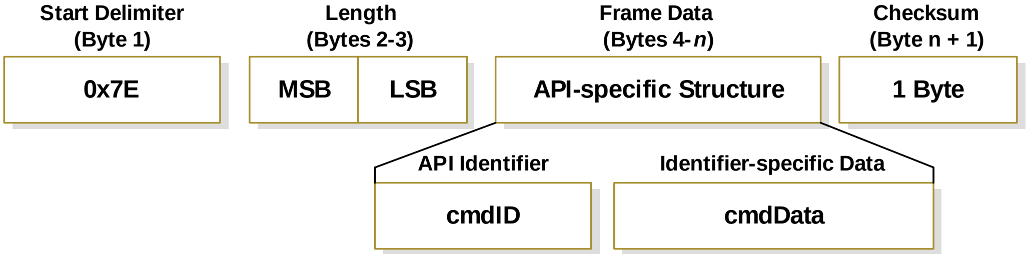
**To calculate**: Not including frame delimiters and length, add all bytes keeping only the lowest 8 bits of the result and subtract from 0xFF.

**To verify**: Add all bytes (include checksum, but not the delimiter and length). If the checksum is correct, the sum will equal 0xFF.

## API Types

Frame data of the UART data frame forms an API-specific structure as follows:

Figure 3-11. UART Data Frame & API-specific Structure:

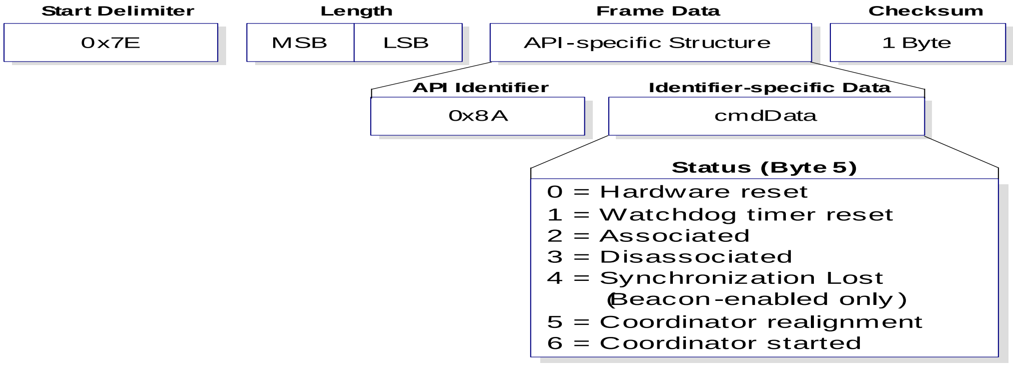


The cmdID frame (API-identifier) indicates which API messages will be contained in the cmdData frame (Identifier-specific data). Refer to the sections that follow for more information regarding the supported API types. Note that multi-byte values are sent big endian.

### Modem Status

API Identifier: 0x8A
RF module status messages are sent from the module in response to specific conditions.

Figure 3-12. Modem Status Frames

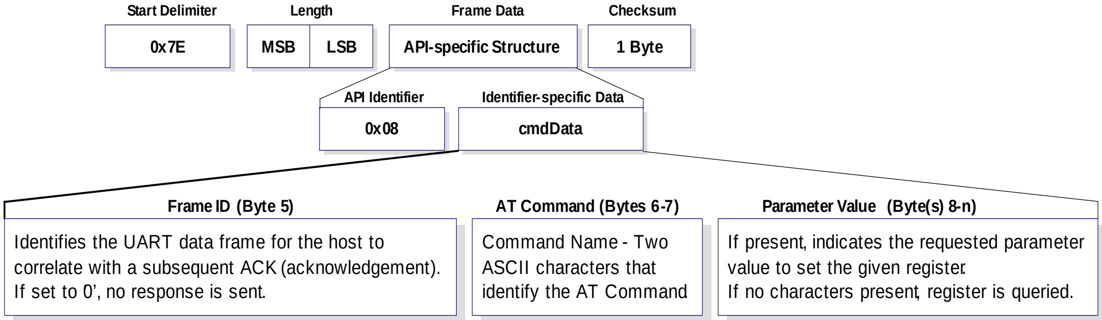


### AT Command

API Identifier Value: 0x08

The “AT Command” API type allows for module parameters to be queried or set. When using this command ID, new parameter values are applied immediately. This includes any register set with the “AT Command - Queue Parameter Value” (0x09) API type.

**Figure 3-13. AT Command Frames**



Identifies the UART data frame for the host to correlate with a subsequent ACK (acknowledgement). If set to 0’, no response is sent.

Command Name - Two ASCII characters that identify the AT Command

If present, indicates the requested parameter value to set the given register. If no characters present, register is queried.

**Figure 3-14. Example: API frames when reading the DL parameter value of the module.**

| Byte 1 | Bytes 2-3 | | Byte 4 | Byte 5 | Bytes 6-7 | | Byte 8 |
|---|---|---|---|---|---|---|---|
| 0x7E | 0x00 | 0x04 | 0x08 | 0x52 (R) | 0x44 (D) | 0x4C (L) | 0x15 |
| Start Delimiter | Length* | | API Identifier | Frame ID** | AT Command | | Checksum |

** Length [Bytes] = API Identifier + Frame ID + AT Command*

*** “R” value was arbitrarily selected.*

**Figure 3-15. Example: API frames when modifying the DL parameter value of the module.**

| Byte 1 | Bytes 2-3 | | Byte 4 | Byte 5 | Bytes 6-7 | | Bytes 8-11 | Byte 12 |
|---|---|---|---|---|---|---|---|---|
| 0x7E | 0x00 | 0x08 | 0x08 | 0x4D (M) | 0x44 (D) | 0x4C (L) | 0x00000FFF | 0x0C |
| Start Delimiter | Length* | | API Identifier | Frame ID** | AT Command | | Parameter Value | Checksum |

** Length [Bytes] = API Identifier + Frame ID + AT Command + Parameter Value*

*** “M” value was arbitrarily selected.*

### AT Command - Queue Parameter Value

API Identifier Value: 0x09

This API type allows module parameters to be queried or set. In contrast to the “AT Command” API type, new parameter values are queued and not applied until either the “AT Command” (0x08) API type or the AC (Apply Changes) command is issued. Register queries (reading parameter values) are returned immediately.

## AT Command Response

API Identifier Value: 0x88

Response to previous command.

In response to an AT Command message, the module will send an AT Command Response message. Some commands will send back multiple frames (for example, the ND (Node Discover) and AS (Active Scan) commands). These commands will end by sending a frame with a status of ATCMD_OK and no cmdData.

Figure 3-16. AT Command Response Frames.

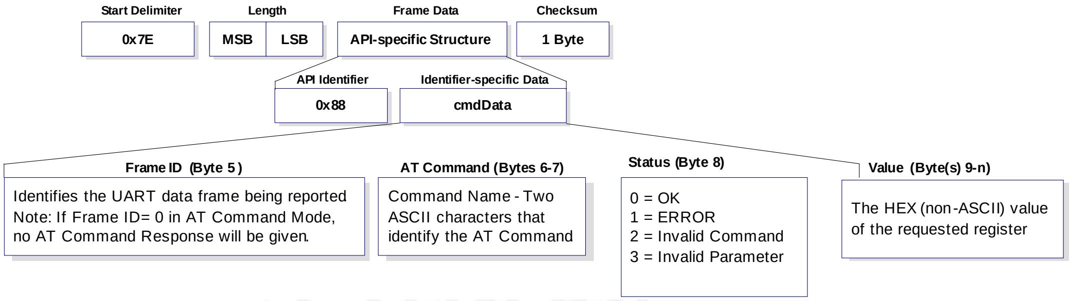
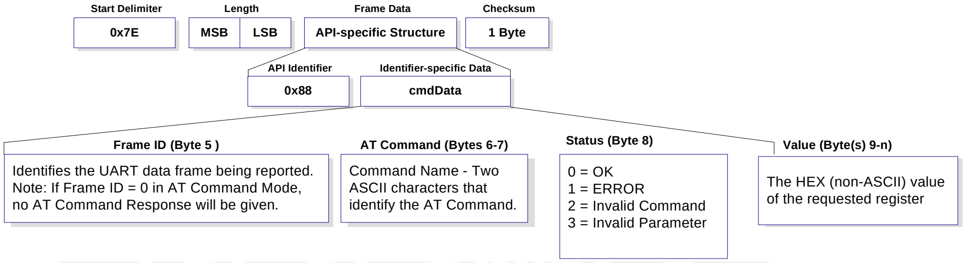


Figure 3-17. AT Command Response Frames.

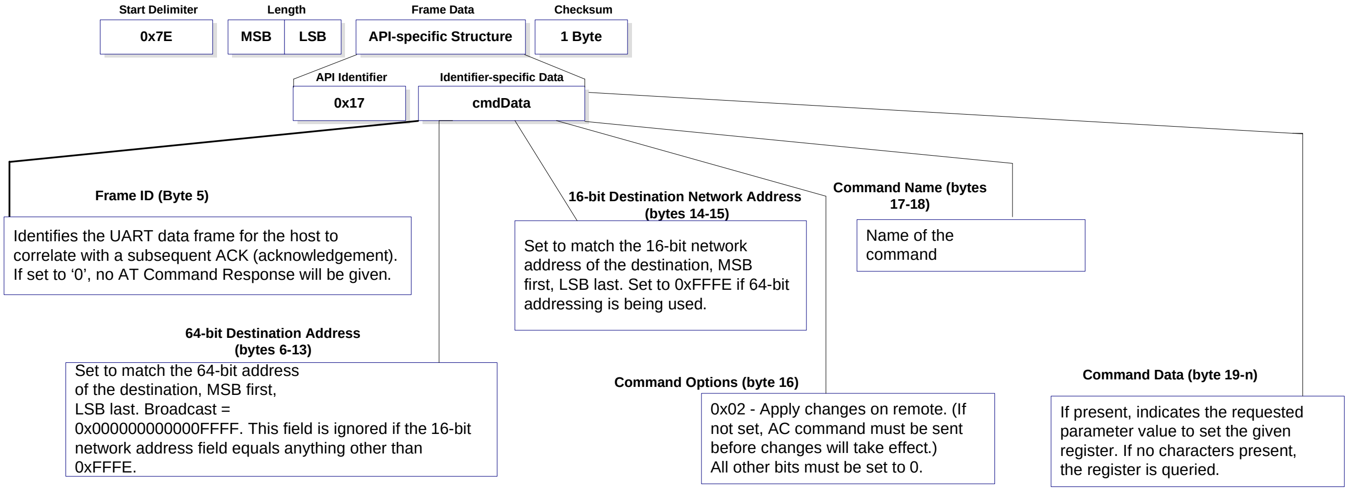


## Remote AT Command Request

API Identifier Value: 0x17

Allows for module parameter registers on a remote device to be queried or set

Figure 3-18. Remote AT Command Request

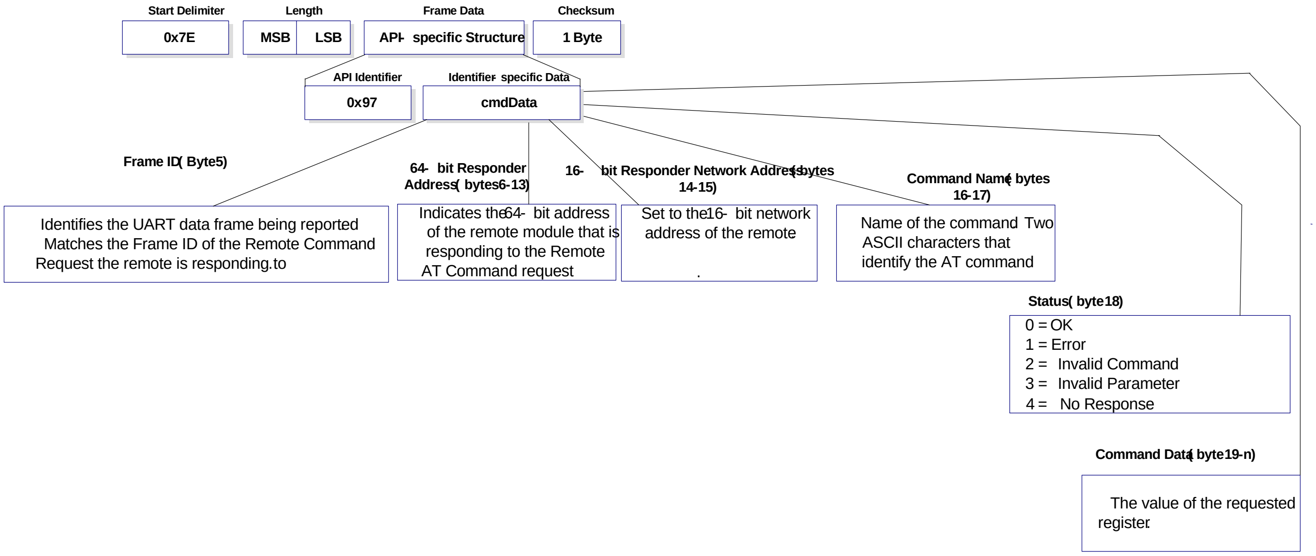

Start Delimiter
0x7E
Length
MSB
LSB
Frame Data
API-specific Structure
Checksum
1 Byte
API Identifier
0x17
Identifier-specific Data
cmdData
Frame ID (Byte 5)
Identifies the UART data frame for the host to correlate with a subsequent ACK (acknowledgement). If set to '0', no AT Command Response will be given.
16-bit Destination Network Address (bytes 14-15)
Set to match the 16-bit network address of the destination, MSB first, LSB last. Set to 0xFFFE if 64-bit addressing is being used.
Command Name (bytes 17-18)
Name of the command
64-bit Destination Address (bytes 6-13)
Set to match the 64-bit address of the destination, MSB first, LSB last. Broadcast = 0x000000000000FFFF. This field is ignored if the 16-bit network address field equals anything other than 0xFFFE.
Command Options (byte 16)
0x02 - Apply changes on remote. (If not set, AC command must be sent before changes will take effect.) All other bits must be set to 0.
Command Data (byte 19-n)
If present, indicates the requested parameter value to set the given register. If no characters present, the register is queried.

## Remote Command Response

API Identifier Value: 0x97

If a module receives a remote command response RF data frame in response to a Remote AT Command Request, the module will send a Remote AT Command Response message out the UART. Some commands may send back multiple frames--for example, Node Discover (ND) command.

Figure 3-19. Remote AT Command Response.

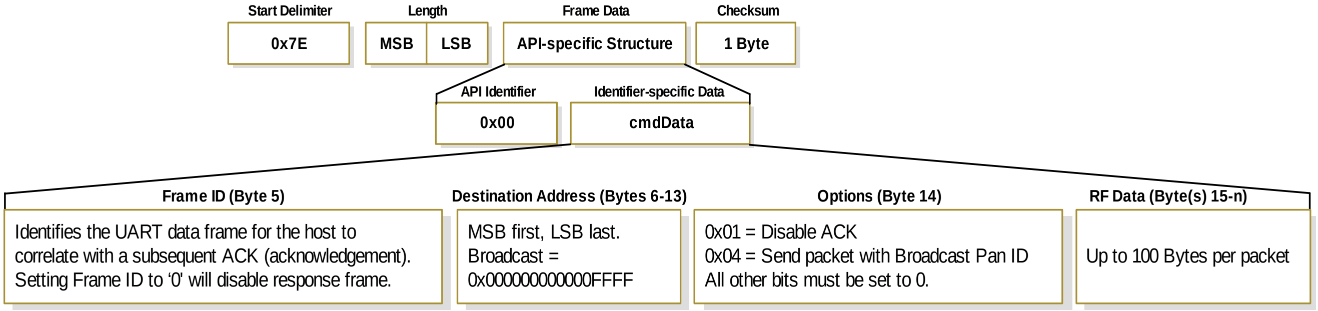

Start Delimiter
0x7E
Length
MSB
LSB
Frame Data
API- specific Structure
Checksum
1 Byte
API Identifier
0x97
Identifier- specific Data
cmdData
Frame ID( Byte5)
Identifies the UART data frame being reported Matches the Frame ID of the Remote Command Request the remote is responding.to
64- bit Responder Address( bytes6-13)
Indicates the64- bit address of the remote module that is responding to the Remote AT Command request
16- bit Responder Network Address( bytes 14-15)
Set to the16- bit network address of the remote
.
Command Name( bytes 16-17)
Name of the command Two ASCII characters that identify the AT command
Status( byte18)
0 = OK
1 = Error
2 = Invalid Command
3 = Invalid Parameter
4 = No Response
Command Data( byte19-n)
The value of the requested register.

## TX (Transmit) Request: 64-bit address

API Identifier Value: 0x00

A TX Request message will cause the module to transmit data as an RF Packet.

Figure 3-20. TX Packet (64-bit address) Frames

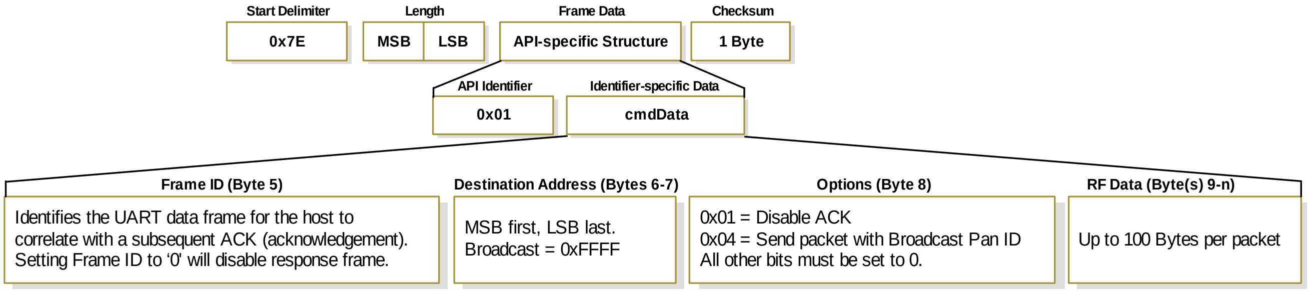

Start Delimiter
0x7E
Length
MSB
LSB
Frame Data
API-specific Structure
Checksum
1 Byte
API Identifier
0x00
Identifier-specific Data
cmdData
Frame ID (Byte 5)
Identifies the UART data frame for the host to correlate with a subsequent ACK (acknowledgement). Setting Frame ID to '0' will disable response frame.
Destination Address (Bytes 6-13)
MSB first, LSB last.
Broadcast =
0x000000000000FFFF
Options (Byte 14)
0x01 = Disable ACK
0x04 = Send packet with Broadcast Pan ID
All other bits must be set to 0.
RF Data (Byte(s) 15-n)
Up to 100 Bytes per packet

### TX (Transmit) Request: 16-bit address

API Identifier Value: 0x01

A TX Request message will cause the module to transmit data as an RF Packet.

**Figure 3-21. TX Packet (16-bit address) Frames**



Identifies the UART data frame for the host to correlate with a subsequent ACK (acknowledgement). Setting Frame ID to '0' will disable response frame.

MSB first, LSB last.
Broadcast = 0xFFFF

0x01 = Disable ACK
0x04 = Send packet with Broadcast Pan ID
All other bits must be set to 0.

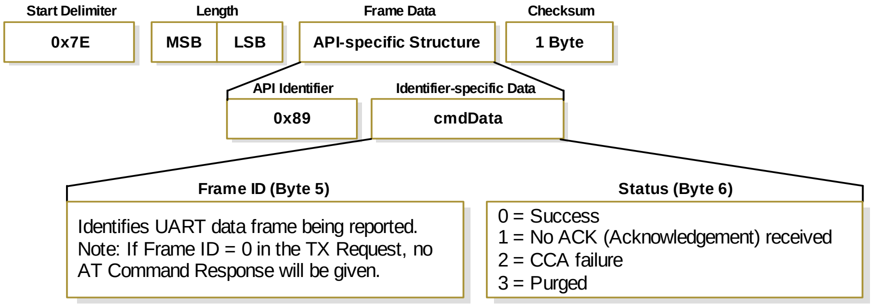
Up to 100 Bytes per packet

### TX (Transmit) Status

API Identifier Value: 0x89

When a TX Request is completed, the module sends a TX Status message. This message will indicate if the packet was transmitted successfully or if there was a failure.

**Figure 3-22. TX Status Frames**



Identifies UART data frame being reported.
Note: If Frame ID = 0 in the TX Request, no AT Command Response will be given.

0 = Success
1 = No ACK (Acknowledgement) received
2 = CCA failure
3 = Purged

NOTES:

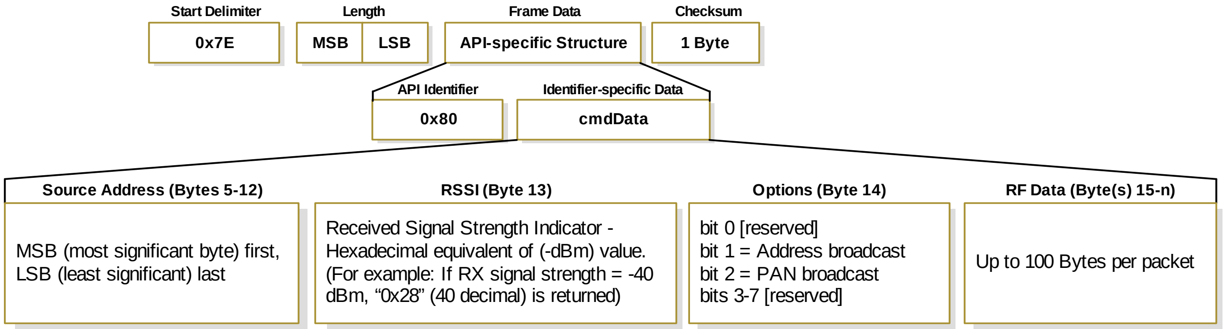
- "STATUS = 1" occurs when all retries are expired and no ACK is received.
- If transmitter broadcasts (destination address = 0x000000000000FFFF), only "STATUS = 0 or 2" will be returned.
- "STATUS = 3" occurs when Coordinator times out of an indirect transmission. Timeout is defined as (2.5 x SP (Cyclic Sleep Period) parameter value).

### RX (Receive) Packet: 64-bit Address

API Identifier Value: 0x80

When the module receives an RF packet, it is sent out the UART using this message type.

**Figure 3-23. RX Packet (64-bit address) Frames**



MSB (most significant byte) first, LSB (least significant) last

Received Signal Strength Indicator - Hexadecimal equivalent of (-dBm) value. (For example: If RX signal strength = -40 dBm, "0x28" (40 decimal) is returned)

bit 0 [reserved]
bit 1 = Address broadcast
bit 2 = PAN broadcast
bits 3-7 [reserved]

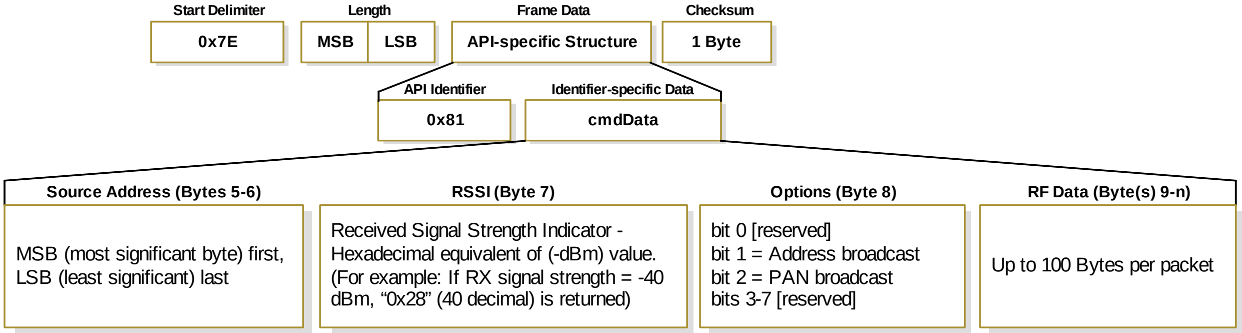
Up to 100 Bytes per packet

### RX (Receive) Packet: 16-bit Address

API Identifier Value: 0x81

When the module receives an RF packet, it is sent out the UART using this message type.

Figure 3-24. RX Packet (16-bit address) Frames

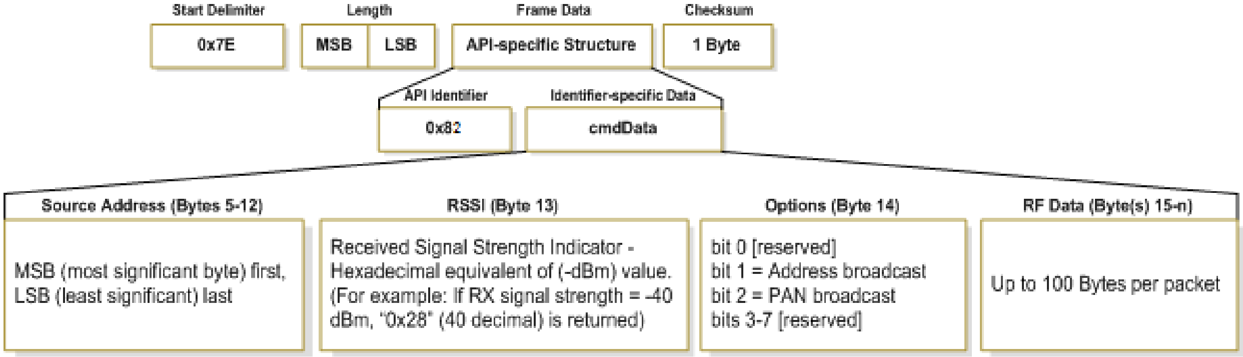

Start Delimiter
0x7E
Length
MSB
LSB
Frame Data
API-specific Structure
Checksum
1 Byte
API Identifier
0x81
Identifier-specific Data
cmdData
Source Address (Bytes 5-6)
MSB (most significant byte) first, LSB (least significant) last
RSSI (Byte 7)
Received Signal Strength Indicator - Hexadecimal equivalent of (-dBm) value. (For example: If RX signal strength = -40 dBm, "0x28" (40 decimal) is returned)
Options (Byte 8)
bit 0 [reserved]
bit 1 = Address broadcast
bit 2 = PAN broadcast
bits 3-7 [reserved]
RF Data (Byte(s) 9-n)
Up to 100 Bytes per packet

### RX (Receive) Packet: 64-bit Address IO

API Identifier Value: 0x82

I/O data is sent out the UART using an API frame.

Figure 3-25. RX Packet (64-bit address) Frames

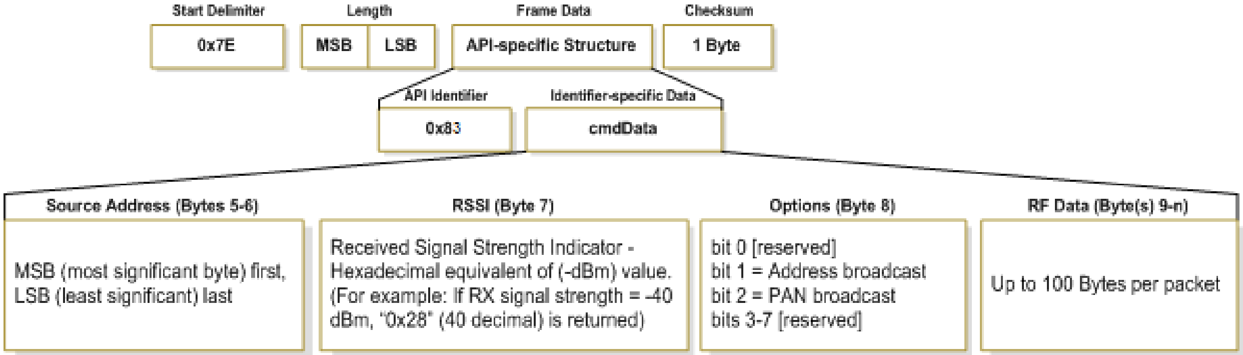

Start Delimiter
0x7E
Length
MSB
LSB
Frame Data
API-specific Structure
Checksum
1 Byte
API Identifier
0x82
Identifier-specific Data
cmdData
Source Address (Bytes 5-12)
MSB (most significant byte) first, LSB (least significant) last
RSSI (Byte 13)
Received Signal Strength Indicator - Hexadecimal equivalent of (-dBm) value. (For example: If RX signal strength = -40 dBm, "0x28" (40 decimal) is returned)
Options (Byte 14)
bit 0 [reserved]
bit 1 = Address broadcast
bit 2 = PAN broadcast
bits 3-7 [reserved]
RF Data (Byte(s) 15-n)
Up to 100 Bytes per packet

### RX (Receive) Packet: 16-bit Address IO

API Identifier Value: 0x83

I/O data is sent out the UART using an API frame.

Figure 3-26. RX Packet (16-bit address) Frames


Start Delimiter
0x7E
Length
MSB
LSB
Frame Data
API-specific Structure
Checksum
1 Byte
API Identifier
0x83
Identifier-specific Data
cmdData
Source Address (Bytes 5-6)
MSB (most significant byte) first, LSB (least significant) last
RSSI (Byte 7)
Received Signal Strength Indicator - Hexadecimal equivalent of (-dBm) value. (For example: If RX signal strength = -40 dBm, "0x28" (40 decimal) is returned)
Options (Byte 8)
bit 0 [reserved]
bit 1 = Address broadcast
bit 2 = PAN broadcast
bits 3-7 [reserved]
RF Data (Byte(s) 9-n)
Up to 100 Bytes per packet

# Appendix A: Agency Certifications

## United States (FCC)

XBee®/XBee-PRO® RF Modules comply with Part 15 of the FCC rules and regulations. Compliance with the labeling requirements, FCC notices and antenna usage guidelines is required.

To fulfill FCC Certification requirements, the OEM must comply with the following regulations:

1. The system integrator must ensure that the text on the external label provided with this device is placed on the outside of the final product [Figure A-01].
2. XBee®/XBee-PRO® RF Modules may only be used with antennas that have been tested and approved for use with this module [refer to the antenna tables in this section].

### OEM Labeling Requirements

WARNING: The Original Equipment Manufacturer (OEM) must ensure that FCC labeling requirements are met. This includes a clearly visible label on the outside of the final product enclosure that displays the contents shown in the figure below.

**Figure 4-01. Required FCC Label for OEM products containing the XBee®/XBee-PRO® RF Module**

Contains FCC ID: OUR-XBEE/OUR-XBEEPRO**

The enclosed device complies with Part 15 of the FCC Rules. Operation is subject to the following two conditions: (*i.*) this device may not cause harmful interference and (*ii.*) this device must accept any interference received, including interference that may cause undesired operation.

* The FCC ID for the XBee is “OUR-XBEE”. The FCC ID for the XBee-PRO is “OUR-XBEEPRO”.

### FCC Notices

**IMPORTANT:** The XBee®/XBee-PRO® RF Module has been certified by the FCC for use with other products without any further certification (as per FCC section 2.1091). Modifications not expressly approved by Digi could void the user's authority to operate the equipment.

**IMPORTANT:** OEMs must test final product to comply with unintentional radiators (FCC section 15.107 & 15.109) before declaring compliance of their final product to Part 15 of the FCC Rules.

**IMPORTANT:** The RF module has been certified for remote and base radio applications. If the module will be used for portable applications, please take note of the following instructions:

- For XBee modules where the antenna gain is less than 13.8 dBi, no additional SAR testing is required. The 20 cm separation distance is not required for antenna gain less than 13.8 dBi.
- For XBee modules where the antenna gain is greater than 13.8 dBi and for all XBee-PRO modules, the device must undergo SAR testing.

This equipment has been tested and found to comply with the limits for a Class B digital device, pursuant to Part 15 of the FCC Rules. These limits are designed to provide reasonable protection against harmful interference in a residential installation. This equipment generates, uses and can radiate radio frequency energy and, if not installed and used in accordance with the instructions, may cause harmful interference to radio communications. However, there is no guarantee that interference will not occur in a particular installation.

If this equipment does cause harmful interference to radio or television reception, which can be determined by turning the equipment off and on, the user is encouraged to try to correct the interference by one or more of the following measures: Re-orient or relocate the receiving antenna, Increase the separation between the equipment and receiver, Connect equipment and receiver to outlets on different circuits, or Consult the dealer or an experienced radio/TV technician for help.

## FCC-Approved Antennas (2.4 GHz)

XBee/XBee-PRO RF Modules can be installed using antennas and cables constructed with standard connectors (Type-N, SMA, TNC, etc.) if the installation is performed professionally and according to FCC guidelines. For installations not performed by a professional, non-standard connectors (RPSMA, RPTNC, etc) must be used.

The modules are FCC-approved for fixed base station and mobile applications on channels 0x0B - 0x1A (XBee) and 0x0C - 0x17 (XBee-PRO). If the antenna is mounted at least 20cm (8 in.) from nearby persons, the application is considered a mobile application. Antennas not listed in the table must be tested to comply with FCC Section 15.203 (Unique Antenna Connectors) and Section 15.247 (Emissions).

**XBee RF Modules (1 mW):** XBee Modules have been tested and approved for use with the antennas listed in the first and second tables below (Cable loss is required as shown).

**XBee-PRO RF Modules (60 mW):** XBee-PRO Modules have been tested and approved for use with the antennas listed in the first and third tables below (Cable loss is required as shown).

The antennas in the tables below have been approved for use with this module. Digi does not carry all of these antenna variants. Contact Digi Sales for available antennas.

**Antennas approved for use with the XBee®/XBee-PRO® RF Modules (Cable loss is not required.)**

| Part Number | Type (Description) | Gain | Application* | Min. Separation |
|---|---|---|---|---|
| A24-HASM-450 | Dipole (Half-wave articulated RPSMA - 4.5”) | 2.1 dBi | Fixed/Mobile | 20 cm |
| A24-HABSM | Dipole (Articulated RPSMA) | 2.1 dBi | Fixed | 20 cm |
| A24-HABUF-P5I | Dipole (Half-wave articulated bulkhead mount U.FL. w/ 5” pigtail) | 2.1 dBi | Fixed | 20 cm |
| A24-HASM-525 | Dipole (Half-wave articulated RPSMA - 5.25") | 2.1 dBi | Fixed/Mobile | 20 cm |
| A24-QI | Monopole (Integrated whip) | 1.5 dBi | Fixed | 20 cm |
| A24-C1 | Surface Mount | -1.5 dBi | Fixed/Mobile | 20 cm |
| 29000430 | Embedded PCB Antenna | -0.5dBi | Fixed/ Mobile | 20 cm |

**Antennas approved for use with the XBee RF Modules (Cable loss is required)**

| Part Number | Type (Description) | Gain | Application* | Min. Separation | Required Cable-loss |
|---|---|---|---|---|---|
| **Yagi Class Antennas** | | | | | |
| A24-Y4NF | Yagi (4-element) | 6.0 dBi | Fixed | 2 m | - |
| A24-Y6NF | Yagi (6-element) | 8.8 dBi | Fixed | 2 m | 1.7 dB |
| A24-Y7NF | Yagi (7-element) | 9.0 dBi | Fixed | 2 m | 1.9 dB |
| A24-Y9NF | Yagi (9-element) | 10.0 dBi | Fixed | 2 m | 2.9 dB |
| A24-Y10NF | Yagi (10-element) | 11.0 dBi | Fixed | 2 m | 3.9 dB |
| A24-Y12NF | Yagi (12-element) | 12.0 dBi | Fixed | 2 m | 4.9 dB |
| A24-Y13NF | Yagi (13-element) | 12.0 dBi | Fixed | 2 m | 4.9 dB |
| A24-Y15NF | Yagi (15-element) | 12.5 dBi | Fixed | 2 m | 5.4 dB |
| A24-Y16NF | Yagi (16-element) | 13.5 dBi | Fixed | 2 m | 6.4 dB |
| A24-Y16RM | Yagi (16-element, RPSMA connector) | 13.5 dBi | Fixed | 2 m | 6.4 dB |
| A24-Y18NF | Yagi (18-element) | 15.0 dBi | Fixed | 2 m | 7.9 dB |
| **Omni-Directional Class Antennas** | | | | | |
| A24-F2NF | Omni-directional (Fiberglass base station) | 2.1 dBi | Fixed/Mobile | 20 cm | |
| A24-F3NF | Omni-directional (Fiberglass base station) | 3.0 dBi | Fixed/Mobile | 20 cm | |
| A24-F5NF | Omni-directional (Fiberglass base station) | 5.0 dBi | Fixed/Mobile | 20 cm | |
| A24-F8NF | Omni-directional (Fiberglass base station) | 8.0 dBi | Fixed | 2 m | |
| A24-F9NF | Omni-directional (Fiberglass base station) | 9.5 dBi | Fixed | 2 m | 0.2 dB |
| A24-F10NF | Omni-directional (Fiberglass base station) | 10.0 dBi | Fixed | 2 m | 0.7 dB |
| A24-F12NF | Omni-directional (Fiberglass base station) | 12.0 dBi | Fixed | 2 m | 2.7 dB |
| A24-F15NF | Omni-directional (Fiberglass base station) | 15.0 dBi | Fixed | 2 m | 5.7 dB |
| A24-W7NF | Omni-directional (Base station) | 7.2 dBi | Fixed | 2 m | |
| A24-M7NF | Omni-directional (Mag-mount base station) | 7.2 dBi | Fixed | 2 m | |
| **Panel Class Antennas** | | | | | |
| A24-P8SF | Flat Panel | 8.5 dBi | Fixed | 2 m | 1.5 dB |
| A24-P8NF | Flat Panel | 8.5 dBi | Fixed | 2 m | 1.5 dB |
| A24-P13NF | Flat Panel | 13.0 dBi | Fixed | 2 m | 6 dB |
| A24-P14NF | Flat Panel | 14.0 dBi | Fixed | 2 m | 7 dB |

| Part Number | Type (Description) | Gain | Application* | Min. Separation | Required Cable-loss |
|---|---|---|---|---|---|
| A24-P15NF | Flat Panel | 15.0 dBi | Fixed | 2 m | 8 dB |
| A24-P16NF | Flat Panel | 16.0 dBi | Fixed | 2 m | 9 dB |

**Antennas approved for use with the XBee®/XBee-PRO® RF Modules (Cable-loss is required)**

| Part Number | Type (Description) | Gain | Application* | Min. Separation | Required Cable-loss |
|---|---|---|---|---|---|
| **Yagi Class Antennas** | | | | | |
| A24-Y4NF | Yagi (4-element) | 6.0 dBi | Fixed | 2 m | 8.1 dB |
| A24-Y6NF | Yagi (6-element) | 8.8 dBi | Fixed | 2 m | 10.9 dB |
| A24-Y7NF | Yagi (7-element) | 9.0 dBi | Fixed | 2 m | 11.1 dB |
| A24-Y9NF | Yagi (9-element) | 10.0 dBi | Fixed | 2 m | 12.1 dB |
| A24-Y10NF | Yagi (10-element) | 11.0 dBi | Fixed | 2 m | 13.1 dB |
| A24-Y12NF | Yagi (12-element) | 12.0 dBi | Fixed | 2 m | 14.1 dB |
| A24-Y13NF | Yagi (13-element) | 12.0 dBi | Fixed | 2 m | 14.1 dB |
| A24-Y15NF | Yagi (15-element) | 12.5 dBi | Fixed | 2 m | 14.6 dB |
| A24-Y16NF | Yagi (16-element) | 13.5 dBi | Fixed | 2 m | 15.6 dB |
| A24-Y16RM | Yagi (16-element, RPSMA connector) | 13.5 dBi | Fixed | 2 m | 15.6 dB |
| A24-Y18NF | Yagi (18-element) | 15.0 dBi | Fixed | 2 m | 17.1 dB |
| **Omni-Directional Class Antennas** | | | | | |
| A24-F2NF | Omni-directional (Fiberglass base station) | 2.1 dBi | Fixed/Mobile | 20 cm | 4.2 dB |
| A24-F3NF | Omni-directional (Fiberglass base station) | 3.0 dBi | Fixed/Mobile | 20 cm | 5.1 dB |
| A24-F5NF | Omni-directional (Fiberglass base station) | 5.0 dBi | Fixed/Mobile | 20 cm | 7.1 dB |
| A24-F8NF | Omni-directional (Fiberglass base station) | 8.0 dBi | Fixed | 2 m | 10.1 dB |
| A24-F9NF | Omni-directional (Fiberglass base station) | 9.5 dBi | Fixed | 2 m | 11.6 dB |
| A24-F10NF | Omni-directional (Fiberglass base station) | 10.0 dBi | Fixed | 2 m | 12.1 dB |
| A24-F12NF | Omni-directional (Fiberglass base station) | 12.0 dBi | Fixed | 2 m | 14.1 dB |
| A24-F15NF | Omni-directional (Fiberglass base station) | 15.0 dBi | Fixed | 2 m | 17.1 dB |
| A24-W7NF | Omni-directional (Base station) | 7.2 dBi | Fixed | 2 m | 9.3 dB |
| A24-M7NF | Omni-directional (Mag-mount base station) | 7.2 dBi | Fixed | 2 m | 9.3 dB |
| **Panel Class Antennas** | | | | | |
| A24-P8SF | Flat Panel | 8.5 dBi | Fixed | 2 m | 8.6 dB |
| A24-P8NF | Flat Panel | 8.5 dBi | Fixed | 2 m | 8.6 dB |
| A24-P13NF | Flat Panel | 13.0 dBi | Fixed | 2 m | 13.1 dB |
| A24-P14NF | Flat Panel | 14.0 dBi | Fixed | 2 m | 14.1 dB |
| A24-P15NF | Flat Panel | 15.0 dBi | Fixed | 2 m | 15.1 dB |
| A24-P16NF | Flat Panel | 16.0 dBi | Fixed | 2 m | 16.1 dB |
| A24-P19NF | Flat Panel | 19.0 dBi | Fixed | 2m | 19.1 dB |
| **Waveguide Class Antennas** | | | | | |
| RSM | Waveguide | 7.1 dBi | Fixed | 2m | 1.5dB |
| **Helical Class Antenna** | | | | | |
| A24-H3UF | Helical | 3.0dBi | Fixed/ Mobile | 20cm | 0dB |

*** If using the RF module in a portable application** (For example - If the module is used in a handheld device and the antenna is less than 20cm from the human body when the device is operation): The integrator is responsible for passing additional SAR (Specific Absorption Rate) testing based on FCC rules 2.1091 and FCC Guidelines for Human Exposure to Radio Frequency Electromagnetic Fields, OET Bulletin and Supplement C. The testing results will be submitted to the FCC for approval prior to selling the integrated unit. The required SAR testing measures emissions from the module and how they affect the person.

### RF Exposure

⚠ WARNING: To satisfy FCC RF exposure requirements for mobile transmitting devices, a separation distance of 20 cm or more should be maintained between the antenna of this device and persons during device operation. To ensure compliance, operations at closer than this distance is not recommended. The antenna used for this transmitter must not be co-located in conjunction with any other antenna or transmitter.

The preceding statement must be included as a CAUTION statement in OEM product manuals in order to alert users of FCC RF Exposure compliance.

### Europe (ETSI)

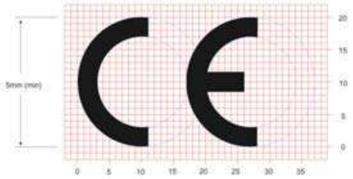
The XBee RF Modules have been certified for use in several European countries. For a complete list, refer to www.digi.com

If the XBee RF Modules are incorporated into a product, the manufacturer must ensure compliance of the final product to the European harmonized EMC and low-voltage/safety standards. A Declaration of Conformity must be issued for each of these standards and kept on file as described in Annex II of the R&TTE Directive.

Furthermore, the manufacturer must maintain a copy of the XBee user manual documentation and ensure the final product does not exceed the specified power ratings, antenna specifications, and/ or installation requirements as specified in the user manual. If any of these specifications are exceeded in the final product, a submission must be made to a notified body for compliance testing to all required standards.

### OEM Labeling Requirements

The 'CE' marking must be affixed to a visible location on the OEM product.



CE Labeling Requirements

The CE mark shall consist of the initials “CE” taking the following form:

- If the CE marking is reduced or enlarged, the proportions given in the above graduated drawing must be respected.
- The CE marking must have a height of at least 5mm except where this is not possible on account of the nature of the apparatus.
- The CE marking must be affixed visibly, legibly, and indelibly.

### Restrictions

**Power Output**: When operating in Europe, XBee-PRO 802.15.4 modules must operate at or below a transmit power output level of 10dBm. Customers have two choices for transmitting at or below 10dBm:

a. Order the standard XBee-PRO module and change the PL command to 0 (10dBm)

b. Order the International variant of the XBee-PRO module, which has a maximum transmit output power of 10dBm (@ PL=4).

Additionally, European regulations stipulate an EIRP power maximum of 12.86 dBm (19 mW) for the XBee-PRO and 12.11 dBm for the XBee when integrating antennas.

**France:** Outdoor use limited to 10 mW EIRP within the band 2454-2483.5 MHz.

**Norway:** Norway prohibits operation near Ny-Alesund in Svalbard. More information can be found at the Norway Posts and Telecommunications site (www.npt.no).

### Declarations of Conformity

Digi has issued Declarations of Conformity for the XBee RF Modules concerning emissions, EMC and safety. Files can be obtained by contacting Digi Support.

Important Note:

Digi does not list the entire set of standards that must be met for each country. Digi customers assume full responsibility for learning and meeting the required guidelines for each country in their distribution market. For more information relating to European compliance of an OEM product incorporating the XBee RF Module, contact Digi, or refer to the following web sites:

CEPT ERC 70-03E - Technical Requirements, European restrictions and general requirements: Available at www.ero.dk/.

R&TTE Directive - Equipment requirements, placement on market: Available at www.ero.dk/.

### Approved Antennas

When integrating high-gain antennas, European regulations stipulate EIRP power maximums. Use the following guidelines to determine which antennas to design into an application.

#### XBee RF Module

The following antenna types have been tested and approved for use with the XBee Module:

**Antenna Type: Yagi**
RF module was tested and approved with 15 dBi antenna gain with 1 dB cable-loss (EIRP Maximum of 14 dBm). Any Yagi type antenna with 14 dBi gain or less can be used with no cable-loss.

**Antenna Type: Omni-directional**
RF module was tested and approved with 15 dBi antenna gain with 1 dB cable-loss (EIRP Maximum of 14 dBm). Any Omni-directional type antenna with 14 dBi gain or less can be used with no cable-loss.

**Antenna Type: Flat Panel**
RF module was tested and approved with 19 dBi antenna gain with 4.8 dB cable-loss (EIRP Maximum of 14.2 dBm). Any Flat Panel type antenna with 14.2 dBi gain or less can be used with no cable-loss.

#### **XBee-PRO RF Module** (@ 10 dBm Transmit Power, PL parameter value must equal 0, or use International variant)

The following antennas have been tested and approved for use with the embedded XBee-PRO RF Module:

- Dipole (2.1 dBi, Omni-directional, Articulated RPSMA, Digi part number A24-HABSM)
- Chip Antenna (-1.5 dBi)
- Attached Monopole Whip (1.5 dBi)
- Integrated PCB Antenna (-0.5 dBi)

## Canada (IC)

### Labeling Requirements

Labeling requirements for Industry Canada are similar to those of the FCC. A clearly visible label on the outside of the final product enclosure must display the following text:

**Contains Model XBee Radio, IC: 4214A-XBEE**
**Contains Model XBee-PRO Radio, IC: 4214A-XBEEPRO**

The integrator is responsible for its product to comply with IC ICES-003 & FCC Part 15, Sub. B - Unintentional Radiators. ICES-003 is the same as FCC Part 15 Sub. B and Industry Canada accepts FCC test report or CISPR 22 test report for compliance with ICES-003.

## Japan

In order to gain approval for use in Japan, the XBee RF module or the International variant of the XBee-PRO RF module (which has 10 dBm transmit output power) must be used.

### Labeling Requirements

A clearly visible label on the outside of the final product enclosure must display the following text:

**ID: 005NYCA0378**

# Appendix B. Additional Information

## 1-Year Warranty

XBee®/XBee-PRO® RF Modules from Digi International, Inc. (the "Product") are warranted against defects in materials and workmanship under normal use, for a period of 1-year from the date of purchase. In the event of a product failure due to materials or workmanship, Digi will repair or replace the defective product. For warranty service, return the defective product to Digi, shipping prepaid, for prompt repair or replacement.

The foregoing sets forth the full extent of Digi's warranties regarding the Product. Repair or replacement at Digi's option is the exclusive remedy. THIS WARRANTY IS GIVEN IN LIEU OF ALL OTHER WARRANTIES, EXPRESS OR IMPLIED, AND DIGI SPECIFICALLY DISCLAIMS ALL WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. IN NO EVENT SHALL DIGI, ITS SUPPLIERS OR LICENSORS BE LIABLE FOR DAMAGES IN EXCESS OF THE PURCHASE PRICE OF THE PRODUCT, FOR ANY LOSS OF USE, LOSS OF TIME, INCONVENIENCE, COMMERCIAL LOSS, LOST PROFITS OR SAVINGS, OR OTHER INCIDENTAL, SPECIAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PRODUCT, TO THE FULL EXTENT SUCH MAY BE DISCLAIMED BY LAW. SOME STATES DO NOT ALLOW THE EXCLUSION OR LIMITATION OF INCIDENTAL OR CONSEQUENTIAL DAMAGES. THEREFORE, THE FOREGOING EXCLUSIONS MAY NOT APPLY IN ALL CASES. This warranty provides specific legal rights. Other rights which vary from state to state may also apply.

# X-CTU
# Configuration & Test Utility Software

## User's Guide

## Contents

Technical Support:

Online support: http://www.digi.com/support/eservice/login.jsp

Phone: (801) 765-9885

90001003_A

2008.08.20

## Introduction

This User’s Guide is intended to discuss the functions of Digi’s X-CTU software utility. Each function will be discussed in detail allowing a better understanding of the program and how it can be used.

X-CTU is a Windows-based application provided by Digi. This program was designed to interact with the firmware files found on Digi’s RF products and to provide a simple-to-use graphical user interface to them.

X-CTU is designed to function with all Windows-based computers running Microsoft Windows 98 SE and above.  X-CTU can either be downloaded from Digi’s Web site or an installation CD. When properly installed it can be launched by clicking on the icon on the PC desktop (see Figure 1) or selecting from the Start menu (see Figure 2).

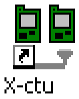


Figure 1

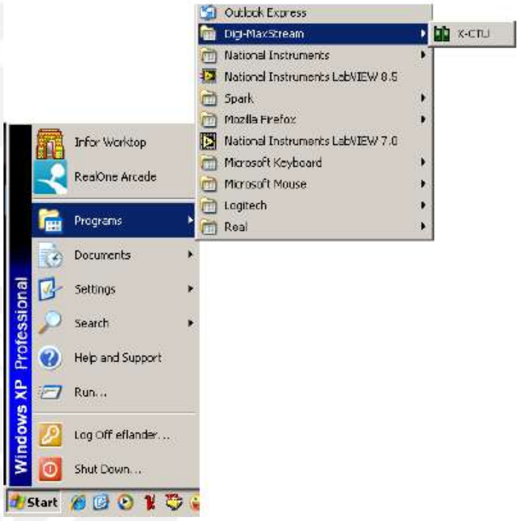


Figure 2

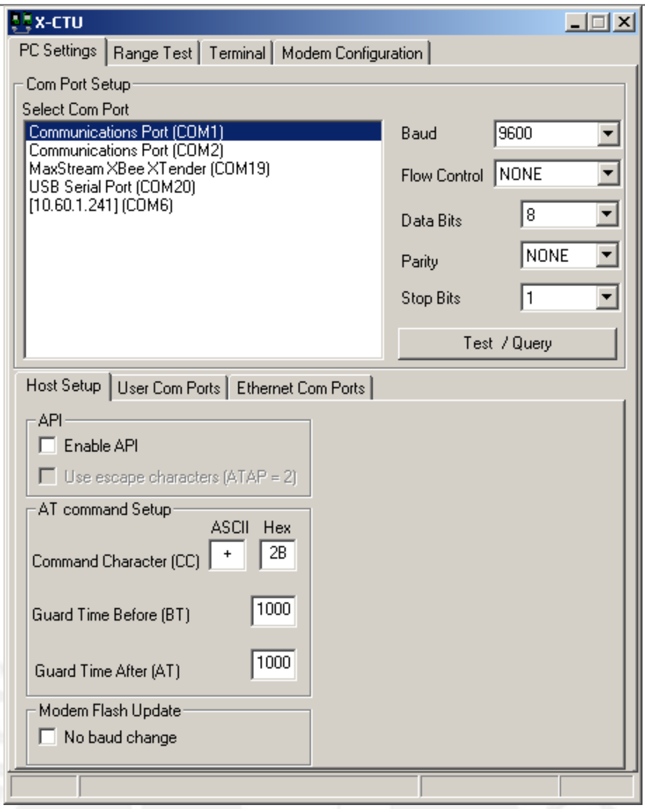
When launched, you will see four tabs across the top of the program (see Figure 3). Each of these tabs has a different function. The four tabs are:

**PC Settings**: Allows a customer to select the desired COM port and configure that port to fit the radios settings.

**Range Test**: Allows a customer to perform a range test between two radios.

**Terminal**: Allows access to the computers COM port with a terminal emulation program. This tab also allows the ability to access the radios’ firmware using AT commands (for a complete listing of the radios’ AT commands, please see the product manuals available online).

**Modem Configuration**: Allows the ability to program the radios’ firmware settings via a graphical user interface. This tab also allows customers the ability to change firmware versions.



Com Port Setup

Select Com Port

Communications Port (COM1)
Communications Port (COM2)
MaxStream XBee XTender (COM19)
USB Serial Port (COM20)
[10.60.1.241] (COM6)

Baud 9600

Flow Control NONE

Data Bits 8

Parity NONE

Stop Bits 1

API

□ Enable API

□ Use escape characters (ATAP = 2)

AT command Setup

| | ASCII | Hex |
|---|---|---|
| Command Character (CC) | + | 2B |
| Guard Time Before (BT) | | 1000 |
| Guard Time After (AT) | | 1000 |

Modem Flash Update

□ No baud change

Figure 3

## PC Settings Tab

When the program is launched, the default tab selected is the “PC Settings” tab. The PC Settings tab is broken down into three basic areas: The COM port setup, the Host Setup, and the User Com ports.

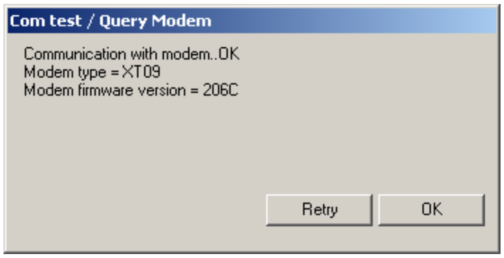
**COM port setup:**

The PC settings tab allows the user to select a COM port and configure the selected COM port settings when accessing the port. Some of these settings include:

| | |
|---|---|
| Baud Rate: | Both standard and non-standard |
| Flow Control: | Hardware, Software (Xon/Xoff), None |
| Data bits: | 4, 5, 6, 7, and 8 data bits |
| Parity: | None, Odd, Even, Mark and Space |
| Stop bit: | 1, 1.5, and 2 |

To change any of the above settings, select the pull down menu on the left of the value and select the desired setting. To enter a non-standard baud rate, type the baud rate into the baud rate box to the left.

The **Test** / **Query** button is used to test the selected COM port and PC settings. If the settings and COM port are correct, you will receive a response similar to the one depicted in Figure 4 below.



Com test / Query Modem

Communication with modem..OK
Modem type = XT09
Modem firmware version = 206C

Figure 4

**Host Setup:**

The Host Setup tab allows the user to configure how the X-CTU program is to interface with a radio's firmware. This includes determining whether API or AT command mode will be used to access the module's firmware as well as the proper command mode character and sequence.

By default, the Host Settings are as follows:

| | |
|---|---|
| API mode: | not enabled (Not checked) |
| Command mode Character: | + (ACSII) 2B (Hex). |
| Before Guard Time: | 1000 (1 Sec) |
| After Guard Time: | 1000 (1 Sec) |

This is the default value of our radios. If this is not the value of the AT, BT, or GT commands of the connected radio, enter the respective value here.

**User COM ports:**

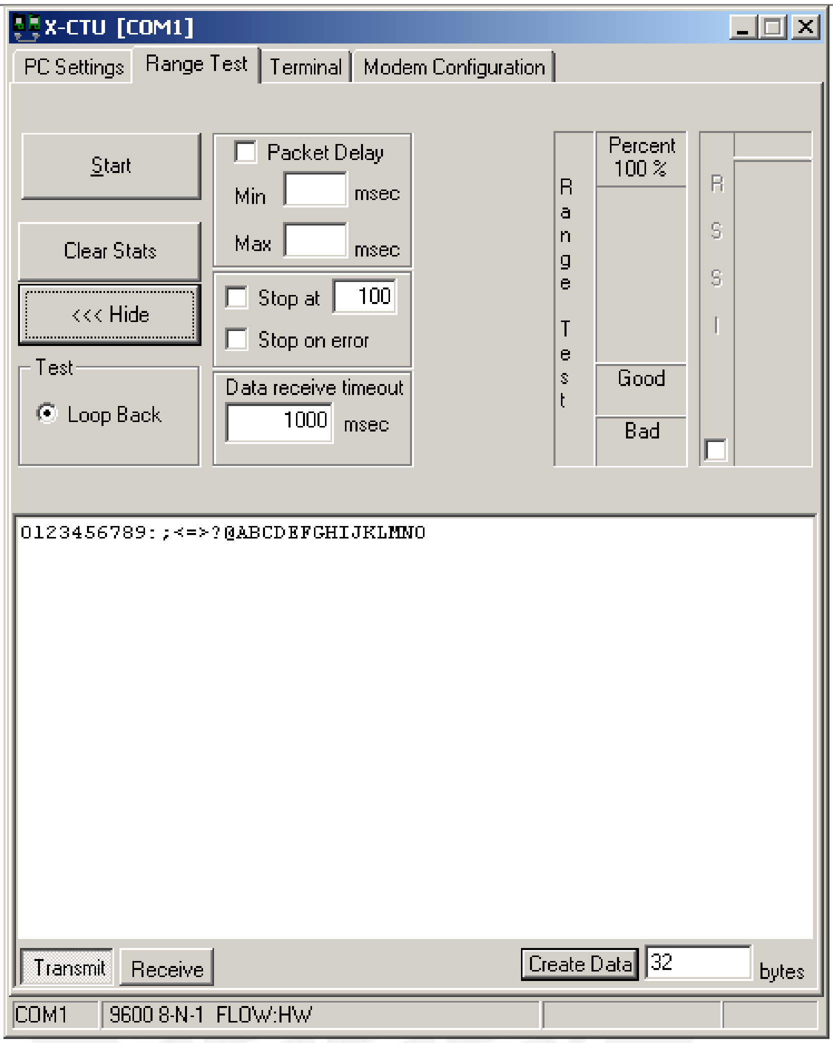
The user COM port option allows the user to "Add" or "Delete" a user-created COM port. This is only for temporary use. Once the program has closed, the user-created COM port will disappear and is no longer accessible to the program.

## Range Test Tab

The range test tab is designed to verify the range of the radio link by sending a user-specified data packet and verifying the response packet is the same, within the time specified. For performing a standard range test, please follow the steps found in most Quick Start or Getting Started Guides that ship with the product.

**Packet Data and Size**

By default, the size of the data packet sent is 32 bytes. This data packet specified can be adjusted in either size or the text sent.



```
0123456789:;<=>?@ABCDEFGHIJKLMNO
```

Figure 5

To modify the size of the packet sent, change the value next to the “Create Data” box and click on the “Create Data” button (see Figure 5). If you want to change the data sent, delete the text in the transmit window and place in your desired text.

By modifying the text, data packet size, packet delay and the data receive timeout; the user is able to simulate a wide range of scenarios.

### RSSI:

The RSSI option of the X-CTU allows the user to see the RSSI (Received Signal Strength Indicator) of a received packet when performing a range test.

### API Function:

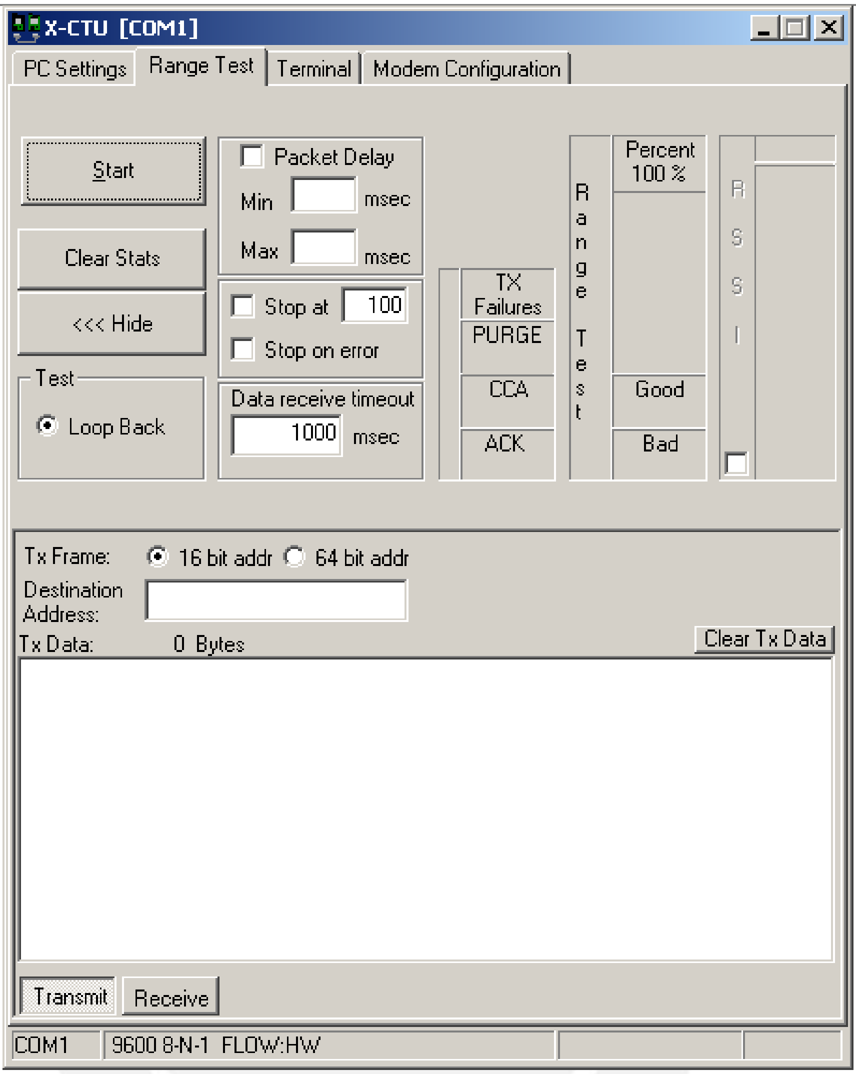
The X-CTU also allows the user to test the API function of a radio during a range test.

To perform a range test with the API function of the radio, follow the steps outlined below:

1: Configure the Base with API enabled and a unique 16 bit or 64 bit source address.
2: Configure the remote radio with a unique source address and set the Destination address to equal the Base radio's source address.
3: Enable the API option of the X-CTU on the PC Settings tab and connect the base radio to the PC (See Figure 3).
4: Connect the red loopback adapter to the remote radio and place them a distance apart.
5: Enter either the 16 bit or 64 bit destination address of the remote radio into the Destination Address box on the Range Test tab (See figure 6).
6: Create a data packet of your choosing by typing in the data in the Transmit box
7: To start a Range test, click on Start.

You will notice the TX failures, Purge, CCA, and ACK messages will increment accordingly while the range test is performed.

To stop a range test, click on the Stop button.



□ Packet Delay Min msec Max msec

□ Stop at 100

□ Stop on error

Data receive timeout 1000 msec

Test: ● Loop Back

Percent 100 %

Tx Frame: ● 16 bit addr ○ 64 bit addr

Destination Address:

Tx Data: 0 Bytes

COM1 | 9600 8-N-1 FLOW:HW

Figure 6

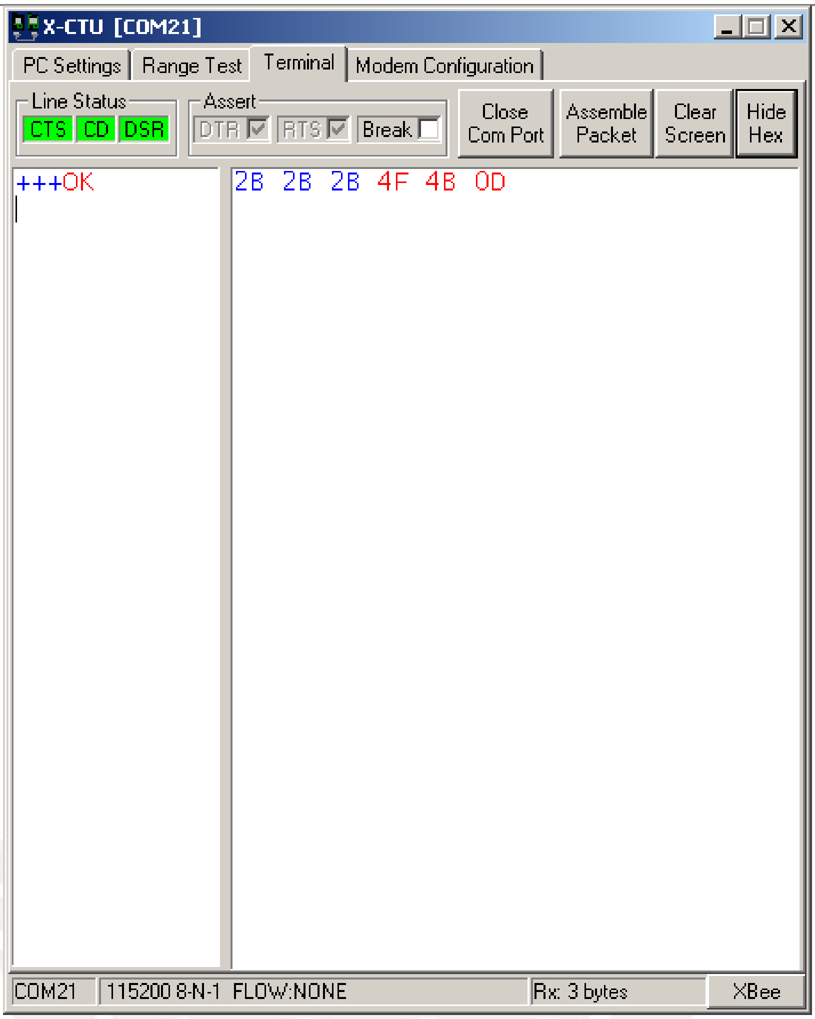
## The Terminal Tab

The Terminal tab has three basic functions:

Terminal emulator
Ability to send and receive predefined data pacts (Assemble packet)
Ability to send and receive data in Hex and ASCII formats (Show/Hide hex)

### The main terminal window

The main white portion of this tab is where most of the communications information will occur while using X-CTU as a terminal emulator. The text in blue is what has been typed in and directed out to the radio's serial port while the red text is the incoming data from the radio's serial port (see Figure 7).



```
+++OK
```

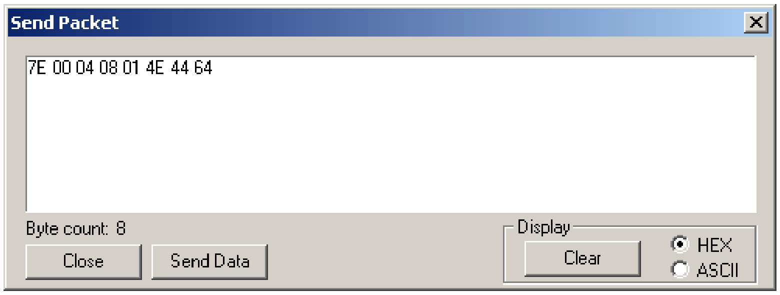
```
2B 2B 2B 4F 4B 0D
```

```
COM21 | 115200 8-N-1  FLOW:NONE | Rx: 3 bytes | XBee
```

Figure 7

**Assemble Packet**

The Assemble Packet option on the Terminal tab is designed to allow the user to assemble a data packet in either ASCII or Hex characters. This is accomplished by selecting the Assemble packet window and choosing either ASCII (default) or Hex. Once selected, the data packet is assembled by typing in the desired characters as depicted in Figure 8.



```
7E 00 04 08 01 4E 44 64
```

Byte count: 8

Display: ● HEX ○ ASCII

Figure 8

The **Line Status** indicators depicted in Figure 5 shows the status of the RS-232 hardware flow control lines.  Green indicates the line is asserted while black indicates de-asserted.

The **Break** option is for engaging the serial line break. This can be accomplished by checking or asserting the Break option. Asserting the Break will place the DI line high and prevent data from being sent to the radio.

## Modem Configuration tab

The Modem configuration tab has four basic functions:

1: Provide a Graphical User Interface with a radio's firmware
2: Read and Write firmware to the radio's microcontroller
3: Download updated firmware files from either the web or from a compressed file
4: Saving or loading a modem profile

### Reading a radio's firmware

To read a radio's firmware, follow the steps outlined below:

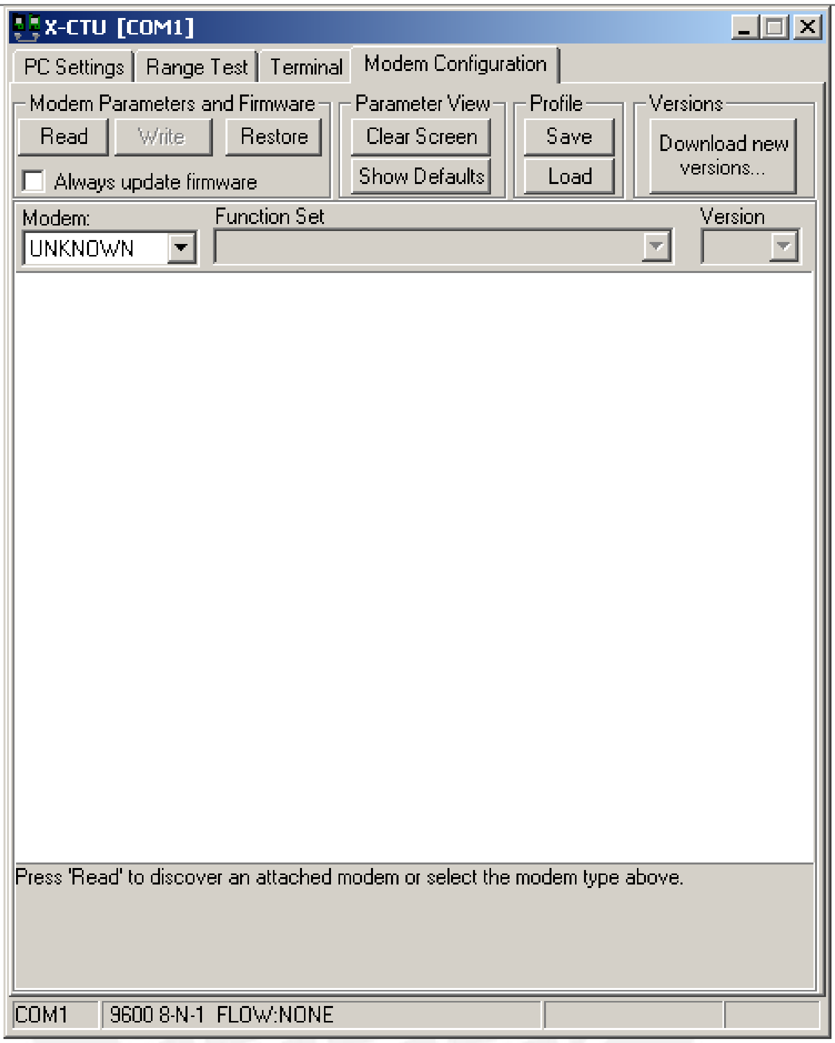
1: Connect the radio module to the interface board and connect this assembly or a packaged radio (PKG) to the PC's corresponding port (IE: USB, RS232, Ethernet etc.).
2: Set the PC Settings tab (see Figure 3) to the radio's default settings.
3: On the Modem Configuration tab, select "Read" from the Modem Parameters and Firmware section (see Figure 9).

### Making changes to a radio's firmware

Once the radio's firmware has been read, the configuration settings are displayed in three colors (see Figure 10):

Black – not settable or read-only
Green – Default value
Blue – User-specified

To modify any of the user-settable parameters, click on the associated command and type in the new value for that parameter. For ease of understanding a specific command, once the command is selected, a quick description along with its limits is provided at the bottom of the screen. Once all of the new values have been entered, the new values are ready to be saved to the radio's non-volatile memory.



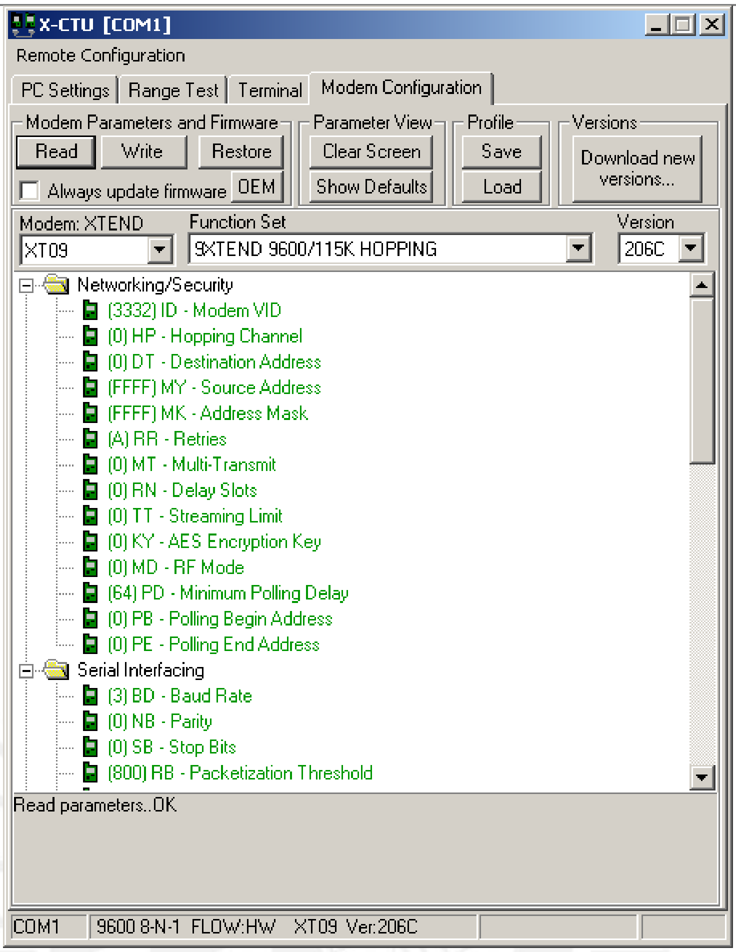
Press 'Read' to discover an attached modem or select the modem type above.

COM1 9600 8-N-1 FLOW:NONE

Figure 9

Writing firmware to the Radio

To write the parameter changes to the radio's non-volatile memory, click on the Write button located in the Modem Parameters and Firmware section (see Figure 10)



Modem: XTEND: XT09
Function Set: 9XTEND 9600/115K HOPPING
Version: 206C

```
⊟ Networking/Security
    (3332) ID - Modem VID
    (0) HP - Hopping Channel
    (0) DT - Destination Address
    (FFFF) MY - Source Address
    (FFFF) MK - Address Mask
    (A) RR - Retries
    (0) MT - Multi-Transmit
    (0) RN - Delay Slots
    (0) TT - Streaming Limit
    (0) KY - AES Encryption Key
    (0) MD - RF Mode
    (64) PD - Minimum Polling Delay
    (0) PB - Polling Begin Address
    (0) PE - Polling End Address
⊟ Serial Interfacing
    (3) BD - Baud Rate
    (0) NB - Parity
    (0) SB - Stop Bits
    (800) RB - Packetization Threshold
```

```
Read parameters..OK
```

COM1 | 9600 8-N-1 FLOW:HW | XT09 Ver:206C

Figure 10

Downloading Updated Firmware Files

Another function of the Modem Configuration tab is allowing the user to download updated firmware files from either the web or install them from a disk or CD. This is accomplished by following the steps below:

1: Click on the Download New Versions… option under the Version section
2a: Click on Web for downloading new firmware files from the web
2b: Click on the File when installing compressed firmware files from a CD or saved file (see Figures 11 and 12)
    2bi: Browse to the location the file is saved at and click on Open (see Figure 13)
3: Click on OK and Done when prompted

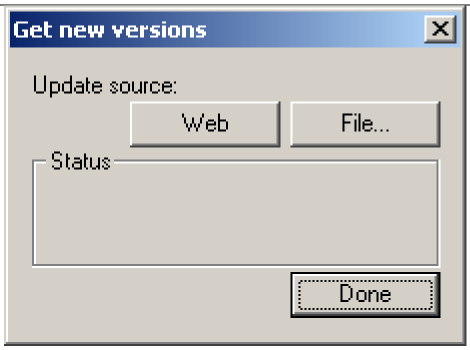
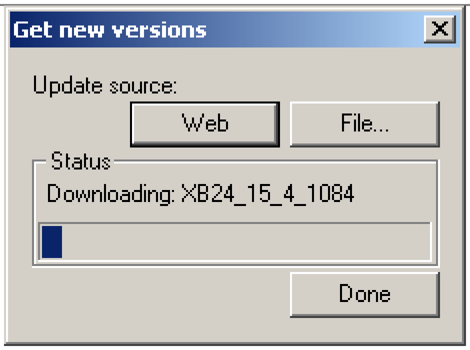
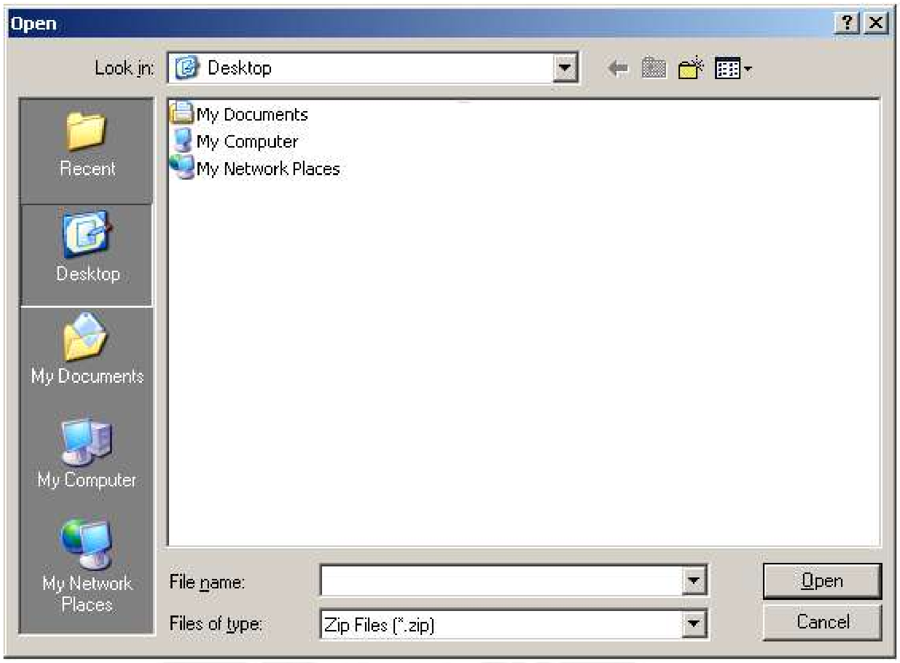


Figure 11



Downloading: XB24_15_4_1084

Figure 12



Files of type: Zip Files (*.zip)

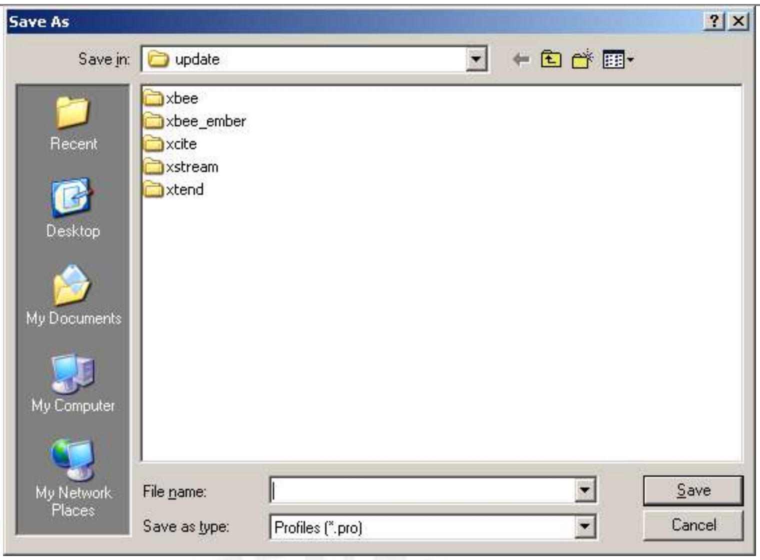
Figure 13

Modem Profiles

The X-CTU has the ability to save and write saved modem profiles or configuration to the radio. This function is useful in a production environment when the same parameters need to be set on multiple radios.

How to save a profile:

1: Set the desired settings within the radio's firmware as described in the Making changes to the radios firmware section
2: Click Save in the Profile section
3: Type in the desired name of this profile in the File Name box (see Figure 14)
4: Browse to the location where you wish to save your profile
5: Click Save



Save in: update

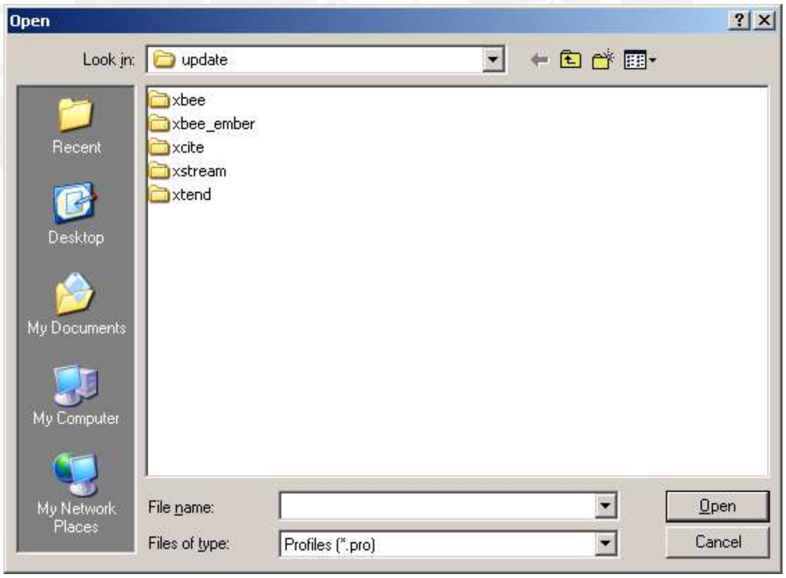
File name:

Save as type: Profiles (*.pro)

Figure 14

How to load a saved profile:

1: Click on Load from the profile section
2: Browse to the location of the file and click on the desired file (see Figure 15)
3: Click Open



Look in: update

File name:

Files of type: Profiles (*.pro)

Figure 15

To save the loaded profile to the radio once you have loaded the file, follow the steps outlined in the Writing firmware to the radio section above.

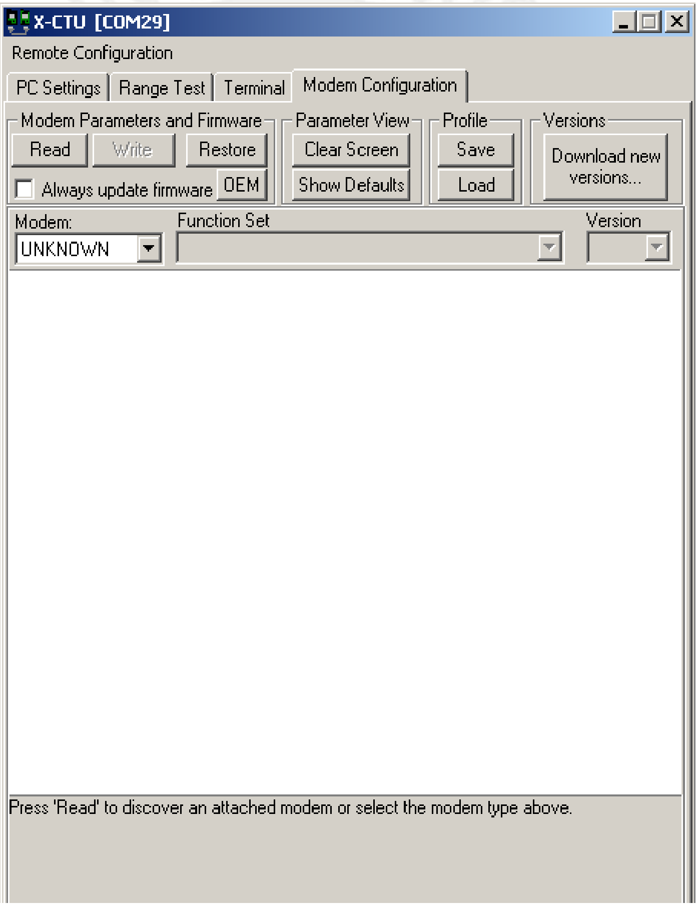
To find out how to load the saved profiles in a production environment from a DOS prompt, please follow the steps outlined in Digi's online Knowledgebase at http://www.maxstream.net/support/knowledgebase/article.php?kb=126

XBee 802.15.4 modules with firmware version 1xCx and above, XBee ZNet 2.5 modules, and XBee ZB modules offer the ability to be configured with over the air commands. With the addition of this new feature, the user is able to configure remote radio parameters with X-CTU or API packets. To use the remote configuration tool, the following is required:

- The radio connected to the PC must be in API mode
- The remote radio must be associated or within range of the base radio

To access remote radios through X-CTU's Modem Configuration tab, perform the steps below:

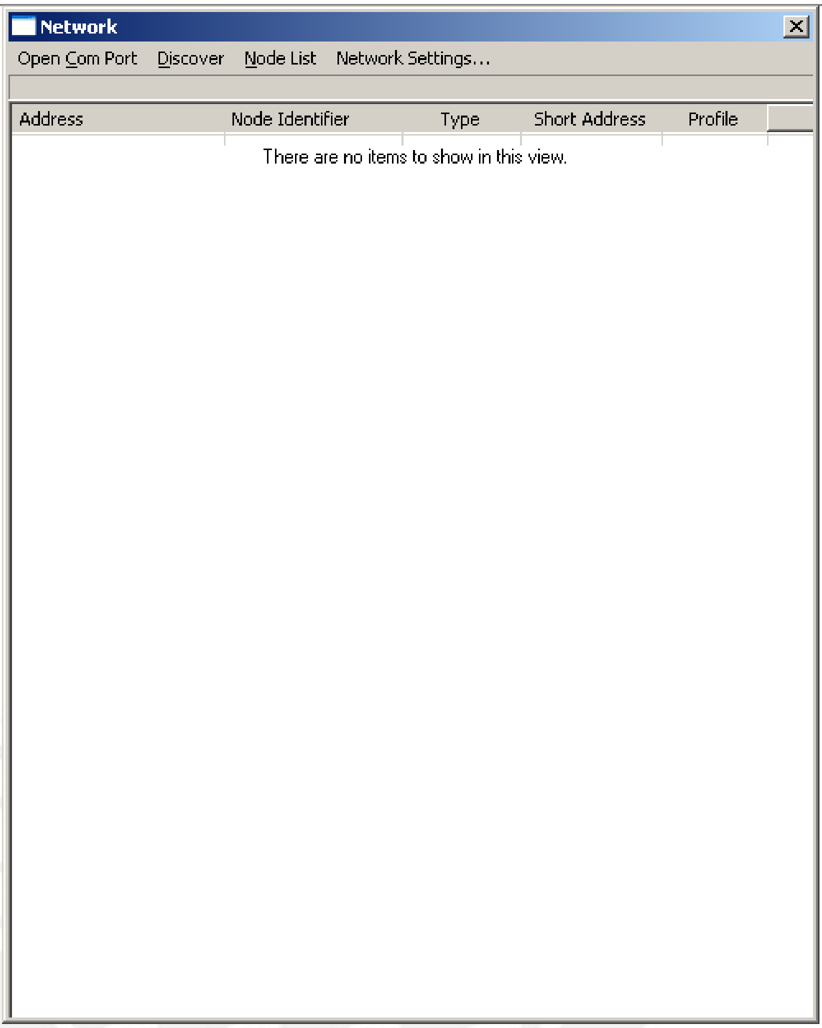
- Enable API on the PC Settings tab
- Verify the COM port selection and settings
- On the Modem Configuration tab, select the Remote Configuration option on the top left corner of the program



Modem: UNKNOWN

Press 'Read' to discover an attached modem or select the modem type above.

- Select Open Com port
- Select Discover



- Select the desired modem from the discovered node list
- On the Modem configuration tab, select Read

The remote radio's configuration is now displayed on the Modem Configuration tab. At this point, the same options exist with respect to Read and Write parameter changes. Please note that the ability to change firmware versions is still limited to the radio's UART.

To clear the discovered node list, click on Node List and Clear.

The Node List option provides several additional options, including:

- Ability to print the discovered list
- Ability to remove a specific node from a list
- Ability to add additional nodes that have not been discovered
- Save the Node List
- Load a saved Node List
- Select/filter All, Routers, or End nodes

For specific questions related to the X-CTU configuration and test utility software, please contact our Support department, Mon – Fri, 8am – 5pm U.S. Mountain Time:

US and Canada Toll free:
(866)765-9885

Local or International calls:
(801) 765-9885

Online support: http://www.digi.com/support/eservice/login.jsp

```
clear all
close all
clc


delete(instrfind({'Port'},{'COM10'})); %ajustar puerto serie!

B = Bluetooth('btspp://0015FFF21097',1)%%ID y luego va el canal
set(B,'InputBufferSize',1024);
s1= B
%%s1 =
serial('COM10','BaudRate',9600,'DataBits',8,'Parity','none','StopBits',1)
s1 =
serial('COM10','BaudRate',9600,'DataBits',8,'Parity','none','StopBits',1,
...
'Flowcontrol','none');
set(s1,'InputBufferSize',2024);
fopen(s1);
%s1=B
%%t=tic;
a=1;
minutoanterior=0;
for j=1:1:36000
 fwrite(s1,'A')
 % Se envía un 'A' para que indique el comienzo de recoleccion de datos
   nMaxMuestras=42;
   reloj= 8;
   potencia= 6
     M= zeros(nMaxMuestras*4+reloj+potencia,1)
M= fread(s1,nMaxMuestras*4+reloj+potencia);
voltaje=zeros(nMaxMuestras,1); %senal auxiliar
corriente=zeros(nMaxMuestras,1); %senal auxiliar
for i=0:1:nMaxMuestras-1
    voltaje(i+1)=M(4*i+3)+ M(4*i+4)*256 ;
    corriente(i+1)=M(4*i+2)*256+M(4*i+1);
end
signo=M(nMaxMuestras*4+1);
millar=M(nMaxMuestras*4+2);
centena= M(nMaxMuestras*4+3);
decena=M(nMaxMuestras*4+4);
unidad=M(nMaxMuestras*4+5);
potenciauc= millar*1000+centena*100+decena*10+unidad;
if (signo==45)
potenciauc= potenciauc*-1;
end

dia=M(nMaxMuestras*4+7);
mes=M(nMaxMuestras*4+8);
anio=M(nMaxMuestras*4+9);
hora= M(nMaxMuestras*4+10);
minuto=M(nMaxMuestras*4+11);
segundo=M(nMaxMuestras*4+12);
diasem=M(nMaxMuestras*4+13);
fin=M(nMaxMuestras*4+14);
```

```
voltaje= (voltaje-72)*(622/869)-311;
subplot(2,2,1);
plot(voltaje,'r')
xlabel('Muestras')
  ylabel('Voltaje')
  title('Voltaje AC')
  grid on

 corriente= (corriente-102)*(60/819)-30;


 maximo= max(corriente);
%minimo= min(corriente);
if (maximo < 0.3 )
 aparato= 0;
else
  aparato= 1;
end

if (mod(minuto,6)==0)
    if minuto ~= minutoanterior
    horario1(a)= diasem;
    horario2(a)= hora;
    horario3(a)=minuto;
    salida(a)= aparato;
    muestrasp(a)=potenciauc;
    a= a+1;
    end
end

 minutoanterior= minuto;
subplot(2,2,2)
plot(corriente,'b')
xlabel('Muestras')
ylabel('Corriente')
title('Corriente AC')
grid on

subplot(2,2,3)
potencia= voltaje.*corriente;
plot(potencia,'g')
xlabel('Muestras')
ylabel('Potencia')
title('Potencia AC')
grid on
P= potencia*(1/42);
Pactiva= sum(P)
subplot(2,2,4)
plot(Pactiva,potenciauc)
pause(0.7);
end

%% Red neuronal
```

```
%ejemplo p=[[1 7 0.1]' [2 13 0]' [3 4 0.2]' [4 12 0.9]' [5 12 0.7]' [6 5
0.3]' [7 1 0.2]' [1 13 1]' [1 13 0]' [1 14 0.5]' [1 15 0.6]'];
% ejemplo t=[1 0 0 1 1 0 0 0 1 1 0];
%horario1
%horario2
%horario3
clear all
close all
clc

%%pruebas
A1=ones(1,160)
A2= 2*ones(1,160)
A3= 3*ones(1,160)
A4= 4*ones(1,160)
A5= 5*ones(1,160)
A6= 6*ones(1,160)
A7= 7*ones(1,160)
Asemana= [A1 A2 A3 A4 A5 A6 A7];%primero

Ap=Asemana

hora=[  7*ones(1,10) 8*ones(1,10) 9*ones(1,10) 10*ones(1,10)
11*ones(1,10) 12*ones(1,10) 13*ones(1,10) 14*ones(1,10) 15*ones(1,10)
16*ones(1,10) 17*ones(1,10) 18*ones(1,10) 19*ones(1,10) 20*ones(1,10)
21*ones(1,10) 22*ones(1,10)]
horasem=[hora hora hora hora hora hora hora];  % segundo
Bp=horasem*(2/23) -1
MIN=[0 1 2 3 4 5 6 7 8 9 ]*6
minutos=[ MIN MIN MIN MIN MIN MIN MIN MIN MIN  MIN MIN MIN MIN MIN MIN
MIN ]

minutossem=[minutos minutos minutos minutos minutos minutos minutos ];
%tercero
Cp= minutossem/0.6

t1=[1*ones(1,10) 0*ones(1,10) 0*ones(1,10) 0*ones(1,10)  0*ones(1,10)
0*ones(1,10)  0*ones(1,10) 0*ones(1,10)  0*ones(1,10) 0*ones(1,10)
0*ones(1,10) 1*ones(1,10)  0*ones(1,10) 1*ones(1,10)  0*ones(1,10)
0*ones(1,10)  ]
t2=[0*ones(1,10) 1*ones(1,10) 0*ones(1,10) 0*ones(1,10)  0*ones(1,10)
0*ones(1,10)  0*ones(1,10) 0*ones(1,10)  0*ones(1,10) 0*ones(1,10)
0*ones(1,10) 0*ones(1,10)  0*ones(1,10) 0*ones(1,10)  1*ones(1,10)
0*ones(1,10)  ]
t3=[0*ones(1,10) 1*ones(1,10) 0*ones(1,10) 0*ones(1,10)  0*ones(1,10)
1*ones(1,10)  0*ones(1,10) 0*ones(1,10)  0*ones(1,10) 0*ones(1,10)
0*ones(1,10) 0*ones(1,10)  0*ones(1,10) 0*ones(1,10)  1*ones(1,10)
0*ones(1,10)  ]
t4=[1*ones(1,10) 0*ones(1,10) 0*ones(1,10) 0*ones(1,10)  0*ones(1,10)
1*ones(1,10)  0*ones(1,10) 0*ones(1,10)  0*ones(1,10) 0*ones(1,10)
0*ones(1,10) 0*ones(1,10)  0*ones(1,10) 0*ones(1,10)  1*ones(1,10)
0*ones(1,10)  ]
t5=[1*ones(1,10) 0*ones(1,10) 0*ones(1,10) 0*ones(1,10)  0*ones(1,10)
1*ones(1,10)  0*ones(1,10) 0*ones(1,10)  0*ones(1,10) 0*ones(1,10)
0*ones(1,10) 0*ones(1,10)  0*ones(1,10) 0*ones(1,10)  1*ones(1,10)
0*ones(1,10)  ]
```

```
t6=[1*ones(1,10) 0*ones(1,10) 0*ones(1,10) 0*ones(1,10)  0*ones(1,10)
0*ones(1,10)  0*ones(1,10) 0*ones(1,10)  0*ones(1,10) 0*ones(1,10)
0*ones(1,10) 0*ones(1,10)  0*ones(1,10) 0*ones(1,10)  1*ones(1,10)
0*ones(1,10)  ]
t7=[0*ones(1,10) 0*ones(1,10) 0*ones(1,10) 0*ones(1,10)  0*ones(1,10)
0*ones(1,10)  0*ones(1,10) 0*ones(1,10)  0*ones(1,10) 0*ones(1,10)
0*ones(1,10) 0*ones(1,10)  0*ones(1,10) 1*ones(1,10)  1*ones(1,10)
0*ones(1,10)  ]
t= [t1 t2 t3 t4 t5 t6 t7]
A= horario1; %dia

B= horario2;% hora
%C= horario3;
C= horario3*(0.1/6);% minutos

p= [Ap' Bp' Cp']';



%p=   [ 3    3    3    3    3    3    3    3   3    3    3    3    3    3
3;
 %   14    14    14       14    14       14   14       14    14    15    15
15     15      15    15;
  %  0.1    0.2   0.3    0.4   0.5      0.6  0.7      0.8   0.9    0
0.1   0.2    0.3    0.4   0.5]
%p =[ 30    30    30    30    30    30    30    30    30    30    30
30     30   30    30;
 %    14    14    14       15    15       15    15       15    15    15
15     15    15       15    15;
  %  57    58    59      0     1         2     3       4       5     6
7       8     9      10    11]

%p= p/100;
%salida=[ 1     1     1     1     1     1     1     1     1     1     1
1     0     0    0]

t= salida;
net=newff(minmax(p),[20,30,1],{'tansig','tansig','purelin'},'traingda');
net.trainParam.show = 20;
net.trainParam.lr = 0.2; % Constante de aprendizaje inicial
net.trainParam.lr_inc = 1.05;
net.trainParam.lr_dec = 0.7;
net.trainParam.epochs = 80000;
net.trainParam.goal = 3*1e-3;
%net.trainParam.
[net,tr]=train(net,p,t);
c= sim(net, p)
subplot(2,1,1)
plot(t*100);
   axis([0 1200 -1 1.5]);
        xlabel('Muestras');
         ylabel('ON/OFF');
          title('Rutina del usuario');
          grid on;

subplot(2,1,2)
plot(c*100,'r');
```

```
   axis([0 1200 -1 1.5]);
  xlabel('Muestras');
         ylabel('ON/OFF');
          title('Rutina del usuario por red neuronal');
          grid on;




 %%
 fwrite(s1,'A');
    nMaxMuestras=325;
     M= zeros(nMaxMuestras*6+7,1);
M= fread(s1,nMaxMuestras*6+7);
%%time= toc(t);
FIN= M(nMaxMuestras*6+7);
if (FIN==13)
   display(['Recepcion exitosa']);
end
   voltaje=zeros(nMaxMuestras,1); %senal auxiliar
corriente=zeros(nMaxMuestras,1); %senal auxiliar
for i=0:1:nMaxMuestras-1
     corriente(i+1)=(M(6*i+2)+ 256*M(6*i+3))/100;
    if (M(6*i+1)==45)
    corriente(i+1)= (corriente(i+1)*-1);
    end
   voltaje(i+1)=(M(6*i+5)+ 256*M(6*i+6))/100;
    if (M(6*i+4)==45)
        voltaje(i+1)=(voltaje(i+1)*-1);

    end
end
dia=M(nMaxMuestras*6+1);
mes=M(nMaxMuestras*6+2);
anio=M(nMaxMuestras*6+3);
hora= M(nMaxMuestras*6+4);
minuto=M(nMaxMuestras*6+5);
segundo=M(nMaxMuestras*6+6);


subplot(2,2,1)
plot(voltaje,'r')
xlabel('Muestras')
  ylabel('Voltaje')
  title('Voltaje AC')

subplot(2,2,2)
plot(corriente,'b')
xlabel('Muestras')
ylabel('Corriente')
title('Corriente AC')

subplot(2,2,3)
potencia= voltaje.*corriente;
plot(potencia,'g')
```

```
xlabel('Muestras')
ylabel('Potencia')
title('Potencia AC')

Pactiva= potencia*(0.05/16.7);
Pactiva= sum(Pactiva)


fclose(s1);

for i=1:160
    lunes(i)=t(i+160*0);
    potencia= 200+20*rand(1)
    consumol(i)= potencia*t(i+160*0)

    martes(i)=t(i+160*1);
    potencia= 200+20*rand(1)
    consumom(i)= potencia*t(i+160*1)

    miercoles(i)=t(i+160*2);
     potencia= 200+20*rand(1)
     consumomi(i)= potencia*t(i+160*2)


     jueves(i)= t(i+160*3);
      potencia= 200+20*rand(1)
    consumoj(i)= potencia*t(i+160*3)

    viernes(i)= t(i+160*4);
     potencia= 200+20*rand(1)
    consumov(i)= potencia*t(i+160*4)

    sabado(i)=t(i+160*5);
     potencia= 200+20*rand(1)
    consumos(i)= potencia*t(i+160*5)

    domingo(i)=t(i+160*6);
     potencia= 200+20*rand(1)
    consumod(i)= potencia*t(i+160*6)

end

; % 200W+-20
for i=1:1120
potencia= 200+20*rand(1)
consumo(i)= potencia*t(i)

end
cosumoWh= (sum(consumo)*6)*(1/60)
costo= cosumoWh*0.3490
plot(1:1120, consumo)

subplot(2,2,1)
bar((1:160)/10 +7 ,lunes,'b')
```

```
axis([7 23 0 1]);
xlabel('Horas')
ylabel('ON/OFF')
title('Rutina del día Lunes')
%%
subplot(2,2, 1)
bar((1:160)/10 +7 ,consumol,'k')
axis([7 23 0 300]);
xlabel('Horas')
ylabel('Potencia (W)')
title('Rutina del día Lunes')


subplot(2,2, 2)
bar((1:160)/10 +7 ,consumom,'r')
axis([7 23 0 300]);
xlabel('Horas')
ylabel('Potencia (W)')
title('Rutina del día Martes')



subplot(2,2, 3)
bar((1:160)/10 +7 ,consumomi,'y')
axis([7 23 0 300]);
xlabel('Horas')
ylabel('Potencia (W)')
title('Rutina del día Miercoles')

subplot(2,2, 4)
bar((1:160)/10 +7 ,consumoj,'g')
axis([7 23 0 300]);
xlabel('Horas')
ylabel('Potencia (W)')
title('Rutina del día Jueves')

subplot(2,2, 1)
bar((1:160)/10 +7 ,consumov,'b')
axis([7 23 0 300]);
xlabel('Horas')
ylabel('Potencia (W)')
title('Rutina del día Viernes')

subplot(2,2, 2)
bar((1:160)/10 +7 ,consumos,'m')
axis([7 23 0 300]);
xlabel('Horas')
ylabel('Potencia (W)')
title('Rutina del día Sabado')

subplot(2,2, 3)
bar((1:160)/10 +7 ,consumod,'c')
axis([7 23 0 300]);
xlabel('Horas')
ylabel('Potencia (W)')
```

```
title('Rutina del día Domingo')

%%
subplot(2,2,3)
bar((1:160)/10 +7 ,martes,'b')
axis([7 23 0 1]);
xlabel('Horas')
ylabel('ON/OFF')
title('Rutina del día Martes')

subplot(2,2,3)
bar((1:160)/10 +7 ,miercoles,'g')
axis([7 23 0 1]);
xlabel('Horas')
ylabel('ON/OFF')
title('Rutina del día Miercoles')

subplot(2,2,4)
bar((1:160)/10 +7 ,jueves,'y')
axis([7 23 0 1]);
xlabel('Horas')
ylabel('ON/OFF')
title('Rutina del día Jueves')


subplot(2,2,1)
bar((1:160)/10 +7 ,viernes,'b')
axis([7 23 0 1]);
xlabel('Horas')
ylabel('ON/OFF')
title('Rutina del día Viernes')

subplot(2,2,2)
bar((1:160)/10 +7 ,sabado,'r')
axis([7 23 0 1]);
xlabel('Horas')
ylabel('ON/OFF')
title('Rutina del día Sabado')

subplot(2,2,3)
bar((1:160)/10 +7 ,domingo,'g')
axis([7 23 0 1]);
xlabel('Horas')
ylabel('ON/OFF')
title('Rutina del día Domingo')
%%
for i=1:160
    lunesr(i)=c(i+160*0);
    martesr(i)=c(i+160*1);
    miercolesr(i)=c(i+160*2);
    juevesr(i)= c(i+160*3);
    viernesr(i)= c(i+160*4);
    sabador(i)=c(i+160*5);
    domingor(i)=c(i+160*6);
```

```
end
subplot(2,2,1)
bar((1:160)/10 +7 ,lunesr,'b')
axis([7 23 0 1]);
xlabel('Horas')
ylabel('ON/OFF')
title('Rutina del día Lunes')

subplot(2,2,2)
bar((1:160)/10 +7 ,martesr,'r')
axis([7 23 0 1]);
xlabel('Horas')
ylabel('ON/OFF')
title('Rutina del día Martes')

subplot(2,2,3)
bar((1:160)/10 +7 ,miercolesr,'g')
axis([7 23 0 1]);
xlabel('Horas')
ylabel('ON/OFF')
title('Rutina del día Miercoles')

subplot(2,2,4)
bar((1:160)/10 +7 ,juevesr,'y')
axis([7 23 0 1]);
xlabel('Horas')
ylabel('ON/OFF')
title('Rutina del día Jueves')


subplot(2,2,1)
bar((1:160)/10 +7 ,viernesr,'b')
axis([7 23 0 1]);
xlabel('Horas')
ylabel('ON/OFF')
title('Rutina del día Viernes')

subplot(2,2,2)
bar((1:160)/10 +7 ,sabador,'r')
axis([7 23 0 1]);
xlabel('Horas')
ylabel('ON/OFF')
title('Rutina del día Sabado')

subplot(2,2,3)
bar((1:160)/10 +7 ,domingor,'g')
axis([7 23 0 1]);
xlabel('Horas')
ylabel('ON/OFF')
title('Rutina del día Domingo')



%% con sensores de presencia
```

```
t21=[1*ones(1,10) 0*ones(1,10) 0*ones(1,10) 0*ones(1,10)  0*ones(1,10)
0*ones(1,10)  0*ones(1,10) 0*ones(1,10)  0*ones(1,10) 0*ones(1,10)
0*ones(1,10) 1*ones(1,10)  0*ones(1,10) 1*ones(1,10)  0*ones(1,10)
0*ones(1,10)  ]
t22=[0*ones(1,10) 1*ones(1,10) 0*ones(1,10) 0*ones(1,10)  0*ones(1,10)
0*ones(1,10)  0*ones(1,10) 0*ones(1,10)  0*ones(1,10) 0*ones(1,10)
0*ones(1,10) 0*ones(1,10)  0*ones(1,10) 0*ones(1,10)  1*ones(1,10)
0*ones(1,10)  ]
t23=[0*ones(1,10) 1*ones(1,10) 0*ones(1,10) 0*ones(1,10)  0*ones(1,10)
0*ones(1,10)  0*ones(1,10) 0*ones(1,10)  0*ones(1,10) 0*ones(1,10)
0*ones(1,10) 0*ones(1,10)  0*ones(1,10) 0*ones(1,10)  1*ones(1,10)
0*ones(1,10)  ]
t24=[1*ones(1,10) 0*ones(1,10) 0*ones(1,10) 0*ones(1,10)  0*ones(1,10)
0*ones(1,10)  0*ones(1,10) 0*ones(1,10)  0*ones(1,10) 0*ones(1,10)
0*ones(1,10) 0*ones(1,10)  0*ones(1,10) 0*ones(1,10)  0*ones(1,10)
0*ones(1,10)  ]
t25=[0*ones(1,10) 0*ones(1,10) 0*ones(1,10) 0*ones(1,10)  0*ones(1,10)
0*ones(1,10)  0*ones(1,10) 0*ones(1,10)  0*ones(1,10) 0*ones(1,10)
0*ones(1,10) 0*ones(1,10)  0*ones(1,10) 0*ones(1,10)  1*ones(1,10)
0*ones(1,10)  ]
t26=[0*ones(1,10) 0*ones(1,10) 0*ones(1,10) 0*ones(1,10)  0*ones(1,10)
0*ones(1,10)  0*ones(1,10) 0*ones(1,10)  0*ones(1,10) 0*ones(1,10)
0*ones(1,10) 0*ones(1,10)  0*ones(1,10) 0*ones(1,10)  1*ones(1,10)
0*ones(1,10)  ]
t27=[0*ones(1,10) 0*ones(1,10) 0*ones(1,10) 0*ones(1,10)  0*ones(1,10)
0*ones(1,10)  0*ones(1,10) 0*ones(1,10)  0*ones(1,10) 0*ones(1,10)
0*ones(1,10) 0*ones(1,10)  0*ones(1,10) 1*ones(1,10)  1*ones(1,10)
0*ones(1,10)  ]
t= [t21 t22 t23 t24 t25 t26 t27]

for i=1:160
    lunes(i)=t(i+160*0);
    potencia= 200+20*rand(1)
    consumol(i)= potencia*t(i+160*0)

    martes(i)=t(i+160*1);
    potencia= 200+20*rand(1)
    consumom(i)= potencia*t(i+160*1)

    miercoles(i)=t(i+160*2);
     potencia= 200+20*rand(1)
     consumomi(i)= potencia*t(i+160*2)


     jueves(i)= t(i+160*3);
      potencia= 200+20*rand(1)
    consumoj(i)= potencia*t(i+160*3)

    viernes(i)= t(i+160*4);
     potencia= 200+20*rand(1)
    consumov(i)= potencia*t(i+160*4)

    sabado(i)=t(i+160*5);
```

```
     potencia= 200+20*rand(1)
    consumos(i)= potencia*t(i+160*5)

    domingo(i)=t(i+160*6);
     potencia= 200+20*rand(1)
    consumod(i)= potencia*t(i+160*6)

end
subplot(2,2, 1)
bar((1:160)/10 +7 ,consumol,'k')
axis([7 23 0 300]);
xlabel('Horas')
ylabel('Potencia (W)')
title('Rutina del día Lunes')


subplot(2,2, 2)
bar((1:160)/10 +7 ,consumom,'r')
axis([7 23 0 300]);
xlabel('Horas')
ylabel('Potencia (W)')
title('Rutina del día Martes')



subplot(2,2, 3)
bar((1:160)/10 +7 ,consumomi,'y')
axis([7 23 0 300]);
xlabel('Horas')
ylabel('Potencia (W)')
title('Rutina del día Miercoles')

subplot(2,2, 4)
bar((1:160)/10 +7 ,consumoj,'g')
axis([7 23 0 300]);
xlabel('Horas')
ylabel('Potencia (W)')
title('Rutina del día Jueves')

subplot(2,2, 1)
bar((1:160)/10 +7 ,consumov,'b')
axis([7 23 0 300]);
xlabel('Horas')
ylabel('Potencia (W)')
title('Rutina del día Viernes')

subplot(2,2, 2)
bar((1:160)/10 +7 ,consumos,'m')
axis([7 23 0 300]);
xlabel('Horas')
ylabel('Potencia (W)')
title('Rutina del día Sabado')

subplot(2,2, 3)
bar((1:160)/10 +7 ,consumod,'c')
```

```
axis([7 23 0 300]);
xlabel('Horas')
ylabel('Potencia (W)')
title('Rutina del día Domingo')

consumopir=sum(consumol) +sum(consumom) +sum(consumomi) +sum(consumoj)
+sum(consumov) +sum(consumos) +sum(consumod)
```

# ANEXO 11

## ECUACIONES DE LA RED NEURONAL

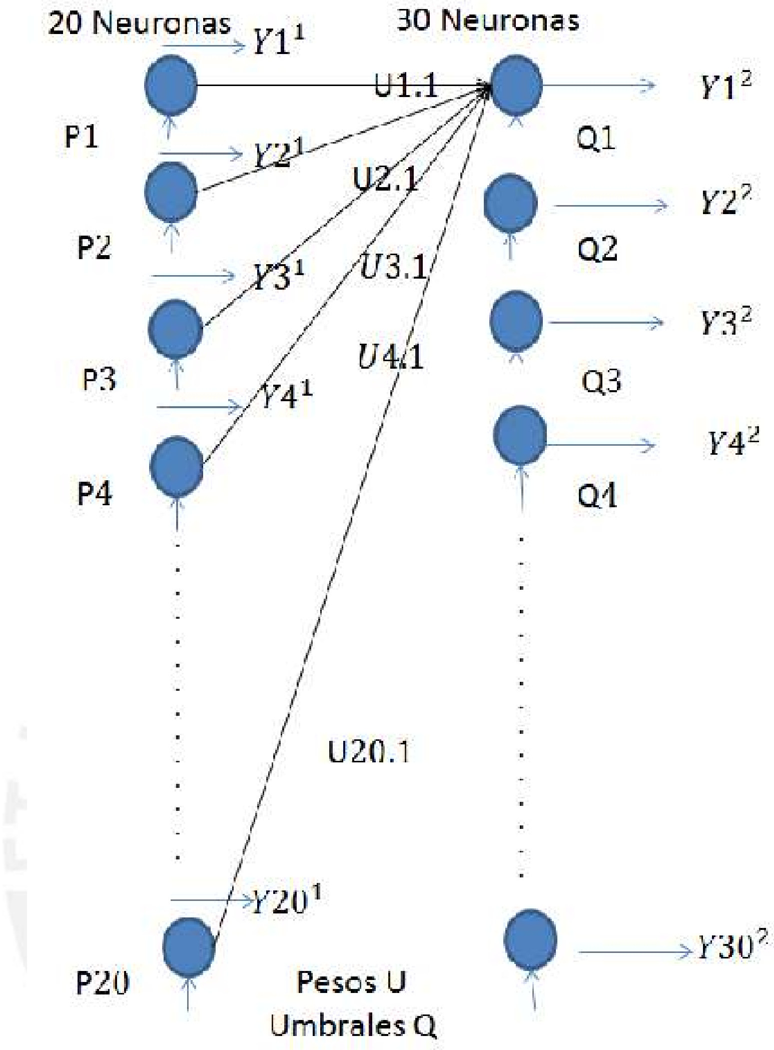
### Pesos U y umbrales Q

Luego, trabajaremos con la matriz de peso de la primera fila U la matriz de umbrales de la primera fila Q.

Reemplazando el peso V1.1 en la ecuación dos:

$$U1.1(n) = U1.1(n-1) + \boldsymbol{\alpha(n)}.\frac{\partial E(n)}{\partial U1.1(n)} \ldots\ldots\ldots\ldots.(a)$$

$$\frac{\partial E(n)}{\partial U1.1(n)} = -\big(S(n) - Y(n)\big).\frac{\partial Y(n)}{\partial U1.1} \ldots\ldots\ldots\ldots..(b)$$



Arquitectura de la segunda capar con la primera capa oculta

Realizando el mismo procedimiento

$$Y(n) = f(V1.1 * Y1^2 + V2.1 * Y2^2 + V3.1 * Y3^2 + \dots\dots\dots\dots + V30.1 * Y30^2 + R1) \dots\dots\dots (c)$$

$$Y1^2(n) = f(U1.1 * Y1^1 + U2.1 * Y2^1 + U3.1 * Y3^1 + \dots\dots\dots\dots + U20.1 * Y20^1 + Q1) \dots\dots (d)$$

Reemplazando c en b tenemos:

$$\frac{\partial E(n)}{\partial U1.1(n)} = -\big(S(n) - Y(n)\big)\frac{\partial Y(n)}{\partial Y1^2(n)}.\frac{\partial Y1^2(n)}{\partial U1.1(n)}$$

$$\frac{\partial E(n)}{\partial U1.1(n)} = -\big(S(n) - Y(n)\big)f'(V1.1 * Y1^2 + V2.1 * Y2^2 + V3.1 * Y3^2 + \dots\dots\dots\dots + V30.1 * Y30^2 + R1).V1.1(n).\frac{\partial Y1^2(n)}{\partial U1.1(n)}$$

Como definimos

$$\boldsymbol{\varphi 3(n)} = -\big(\boldsymbol{S(n)} - \boldsymbol{Y(n)}\big)\boldsymbol{f'}\big(\boldsymbol{V1.1} * \boldsymbol{Y1^2} + \boldsymbol{V2.1} * \boldsymbol{Y2^2} + \boldsymbol{V3.1} * \boldsymbol{Y3^2} + \ldots\ldots\ldots\ldots + \boldsymbol{V30.1} * \boldsymbol{Y30^2} + \boldsymbol{R1}\big)$$

$$\frac{\partial E(n)}{\partial U1.1(n)} =. \boldsymbol{\varphi 3(n)}. V1.1. f'(U1.1 * Y1^1 + U2.1 * Y2^1 + U3.1 * Y3^1 + \ldots\ldots\ldots\ldots + U20.1 * Y20^1 + Q1)Y1^1(n)$$

Definimos:

$$\boldsymbol{\varphi 2.1(n)} = \boldsymbol{\varphi 3(n)}. f'(U1.1 * Y1^1 + U2.1 * Y2^1 + U3.1 * Y3^1 + \ldots\ldots\ldots\ldots + U20.1 * Y20^1 + Q1)V1.1(n)$$

Reemplazando en a)

$$U1.1(n) = U1.1(n-1) + \boldsymbol{\alpha(n)}. \varphi 2.1(n). Y1^1(n)$$

Aplicando el mismo criterio para el resto de los pesos sinópticos de la primera fila se obtiene:

$$U2.1(n) = U1.1(n-1) + \boldsymbol{\alpha(n)}. \varphi 2.1(n). Y2^1(n)$$

$$U3.1(n) = U1.1(n-1) + \boldsymbol{\alpha(n)}. \varphi 2.1(n). Y3^1(n)$$

$$U4.1(n) = U1.1(n-1) + \boldsymbol{\alpha(n)}. \varphi 2.1(n). Y4^1(n)$$

$$\vdots \qquad\qquad \vdots \qquad\qquad \vdots \qquad \vdots \qquad \vdots$$

$$\vdots \qquad\qquad \vdots \qquad\qquad \vdots \qquad \vdots \qquad \vdots$$

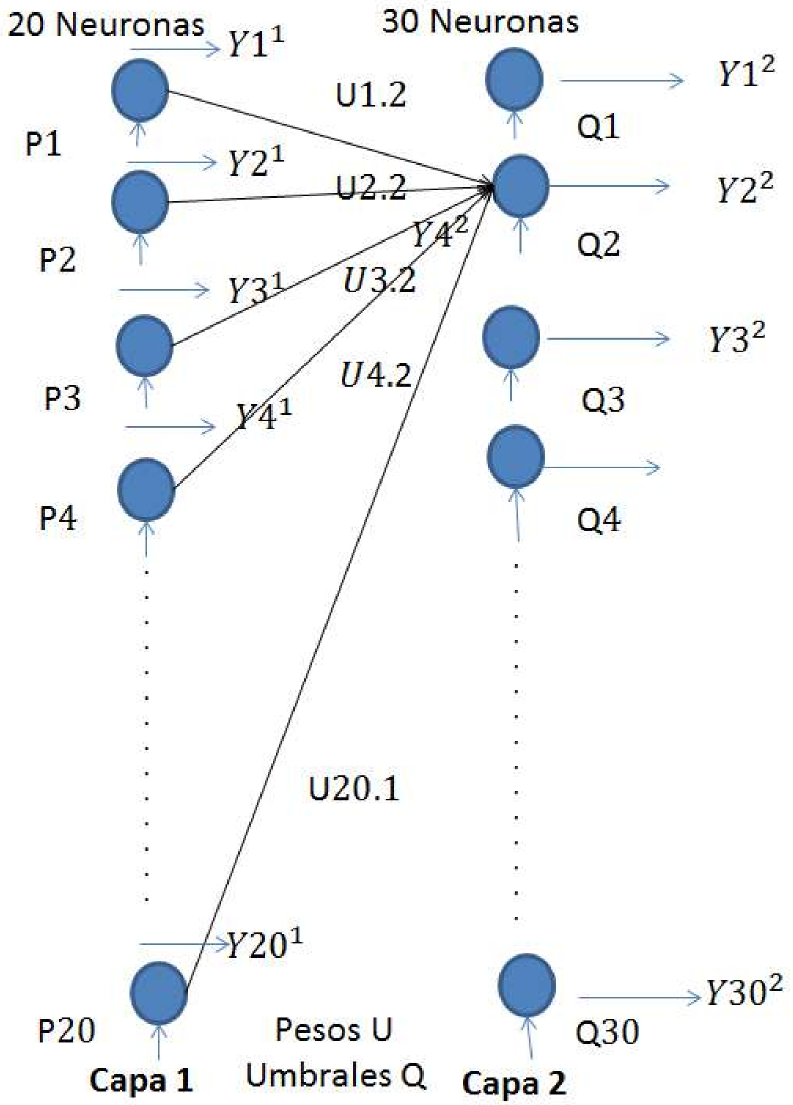
$$U20.1(n) = U20.1(n-1) + \boldsymbol{\alpha(n)}. \varphi 2.1(n). Y20^1(n)$$

De igual forma para el umbral Q1

$$\frac{\partial E(n)}{\partial Q1(n)} = -\big(S(n) - Y(n)\big). \frac{\partial Y(n)}{\partial Q1}$$

$$Q1(n) = Q1(n-1) + \boldsymbol{\alpha(n)}. \varphi 2.1(n)$$

Realizando el mismo procedimiento para la segunda fila



$$Y(n) = f(V1.1 * Y1^2 + \ V2.1 * Y2^2 + \ V3.1 * Y3^2 + \ \ldots\ldots\ldots\ldots + \ V30.1 * Y30^2 + R1) \ldots\ldots\ldots (c)$$

$$Y2^2(n) = f(U1.2 * Y1^1 + \ U2.2 * Y2^1 + \ U3.2 * Y3^1 + \ \ldots\ldots\ldots\ldots + \ U20.2 * Y20^1 + Q2) \ldots (d)$$

Reemplazando c en b tenemos:

$$\frac{\partial E(n)}{\partial U1.2(n)} = -\big(S(n) - Y(n)\big)\frac{\partial Y(n)}{\partial Y2^2(n)}.\frac{\partial Y2^2(n)}{\partial U1.2(n)}$$

$$\frac{\partial E(n)}{\partial U1.2(n)} = -\big(S(n) - Y(n)\big)f'(V1.1 * Y1^2 + \ V2.1 * Y2^2 + \ V3.1 * Y3^2 + \ \ldots\ldots\ldots\ldots + \ V30.1 * Y30^2 + R1).V2.1(n).\frac{\partial Y2^2(n)}{\partial U1.2(n)}$$

Como definimos

$$\boldsymbol{\varphi 3(n)} = -\left(\boldsymbol{S(n)} - \boldsymbol{Y(n)}\right)\boldsymbol{f'}\left(\boldsymbol{V1.1 * Y1^2} + \boldsymbol{V2.1 * Y2^2} + \boldsymbol{V3.1 * Y3^2} + \ \dots\dots\dots\dots + \boldsymbol{V30.1 * Y30^2} + \boldsymbol{R1}\right)$$

$$\frac{\partial E(n)}{\partial U1.2(n)} =. \boldsymbol{\varphi 3(n)}. V2.1. f'(U1.2 * Y1^1 + \ U2.2 * Y2^1 + \ U3.2 * Y3^1 + \ \dots\dots\dots\dots + \ U20.2 * Y20^1 + Q2)Y1^1(n)$$

Definimos:

$$\boldsymbol{\varphi 2.2(n)} = \boldsymbol{\varphi 3(n)}. f'(U1.2 * Y1^1 + \ U2.2 * Y2^1 + \ U3.2 * Y3^1 + \ \dots\dots\dots\dots + \ U20.2 * Y20^1 + Q2). V2.1(n)$$

Reemplazando en a)

$$U1.2(n) = U1.2(n-1) + \boldsymbol{\alpha(n)}. \varphi 2.2(n). Y2^1(n)$$

Aplicando el mismo criterio para el resto de los pesos sinópticos de la primera fila se obtiene:

$$U2.2(n) = U2.1(n-1) + \boldsymbol{\alpha(n)}. \varphi 2.2(n). Y2^1(n)$$

$$U3.2(n) = U3.1(n-1) + \boldsymbol{\alpha(n)}. \varphi 2.2(n). Y3^1(n)$$

$$U4.2(n) = U4.1(n-1) + \boldsymbol{\alpha(n)}. \varphi 2.2(n). Y4^1(n)$$

$$\vdots \qquad\qquad \vdots \qquad\qquad \vdots \qquad\quad \vdots \qquad \vdots$$

$$\vdots \qquad\qquad \vdots \qquad\qquad \vdots \qquad\quad \vdots \qquad \vdots$$

$$U20.2(n) = U20.1(n-1) + \boldsymbol{\alpha(n)}. \varphi 2.2(n). Y20^1(n)$$

$$Q2(n) = Q1(n-1) + \boldsymbol{\alpha(n)}. \varphi 2.2(n)$$

Generalizando

Para la tercera fila

$$U1.3(n) = U1.3(n-1) + \boldsymbol{\alpha(n)}. \varphi 2.3(n). Y1^1(n)$$

$$U2.3(n) = U2.3(n-1) + \boldsymbol{\alpha(n)}. \varphi 2.3(n). Y2^1(n)$$

$$U3.3(n) = U3.3(n-1) + \boldsymbol{\alpha(n)}. \varphi 2.3(n). Y3^1(n)$$

$U4.3(n) = U4.3(n-1) + \boldsymbol{\alpha}(\boldsymbol{n}).\varphi 2.3(n).Y4^1(n)$

$\vdots \qquad\qquad\qquad \vdots \qquad\qquad\qquad \vdots \qquad\qquad \vdots \qquad\quad \vdots$

$\vdots \qquad\qquad\qquad \vdots \qquad\qquad\qquad \vdots \qquad\qquad \vdots \qquad\quad \vdots$

$U20.3(n) = U20.3(n-1) + \boldsymbol{\alpha}(\boldsymbol{n}).\varphi 2.3(n).Y20^1(n)$

$Q3(n) = Q3(n-1) + \boldsymbol{\alpha}(\boldsymbol{n}).\varphi 2.3(n)$

$\boldsymbol{\varphi 2.3}(\boldsymbol{n}) = \boldsymbol{\varphi 3}(\boldsymbol{n}).f'(U1.3 * Y1^1 + U2.3 * Y2^1 + U3.3 * Y3^1 + \ldots\ldots\ldots\ldots + U20.3 * Y20^1 + Q3).V3.1(n)$

Para la cuarta fila

$U1.4(n) = U1.4(n-1) + \boldsymbol{\alpha}(\boldsymbol{n}).\varphi 2.4(n).Y1^1(n)$

$U2.4(n) = U2.4(n-1) + \boldsymbol{\alpha}(\boldsymbol{n}).\varphi 2.4(n).Y2^1(n)$

$U3.4(n) = U3.4(n-1) + \boldsymbol{\alpha}(\boldsymbol{n}).\varphi 2.4(n).Y3^1(n)$

$U4.4(n) = U4.4(n-1) + \boldsymbol{\alpha}(\boldsymbol{n}).\varphi 2.4(n).Y4^1(n)$

$\vdots \qquad\qquad\qquad \vdots \qquad\qquad\qquad \vdots \qquad\qquad \vdots \qquad\quad \vdots$

$\vdots \qquad\qquad\qquad \vdots \qquad\qquad\qquad \vdots \qquad\qquad \vdots \qquad\quad \vdots$

$U20.4(n) = U20.4(n-1) + \boldsymbol{\alpha}(\boldsymbol{n}).\varphi 2.4(n).Y20^1(n)$

$Q4(n) = Q4(n-1) + \boldsymbol{\alpha}(\boldsymbol{n}).\varphi 2.4(n)$

$\boldsymbol{\varphi 2.4}(\boldsymbol{n}) = \boldsymbol{\varphi 3}(\boldsymbol{n}).f'(U1.4 * Y1^1 + U2.4 * Y2^1 + U3.4 * Y3^1 + \ldots\ldots\ldots\ldots + U20.4 * Y20^1 + Q4).V4.1(n)$

Para la trigésima fila

$U1.30(n) = U1.30(n-1) + \boldsymbol{\alpha}(\boldsymbol{n}).\varphi 2.30(n).Y1^1(n)$

$U2.30(n) = U1.30(n-1) + \boldsymbol{\alpha}(\boldsymbol{n}).\varphi 2.30(n).Y2^1(n)$

$U3.30(n) = U1.30(n-1) + \boldsymbol{\alpha}(\boldsymbol{n}).\varphi 2.30(n).Y3^1(n)$

$U4.30(n) = U1.30(n-1) + \boldsymbol{\alpha}(\boldsymbol{n}).\varphi 2.30(n).Y4^1(n)$

⋮ ⋮ ⋮ ⋮ ⋮

⋮ ⋮ ⋮ ⋮ ⋮

$$U20.4(n) = U20.4(n-1) + \boldsymbol{\alpha}(\boldsymbol{n}).\varphi 2.30(n).Y20^1(n)$$

$$Q4(n) = Q4(n-1) + \boldsymbol{\alpha}(\boldsymbol{n}).\varphi 2.30(n)$$

$$\boldsymbol{\varphi 2.30}(\boldsymbol{n}) = \boldsymbol{\varphi 3}(\boldsymbol{n}).f'(U1.30 * Y1^1 + \ U2.30 * Y2^1 + \ U3.30 * Y3^1 + \ \ldots\ldots + \ U20.30 * Y20^1 + Q4).V30.1(n)$$

**<u>Pesos W y umbrales P</u>**

Luego, trabajaremos con la matriz de peso de la primera fila W la matriz de umbrales de la primera fila P.

Reemplazando el peso W1.1 en la ecuación dos:

$$W1.1(n) = W1.1(n-1) + \boldsymbol{\alpha}(\boldsymbol{n}).\frac{\partial E(n)}{\partial W1.1(n)} \ldots\ldots\ldots\ldots.(a)$$

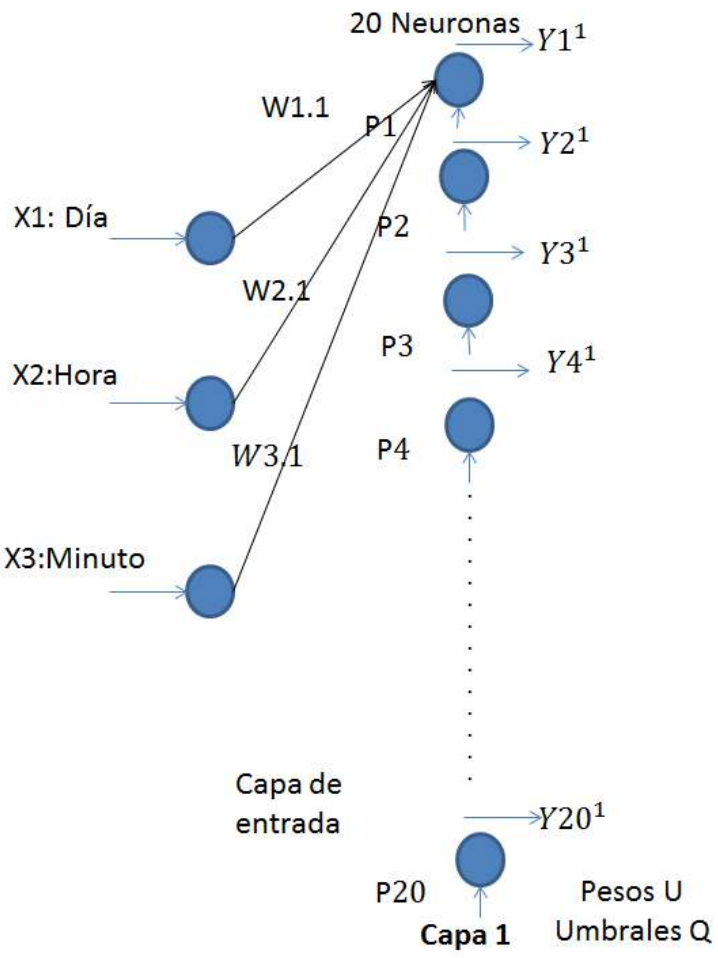
$$\frac{\partial E(n)}{\partial W1.1(n)} = -\big(S(n) - Y(n)\big)\frac{\partial Y(n)}{\partial Y1^2(n)}.\frac{\partial Y1^2(n)}{\partial Y1^1(n)}.\frac{\partial Y1^1(n)}{\partial W1.1(n)}$$

Como ya se definió:

$$Y(n) = f(V1.1 * Y1^2 + \ V2.1 * Y2^2 + \ V3.1 * Y3^2 + \ \ldots\ldots\ldots\ldots + \ V30.1 * Y30^2 + R1) \ldots\ldots\ldots(c)$$

$$Y1^2(n) = f(U1.1 * Y1^1 + \ U2.1 * Y2^1 + \ U3.1 * Y3^1 + \ \ldots\ldots\ldots\ldots + \ U20.1 * Y20^1 + Q1) \ldots(d)$$

$$Y1^1(n) = f(W1.1 * X1 + W2.1 * X2 + W3.1 * X3 + P1) \ldots(e)$$



$$\frac{\partial E(n)}{\partial W1.1(n)} = -\big(S(n) - Y(n)\big)\frac{\partial Y(n)}{\partial Y1^2(n)}.\frac{\partial Y1^2(n)}{\partial Y1(n)}.\frac{\partial Y1(n)}{\partial W1.1(n)}$$

$$\frac{\partial E(n)}{\partial W1.1(n)} = \varphi 3(n).V1.1.U1.1.f'(U1.1 * Y1^1 + \ U2.1 * Y2^1 + \ U3.1 * Y3^1 + \ \ldots\ldots\ldots\ldots + \ U20.1 * Y20^1 + Q1).\frac{\partial Y1^1(n)}{\partial W1.1(n)}$$

Como definimos

$$\boldsymbol{\varphi 2.1(n)} = \boldsymbol{\varphi 3(n)}.f'(U1.1 * Y1^1 + \ U2.1 * Y2^1 + \ U3.1 * Y3^1 + \ \ldots\ldots\ldots\ldots + \ U20.1 * Y20^1 + Q1)V1.1(n)$$

$$\frac{\partial E(n)}{\partial W1.1(n)} = \varphi 2.1(n).U1.1.\frac{\partial Y1^1(n)}{\partial W1.1(n)}$$

$$\frac{\partial E(n)}{\partial W1.1(n)} = \varphi 2.1(n).U1.1.f'(W1.1 * X1 + W2.1 * X2 + W3.1 * X3 + P1).X1$$

Definimos:

$$\varphi 1.1(n) = \varphi 2.1(n). U1.1. f'(W1.1 * X1 + W2.1 * X2 + W3.1 * X3 + P1)$$

$$\frac{\partial E(n)}{\partial W1.1(n)} = \varphi 1.1(n). X1$$

$$W1.1(n) = W1.1(n-1) + \boldsymbol{\alpha}(\boldsymbol{n}). \varphi 1.1(n). X1$$

Aplicando el mismo criterio para el resto de los pesos sinópticos se obtiene:

$$W2.1(n) = W2.1(n-1) + \boldsymbol{\alpha}(\boldsymbol{n}). \varphi 1.1(n). X2$$

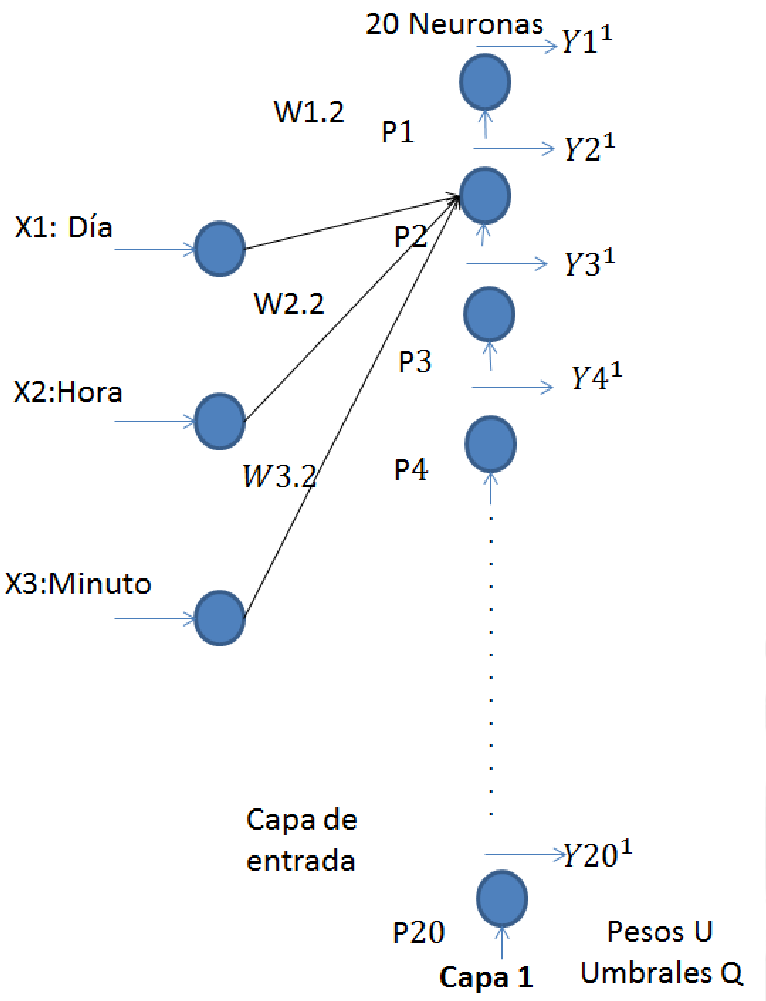
$$W3.1(n) = W3.1(n-1) + \boldsymbol{\alpha}(\boldsymbol{n}). \varphi 1.1(n). X3$$

De igual forma para el umbral Q1

$$\frac{\partial E(n)}{\partial P1(n)} = -\big(S(n) - Y(n)\big). \frac{\partial Y(n)}{\partial P1}$$

$$P1(n) = P1(n-1) + \boldsymbol{\alpha}(\boldsymbol{n}). \varphi 1.1(n)$$

Realizando el mismo procedimiento para la segunda fila



$$W1.2(n) = W1.2(n-1) + \boldsymbol{\alpha}(\boldsymbol{n}).\varphi 1.2(n).X1$$

$$W1.2(n) = W1.2(n-1) + \boldsymbol{\alpha}(\boldsymbol{n}).\frac{\partial E(n)}{\partial W1.2(n)} \ldots\ldots\ldots\ldots.(a)$$

$$\frac{\partial E(n)}{\partial W1.2(n)} = -\big(S(n) - Y(n)\big)\frac{\partial Y(n)}{\partial Y2^2(n)}.\frac{\partial Y2^2(n)}{\partial Y2^1(n)}.\frac{\partial Y2^1(n)}{\partial W1.2(n)}$$

Como ya se definió:

$$Y(n) = f(V1.1 * Y1^2 +\ V2.1 * Y2^2 +\ V3.1 * Y3^2 +\ \ldots\ldots\ldots\ldots +\ V30.1 * Y30^2 + R1) \ldots\ldots\ldots(c)$$

$$Y2^2(n) = f(U1.2 * Y1^1 +\ U2.2 * Y2^1 +\ U3.2 * Y3^1 +\ \ldots\ldots\ldots\ldots +\ U20.2 * Y20^1 + Q2) \ldots(d)$$

$$Y2^1(n) = f(W1.2 * X1 + W2.2 * X2 + W3.2 * X3 + P2) \ldots(e)$$

$$\frac{\partial E(n)}{\partial W1.2(n)} = \varphi 3(n).V2.1.U2.2.f'(U1.2 * Y1^1 + U2.2 * Y2^1 + U3.2 * Y3^1 + \ldots\ldots\ldots\ldots + U20.2 * Y20^1 + Q2).\frac{\partial Y2^1(n)}{\partial W2.1(n)}$$

Como definimos

$$\boldsymbol{\varphi 2.2(n)} = \boldsymbol{\varphi 3(n)}.f'(U1.2 * Y1^1 + U2.2 * Y2^1 + U3.2 * Y3^1 + \ldots\ldots\ldots\ldots + U20.2 * Y20^1 + Q2).V2.1(n)$$

$$\frac{\partial E(n)}{\partial W1.2(n)} = \varphi 2.2(n).U2.2.\frac{\partial Y2^1(n)}{\partial W1.2(n)}$$

$$\frac{\partial E(n)}{\partial W1.2(n)} = \varphi 2.2(n).U2.2.f'(W1.2 * X1 + W2.2 * X2 + W3.2 * X3 + P2).X1$$

Definimos:

$$\varphi 1.2(n) = \varphi 2.2(n).U2.2.f'(W1.2 * X1 + W2.2 * X2 + W3.2 * X3 + P2)$$

$$\frac{\partial E(n)}{\partial W1.2(n)} = \varphi 1.2(n).X1$$

$$W1.2(n) = W1.2(n-1) + \boldsymbol{\alpha(n)}.\varphi 1.2(n).X1$$

Aplicando el mismo criterio para el resto de los pesos sinópticos se obtiene:

$$W2.2(n) = W2.2(n-1) + \boldsymbol{\alpha(n)}.\varphi 1.2(n).X2$$

$$W3.2(n) = W3.2(n-1) + \boldsymbol{\alpha(n)}.\varphi 1.2(n).X3$$

De igual forma para el umbral Q1

$$\frac{\partial E(n)}{\partial P1(n)} = -\big(S(n) - Y(n)\big).\frac{\partial Y(n)}{\partial P1}$$

$$P2(n) = P1(n-1) + \boldsymbol{\alpha(n)}.\varphi 1.2(n)$$

Generalizando:

Para la tercera fila

$W1.3(n) = W1.3(n-1) + \boldsymbol{\alpha}(\boldsymbol{n}).\varphi1.3(n).X1$

$W2.3(n) = W2.3(n-1) + \boldsymbol{\alpha}(\boldsymbol{n}).\varphi1.3(n).X2$

$W3.3(n) = W3.3(n-1) + \boldsymbol{\alpha}(\boldsymbol{n}).\varphi1.3(n).X3$

$P3(n) = P3(n-1) + \boldsymbol{\alpha}(\boldsymbol{n}).\varphi1.3(n)$

$\varphi1.3(n) = \varphi2.3(n).U3.3.f'(W1.3 * X1 + W2.3 + X2 + W3.3 * X3 + P3)$

Para la vigésima fial

$W1.20(n) = W1.20(n-1) + \boldsymbol{\alpha}(\boldsymbol{n}).\varphi1.20(n).X1$

$W2.20(n) = W2.20(n-1) + \boldsymbol{\alpha}(\boldsymbol{n}).\varphi1.20(n).X2$

$W3.20(n) = W3.20(n-1) + \boldsymbol{\alpha}(\boldsymbol{n}).\varphi1.20(n).X3$

$P20(n) = P20(n-1) + \boldsymbol{\alpha}(\boldsymbol{n}).\varphi1.20(n)$

$$\varphi1.20(n) = \varphi2.20(n).U20.20.f'(W1.20 * X1 + W2.20 + X2 + W3.20 * X3 + P20)$$

# Neural Network Toolbox

## For Use with MATLAB®

*Howard Demuth*
*Mark Beale*

- Computation
- Visualization
- Programming

User's Guide

*Version 4*

The MathWorks

**How to Contact The MathWorks:**

| | |
|---|---|
| `www.mathworks.com` | Web |
| `comp.soft-sys.matlab` | Newsgroup |
| `support@mathworks.com` | Technical support |
| `suggest@mathworks.com` | Product enhancement suggestions |
| `bugs@mathworks.com` | Bug reports |
| `doc@mathworks.com` | Documentation error reports |
| `service@mathworks.com` | Order status, license renewals, passcodes |
| `info@mathworks.com` | Sales, pricing, and general information |
| 508-647-7000 | Phone |
| 508-647-7001 | Fax |
| The MathWorks, Inc.<br>3 Apple Hill Drive<br>Natick, MA 01760-2098 | Mail |

For contact information about worldwide offices, see the MathWorks Web site.

*Neural Network Toolbox User's Guide*

History:

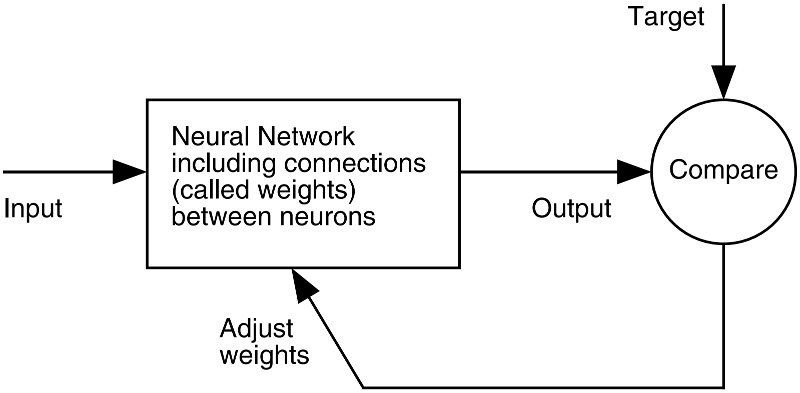
| | | |
|---|---|---|
| June 1992 | First printing | |
| April 1993 | Second printing | |
| January 1997 | Third printing | |
| July 1997 | Fourth printing | |
| January 1998 | Fifth printing | Revised for Version 3 (Release 11) |
| September 2000 | Sixth printing | Revised for Version 4 (Release 12) |
| June 2001 | Seventh printing | Minor revisions (Release 12.1) |
| July 2002 | Online only | Minor revisions (Release 13) |
| January 2003 | Online only | Minor revisions (Release 13SP1) |
| June 2004 | Online only | Revised for Release 14 |
| October 2004 | Online only | Revised for Version 4.0.4 (Release 14SP1) |

# Preface

# Neural Networks

Neural networks are composed of simple elements operating in parallel. These elements are inspired by biological nervous systems. As in nature, the network function is determined largely by the connections between elements. We can train a neural network to perform a particular function by adjusting the values of the connections (weights) between elements.

Commonly neural networks are adjusted, or trained, so that a particular input leads to a specific target output. Such a situation is shown below. There, the network is adjusted, based on a comparison of the output and the target, until the network output matches the target. Typically many such input/target pairs are used, in this *supervised learning*, to train a network.



Batch training of a network proceeds by making weight and bias changes based on an entire set (batch) of input vectors. Incremental training changes the weights and biases of a network as needed after presentation of each individual input vector. Incremental training is sometimes referred to as “on line” or “adaptive” training.

Neural networks have been trained to perform complex functions in various fields of application including pattern recognition, identification, classification, speech, vision and control systems. A list of applications is given in Chapter 1.

Today neural networks can be trained to solve problems that are difficult for conventional computers or human beings. Throughout the toolbox emphasis is placed on neural network paradigms that build up to or are themselves used in engineering, financial and other practical applications.

The supervised training methods are commonly used, but other networks can be obtained from *unsupervised training* techniques or from direct *design* methods. Unsupervised networks can be used, for instance, to identify groups of data. Certain kinds of linear networks and Hopfield networks are designed directly. In summary, there are a variety of kinds of design and learning techniques that enrich the choices that a user can make.

The field of neural networks has a history of some five decades but has found solid application only in the past fifteen years, and the field is still developing rapidly. Thus, it is distinctly different from the fields of control systems or optimization where the terminology, basic mathematics, and design procedures have been firmly established and applied for many years. We do not view the Neural Network Toolbox as simply a summary of established procedures that are known to work well. Rather, we hope that it will be a useful tool for industry, education and research, a tool that will help users find what works and what doesn't, and a tool that will help develop and extend the field of neural networks. Because the field and the material are so new, this toolbox will explain the procedures, tell how to apply them, and illustrate their successes and failures with examples. We believe that an understanding of the paradigms and their application is essential to the satisfactory and successful use of this toolbox, and that without such understanding user complaints and inquiries would bury us. So please be patient if we include a lot of explanatory material. We hope that such material will be helpful to you.

## Basic Chapters

The Neural Network Toolbox is written so that if you read Chapter 2, Chapter 3 and Chapter 4 you can proceed to a later chapter, read it and use its functions without difficulty. To make this possible, Chapter 2 presents the fundamentals of the neuron model, the architectures of neural networks. It also will discuss notation used in the architectures. All of this is basic material. It is to your advantage to understand this Chapter 2 material thoroughly.

The neuron model and the architecture of a neural network describe how a network transforms its input into an output. This transformation can be viewed as a computation. The model and the architecture each place limitations on what a particular neural network can compute. The way a network computes its output must be understood before training methods for the network can be explained.

# Mathematical Notation for Equations and Figures

## Basic Concepts

Scalars-small *italic* letters.....*a,b,c*

Vectors - small **bold** non-italic letters.....**a,b,c**

Matrices - capital **BOLD** non-italic letters.....**A,B,C**

## Language

*Vector* means a column of numbers.

## Weight Matrices

### Scalar Element $w_{i,j}(t)$

$i$ - row, $j$ - column, $t$ - time or iteration

### Matrix $\mathbf{W}(t)$

### Column Vector $\mathbf{w}_j(t)$

**Row Vector** ${}_i\mathbf{w}(t)$ **...**vector made of ith row of weight matrix **W**

### Bias Vector

### Scalar Element $b_i(t)$

### Vector $\mathbf{b}(t)$

## Layer Notation

A single superscript is used to identify elements of layer. For instance, the net input of layer 3 would be shown as $\mathbf{n}^3$.

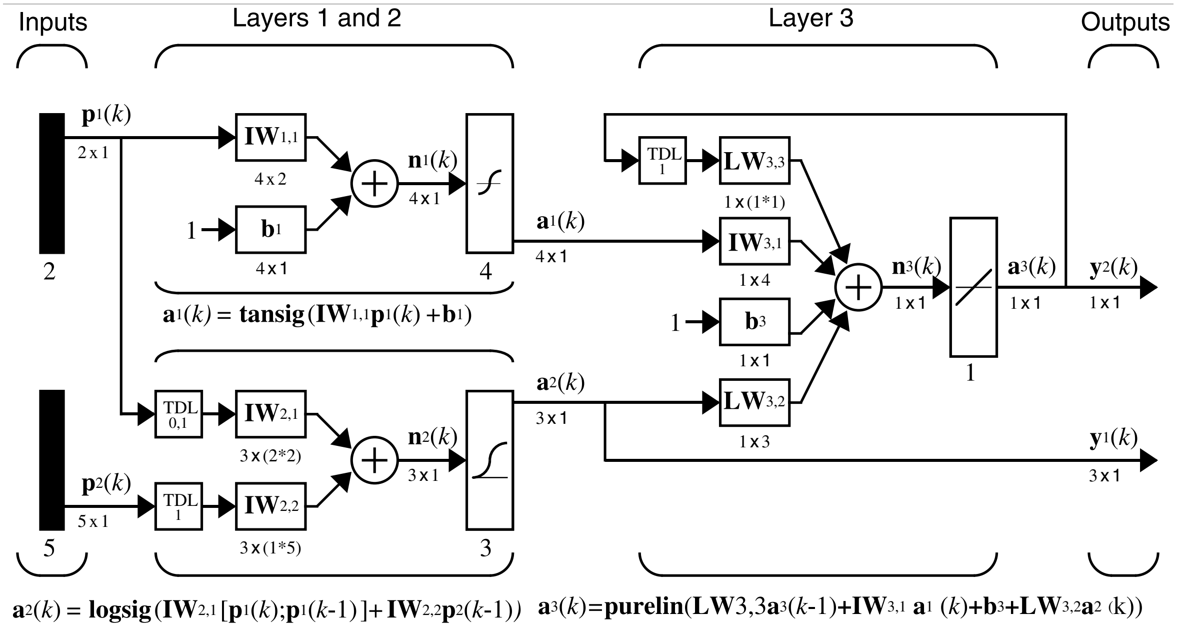
Superscripts $k, l$ are used to identify the source (l) connection and the destination (k) connection of layer weight matrices ans input weight matrices. For instance, the layer weight matrix from layer 2 to layer 4 would be shown as $\mathbf{LW}^{4,2}$.

### Input Weight Matrix $\mathbf{IW}^{k,l}$

### Layer Weight Matrix $\mathbf{LW}^{k,l}$

## Figure and Equation Examples

The following figure, taken from Chapter 12 illustrates notation used in such advanced figures.



# Mathematics and Code Equivalents

The transition from mathematics to code or vice versa can be made with the aid of a few rules. They are listed here for future reference.

To change from mathematics notation to MATLAB® notation, the user needs to:

- Change superscripts to cell array indices.

  For example, $p^{1} \rightarrow p\{1\}$
- Change subscripts to parentheses indices.

  For example, $p_2 \rightarrow p(2)$, and $p_2^1 \rightarrow p\{1\}(2)$
- Change parentheses indices to a second cell array index.

  For example, $p^{1}(k-1) \rightarrow p\{1, k-1\}$
- Change mathematics operators to MATLAB operators and toolbox functions.

  For example, $ab \rightarrow a$*b

The following equations illustrate the notation used in figures.

$$n = w_{1,1}p_1 + w_{1,2}p_2 + \ldots + w_{1,R}p_R + b$$

$$\mathbf{W} = \begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,R} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,R} \\ & & & \\ w_{S,1} & w_{S,2} & \cdots & w_{S,R} \end{bmatrix}$$

## Neural Network Design Book

Professor Martin Hagan of Oklahoma State University, and Neural Network Toolbox authors Howard Demuth and Mark Beale have written a textbook, *Neural Network Design* (ISBN 0-9717321-0-8). The book presents the theory of neural networks, discusses their design and application, and makes considerable use of MATLAB and the Neural Network Toolbox. Demonstration programs from the book are used in various chapters of this Guide. (You can find all the book demonstration programs in the Neural Network Toolbox by typing `nnd`.)

The book has:

- An INSTRUCTOR'S MANUAL for adopters and
- TRANSPARENCY OVERHEADS for class use.

This book can be obtained from the University of Colorado Bookstore at 1-303-492-3648 or at the online purchase web site, cubooks.colorado.edu.

To obtain a copy of the INSTRUCTOR'S MANUAL contact the University of Colorado Bookstore phone 1-303-492-3648. Ask specifically for an instructor's manual if you are instructing a class and want one.

You can go directly to the Neural Network Design page at

http://ee.okstate.edu/mhagan/nnd.html

Once there, you can download the TRANSPARENCY MASTERS with a click on “Transparency Masters(3.6MB)”.

You can get the Transparency Masters in Powerpoint or PDF format. You can obtain sample book chapters in PDF format as well.

# Acknowledgments

The authors would like to thank:

**Martin Hagan** of Oklahoma State University for providing the original Levenberg-Marquardt algorithm in the Neural Network Toolbox version 2.0 and various algorithms found in version 3.0, including the new reduced memory use version of the Levenberg-Marquardt algorithm, the conjugate gradient algorithm, RPROP, and generalized regression method. Martin also wrote Chapter 5 and Chapter 6 of this toolbox. Chapter 5 on Chapter  describes new algorithms, suggests algorithms for pre- and post-processing of data, and presents a comparison of the efficacy of various algorithms. Chapter 6 on control system applications describes practical applications including neural network model predictive control, model reference adaptive control, and a feedback linearization controller.

**Joe Hicklin** of The MathWorks for getting Howard into neural network research years ago at the University of Idaho, for encouraging Howard to write the toolbox, for providing crucial help in getting the first toolbox version 1.0 out the door, and for continuing to be a good friend.

**Jim Tung** of The MathWorks for his long-term support for this project.

**Liz Callanan** of The MathWorks for getting us off the such a good start with the Neural Network Toolbox version 1.0.

**Roy Lurie** of The MathWorks for his vigilant reviews of the developing material in this version of the toolbox.

**Matthew Simoneau** of The MathWorks for his help with demos, test suite routines, for getting user feedback, and for helping with other toolbox matters.

**Sean McCarthy** for his many questions from users about the toolbox operation

**Jane Carmody** of The MathWorks for editing help and for always being at her phone to help with documentation problems.

**Donna Sullivan** and **Peg Theriault** of The MathWorks for their editing and other help with the Mac document.

**Jane Price** of The MathWorks for getting constructive user feedback on the toolbox document and its Graphical User's Interface.

Orlando De Jesús of Oklahoma State University for his excellent work in programming the neural network controllers described in Chapter 6.

**Bernice Hewitt** for her wise New Zealand counsel, encouragement, and tea, and for the company of her cats **Tiny** and **Mr. Britches**.

**Joan Pilgram** for her business help, general support, and good cheer.

**Teri Beale** for running the show and having Valerie and Asia Danielle while Mark worked on this toolbox.

**Martin Hagan and Howard Demuth** for permission to include various problems, demonstrations, and other material from *Neural Network Design*, Jan. 1996.

# Contents

## Preface

## 1 Introduction

## 2 Neuron Model and Network Architectures

## 3 Perceptrons

## 4 Linear Filters

## 5 Backpropagation

## 6 Control Systems

## 9 Recurrent Networks

## 10 Adaptive Filters and Adaptive Training

## 11 Applications

## 12 Advanced Topics

## 13 Network Object Reference

# 14 Reference

## A Glossary

## B Bibliography

## C Demonstrations and Applications

## D Simulink

## E Code Notes

# 1

# Introduction

# Getting Started

## Basic Chapters

Chapter 2 contains basic material about network architectures and notation specific to this toolbox.Chapter 3 includes the first reference to basic functions such as `init` and `adapt`. Chapter 4 describes the use of the functions `designd` and `train`, and discusses delays. Chapter 2, Chapter 3, and Chapter 4 should be read before going to later chapters

## Help and Installation

The Neural Network Toolbox is contained in a directory called `nnet`. Type `help nnet` for a listing of help topics.

A number of demonstrations are included in the toolbox. Each example states a problem, shows the network used to solve the problem, and presents the final results. Lists of the neural network demonstration and application scripts that are discussed in this guide can be found by typing `help nndemos`

Instructions for installing the Neural Network Toolbox are found in one of two MATLAB® documents: the *Installation Guide for PC* or the *Installation Guide for UNIX*.

# What’s New in Version 4.0

A few of the new features and improvements introduced with this version of the Neural Network Toolbox are discussed below.

## Control System Applications

A new Chapter 6 presents three practical control systems applications:

- Network model predictive control
- Model reference adaptive control
- Feedback linearization controller

## Graphical User Interface

A graphical user interface has been added to the toolbox. This interface allows you to:

- Create networks
- Enter data into the GUI
- Initialize, train, and simulate networks
- Export the training results from the GUI to the command line workspace
- Import data from the command line workspace to the GUI

To open the Network/Data Manager window type `nntool`.

## New Training Functions

The toolbox now has four training algorithms that apply weight and bias learning rules. One algorithm applies the learning rules in batch mode. Three algorithms apply learning rules in three different incremental modes:

- `trainb` - Batch training function
- `trainc` - Cyclical order incremental training function
- `trainr` - Random order incremental training function
- `trains` - Sequential order incremental training function

All four functions present the whole training set in each epoch (pass through the entire input set).

**Note** We no longer recommend using trainwb and trainwb1, which have been replaced by trainb and trainr. The function trainr differs from trainwb1 in that trainwb1 only presented a single vector each epoch instead of going through all vectors, as is done by trainr.

These new training functions are relatively fast because they generate M-code. The functions trainb, trainc, trainr, and trains all generate a temporary M-file consisting of specialized code for training the current network in question.

## Design of General Linear Networks

The function newlind now allows the design of linear networks with multiple inputs, outputs, and input delays.

## Improved Early Stopping

Early stopping can now be used in combination with Bayesian regularization. In some cases this can improve the generalization capability of the trained network.

## Generalization and Speed Benchmarks

Generalization benchmarks comparing the performance of Bayesian regularization and early stopping are provided. We also include speed benchmarks, which compare the speed of convergence of the various training algorithms on a variety of problems in pattern recognition and function approximation. These benchmarks can aid users in selecting the appropriate algorithm for their problem.

## Demonstration of a Sample Training Session

A new demonstration that illustrates a sample training session is included in Chapter 5. A sample training session script is also provided. Users can modify this script to fit their problem.

# Neural Network Applications

## Applications in this Toolbox

Chapter 6 describes three practical neural network control system applications, including neural network model predictive control, model reference adaptive control, and a feedback linearization controller.

Other neural network applications are described in Chapter 11.

## Business Applications

The *1988 DARPA Neural Network Study* [DARP88] lists various neural network applications, beginning in about 1984 with the adaptive channel equalizer. This device, which is an outstanding commercial success, is a single-neuron network used in long-distance telephone systems to stabilize voice signals. The *DARPA* report goes on to list other commercial applications, including a small word recognizer, a process monitor, a sonar classifier, and a risk analysis system.

Neural networks have been applied in many other fields since the *DARPA* report was written. A list of some applications mentioned in the literature follows.

## Aerospace

- High performance aircraft autopilot, flight path simulation, aircraft control systems, autopilot enhancements, aircraft component simulation, aircraft component fault detection

## Automotive

- Automobile automatic guidance system, warranty activity analysis

## Banking

- Check and other document reading, credit application evaluation

## Credit Card Activity Checking

- Neural networks are used to spot unusual credit card activity that might possibly be associated with loss of a credit card

### Defense

- Weapon steering, target tracking, object discrimination, facial recognition, new kinds of sensors, sonar, radar and image signal processing including data compression, feature extraction and noise suppression, signal/image identification

### Electronics

- Code sequence prediction, integrated circuit chip layout, process control, chip failure analysis, machine vision, voice synthesis, nonlinear modeling

### Entertainment

- Animation, special effects, market forecasting

### Financial

- Real estate appraisal, loan advisor, mortgage screening, corporate bond rating, credit-line use analysis, portfolio trading program, corporate financial analysis, currency price prediction

### Industrial

- Neural networks are being trained to predict the output gasses of furnaces and other industrial processes. They then replace complex and costly equipment used for this purpose in the past.

### Insurance

- Policy application evaluation, product optimization

### Manufacturing

- Manufacturing process control, product design and analysis, process and machine diagnosis, real-time particle identification, visual quality inspection systems, beer testing, welding quality analysis, paper quality prediction, computer-chip quality analysis, analysis of grinding operations, chemical product design analysis, machine maintenance analysis, project bidding, planning and management, dynamic modeling of chemical process system

## Medical

- Breast cancer cell analysis, EEG and ECG analysis, prosthesis design, optimization of transplant times, hospital expense reduction, hospital quality improvement, emergency-room test advisement

## Oil and Gas

- Exploration

## Robotics

- Trajectory control, forklift robot, manipulator controllers, vision systems

## Speech

- Speech recognition, speech compression, vowel classification, text-to-speech synthesis

## Securities

- Market analysis, automatic bond rating, stock trading advisory systems

## Telecommunications

- Image and data compression, automated information services, real-time translation of spoken language, customer payment processing systems

## Transportation

- Truck brake diagnosis systems, vehicle scheduling, routing systems

## Summary

The list of additional neural network applications, the money that has been invested in neural network software and hardware, and the depth and breadth of interest in these devices have been growing rapidly. The authors hope that this toolbox will be useful for neural network educational and design purposes within a broad field of neural network applications.

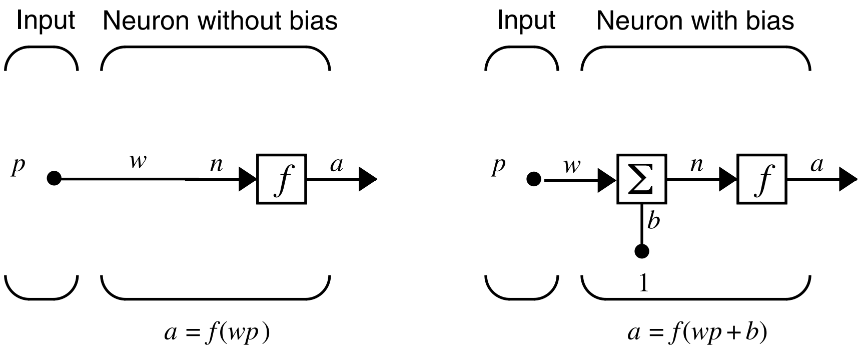
2

# Neuron Model and Network Architectures

# Neuron Model

## Simple Neuron

A neuron with a single scalar input and no bias appears on the left below.



The scalar input $p$ is transmitted through a connection that multiplies its strength by the scalar weight $w$, to form the product $wp$, again a scalar. Here the weighted input $wp$ is the only argument of the transfer function $f$, which produces the scalar output $a$. The neuron on the right has a scalar bias, $b$. You may view the bias as simply being added to the product $wp$ as shown by the summing junction or as shifting the function $f$ to the left by an amount $b$. The bias is much like a weight, except that it has a constant input of 1.

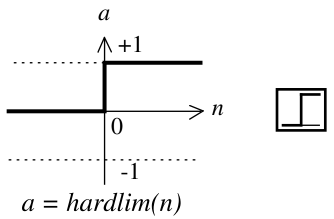
The transfer function net input $n$, again a scalar, is the sum of the weighted input $wp$ and the bias $b$. This sum is the argument of the transfer function $f$. (Chapter 7, “Radial Basis Networks” discusses a different way to form the net input $n$.) Here $f$ is a transfer function, typically a step function or a sigmoid function, which takes the argument $n$ and produces the output $a$. Examples of various transfer functions are given in the next section. Note that $w$ and $b$ are both *adjustable* scalar parameters of the neuron. The central idea of neural networks is that such parameters can be adjusted so that the network exhibits some desired or interesting behavior. Thus, we can train the network to do a particular job by adjusting the weight or bias parameters, or perhaps the network itself will adjust these parameters to achieve some desired end.

All of the neurons in this toolbox have provision for a bias, and a bias is used in many of our examples and will be assumed in most of this toolbox. However, you may omit a bias in a neuron if you want.

As previously noted, the bias $b$ is an adjustable (scalar) parameter of the neuron. It is *not* an input. However, the constant *1* that drives the bias is an input and must be treated as such when considering the linear dependence of input vectors in Chapter 4, "Linear Filters."

## Transfer Functions

Many transfer functions are included in this toolbox. A complete list of them can be found in "Transfer Function Graphs" on page 14-14. Three of the most commonly used functions are shown below.



Hard-Limit Transfer Function

The hard-limit transfer function shown above limits the output of the neuron to either 0, if the net input argument $n$ is less than 0; or 1, if $n$ is greater than or equal to 0. We will use this function in Chapter 3 "Perceptrons" to create neurons that make classification decisions.

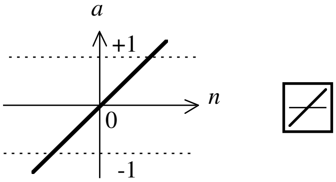
The toolbox has a function, `hardlim`, to realize the mathematical hard-limit transfer function shown above. Try the code shown below.

```
n = -5:0.1:5;
plot(n,hardlim(n),'c+:');
```

It produces a plot of the function `hardlim` over the range -5 to +5.

All of the mathematical transfer functions in the toolbox can be realized with a function having the same name.

The linear transfer function is shown below.

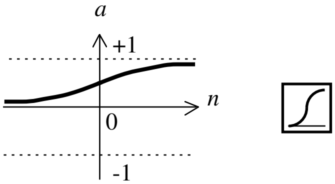


*a = purelin(n)*

Linear Transfer Function

Neurons of this type are used as linear approximators in “Linear Filters” on page 4-1.

The sigmoid transfer function shown below takes the input, which may have any value between plus and minus infinity, and squashes the output into the range 0 to 1.



*a = logsig(n)*

Log-Sigmoid Transfer Function

This transfer function is commonly used in backpropagation networks, in part because it is differentiable.

The symbol in the square to the right of each transfer function graph shown above represents the associated transfer function. These icons will replace the general *f* in the boxes of network diagrams to show the particular transfer function being used.

For a complete listing of transfer functions and their icons, see the “Transfer Function Graphs” on page 14-14. You can also specify your own transfer functions. You are not limited to the transfer functions listed in Chapter 14, “Reference.”

You can experiment with a simple neuron and various transfer functions by running the demonstration program nnd2n1.

## Neuron with Vector Input

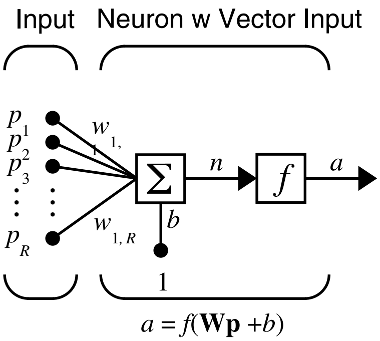
A neuron with a single R-element input vector is shown below. Here the individual element inputs

$$p_1, p_2, \dots p_R$$

are multiplied by weights

$$w_{1,1}, w_{1,2}, \dots w_{1,R}$$

and the weighted values are fed to the summing junction. Their sum is simply **Wp**, the dot product of the (single row) matrix **W** and the vector **p**.



Where...

*R* = number of elements in input vector

The neuron has a bias *b*, which is summed with the weighted inputs to form the net input $n$. This sum, *n*, is the argument of the transfer function *f*.

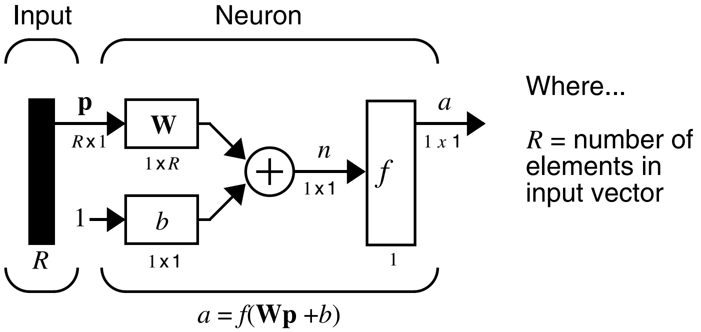
$$n = w_{1,1}p_1 + w_{1,2}p_2 + \dots + w_{1,R}p_R + b$$

This expression can, of course, be written in MATLAB® code as:

```
n = W*p + b
```

However, the user will seldom be writing code at this low level, for such code is already built into functions to define and simulate entire networks.

The figure of a single neuron shown above contains a lot of detail. When we consider networks with many neurons and perhaps layers of many neurons, there is so much detail that the main thoughts tend to be lost. Thus, the authors have devised an abbreviated notation for an individual neuron. This notation, which will be used later in circuits of multiple neurons, is illustrated in the diagram shown below.

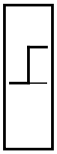


Here the input vector **p** is represented by the solid dark vertical bar at the left. The dimensions of **p** are shown below the symbol **p** in the figure as $R$x1. (Note that we will use a capital letter, such as $R$ in the previous sentence, when referring to the *size* of a vector.) Thus, **p** is a vector of $R$ input elements. These inputs post multiply the single row, $R$ column matrix **W**. As before, a constant 1 enters the neuron as an input and is multiplied by a scalar bias $b$. The net input to the transfer function $f$ is $n$, the sum of the bias $b$ and the product **Wp**. This sum is passed to the transfer function $f$ to get the neuron's output $a$, which in this case is a scalar. Note that if we had more than one neuron, the network output would be a vector.

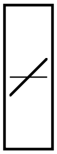
A *layer* of a network is defined in the figure shown above. A layer includes the combination of the weights, the multiplication and summing operation (here realized as a vector product **Wp**), the bias $b$, and the transfer function $f$. The array of inputs, vector **p**, is not included in or called a layer.

Each time this abbreviated network notation is used, the size of the matrices will be shown just below their matrix variable names. We hope that this notation will allow you to understand the architectures and follow the matrix mathematics associated with them.

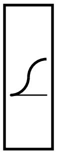
As discussed previously, when a specific transfer function is to be used in a figure, the symbol for that transfer function will replace the *f* shown above. Here are some examples.



*hardlim*



*purelin*



*logsig*

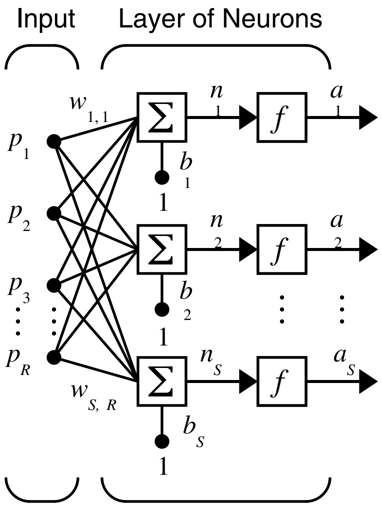
You can experiment with a two-element neuron by running the demonstration program `nnd2n2`.

# Network Architectures

Two or more of the neurons shown earlier can be combined in a layer, and a particular network could contain one or more such layers. First consider a single layer of neurons.

## A Layer of Neurons

A one-layer network with *R* input elements and *S* neurons follows.



$$\mathbf{a} = \mathbf{f}\,(\mathbf{Wp} + \mathbf{b})$$

Where...

*R* = number of elements in input vector

*S* = number of neurons in layer

In this network, each element of the input vector **p** is connected to each neuron input through the weight matrix **W**. The *i*th neuron has a summer that gathers its weighted inputs and bias to form its own scalar output *n(i)*. The various *n(i)* taken together form an *S*-element net input vector **n**. Finally, the neuron layer outputs form a column vector **a**. We show the expression for **a** at the bottom of the figure.

Note that it is common for the number of inputs to a layer to be different from the number of neurons (i.e., *R* ¦ *S*). A layer is not constrained to have the number of its inputs equal to the number of its neurons.

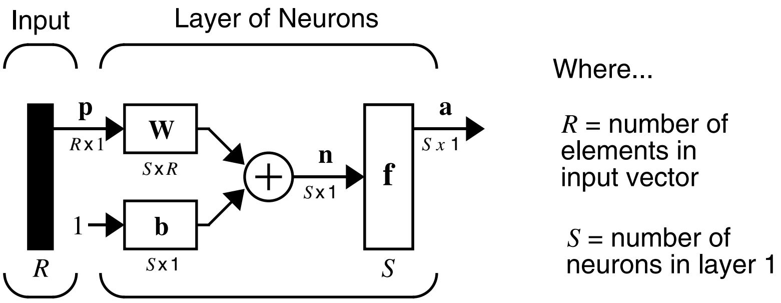
You can create a single (composite) layer of neurons having different transfer functions simply by putting two of the networks shown earlier in parallel. Both networks would have the same inputs, and each network would create some of the outputs.

The input vector elements enter the network through the weight matrix **W**.

$$\mathbf{W} = \begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,R} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,R} \\ \\ w_{S,1} & w_{S,2} & \cdots & w_{S,R} \end{bmatrix}$$

Note that the row indices on the elements of matrix **W** indicate the destination neuron of the weight, and the column indices indicate which source is the input for that weight. Thus, the indices in $w_{1,2}$ say that the strength of the signal *from* the second input element *to* the first (and only) neuron is $w_{1,2}$.

The $S$ neuron $R$ input one-layer network also can be drawn in abbreviated notation.


Input
Layer of Neurons
p
R x 1
W
S x R
1
b
S x 1
n
S x 1
f
a
S x 1
R
S
Where...
R = number of elements in input vector
S = number of neurons in layer 1

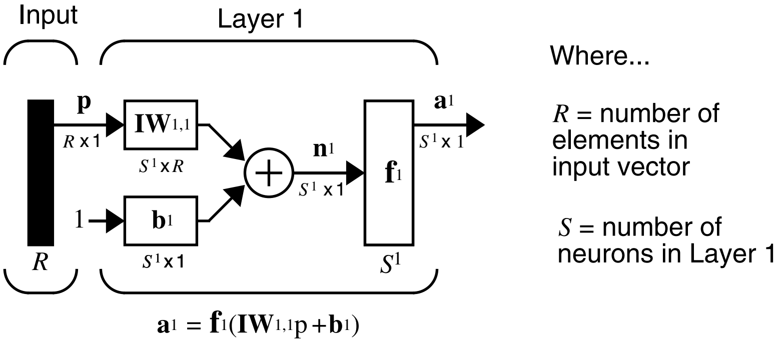
$$\mathbf{a} = \mathbf{f}\,(\mathbf{W}\mathbf{p} + \mathbf{b})$$

Here **p** is an $R$ length input vector, **W** is an $SxR$ matrix, and **a** and **b** are $S$ length vectors. As defined previously, the neuron layer includes the weight matrix, the multiplication operations, the bias vector **b**, the summer, and the transfer function boxes.

### Inputs and Layers

We are about to discuss networks having multiple layers so we will need to extend our notation to talk about such networks. Specifically, we need to make a distinction between weight matrices that are connected to inputs and weight matrices that are connected between layers. We also need to identify the source and destination for the weight matrices.

We will call weight matrices connected to inputs, *input weights;* and we will call weight matrices coming from layer outputs, *layer weights.* Further, we will use superscripts to identify the source (second index) and the destination (first index) for the various weights and other elements of the network. To illustrate, we have taken the one-layer multiple input network shown earlier and redrawn it in abbreviated form below.



As you can see, we have labeled the weight matrix connected to the input vector **p** as an Input Weight matrix ($\mathbf{IW}^{1,1}$) having a source 1 (second index) and a destination 1 (first index). Also, elements of layer one, such as its bias, net input, and output have a superscript 1 to say that they are associated with the first layer.

In the next section, we will use Layer Weight (**LW**) matrices as well as Input Weight (**IW**) matrices.

You might recall from the notation section of the Preface that conversion of the layer weight matrix from math to code for a particular network called *net* is:

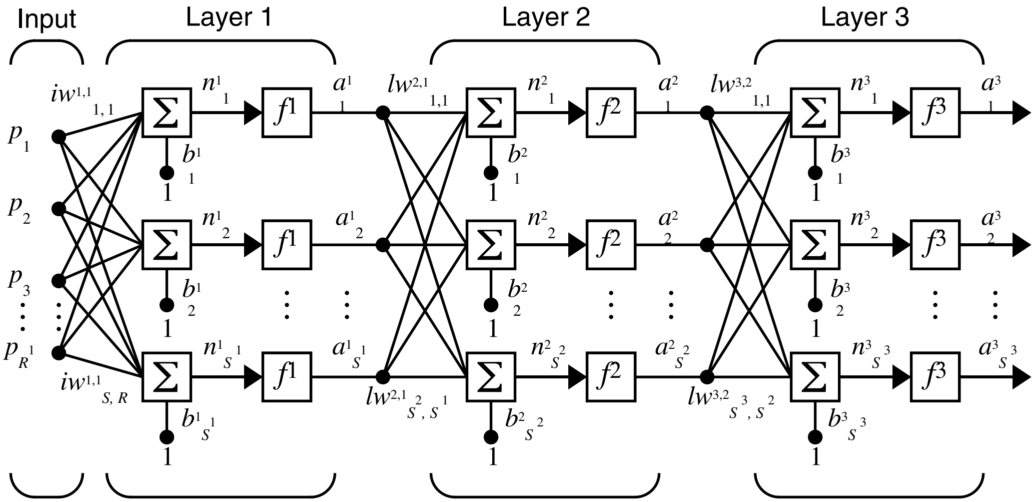
$$\mathbf{IW}^{1,1} \rightarrow \text{net.IW}\{1, 1\}$$

Thus, we could write the code to obtain the net input to the transfer function as:

```
n{1} = net.IW{1,1}*p + net.b{1};
```

## Multiple Layers of Neurons

A network can have several layers. Each layer has a weight matrix **W**, a bias vector **b**, and an output vector **a**. To distinguish between the weight matrices, output vectors, etc., for each of these layers in our figures, we append the number of the layer as a superscript to the variable of interest. You can see the use of this layer notation in the three-layer network shown below, and in the equations at the bottom of the figure.



$$\mathbf{a}^1 = \mathbf{f}^1(\mathbf{IW}^{1,1}\mathbf{p} + \mathbf{b}^1) \qquad \mathbf{a}^2 = \mathbf{f}^2(\mathbf{LW}^{2,1}\mathbf{a}^1 + \mathbf{b}^2) \qquad \mathbf{a}^3 = \mathbf{f}^3(\mathbf{LW}^{3,2}\mathbf{a}^2 + \mathbf{b}^3)$$

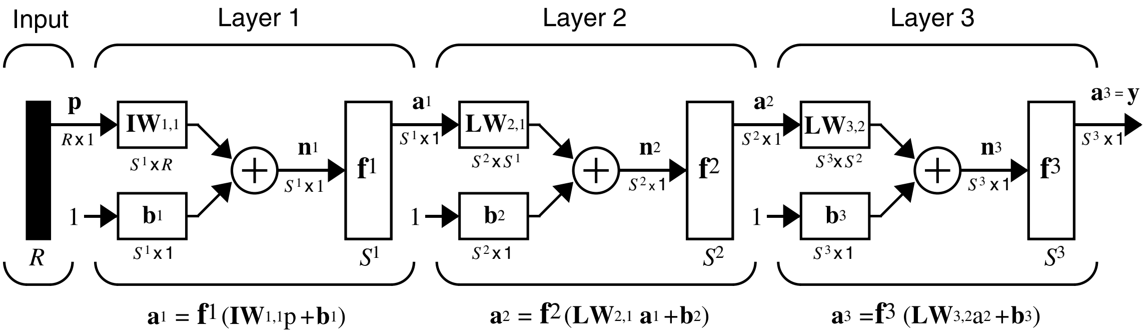
$$\mathbf{a}^3 = \mathbf{f}^3(\mathbf{LW}^{3,2}\,\mathbf{f}^2(\mathbf{LW}^{2,1}\mathrm{f}^1(\mathbf{IW}^{1,1}\mathbf{p} + \mathbf{b}^1) + \mathbf{b}^2) + \mathbf{b}^3)$$

The network shown above has $R^1$ inputs, $S^1$ neurons in the first layer, $S^2$ neurons in the second layer, etc. It is common for different layers to have different numbers of neurons. A constant input 1 is fed to the biases for each neuron.

Note that the outputs of each intermediate layer are the inputs to the following layer. Thus layer 2 can be analyzed as a one-layer network with $S^1$ inputs, $S^2$ neurons, and an $S^2$x$S^1$ weight matrix W$^2$. The input to layer 2 is $\mathbf{a}^1$; the output is $\mathbf{a}^2$. Now that we have identified all the vectors and matrices of layer 2, we can treat it as a single-layer network on its own. This approach can be taken with any layer of the network.

The layers of a multilayer network play different roles. A layer that produces the network output is called an *output layer*. All other layers are called *hidden layers*. The three-layer network shown earlier has one output layer (layer 3) and two hidden layers (layer 1 and layer 2). Some authors refer to the inputs as a fourth layer. We will not use that designation.

The same three-layer network discussed previously also can be drawn using our abbreviated notation.



$$\mathbf{a}^3 = \mathbf{f}^3\,(\mathbf{LW}^{3,2}\,\mathbf{f}^2\,(\mathbf{LW}^{2,1}\mathrm{f}^1\,(\mathbf{IW}^{1,1}\mathbf{p} + \mathbf{b}^1) + \mathbf{b}^2) + \mathbf{b}^3 = \mathbf{y}$$

Multiple-layer networks are quite powerful. For instance, a network of two layers, where the first layer is sigmoid and the second layer is linear, can be trained to approximate any function (with a finite number of discontinuities) arbitrarily well. This kind of two-layer network is used extensively in Chapter 5, "Backpropagation."

Here we assume that the output of the third layer, $\mathbf{a}^3$, is the network output of interest, and we have labeled this output as $\mathbf{y}$. We will use this notation to specify the output of multilayer networks.

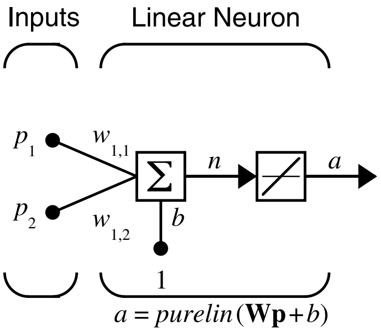
# Data Structures

This section discusses how the format of input data structures affects the simulation of networks. We will begin with static networks, and then move to dynamic networks.

We are concerned with two basic types of input vectors: those that occur *concurrently* (at the same time, or in no particular time sequence), and those that occur *sequentially* in time. For concurrent vectors, the order is not important, and if we had a number of networks running in parallel, we could present one input vector to each of the networks. For sequential vectors, the order in which the vectors appear is important.

## Simulation With Concurrent Inputs in a Static Network

The simplest situation for simulating a network occurs when the network to be simulated is static (has no feedback or delays). In this case, we do not have to be concerned about whether or not the input vectors occur in a particular time sequence, so we can treat the inputs as concurrent. In addition, we make the problem even simpler by assuming that the network has only one input vector. Use the following network as an example.



To set up this feedforward network, we can use the following command.

```
net = newlin([1 3;1 3],1);
```

For simplicity assign the weight matrix and bias to be

$\mathbf{W} = \begin{bmatrix} 1 & 2 \end{bmatrix}$ and $b = \begin{bmatrix} 0 \end{bmatrix}$.

The commands for these assignments are

```
net.IW{1,1} = [1 2];
net.b{1} = 0;
```

Suppose that the network simulation data set consists of $Q = 4$ concurrent vectors:

$$\mathbf{p}_1 = \begin{bmatrix} 1 \\ 2 \end{bmatrix},\quad \mathbf{p}_2 = \begin{bmatrix} 2 \\ 1 \end{bmatrix},\quad \mathbf{p}_3 = \begin{bmatrix} 2 \\ 3 \end{bmatrix},\quad \mathbf{p}_4 = \begin{bmatrix} 3 \\ 1 \end{bmatrix},$$

Concurrent vectors are presented to the network as a single matrix:

```
P = [1 2 2 3; 2 1 3 1];
```

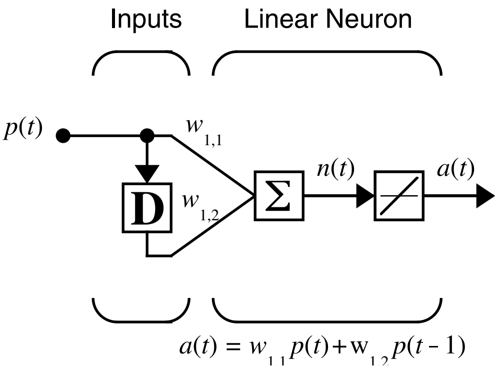
We can now simulate the network:

```
A = sim(net,P)
A =
     5     4     8     5
```

A single matrix of concurrent vectors is presented to the network and the network produces a single matrix of concurrent vectors as output. The result would be the same if there were four networks operating in parallel and each network received one of the input vectors and produced one of the outputs. The ordering of the input vectors is not important as they do not interact with each other.

## Simulation With Sequential Inputs in a Dynamic Network

When a network contains delays, the input to the network would normally be a sequence of input vectors that occur in a certain time order. To illustrate this case, we use a simple network that contains one delay.



The following commands create this network:

```
net = newlin([-1 1],1,[0 1]);
net.biasConnect = 0;
```

Assign the weight matrix to be

$\mathbf{W} = \begin{bmatrix} 1 & 2 \end{bmatrix}$ .

The command is

```
net.IW{1,1} = [1 2];
```

Suppose that the input sequence is

$$\mathbf{p}1 = \begin{bmatrix} 1 \end{bmatrix},\ \mathbf{p}2 = \begin{bmatrix} 2 \end{bmatrix},\ \mathbf{p}3 = \begin{bmatrix} 3 \end{bmatrix},\ \mathbf{p}4 = \begin{bmatrix} 4 \end{bmatrix},$$

Sequential inputs are presented to the network as elements of a cell array:

```
P = {1 2 3 4};
```

We can now simulate the network:

```
A = sim(net,P)
A =
    [1]    [4]    [7]    [10]
```

We input a cell array containing a sequence of inputs, and the network produced a cell array containing a sequence of outputs. Note that the order of the inputs is important when they are presented as a sequence. In this case, the current output is obtained by multiplying the current input by 1 and the preceding input by 2 and summing the result. If we were to change the order of the inputs, it would change the numbers we would obtain in the output.

## Simulation With Concurrent Inputs in a Dynamic Network

If we were to apply the same inputs from the previous example as a set of concurrent inputs instead of a sequence of inputs, we would obtain a completely different response. (Although, it is not clear why we would want to do this with a dynamic network.) It would be as if each input were applied concurrently to a separate parallel network. For the previous example, if we use a concurrent set of inputs we have

$$\mathbf{p}_1 = \begin{bmatrix}1\end{bmatrix},\quad \mathbf{p}_2 = \begin{bmatrix}2\end{bmatrix},\quad \mathbf{p}_3 = \begin{bmatrix}3\end{bmatrix},\quad \mathbf{p}_4 = \begin{bmatrix}4\end{bmatrix}$$

which can be created with the following code:

```
P = [1 2 3 4];
```

When we simulate with concurrent inputs we obtain

```
A = sim(net,P)
A =
     1     2     3     4
```

The result is the same as if we had concurrently applied each one of the inputs to a separate network and computed one output. Note that since we did not assign any initial conditions to the network delays, they were assumed to be zero. For this case the output will simply be 1 times the input, since the weight that multiplies the current input is 1.

In certain special cases, we might want to simulate the network response to several different sequences at the same time. In this case, we would want to present the network with a concurrent set of sequences. For example, let's say we wanted to present the following two sequences to the network:

$$\mathbf{p}_1(1) = \begin{bmatrix}1\end{bmatrix},\ \mathbf{p}_1(2) = \begin{bmatrix}2\end{bmatrix},\ \mathbf{p}_1(3) = \begin{bmatrix}3\end{bmatrix},\ \mathbf{p}_1(4) = \begin{bmatrix}4\end{bmatrix}$$

$$\mathbf{p}_2(1) = \begin{bmatrix}4\end{bmatrix},\ \mathbf{p}_2(2) = \begin{bmatrix}3\end{bmatrix},\ \mathbf{p}_2(3) = \begin{bmatrix}2\end{bmatrix},\ \mathbf{p}_2(4) = \begin{bmatrix}1\end{bmatrix}$$

The input P should be a cell array, where each element of the array contains the two elements of the two sequences that occur at the same time:

```
P = {[1 4] [2 3] [3 2] [4 1]};
```

We can now simulate the network:

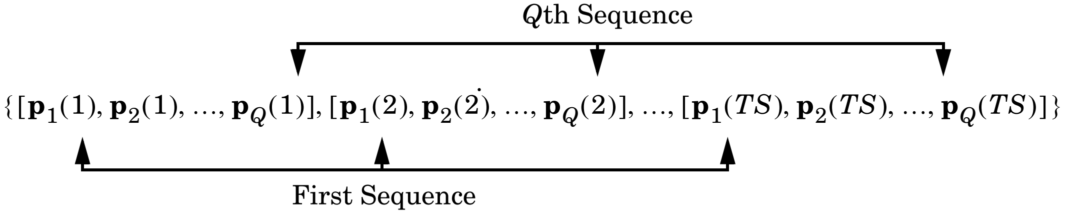
```
A = sim(net,P);
```

The resulting network output would be

```
A = {[ 1 4] [4 11] [7 8] [10 5]}
```

As you can see, the first column of each matrix makes up the output sequence produced by the first input sequence, which was the one we used in an earlier example. The second column of each matrix makes up the output sequence produced by the second input sequence. There is no interaction between the two concurrent sequences. It is as if they were each applied to separate networks running in parallel.

The following diagram shows the general format for the input P to the sim function when we have $Q$ concurrent sequences of $TS$ time steps. It covers all cases where there is a single input vector. Each element of the cell array is a matrix of concurrent vectors that correspond to the same point in time for each sequence. If there are multiple input vectors, there will be multiple rows of matrices in the cell array.



$$\{[\mathbf{p}_1(1), \mathbf{p}_2(1), \ldots, \mathbf{p}_Q(1)], [\mathbf{p}_1(2), \mathbf{p}_2(2), \ldots, \mathbf{p}_Q(2)], \ldots, [\mathbf{p}_1(TS), \mathbf{p}_2(TS), \ldots, \mathbf{p}_Q(TS)]\}$$

In this section, we have applied sequential and concurrent inputs to dynamic networks. In the previous section, we applied concurrent inputs to static networks. It is also possible to apply sequential inputs to static networks. It will not change the simulated response of the network, but it can affect the way in which the network is trained. This will become clear in the next section.

# Training Styles

In this section, we describe two different styles of training. In *incremental* training the weights and biases of the network are updated each time an input is presented to the network. In *batch* training the weights and biases are only updated after all of the inputs are presented.

## Incremental Training (of Adaptive and Other Networks)

Incremental training can be applied to both static and dynamic networks, although it is more commonly used with dynamic networks, such as adaptive filters. In this section, we demonstrate how incremental training is performed on both static and dynamic networks.

### Incremental Training with Static Networks

Consider again the static network we used for our first example. We want to train it incrementally, so that the weights and biases will be updated after each input is presented. In this case we use the function `adapt`, and we present the inputs and targets as sequences.

Suppose we want to train the network to create the linear function

$$t = 2p_1 + p_2 \ .$$

Then for the previous inputs we used,

$$\mathbf{p}_1 = \begin{bmatrix} 1 \\ 2 \end{bmatrix},\ \mathbf{p}_2 = \begin{bmatrix} 2 \\ 1 \end{bmatrix},\ \mathbf{p}_3 = \begin{bmatrix} 2 \\ 3 \end{bmatrix},\ \mathbf{p}_4 = \begin{bmatrix} 3 \\ 1 \end{bmatrix}$$

the targets would be

$$\mathbf{t}_1 = \begin{bmatrix} 4 \end{bmatrix},\ \mathbf{t}_2 = \begin{bmatrix} 5 \end{bmatrix},\ \mathbf{t}_3 = \begin{bmatrix} 7 \end{bmatrix},\ \mathbf{t}_4 = \begin{bmatrix} 7 \end{bmatrix}$$

We first set up the network with zero initial weights and biases. We also set the learning rate to zero initially, to show the effect of the incremental training.

```
net = newlin([-1 1;-1 1],1,0,0);
net.IW{1,1} = [0 0];
net.b{1} = 0;
```

For incremental training we want to present the inputs and targets as sequences:

```
P = {[1;2] [2;1] [2;3] [3;1]};
T = {4 5 7 7};
```

Recall from the earlier discussion that for a static network the simulation of the network produces the same outputs whether the inputs are presented as a matrix of concurrent vectors or as a cell array of sequential vectors. This is not true when training the network, however. When using the adapt function, if the inputs are presented as a cell array of sequential vectors, then the weights are updated as each input is presented (incremental mode). As we see in the next section, if the inputs are presented as a matrix of concurrent vectors, then the weights are updated only after all inputs are presented (batch mode).

We are now ready to train the network incrementally.

```
[net,a,e,pf] = adapt(net,P,T);
```

The network outputs will remain zero, since the learning rate is zero, and the weights are not updated. The errors will be equal to the targets:

```
a = [0]    [0]    [0]    [0]
e = [4]    [5]    [7]    [7]
```

If we now set the learning rate to 0.1 we can see how the network is adjusted as each input is presented:

```
net.inputWeights{1,1}.learnParam.lr=0.1;
net.biases{1,1}.learnParam.lr=0.1;
[net,a,e,pf] = adapt(net,P,T);
a = [0]    [2]    [6.0]    [5.8]
e = [4]    [3]    [1.0]    [1.2]
```

The first output is the same as it was with zero learning rate, since no update is made until the first input is presented. The second output is different, since the weights have been updated. The weights continue to be modified as each error is computed. If the network is capable and the learning rate is set correctly, the error will eventually be driven to zero.

## Incremental Training with Dynamic Networks

We can also train dynamic networks incrementally. In fact, this would be the most common situation. Let’s take the linear network with one delay at the input that we used in a previous example. We initialize the weights to zero and set the learning rate to 0.1.

```
net = newlin([-1 1],1,[0 1],0.1);
net.IW{1,1} = [0 0];
net.biasConnect = 0;
```

To train this network incrementally we present the inputs and targets as elements of cell arrays.

```
Pi = {1};
P = {2 3 4};
T = {3 5 7};
```

Here we attempt to train the network to sum the current and previous inputs to create the current output. This is the same input sequence we used in the previous example of using `sim`, except that we assign the first term in the sequence as the initial condition for the delay. We now can sequentially train the network using adapt.

```
[net,a,e,pf] = adapt(net,P,T,Pi);
a = [0] [2.4] [ 7.98]
e = [3] [2.6] [-0.98]
```

The first output is zero, since the weights have not yet been updated. The weights change at each subsequent time step.

## Batch Training

Batch training, in which weights and biases are only updated after all of the inputs and targets are presented, can be applied to both static and dynamic networks. We discuss both types of networks in this section.

### Batch Training with Static Networks

Batch training can be done using either `adapt` or `train`, although `train` is generally the best option, since it typically has access to more efficient training algorithms. Incremental training can only be done with `adapt`; `train` can only perform batch training.

Let's begin with the static network we used in previous examples. The learning rate will be set to 0.1.

```
net = newlin([-1 1;-1 1],1,0,0.1);
```

```
net.IW{1,1} = [0 0];
net.b{1} = 0;
```

For batch training of a static network with adapt, the input vectors must be placed in one matrix of concurrent vectors.

```
P = [1 2 2 3; 2 1 3 1];
T = [4 5 7 7];
```

When we call adapt, it will invoke trains (which is the default adaptation function for the linear network) and learnwh (which is the default learning function for the weights and biases). Therefore, Widrow-Hoff learning is used.

```
[net,a,e,pf] = adapt(net,P,T);
a = 0 0 0 0
e = 4 5 7 7
```

Note that the outputs of the network are all zero, because the weights are not updated until all of the training set has been presented. If we display the weights we find:

```
»net.IW{1,1}
  ans = 4.9000    4.1000
»net.b{1}
  ans =
    2.3000
```

This is different that the result we had after one pass of adapt with incremental updating.

Now let’s perform the same batch training using train. Since the Widrow-Hoff rule can be used in incremental or batch mode, it can be invoked by adapt or train. There are several algorithms that can only be used in batch mode (e.g., Levenberg-Marquardt), and so these algorithms can only be invoked by train.

The network will be set up in the same way.

```
net = newlin([-1 1;-1 1],1,0,0.1);
net.IW{1,1} = [0 0];
net.b{1} = 0;
```

For this case, the input vectors can either be placed in a matrix of concurrent vectors or in a cell array of sequential vectors. Within train any cell array of sequential vectors is converted to a matrix of concurrent vectors. This is because the network is static, and because train always operates in the batch mode. Concurrent mode operation is generally used whenever possible, because it has a more efficient MATLAB implementation.

```
P = [1 2 2 3; 2 1 3 1];
T = [4 5 7 7];
```

Now we are ready to train the network. We will train it for only one epoch, since we used only one pass of adapt. The default training function for the linear network is trainc, and the default learning function for the weights and biases is learnwh, so we should get the same results that we obtained using adapt in the previous example, where the default adaptation function was trains.

```
net.inputWeights{1,1}.learnParam.lr = 0.1;
net.biases{1}.learnParam.lr = 0.1;
net.trainParam.epochs = 1;
net = train(net,P,T);
```

If we display the weights after one epoch of training we find:

```
»net.IW{1,1}
  ans = 4.9000    4.1000
»net.b{1}
  ans =
    2.3000
```

This is the same result we had with the batch mode training in adapt. With static networks, the adapt function can implement incremental or batch training depending on the format of the input data. If the data is presented as a matrix of concurrent vectors, batch training will occur. If the data is presented as a sequence, incremental training will occur. This is not true for train, which always performs batch training, regardless of the format of the input.

## Batch Training With Dynamic Networks

Training static networks is relatively straightforward. If we use train the network is trained in the batch mode and the inputs is converted to concurrent vectors (columns of a matrix), even if they are originally passed as a sequence (elements of a cell array). If we use adapt, the format of the input determines the method of training. If the inputs are passed as a sequence, then the network is trained in incremental mode. If the inputs are passed as concurrent vectors, then batch mode training is used.

With dynamic networks, batch mode training is typically done with `train` only, especially if only one training sequence exists. To illustrate this, let's consider again the linear network with a delay. We use a learning rate of 0.02 for the training. (When using a gradient descent algorithm, we typically use a smaller learning rate for batch mode training than incremental training, because all of the individual gradients are summed together before determining the step change to the weights.)

```
net = newlin([-1 1],1,[0 1],0.02);
net.IW{1,1}=[0 0];
net.biasConnect=0;
net.trainParam.epochs = 1;
Pi = {1};
P = {2 3 4};
T = {3 5 6};
```

We want to train the network with the same sequence we used for the incremental training earlier, but this time we want to update the weights only after all of the inputs are applied (batch mode). The network is simulated in sequential mode because the input is a sequence, but the weights are updated in batch mode.

```
net=train(net,P,T,Pi);
```

The weights after one epoch of training are

```
»net.IW{1,1}
ans = 0.9000    0.6200
```

These are different weights than we would obtain using incremental training, where the weights would be updated three times during one pass through the training set. For batch training the weights are only updated once in each epoch.

## Summary

The inputs to a neuron include its bias and the sum of its weighted inputs (using the inner product). The output of a neuron depends on the neuron's inputs and on its transfer function. There are many useful transfer functions.

A single neuron cannot do very much. However, several neurons can be combined into a layer or multiple layers that have great power. Hopefully this toolbox makes it easy to create and understand such large networks.

The architecture of a network consists of a description of how many layers a network has, the number of neurons in each layer, each layer's transfer function, and how the layers connect to each other. The best architecture to use depends on the type of problem to be represented by the network.

A network effects a computation by mapping input values to output values. The particular mapping problem to be performed fixes the number of inputs, as well as the number of outputs for the network.

Aside from the number of neurons in a network's output layer, the number of neurons in each layer is up to the designer. Except for purely linear networks, the more neurons in a hidden layer, the more powerful the network.

If a linear mapping needs to be represented linear neurons should be used. However, linear networks cannot perform any nonlinear computation. Use of a nonlinear transfer function makes a network capable of storing nonlinear relationships between input and output.

A very simple problem can be represented by a single layer of neurons. However, single-layer networks cannot solve certain problems. Multiple feed-forward layers give a network greater freedom. For example, any reasonable function can be represented with a two-layer network: a sigmoid layer feeding a linear output layer.

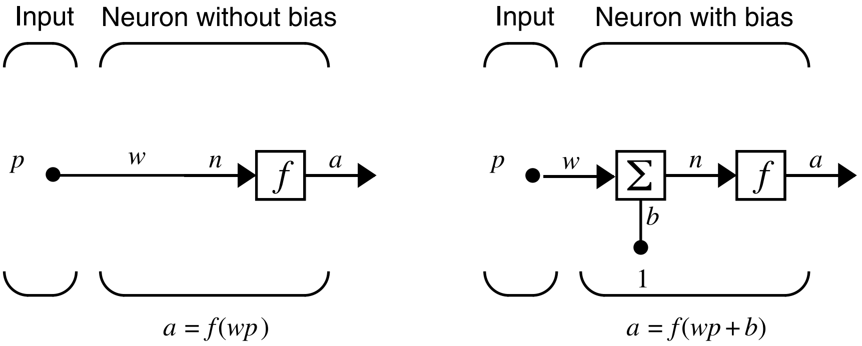
Networks with biases can represent relationships between inputs and outputs more easily than networks without biases. (For example, a neuron without a bias will always have a net input to the transfer function of zero when all of its inputs are zero. However, a neuron with a bias can learn to have any net transfer function input under the same conditions by learning an appropriate value for the bias.)

Feed-forward networks cannot perform temporal computation. More complex networks with internal feedback paths are required for temporal behavior.

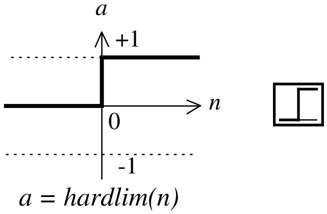
If several input vectors are to be presented to a network, they may be presented sequentially or concurrently. Batching of concurrent inputs is computationally more efficient and may be what is desired in any case. The matrix notation used in MATLAB makes batching simple.

## Figures and Equations

### Simple Neuron

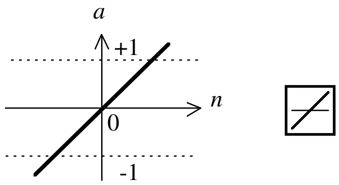


### Hard Limit Transfer Function



Hard-Limit Transfer Function

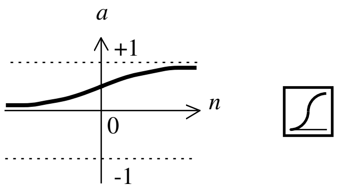
### Purelin Transfer Function



*a = purelin(n)*

Linear Transfer Function

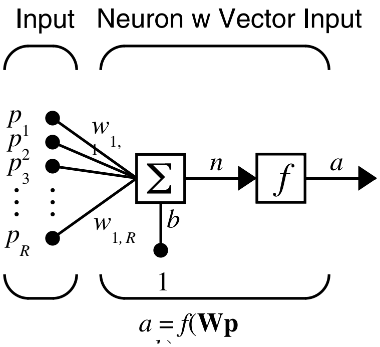
### Log Sigmoid Transfer Function



*a = logsig(n)*

Log-Sigmoid Transfer Function

### Neuron With Vector Input



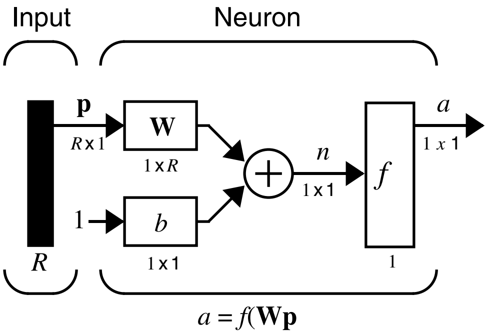
Where...

*R* = number of elements in input vector

### Net Input

$$n = w_{1,1}p_1 + w_{1,2}p_2 + \ldots + w_{1,R}p_R + b$$

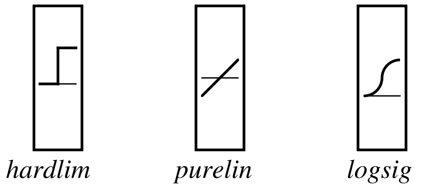
### Single Neuron Using Abbreviated Notation



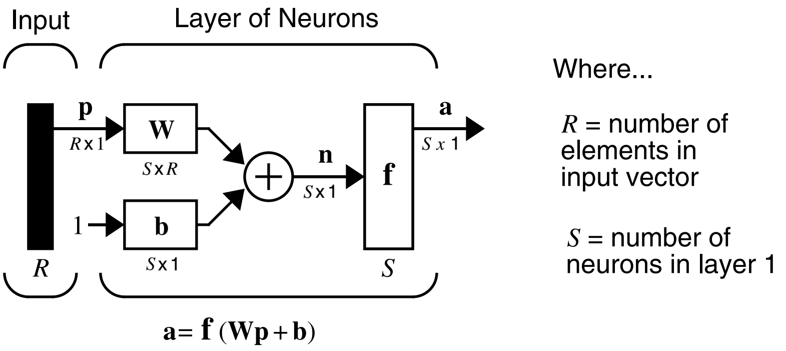
Where...

*R* = number of elements in input vector

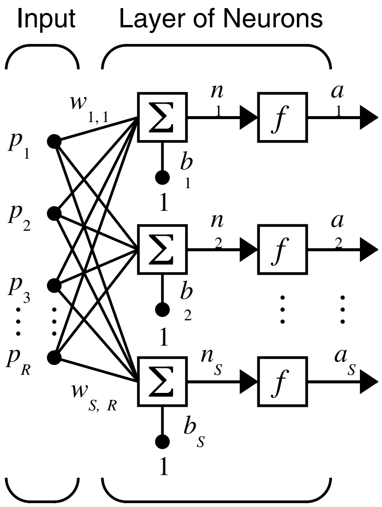
## Icons for Transfer Functions



## Layer of Neurons



## A Layer of Neurons



$$\mathbf{a} = \mathbf{f}(\mathbf{W}\mathbf{p} + \mathbf{b})$$

Where...

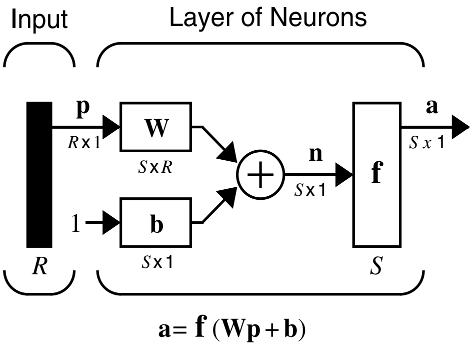
$R$ = number of elements in input vector

$S$ = number of neurons in layer

## Weight Matrix

$$\mathbf{W} = \begin{bmatrix} w_{1,1} & w_{1,2} & \dots & w_{1,R} \\ w_{2,1} & w_{2,2} & \dots & w_{2,R} \\ & & & \\ w_{S,1} & w_{S,2} & \dots & w_{S,R} \end{bmatrix}$$

### Layer of Neurons, Abbreviated Notation

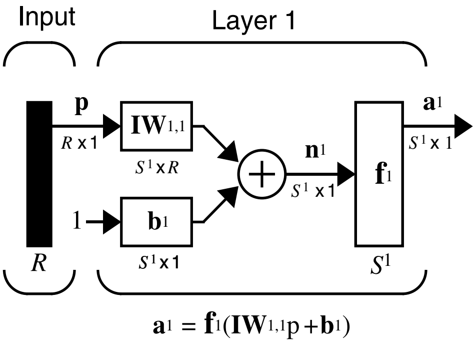


Where...

$R$ = number of elements in input vector

$S$ = number of neurons in layer 1

### Layer of Neurons Showing Indices

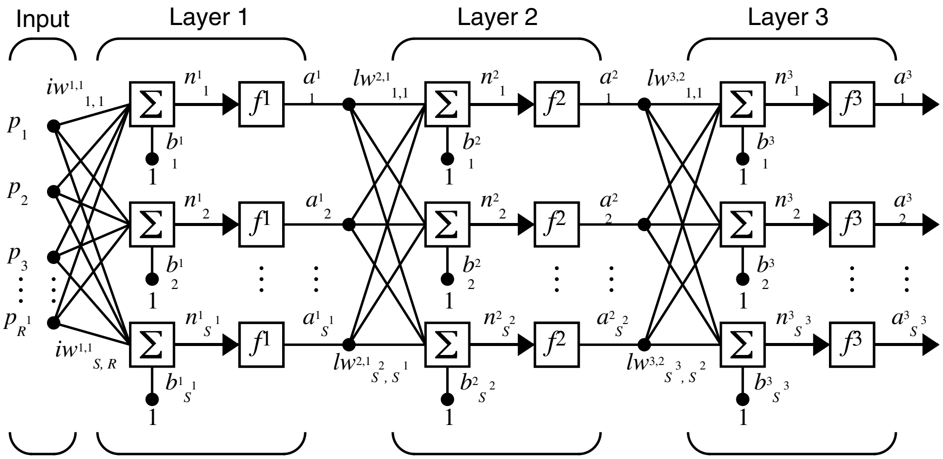


Where...

$R$ = number of elements in input vector

$S$ = number of neurons in Layer 1

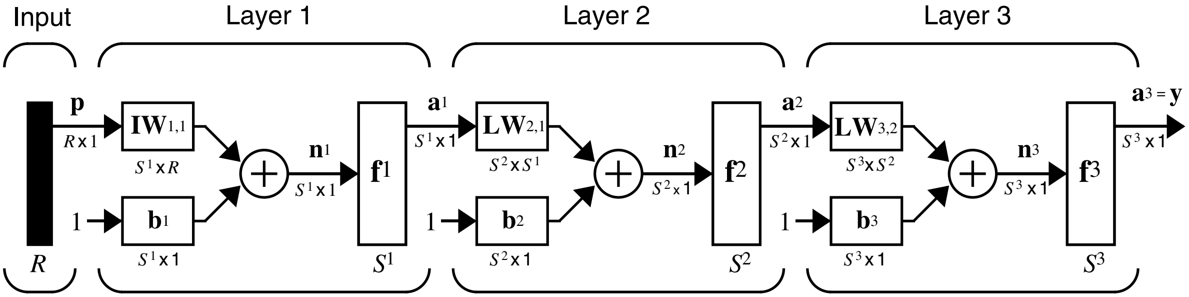
### Three Layers of Neurons



$$\mathbf{a}^1 = \mathbf{f}^1(\mathbf{IW}^{1,1}\mathbf{p} + \mathbf{b}^1) \qquad \mathbf{a}^2 = \mathbf{f}^2(\mathbf{LW}^{2,1}\mathbf{a}^1 + \mathbf{b}^2) \qquad \mathbf{a}^3 = \mathbf{f}^3(\mathbf{LW}^{3,2}\mathbf{a}^2 + \mathbf{b}^3)$$

$$\mathbf{a}^3 = \mathbf{f}^3(\mathbf{LW}^{3,2}\,\mathbf{f}^2(\mathbf{LW}^{2,1}\mathrm{f}^1(\mathbf{IW}^{1,1}\mathbf{p} + \mathbf{b}^1) + \mathbf{b}^2) + \mathbf{b}^3)$$

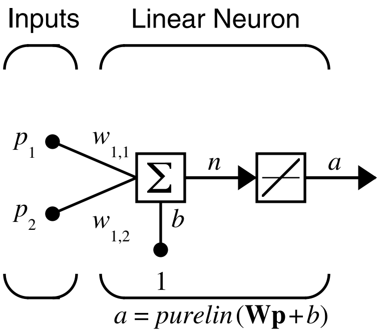
### Three Layers, Abbreviated Notation



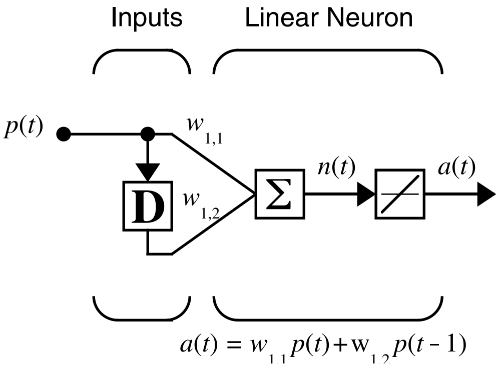
$$\mathbf{a}^1 = \mathbf{f}^1(\mathbf{IW}^{1,1}\mathrm{p} + \mathbf{b}^1) \qquad \mathbf{a}^2 = \mathbf{f}^2(\mathbf{LW}^{2,1}\,\mathbf{a}^1 + \mathbf{b}^2) \qquad \mathbf{a}^3 = \mathbf{f}^3(\mathbf{LW}^{3,2}\mathrm{a}^2 + \mathbf{b}^3)$$

$$\mathbf{a}^3 = \mathbf{f}^3(\mathbf{LW}^{3,2}\,\mathbf{f}^2(\mathbf{LW}^{2,1}\mathrm{f}^1(\mathbf{IW}^{1,1}\mathbf{p} + \mathbf{b}^1) + \mathbf{b}^2) + \mathbf{b}^3 = \mathbf{y}$$

### Linear Neuron With Two-Element Vector Input



### Dynamic Network With One Delay



3

# Perceptrons

# Introduction

This chapter has a number of objectives. First we want to introduce you to learning rules, methods of deriving the next changes that might be made in a network, and training, a procedure whereby a network is actually adjusted to do a particular job. Along the way we discuss a toolbox function to create a simple perceptron network, and we also cover functions to initialize and simulate such networks. We use the perceptron as a vehicle for tying these concepts together.

Rosenblatt [Rose61] created many variations of the perceptron. One of the simplest was a single-layer network whose weights and biases could be trained to produce a correct target vector when presented with the corresponding input vector. The training technique used is called the perceptron learning rule. The perceptron generated great interest due to its ability to generalize from its training vectors and learn from initially randomly distributed connections. Perceptrons are especially suited for simple problems in pattern classification. They are fast and reliable networks for the problems they can solve. In addition, an understanding of the operations of the perceptron provides a good basis for understanding more complex networks.

In this chapter we define what we mean by a learning rule, explain the perceptron network and its learning rule, and tell you how to initialize and simulate perceptron networks.

The discussion of perceptron in this chapter is necessarily brief. For a more thorough discussion, see Chapter 4 “Perceptron Learning Rule” of [HDB1996], which discusses the use of multiple layers of perceptrons to solve more difficult problems beyond the capability of one layer.

You also may want to refer to the original book on the perceptron, Rosenblatt, F., *Principles of Neurodynamics,* Washington D.C.: Spartan Press, 1961. [Rose61].

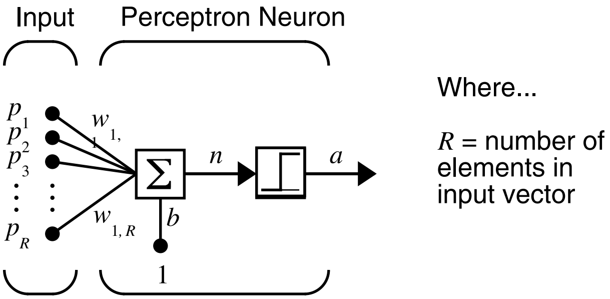
## Important Perceptron Functions

Entering `help percept` at the MATLAB® command line displays all the functions that are related to perceptrons.

Perceptron networks can be created with the function `newp`. These networks can be initialized, simulated and trained with the `init`, `sim` and `train`. The following material describes how perceptrons work and introduces these functions.

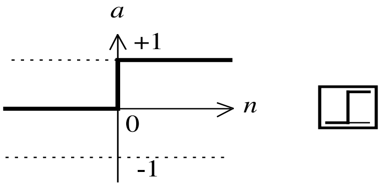
## Neuron Model

A perceptron neuron, which uses the hard-limit transfer function `hardlim`, is shown below.



$a = \mathbf{hardlim}(\mathbf{Wp} + b)$

Each external input is weighted with an appropriate weight $w_{1j}$, and the sum of the weighted inputs is sent to the hard-limit transfer function, which also has an input of 1 transmitted to it through the bias. The hard-limit transfer function, which returns a 0 or a 1, is shown below.

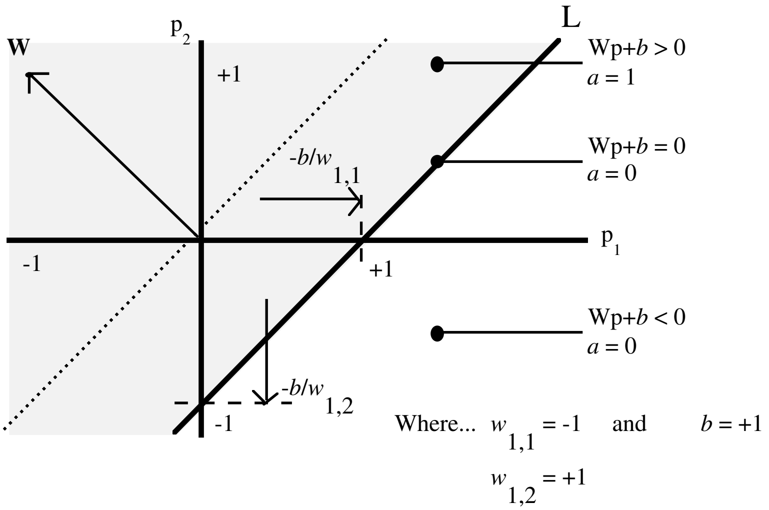


$a = hardlim(n)$

Hard-Limit Transfer Function

The perceptron neuron produces a 1 if the net input into the transfer function is equal to or greater than 0; otherwise it produces a 0.

The hard-limit transfer function gives a perceptron the ability to classify input vectors by dividing the input space into two regions. Specifically, outputs will be 0 if the net input $n$ is less than 0, or 1 if the net input $n$ is 0 or greater. The input space of a two-input hard limit neuron with the weights $w_{1,1} = -1,\ w_{1,2} = 1$ and a bias $b = 1$, is shown below.



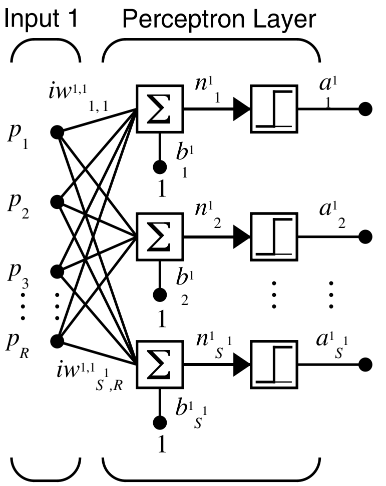
Two classification regions are formed by the *decision boundary* line L at $\mathbf{Wp} + b = 0$. This line is perpendicular to the weight matrix **W** and shifted according to the bias $b$. Input vectors above and to the left of the line L will result in a net input greater than 0; and therefore, cause the hard-limit neuron to output a 1. Input vectors below and to the right of the line L cause the neuron to output 0. The dividing line can be oriented and moved anywhere to classify the input space as desired by picking the weight and bias values.

Hard-limit neurons without a bias will always have a classification line going through the origin. Adding a bias allows the neuron to solve problems where the two sets of input vectors are not located on different sides of the origin. The bias allows the decision boundary to be shifted away from the origin as shown in the plot above.

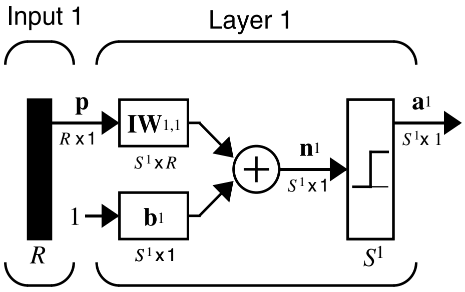
You may want to run the demonstration program `nnd4db`. With it you can move a decision boundary around, pick new inputs to classify, and see how the repeated application of the learning rule yields a network that does classify the input vectors properly.

## Perceptron Architecture

The perceptron network consists of a single layer of S perceptron neurons connected to $R$ inputs through a set of weights $w_{i,j}$ as shown below in two forms. As before, the network indices $i$ and $j$ indicate that $w_{i,j}$ is the strength of the connection from the $j$th input to the $i$th neuron.



$\mathbf{a}^1 = \mathbf{hardlim}(\mathbf{IW}^{1,1}\mathbf{p}^1 + \mathbf{b}^1)$



$\mathbf{a}^1 = \mathbf{hardlim}(\mathbf{IW}^{1,1}\mathbf{p}^1 + \mathbf{b}^1)$

Where...

$R$ = number of elements in Input

$S^1$ = number of neurons in layer 1

The perceptron learning rule that we will describe shortly is capable of training only a single layer. Thus, here we will consider only one-layer networks. This restriction places limitations on the computation a perceptron can perform. The types of problems that perceptrons are capable of solving are discussed later in this chapter in the "Limitations and Cautions" section.

# Creating a Perceptron (newp)

A perceptron can be created with the function newp

```
net = newp(PR, S)
```

where input arguments:

PR is an R-by-2 matrix of minimum and maximum values for R input elements.

S is the number of neurons.

Commonly the hardlim function is used in perceptrons, so it is the default.

The code below creates a perceptron network with a single one-element input vector and one neuron. The range for the single element of the single input vector is [0 2].

```
net = newp([0 2],1);
```

We can see what network has been created by executing the following code

```
inputweights = net.inputweights{1,1}
```

which yields:

```
inputweights =
        delays: 0
       initFcn: 'initzero'
         learn: 1
      learnFcn: 'learnp'
    learnParam: []
          size: [1 1]
      userdata: [1x1 struct]
     weightFcn: 'dotprod'
```

Note that the default learning function is learnp, which is discussed later in this chapter. The net input to the hardlim transfer function is dotprod, which generates the product of the input vector and weight matrix and adds the bias to compute the net input.

Also note that the default initialization function, initzero, is used to set the initial values of the weights to zero.

Similarly,

```
biases = net.biases{1}
```

gives

```
biases =
       initFcn: 'initzero'
         learn: 1
      learnFcn: 'learnp'
    learnParam: []
          size: 1
      userdata: [1x1 struct]
```

We can see that the default initialization for the bias is also 0.

## Simulation (sim)

To show how `sim` works we examine a simple problem.

Suppose we take a perceptron with a single two-element input vector, like that discussed in the *decision boundary* figure. We define the network with

```
net = newp([-2 2;-2 +2],1);
```

As noted above, this gives us zero weights and biases, so if we want a particular set other than zeros, we have to create them. We can set the two weights and the one bias to -1, 1 and 1 as they were in the decision boundary figure with the following two lines of code.

```
net.IW{1,1}= [-1 1];
net.b{1} = [1];
```

To make sure that these parameters were set correctly, we check them with

```
net.IW{1,1}
ans =
    -1     1
net.b{1}
ans =

     1
```

Now let us see if the network responds to two signals, one on each side of the perceptron boundary.

```
p1 = [1;1];
```

```
a1 = sim(net,p1)
a1 =

     1
```

and for

```
p2 = [1;-1]
a2 = sim(net,p2)
a2 =

     0
```

Sure enough, the perceptron classified the two inputs correctly.

Note that we could present the two inputs in a sequence and get the outputs in a sequence as well.

```
p3 = {[1;1] [1;-1]};
a3 = sim(net,p3)
a3 =

    [1]    [0]
```

You may want to read more about `sim` in “Advanced Topics” in Chapter 12.

## Initialization (init)

You can use the function `init` to reset the network weights and biases to their original values. Suppose, for instance that you start with the network

```
net = newp([-2 2;-2 +2],1);
```

Now check its weights with

```
wts = net.IW{1,1}
```

which gives, as expected,

```
wts =

     0     0
```

In the same way, you can verify that the bias is 0 with

```
bias = net.b{1}
```

which gives

```
bias =

     0
```

Now set the weights to the values 3 and 4 and the bias to the value 5 with

```
net.IW{1,1} = [3,4];
net.b{1} = 5;
```

Recheck the weights and bias as shown above to verify that the change has been made. Sure enough,

```
wts =
     3     4
bias =

     5
```

Now use `init` to reset the weights and bias to their original values.

```
net = init(net);
```

We can check as shown above to verify that.

```
wts =
     0     0
bias =

     0
```

We can change the way that a perceptron is initialized with `init`. For instance, we can redefine the network input weights and bias `initFcns` as `rands`, and then apply `init` as shown below.

```
net.inputweights{1,1}.initFcn = 'rands';
net.biases{1}.initFcn = 'rands';
net = init(net);
```

Now check on the weights and bias.

```
wts =
    0.2309    0.5839
biases =
    -0.1106
```

We can see that the weights and bias have been given random numbers.

You may want to read more about `init` in “Advanced Topics” in Chapter 12.

## Learning Rules

We define a *learning rule* as a procedure for modifying the weights and biases of a network. (This procedure may also be referred to as a training algorithm.) The learning rule is applied to train the network to perform some particular task. Learning rules in this toolbox fall into two broad categories: supervised learning, and unsupervised learning.

In *supervised learning*, the learning rule is provided with a set of examples (the *training set*) of proper network behavior

$$\{\mathbf{p}_1, \mathbf{t}_1\}, \{\mathbf{p}_2, \mathbf{t}_2\}, \ldots, \{\mathbf{p}_Q, \mathbf{t}_Q\}$$

where $\mathbf{p}_q$ is an input to the network, and $\mathbf{t}_q$ is the corresponding correct (*target*) output. As the inputs are applied to the network, the network outputs are compared to the targets. The learning rule is then used to adjust the weights and biases of the network in order to move the network outputs closer to the targets. The perceptron learning rule falls in this supervised learning category.

In *unsupervised learning*, the weights and biases are modified in response to network inputs only. There are no target outputs available. Most of these algorithms perform clustering operations. They categorize the input patterns into a finite number of classes. This is especially useful in such applications as vector quantization.

As noted, the perceptron discussed in this chapter is trained with supervised learning. Hopefully, a network that produces the right output for a particular input will be obtained.

## Perceptron Learning Rule (learnp)

Perceptrons are trained on examples of desired behavior. The desired behavior can be summarized by a set of input, output pairs

$$\mathbf{p}_1\mathbf{t}_1,\mathbf{p}_2\mathbf{t}_1,\ldots,\mathrm{p}_Q\mathbf{t}_Q$$

where **p** is an input to the network and **t** is the corresponding correct (target) output. The objective is to reduce the error **e**, which is the difference $\mathbf{t} - \mathbf{a}$ between the neuron response **a,** and the target vector **t**. The *perceptron learning rule* `learnp` calculates desired changes to the perceptron's weights and biases given an input vector **p**, and the associated error **e**. The target vector **t** must contain values of either 0 or 1, as perceptrons (with `hardlim` transfer functions) can only output such values.

Each time `learnp` is executed, the perceptron has a better chance of producing the correct outputs. The perceptron rule is proven to converge on a solution in a finite number of iterations if a solution exists.

If a bias is not used, `learnp` works to find a solution by altering only the weight vector **w** to point toward input vectors to be classified as 1, and away from vectors to be classified as 0. This results in a decision boundary that is perpendicular to **w**, and which properly classifies the input vectors.

There are three conditions that can occur for a single neuron once an input vector **p** is presented and the network's response **a** is calculated:

**CASE 1.** If an input vector is presented and the output of the neuron is correct ($\mathbf{a} = \mathbf{t}$, and $\mathbf{e} = \mathbf{t} - \mathbf{a} = 0$), then the weight vector **w** is not altered.

**CASE 2.** If the neuron output is 0 and should have been 1 ($\mathbf{a} = 0$ and $\mathbf{t} = 1$, and $\mathbf{e} = \mathbf{t} - \mathbf{a} = 1$), the input vector **p** is added to the weight vector **w**. This makes the weight vector point closer to the input vector, increasing the chance that the input vector will be classified as a 1 in the future.

**CASE 3.** If the neuron output is 1 and should have been 0 ($\mathbf{a} = 1$and $\mathbf{t} = 0$, and $\mathbf{e} = \mathbf{t} - \mathbf{a} = -1$), the input vector **p** is subtracted from the weight vector **w**. This makes the weight vector point farther away from the input vector, increasing the chance that the input vector is classified as a 0 in the future.

The perceptron learning rule can be written more succinctly in terms of the error $\mathbf{e} = \mathbf{t} - \mathbf{a}$, and the change to be made to the weight vector $\Delta\mathbf{w}$:

**CASE 1.** If $\mathbf{e} = 0$, then make a change $\Delta\mathbf{w}$ equal to 0.

**CASE 2.** If $\mathbf{e} = 1$, then make a change $\Delta\mathbf{w}$ equal to $\mathbf{p}^{\mathrm{T}}$.

**CASE 3.** If $\mathbf{e} = -1$, then make a change $\Delta\mathbf{w}$ equal to $-\mathbf{p}^{\mathrm{T}}$.

All three cases can then be written with a single expression:

$$\Delta\mathbf{w} = (t-a)\mathbf{p}^{\mathrm{T}} = e\mathbf{p}^{\mathrm{T}}$$

We can get the expression for changes in a neuron's bias by noting that the bias is simply a weight that always has an input of 1:

$$\Delta b = (t-a)(1) = e$$

For the case of a layer of neurons we have:

$$\Delta\mathbf{W} = (\mathbf{t}-\mathbf{a})(\mathbf{p})^{\mathrm{T}} = \mathbf{e}(\mathbf{p})^{\mathrm{T}} \text{ and}$$

$$\Delta\mathbf{b} = (\mathbf{t}-\mathbf{a}) = \mathbf{E}$$

The Perceptron Learning Rule can be summarized as follows

$$\mathbf{W}^{new} = \mathbf{W}^{old} + \mathbf{e}\mathbf{p}^{T} \text{ and}$$

$$\mathbf{b}^{new} = \mathbf{b}^{old} + \mathbf{e}$$

where $\mathbf{e} = \mathbf{t}-\mathbf{a}$.

Now let us try a simple example. We start with a single neuron having a input vector with just two elements.

```
net = newp([-2 2;-2 +2],1);
```

To simplify matters we set the bias equal to 0 and the weights to 1 and -0.8.

```
net.b{1} =  [0];
w = [1 -0.8];
net.IW{1,1} = w;
```

The input target pair is given by

```
p = [1; 2];
t = [1];
```

We can compute the output and error with

```
a = sim(net,p)
a =
     0
e = t-a
e =
     1
```

and finally use the function learnp to find the change in the weights.

```
dw = learnp(w,p,[],[],[],[],e,[],[],[])
dw =
     1     2
```

The new weights, then, are obtained as

```
w = w + dw
w =
    2.0000    1.2000
```

The process of finding new weights (and biases) can be repeated until there are no errors. Note that the perceptron learning rule is guaranteed to converge in a finite number of steps for all problems that can be solved by a perceptron. These include all classification problems that are “linearly separable.” The objects to be classified in such cases can be separated by a single line.

You might want to try demo `nnd4pr`. It allows you to pick new input vectors and apply the learning rule to classify them.

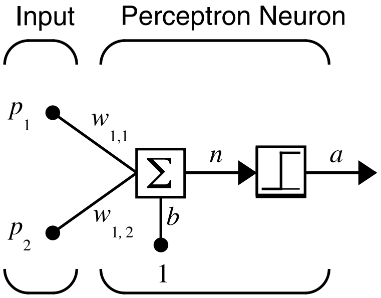
## Training (train)

If sim and learnp are used repeatedly to present inputs to a perceptron, and to change the perceptron weights and biases according to the error, the perceptron will eventually find weight and bias values that solve the problem, given that the perceptron *can* solve it. Each traverse through all of the training input and target vectors is called a *pass*.

The function train carries out such a loop of calculation. In each pass the function train proceeds through the specified sequence of inputs, calculating the output, error and network adjustment for each input vector in the sequence as the inputs are presented.

Note that train  does not guarantee that the resulting network does its job. The new values of **W** and **b** must be checked by computing the network output for each input vector to see if all targets are reached. If a network does not perform successfully it can be trained further by again calling train with the new weights and biases for more training passes, or the problem can be analyzed to see if it is a suitable problem for the perceptron. Problems which are not solvable by the perceptron network are discussed in the “Limitations and Cautions” section.

To illustrate the training procedure, we will work through a simple problem. Consider a one neuron perceptron with a single vector input having two elements.



$$= \mathbf{hardlim}(\mathbf{Wp} + b)$$

This network, and the problem we are about to consider are simple enough that you can follow through what is done with hand calculations if you want. The problem discussed below follows that found in [HDB1996].

Let us suppose we have the following classification problem and would like to solve it with our single vector input, two-element perceptron network.

$$\left\{\mathbf{p}_1 = \begin{bmatrix}2\\2\end{bmatrix}, t_1 = 0\right\} \left\{\mathbf{p}_2 = \begin{bmatrix}1\\-2\end{bmatrix}, t_2 = 1\right\} \left\{\mathbf{p}_3 = \begin{bmatrix}-2\\2\end{bmatrix}, t_3 = 0\right\} \left\{\mathbf{p}_4 = \begin{bmatrix}-1\\1\end{bmatrix}, t_4 = 1\right\}$$

Use the initial weights and bias. We denote the variables at each step of this calculation by using a number in parentheses after the variable. Thus, above, we have the initial values, $\mathbf{W}(0)$ and $b(0)$.

$$\mathbf{W}(0) = \begin{bmatrix}0 & 0\end{bmatrix} \qquad b(0) = 0$$

We start by calculating the perceptron's output $a$ for the first input vector $\mathbf{p_1}$, using the initial weights and bias.

$$a = hardlim(\mathbf{W}(0)\mathbf{p}_1 + b(0))$$

$$= hardlim\left(\begin{bmatrix}0 & 0\end{bmatrix}\begin{bmatrix}2\\2\end{bmatrix} + 0\right) = hardlim(0) = 1$$

The output $a$ does not equal the target value $t_1$, so we use the perceptron rule to find the incremental changes to the weights and biases based on the error.

$$e = t_1 - a = 0 - 1 = -1$$
$$\Delta\mathbf{W} = e\mathbf{p}_1^T = (-1)\begin{bmatrix}2 & 2\end{bmatrix} = \begin{bmatrix}-2 & -2\end{bmatrix}$$
$$\Delta b = e = (-1) = -1$$

You can calculate the new weights and bias using the Perceptron update rules shown previously.

$$\mathbf{W}^{new} = \mathbf{W}^{old} + \mathbf{ep}^T = \begin{bmatrix}0 & 0\end{bmatrix} + \begin{bmatrix}-2 & -2\end{bmatrix} = \begin{bmatrix}-2 & -2\end{bmatrix} = \mathbf{W}(1)$$

$$b^{new} = b^{old} + e = 0 + (-1) = -1 = b(1)$$

Now present the next input vector, $\mathbf{p}_{2.}$ The output is calculated below.

$$a = hardlim(\mathbf{W}(1)\mathbf{p}_2 + b(1))$$

$$= hardlim\left(\begin{bmatrix}-2 & -2\end{bmatrix}\begin{bmatrix}-2\\-2\end{bmatrix} - 1\right) = hardlim(1) = 1$$

On this occasion, the target is 1, so the error is zero. Thus there are no changes in weights or bias, so $\mathbf{W}(2) = \mathbf{W}(1) = \begin{bmatrix}-2 & -2\end{bmatrix}$ and $p(2) = p(1) = -1$

We can continue in this fashion, presenting $\mathbf{p}_3$ next, calculating an output and the error, and making changes in the weights and bias, etc. After making one pass through all of the four inputs, you get the values: $\mathbf{W}(4) = \begin{bmatrix}-3 & -1\end{bmatrix}$ and $b(4) = 0$. To determine if we obtained a satisfactory solution, we must make one pass through all input vectors to see if they all produce the desired target values. This is not true for the 4th input, but the algorithm does converge on the 6th presentation of an input. The final values are:

$\mathbf{W}(6) = \begin{bmatrix}-2 & -3\end{bmatrix}$ and $b(6) = 1$

This concludes our hand calculation. Now, how can we do this using the `train` function?

The following code defines a perceptron like that shown in the previous figure, with initial weights and bias values of 0.

```
net = newp([-2 2;-2 +2],1);
```

Now consider the application of a single input.

```
p =[2; 2];
```

```
having the target
```

```
t =[0];
```

Now set `epochs` to 1, so that `train` will go through the input vectors (only one here) just one time.

```
net.trainParam.epochs = 1;
net = train(net,p,t);
```

The new weights and bias are

```
w =
     -2    -2
```

```
b =
    -1
```

Thus, the initial weights and bias are 0, and after training on only the first vector, they have the values [-2 -2] and -1, just as we hand calculated.

We now apply the second input vector $\mathbf{p}_2$. The output is 1, as it will be until the weights and bias are changed, but now the target is 1, the error will be 0 and the change will be zero. We could proceed in this way, starting from the previous result and applying a new input vector time after time. But we can do this job automatically with `train`.

Now let's apply `train` for one epoch, a single pass through the sequence of all four input vectors. Start with the network definition.

```
net = newp([-2 2;-2 +2],1);
net.trainParam.epochs = 1;
```

The input vectors and targets are

```
p = [[2;2] [1;-2] [-2;2] [-1;1]]
t =[0 1 0 1]
```

Now train the network with

```
net = train(net,p,t);
```

The new weights and bias are

```
w =
    -3    -1
b =
     0
```

Note that this is the same result as we got previously by hand. Finally simulate the trained network for each of the inputs.

```
a = sim(net,p)
a =
    [0]    [0]    [1]    [1]
```

The outputs do not yet equal the targets, so we need to train the network for more than one pass. We will try four epochs. This run gives the following results.

```
TRAINC, Epoch 0/20
TRAINC, Epoch 3/20
TRAINC, Performance goal met.
```

Thus, the network was trained by the time the inputs were presented on the third epoch. (As we know from our hand calculation, the network converges on the presentation of the sixth input vector. This occurs in the middle of the second epoch, but it takes the third epoch to detect the network convergence.) The final weights and bias are

```
w =
    -2    -3
b =
     1
```

The simulated output and errors for the various inputs are

```
a =
            0          1.00              0          1.00
error = [a(1)-t(1) a(2)-t(2) a(3)-t(3) a(4)-t(4)]
error =
            0              0              0              0
```

Thus, we have checked that the training procedure was successful. The network converged and produces the correct target outputs for the four input vectors.

Note that the default training function for networks created with `newp` is `trains`. (You can find this by executing `net.trainFcn`.) This training function applies the perceptron learning rule in its pure form, in that individual input vectors are applied individually in sequence, and corrections to the weights and bias are made after each presentation of an input vector. Thus, perceptron training with `train` will converge in a finite number of steps unless the problem presented can not be solved with a simple perceptron.

The function `train` can be used in various ways by other networks as well. Type `help train` to read more about this basic function.

You may want to try various demonstration programs. For instance, `demop1` illustrates classification and training of a simple perceptron.

# Limitations and Cautions

Perceptron networks should be trained with `adapt`, which presents the input vectors to the network one at a time and makes corrections to the network based on the results of each presentation. Use of adapt in this way guarantees that any linearly separable problem is solved in a finite number of training presentations. Perceptrons can also be trained with the function `train`, which is presented in the next chapter. When `train` is used for perceptrons, it presents the inputs to the network in batches, and makes corrections to the network based on the sum of all the individual corrections. Unfortunately, there is no proof that such a training algorithm converges for perceptrons. On that account the use of `train` for perceptrons is not recommended.

Perceptron networks have several limitations. First, the output values of a perceptron can take on only one of two values (0 or 1) due to the hard-limit transfer function. Second, perceptrons can only classify linearly separable sets of vectors. If a straight line or a plane can be drawn to separate the input vectors into their correct categories, the input vectors are linearly separable. If the vectors are not linearly separable, learning will never reach a point where all vectors are classified properly. Note, however, that it has been proven that if the vectors are linearly separable, perceptrons trained adaptively will always find a solution in finite time. You might want to try `demop6`. It shows the difficulty of trying to classify input vectors that are not linearly separable.

It is only fair, however, to point out that networks with more than one perceptron can be used to solve more difficult problems. For instance, suppose that you have a set of four vectors that you would like to classify into distinct groups, and that two lines can be drawn to separate them. A two neuron network can be found such that its two decision boundaries classify the inputs into four categories. For additional discussion about perceptrons and to examine more complex perceptron problems, see [HDB1996].

## Outliers and the Normalized Perceptron Rule

Long training times can be caused by the presence of an *outlier* input vector whose length is much larger or smaller than the other input vectors. Applying the perceptron learning rule involves adding and subtracting input vectors from the current weights and biases in response to error. Thus, an input vector with large elements can lead to changes in the weights and biases that take a long time for a much smaller input vector to overcome. You might want to try `demop4` to see how an outlier affects the training.

By changing the perceptron learning rule slightly, training times can be made insensitive to extremely large or small outlier input vectors.

Here is the original rule for updating weights:

$$\Delta \mathbf{w} = (t-a)\mathbf{p}^{\mathrm{T}} = e\mathbf{p}^{\mathrm{T}}$$

As shown above, the larger an input vector **p**, the larger its effect on the weight vector **w**. Thus, if an input vector is much larger than other input vectors, the smaller input vectors must be presented many times to have an effect.

The solution is to normalize the rule so that effect of each input vector on the weights is of the same magnitude:

$$\Delta \mathbf{w} = (t-a)\frac{\mathbf{p}^{\mathrm{T}}}{\|\mathbf{p}\|} = e\frac{\mathbf{p}^{\mathrm{T}}}{\|\mathbf{p}\|}$$

The normalized perceptron rule is implemented with the function `learnpn`, which is called exactly like `learnpn`. The normalized perceptron rule function `learnpn` takes slightly more time to execute, but reduces number of epochs considerably if there are outlier input vectors. You might try `demop5` to see how this normalized training rule works.

# Graphical User Interface

## Introduction to the GUI

The graphical user interface (GUI) is designed to be simple and user friendly, but we will go through a simple example to get you started.

In what follows you bring up a GUI **Network/Data Manager** window. This window has its own work area, separate from the more familiar command line workspace. Thus, when using the GUI, you might “export” the GUI results to the (command line) workspace. Similarly you may want to “import” results from the command line workspace to the GUI.

Once the **Network/Data Manager** is up and running, you can create a network, view it, train it, simulate it and export the final results to the workspace. Similarly, you can import data from the workspace for use in the GUI.

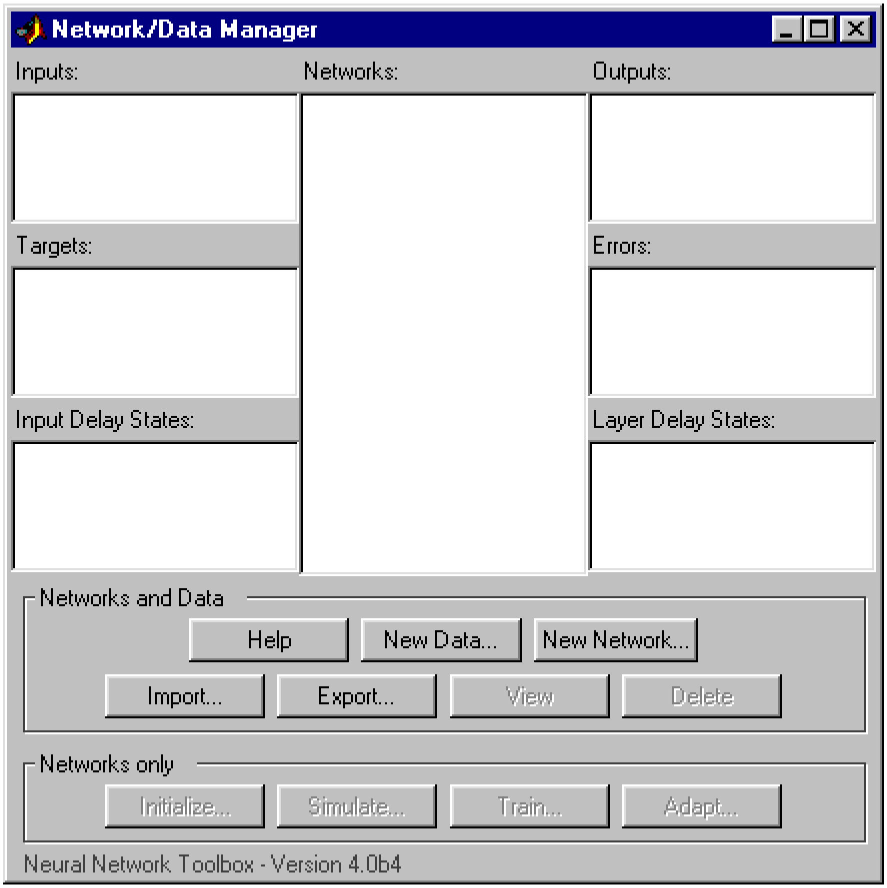
The following example deals with a perceptron network. We go through all the steps of creating a network and show you what you might expect to see as you go along.

## Create a Perceptron Network (nntool)

We create a perceptron network to perform the AND function in this example. It has an input vector `p= [0 0 1 1;0 1 0 1]` and a target vector `t=[0 0 0 1]`. We call the network `ANDNet`. Once created, the network will be trained. We can then save the network, its output, etc., by “exporting” it to the command line.

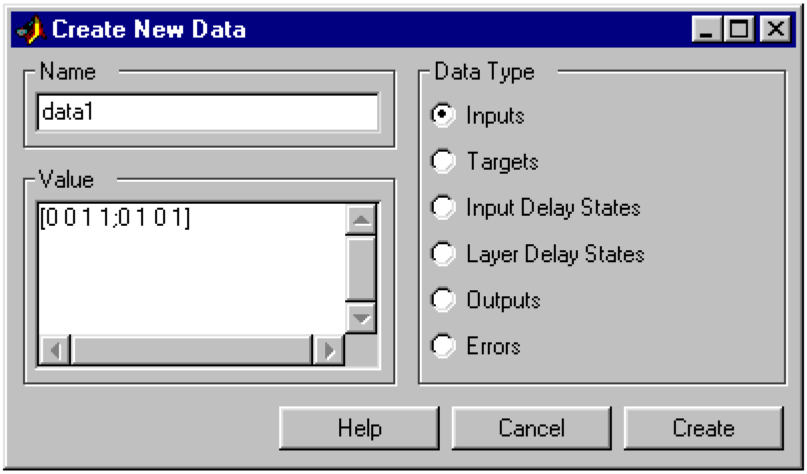
### Input and target

To start, type `nntool`. The following window appears.



Click on **Help** to get started on a new problem and to see descriptions of the buttons and lists.

First, we want to define the network input, which we call p, as having the particular value [0 0 1 1;0 1 0 1]. Thus, the network had a two-element input and four sets of such two-element vectors are presented to it in training. To define this data, click on **New Data**, and a new window, **Create New Data** appears. Set the **Name** to p, the **Value** to [0 0 1 1;0 1 0 1], and make sure that **Data Type** is set to **Inputs**.The **Create New Data** window will then look like this:



Create New Data

Name: data1

Value: [0 0 1 1;0 1 0 1]

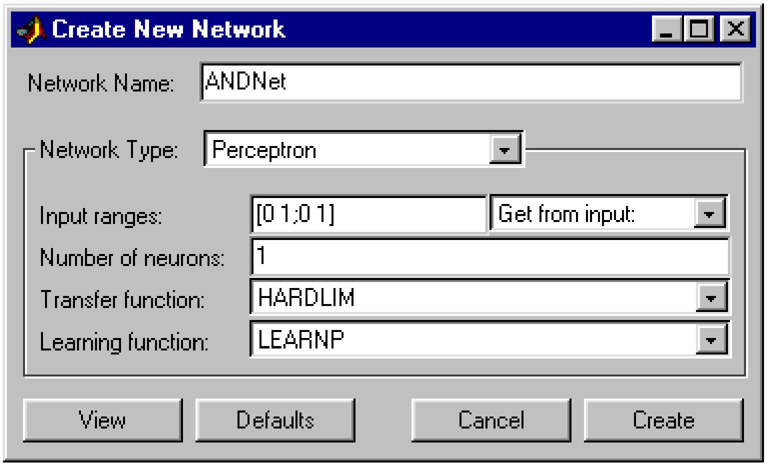
Data Type:
- ● Inputs
- ○ Targets
- ○ Input Delay States
- ○ Layer Delay States
- ○ Outputs
- ○ Errors

Now click **Create** to actually create an input file p. The **Network/Data Manager** window comes up and p shows as an input.

Next we create a network target. Click on **New Data** again, and this time enter the variable name `t`, specify the value `[0 0 0 1]`, and click on **Target** under data type. Again click on **Create** and you will see in the resulting **Network/Data Manager** window that you now have t as a target as well as the previous p as an input.

## Create Network

Now we want to create a new network, which we will call **ANDNet**.To do this, click on **New Network**, and a **CreateNew Network** window appears. Enter `ANDNet` under **Network Name**. Set the **Network Type** to `Perceptron`, for that is the kind of network we want to create. The input ranges can be set by entering numbers in that field, but it is easier to get them from the particular input data that you want to use. To do this, click on the down arrow at the right side of **Input Range**. This pull-down menu shows that you can get the input ranges from the file p if you want. That is what we want to do, so click on p. This should lead to input ranges `[0 1;0 1]`.We want to use a `hardlim` transfer function and a `learnp` learning function, so set those values using the arrows for **Transfer function** and **Learning function** respectively. By now your **Create New Network** window should look like:



Create New Network

Network Name: ANDNet

Network Type: Perceptron

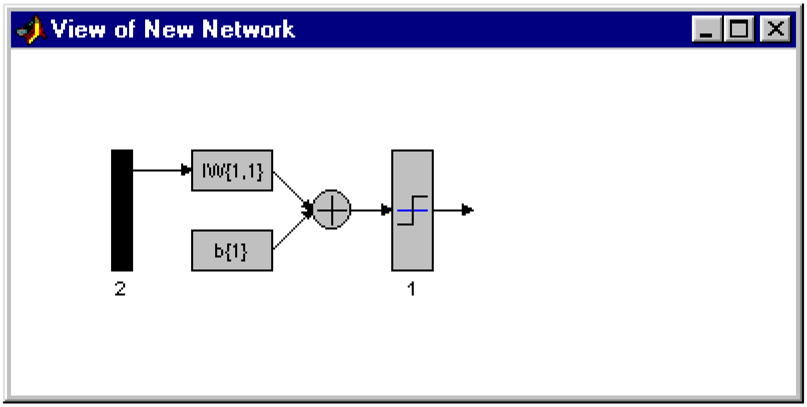
Input ranges: [0 1;0 1] Get from input:

Number of neurons: 1

Transfer function: HARDLIM

Learning function: LEARNP

Next you might look at the network by clicking on **View**. For example:

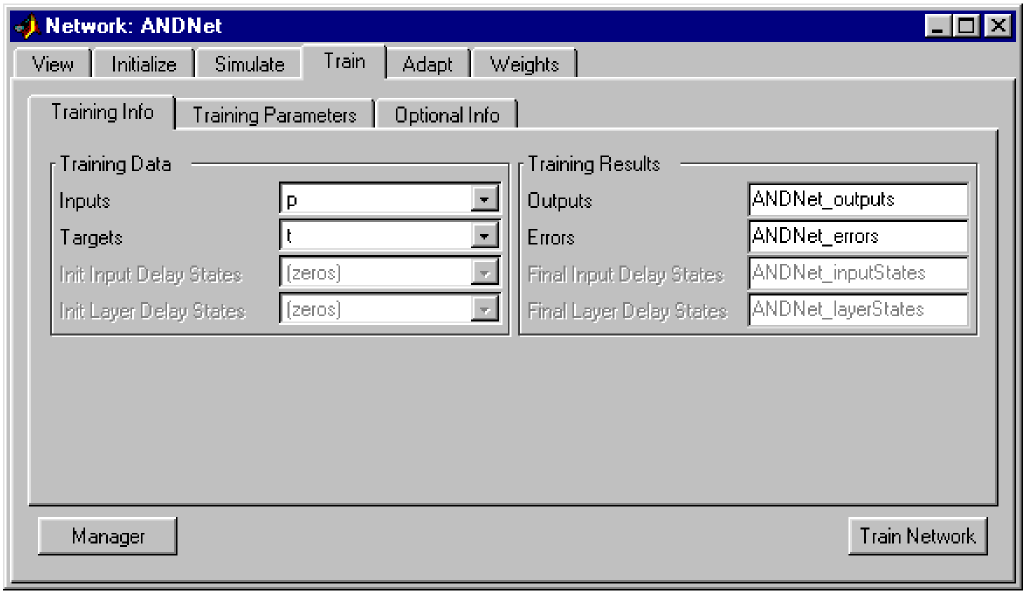


This picture shows that you are about to create a network with a single input (composed of two elements), a `hardlim` transfer function, and a single output. This is the perceptron network that we wanted.

Now click **Create** to generate the network. You will get back the **Network/Data Manager** window. Note that `ANDNet` is now listed as a network.

## Train the Perceptron

To train the network, click on `ANDNet` to highlight it. Then click on **Train**. This leads to a new window labeled **Network:ANDNet**. At this point you can view the network again by clicking on the top tab **Train**. You can also check on the initialization by clicking on the top tab **Initialize**. Now click on the top tab **Train**. Specify the inputs and output by clicking on the left tab **Training Info** and selecting `p` from the pop-down list of inputs and `t` from the pull-down list of targets. The **Network:ANDNet** window should look like:



Training Data

Inputs: p

Targets: t

Init Input Delay States: (zeros)

Init Layer Delay States: (zeros)

Training Results

Outputs: ANDNet_outputs

Errors: ANDNet_errors

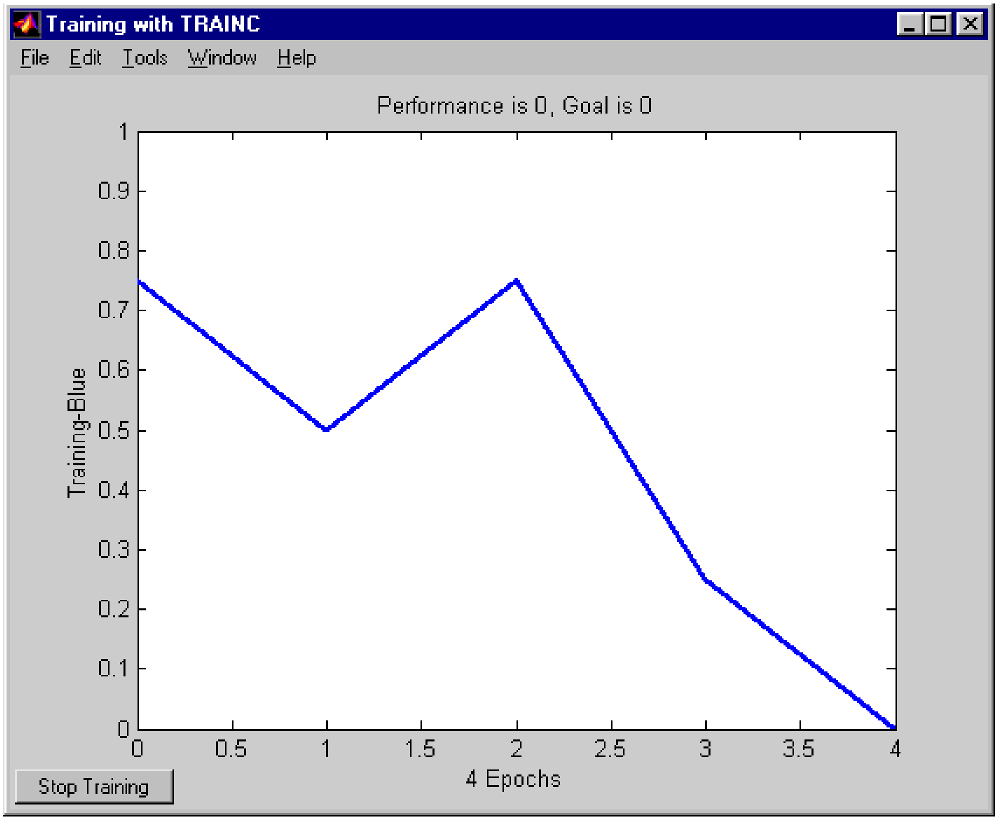
Final Input Delay States: ANDNet_inputStates

Final Layer Delay States: ANDNet_layerStates

Note that the **Training Result Outputs** and **Errors** have the name ANDNet appended to them. This makes them easy to identify later when they are exported to the command line.

While you are here, click on the **Training Parameters** tab. It shows you parameters such as the epochs and error goal. You can change these parameters at this point if you want.

Now click **Train Network** to train the perceptron network. You will see the following training results.



Thus, the network was trained to zero error in four epochs. (Note that other kinds of networks commonly do not train to zero error and their error commonly cover a much larger range. On that account, we plot their errors on a log scale rather than on a linear scale such as that used above for perceptrons.)

You can check that the trained network does indeed give zero error by using the input `p` and simulating the network. To do this, get to the **Network/Data Manager** window and click on **Network Only: Simulate)**. This will bring up the **Network:ANDNet** window. Click there on **Simulate**. Now use the **Input** pull-down menu to specify `p` as the input, and label the output as `ANDNet_outputsSim` to distinguish it from the training output. Now click **Simulate Network** in the lower right corner. Look at the **Network/Data Manager** and you will see a new variable in the output: `ANDNet_outputsSim`.

Double-click on it and a small window **Data:ANDNet_outputsSim** appears with the value

```
[0 0 0 1]
```

Thus, the network does perform the AND of the inputs, giving a 1 as an output only in this last case, when both inputs are 1.

## Export Perceptron Results to Workspace

To export the network outputs and errors to the MATLAB command line workspace, click in the lower left of the **Network:ANDNet** window to go back to the **Network/Data Manager**. Note that the output and error for the ANDNet are listed in the **Outputs and Error** lists on the right side. Next click on **Export** This will give you an **Export** or **Save from Network/Data Manager** window. Click on ANDNet_outputs and ANDNet_errors to highlight them, and then click the **Export** button. These two variables now should be in the command line workspace. To check this, go to the command line and type who to see all the defined variables. The result should be

```
who
Your variables are:
ANDNet_errors       ANDNet_outputs
```

You might type ANDNet_outputs and ANDNet_errors to obtain the following

```
ANDNet_outputs =
0     0     0     1
```

and

```
ANDNet_errors =
0     0     0     0.
```

You can export p, t, and ANDNet in a similar way. You might do this and check with who to make sure that they got to the command line.

Now that ANDNet is exported you can view the network description and examine the network weight matrix. For instance, the command

```
ANDNet.iw{1,1}
```

gives

```
ans =
   2 1
```

Similarly,

```
ANDNet.b{1}
```

yields

```
   ans =
   -3.
```

## Clear Network/Data Window

You can clear the **Network/Data Manager** window by highlighting a variable such as `p` and clicking the **Delete** button until all entries in the list boxes are gone. By doing this, we start from clean slate.

Alternatively, you can quit MATLAB. A restart with a new MATLAB, followed by `nntool`, gives a clean **Network/Data Manager** window.

Recall however, that we exported `p`, `t`, etc., to the command line from the perceptron example. They are still there for your use even after you clear the **Network/Data Manager**.

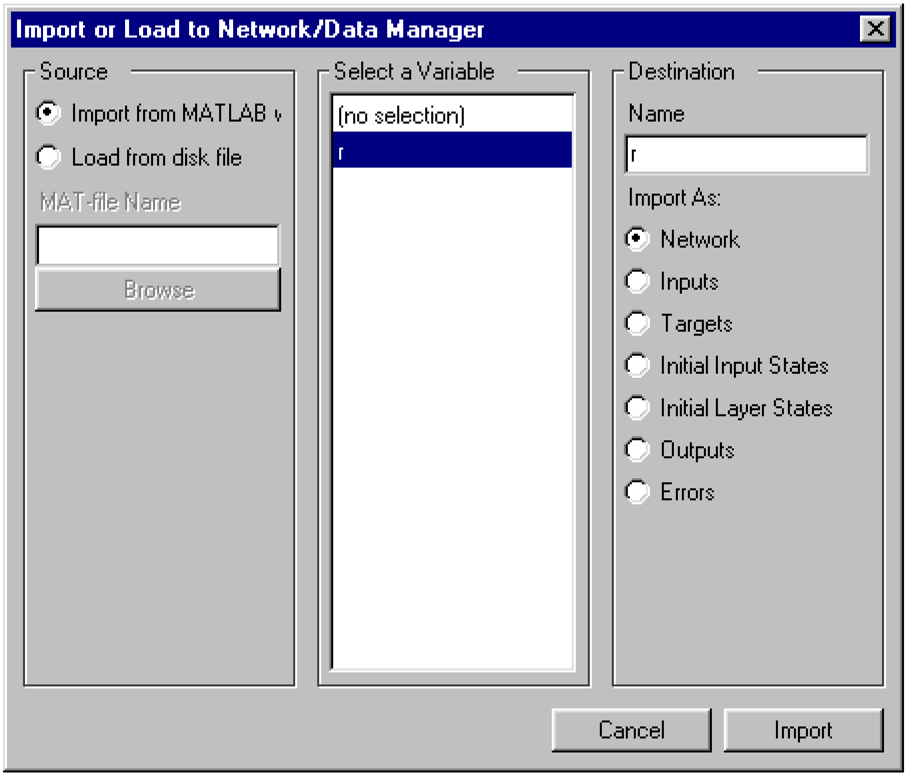
## Importing from the Command Line

To make thing simple, quit MATLAB. Start it again, and type `nntool` to begin a new session.

Create a new vector.

```
   r= [0; 1; 2; 3]
   r =
        0
        1
        2
        3
```

Now click on **Import**, and set the destination **Name** to `R` (to distinguish between the variable named at the command line and the variable in the GUI). You will have a window that looks like this



Import or Load to Network/Data Manager

Source: ● Import from MATLAB w ○ Load from disk file

MAT-file Name:

Select a Variable: (no selection), r (selected)

Destination Name: r

Import As: ● Network ○ Inputs ○ Targets ○ Initial Input States ○ Initial Layer States ○ Outputs ○ Errors

Now click **Import** and verify by looking at the **Network/DAta Manager** that the variable R is there as an input.

## Save a Variable to a File and Load It Later

Bring up the **Network/Data Manager** and click on **New Network**. Set the name to mynet. Click on **Create.** The network name mynet should appear in the **Network/Data Manager.** In this same manager window click on **Export**. Select mynet in the variable list of the **Export or Save** window and click on **Save**. This leads to the **Save to a MAT file** window. Save to a file mynetfile.

Now lets get rid of mynet in the GUI and retrieve it from the saved file. First go to the **Data/Network Manager**, highlight mynet, and click **Delete**. Next click on **Import**. This brings up the **Import or Load to Network/Data Manager** window. Select the **Load from Disk** button and type `mynetfile` as the **MAT-file Name**. Now click on **Browse**. This brings up the **Select MAT file** window with `mynetfile` as an option that you can select as a variable to be imported. Highlight `mynetfile`, press **Open**, and you return to the **Import or Load to Network/Data Manager** window. On the **Import As** list, select **Network**. Highlight `mynet` and lick on **Load** to bring `mynet` to the GUI. Now mynet is back in the GUI **Network/Data Manager** window.

# Summary

Perceptrons are useful as classifiers. They can classify linearly separable input vectors very well. Convergence is guaranteed in a finite number of steps providing the perceptron can solve the problem.

The design of a perceptron network is constrained completely by the problem to be solved. Perceptrons have a single layer of hard-limit neurons. The number of network inputs and the number of neurons in the layer are constrained by the number of inputs and outputs required by the problem.

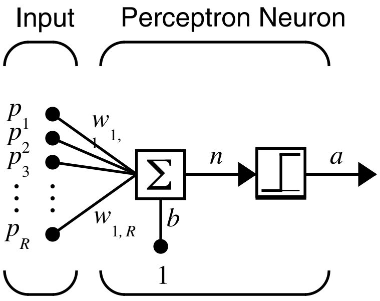
Training time is sensitive to outliers, but outlier input vectors do not stop the network from finding a solution.

Single-layer perceptrons can solve problems only when data is linearly separable. This is seldom the case. One solution to this difficulty is to use a preprocessing method that results in linearly separable vectors. Or you might use multiple perceptrons in multiple layers. Alternatively, you can use other kinds of networks such as linear networks or backpropagation networks, which can classify nonlinearly separable input vectors.

A graphical user interface can be used to create networks and data, train the networks, and export the networks and data to the command line workspace.

## Figures and Equations

### Perceptron Neuron

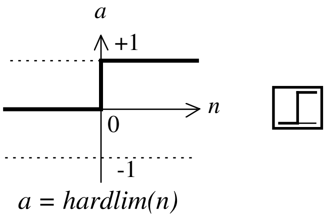


$$a = \mathbf{hardlim}(\mathbf{Wp} + b)$$

Where...

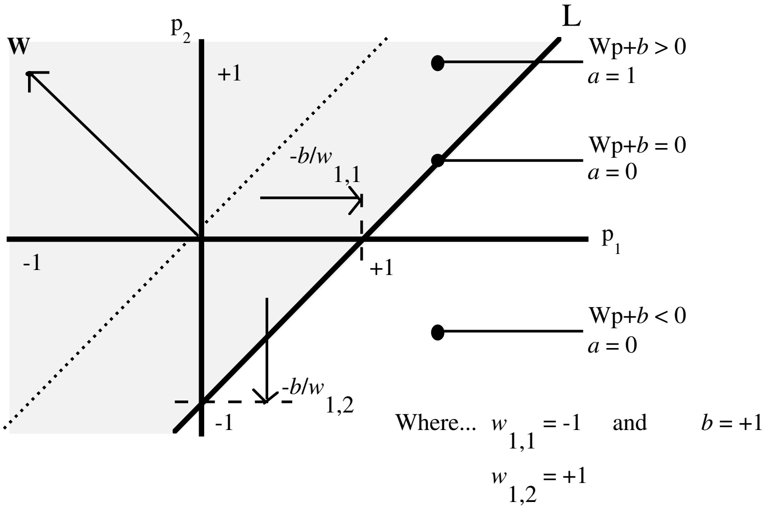
$R$ = number of elements in input vector

### Perceptron Transfer Function, `hardlim`

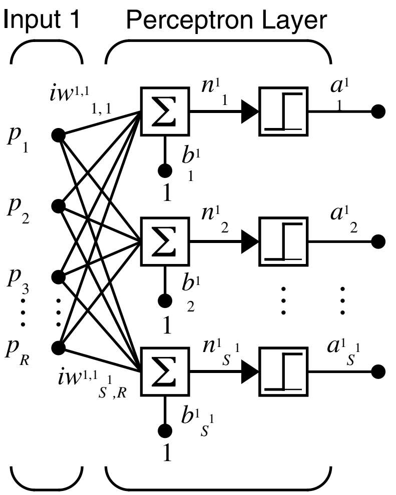


Hard-Limit Transfer Function

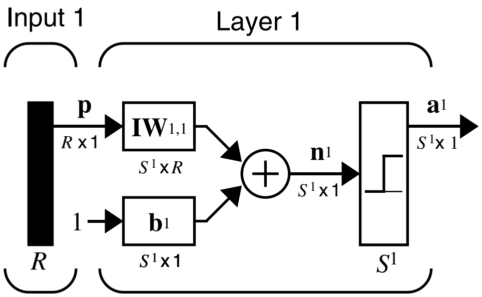
### Decision Boundary



## Perceptron Architecture



$$\mathbf{a}^1 = \mathbf{hardlim}(\mathbf{IW}^{1,1}\mathbf{p}^1 + \mathbf{b}^1)$$



$$\mathbf{a}^1 = \mathbf{hardlim}(\mathbf{IW}^{1,1}\mathbf{p}^1 + \mathbf{b}^1)$$

Where...

$R$ = number of elements in Input

$S^1$ = number of neurons in layer 1

## The Perceptron Learning Rule

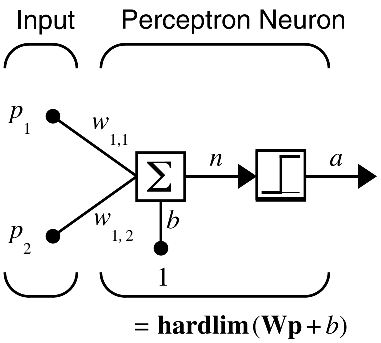
$$\mathbf{W}^{new} = \mathbf{W}^{old} + \mathbf{e}\mathbf{p}^T$$

$$\mathbf{b}^{new} = \mathbf{b}^{old} + \mathbf{e}$$

where

$$\mathbf{e} = \mathbf{t} - \mathbf{a}$$

### One Perceptron Neuron



## New Functions

This chapter introduces the following new functions.

| Function | Description |
|---|---|
| `hardlim` | A hard limit transfer function |
| `initzero` | Zero weight/bias initialization function |
| `dotprod` | Dot product weight function |
| `newp` | Creates a new perceptron network. |
| `sim` | Simulates a neural network. |
| `init` | Initializes a neural network |
| `learnp` | Perceptron learning function |
| `learnpn` | Normalized perceptron learning function |
| `nntool` | Starts the Graphical User Interface (GUI) |

4

# Linear Filters

# Introduction

The linear networks discussed in this chapter are similar to the perceptron, but their transfer function is linear rather than hard-limiting. This allows their outputs to take on any value, whereas the perceptron output is limited to either 0 or 1. Linear networks, like the perceptron, can only solve linearly separable problems.

Here we will design a linear network that, when presented with a set of given input vectors, produces outputs of corresponding target vectors. For each input vector we can calculate the network's output vector. The difference between an output vector and its target vector is the error. We would like to find values for the network weights and biases such that the sum of the squares of the errors is minimized or below a specific value. This problem is manageable because linear systems have a single error minimum. In most cases, we can calculate a linear network directly, such that its error is a minimum for the given input vectors and targets vectors. In other cases, numerical problems prohibit direct calculation. Fortunately, we can always train the network to have a minimum error by using the Least Mean Squares (Widrow-Hoff) algorithm.

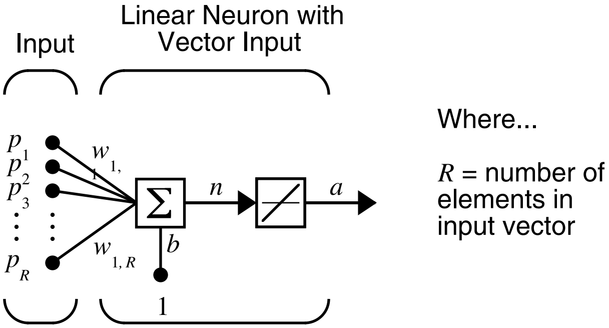
Note that the use of linear filters in adaptive systems is discussed in Chapter 10.

This chapter introduces `newlin`, a function that creates a linear layer, and `newlind`, a function that designs a linear layer for a specific purpose.

You can type `help linnet` to see a list of linear network functions, demonstrations, and applications.

# Neuron Model

A linear neuron with $R$ inputs is shown below.

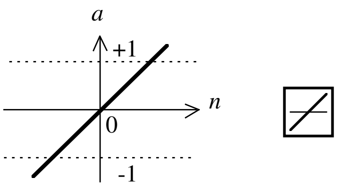


Where...

$R$ = number of elements in input vector

$a = purelin(\mathbf{Wp} + b)$

This network has the same basic structure as the perceptron. The only difference is that the linear neuron uses a linear transfer function, which we will give the name `purelin`.



$a = purelin(n)$

Linear Transfer Function

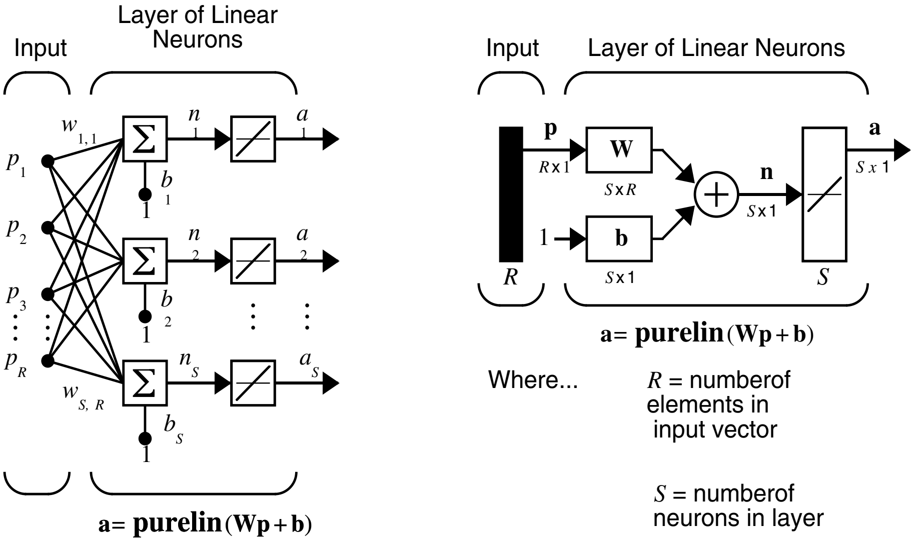
The linear transfer function calculates the neuron's output by simply returning the value passed to it.

$$a = purelin(n) = purelin(\mathbf{Wp} + b) = \mathbf{Wp} + b$$

This neuron can be trained to learn an affine function of its inputs, or to find a linear approximation to a nonlinear function. A linear network cannot, of course, be made to perform a nonlinear computation.

# Network Architecture

The linear network shown below has one layer of $S$ neurons connected to $R$ inputs through a matrix of weights **W**.

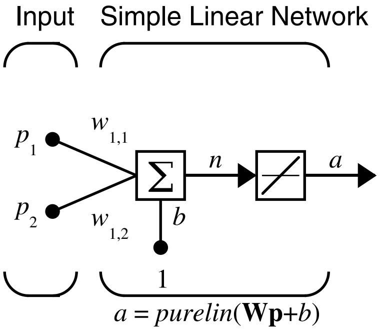


Note that the figure on the right defines an $S$-length output vector **a**.

We have shown a single-layer linear network. However, this network is just as capable as multilayer linear networks. For every multilayer linear network, there is an equivalent single-layer linear network.

## Creating a Linear Neuron (newlin)

Consider a single linear neuron with two inputs. The diagram for this network is shown below.

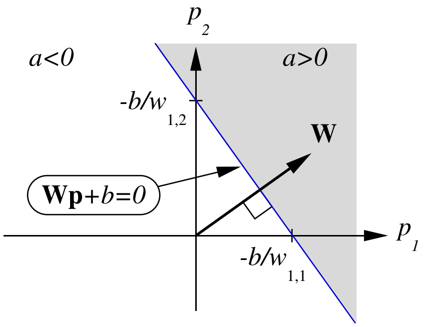


The weight matrix **W** in this case has only one row. The network output is:

$$a = purelin(n) = purelin(\mathbf{Wp} + b) = \mathbf{Wp} + b \qquad \text{or}$$

$$a = w_{1,1}p_1 + w_{1,2}p_2 + b$$

Like the perceptron, the linear network has a *decision boundary* that is determined by the input vectors for which the net input $n$ is zero. For $n = 0$ the equation $\mathbf{Wp} + b = 0$ specifies such a decision boundary as shown below (adapted with thanks from [HDB96]).



Input vectors in the upper right gray area will lead to an output greater than 0. Input vectors in the lower left white area will lead to an output less than 0. Thus, the linear network can be used to classify objects into two categories. However, it can classify in this way only if the objects are linearly separable. Thus, the linear network has the same limitation as the perceptron.

We can create a network like that shown above with the command

```
net = newlin( [-1 1; -1 1],1);
```

The first matrix of arguments specify the range of the two scalar inputs. The last argument, 1, says that the network has a single output.

The network weights and biases are set to zero by default. You can see the current values with the commands

```
W = net.IW{1,1}
W =
     0     0
```

and

```
b= net.b{1}
b =
     0
```

However, you can give the weights any value that you want, such as 2 and 3 respectively, with

```
net.IW{1,1} = [2 3];
W = net.IW{1,1}
W =
     2     3
```

The bias can be set and checked in the same way.

```
net.b{1} =[-4];
b = net.b{1}
b =
     -4
```

You can simulate the linear network for a particular input vector. Try

```
p = [5;6];
```

Now you can find the network output with the function `sim`.

```
a = sim(net,p)
a =
    24
```

To summarize, you can create a linear network with newlin, adjust its elements as you want, and simulate it with sim. You can find more about newlin by typing help newlin.

## Mean Square Error

Like the perceptron learning rule, the least mean square error (LMS) algorithm is an example of supervised training, in which the learning rule is provided with a set of examples of desired network behavior:

$$\{\mathbf{p}_1, \mathbf{t}_1\}, \{\mathbf{p}_2, \mathbf{t}_2\}, \ldots, \{\mathbf{p}_Q, \mathbf{t}_Q\}$$

Here $\mathbf{p}_q$ is an input to the network, and $\mathbf{t}_q$ is the corresponding target output. As each input is applied to the network, the network output is compared to the target. The error is calculated as the difference between the target output and the network output. We want to minimize the average of the sum of these errors.

$$mse = \frac{1}{Q}\sum_{k=1}^{Q} e(k)^2 = \frac{1}{Q}\sum_{k=1}^{Q} (t(k) - a(k))^2$$

The LMS algorithm adjusts the weights and biases of the linear network so as to minimize this mean square error.

Fortunately, the mean square error performance index for the linear network is a quadratic function. Thus, the performance index will either have one global minimum, a weak minimum or no minimum, depending on the characteristics of the input vectors. Specifically, the characteristics of the input vectors determine whether or not a unique solution exists.

You can find more about this topic in Chapter 10 of [HDB96].

# Linear System Design (newlind)

Unlike most other network architectures, linear networks can be designed directly if input/target vector pairs are known. Specific network values for weights and biases can be obtained to minimize the mean square error by using the function `newlind`.

Suppose that the inputs and targets are

```
P = [1 2 3];
T= [2.0 4.1 5.9];
```

Now you can design a network.

```
net = newlind(P,T);
```

You can simulate the network behavior to check that the design was done properly.

```
Y = sim(net,P)
Y =
    2.0500    4.0000    5.9500
```

Note that the network outputs are quite close to the desired targets.

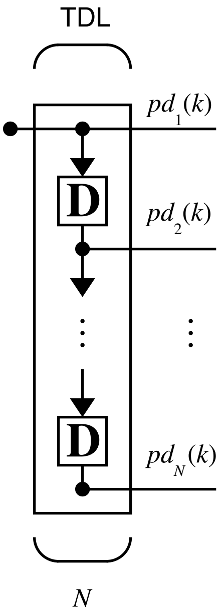
You might try `demolin1`. It shows error surfaces for a particular problem, illustrates the design and plots the designed solution.

The function `newlind` can also be used to design linear networks having delays in the input. Such networks are discussed later in this chapter. First, however, we need to discuss delays.

# Linear Networks with Delays

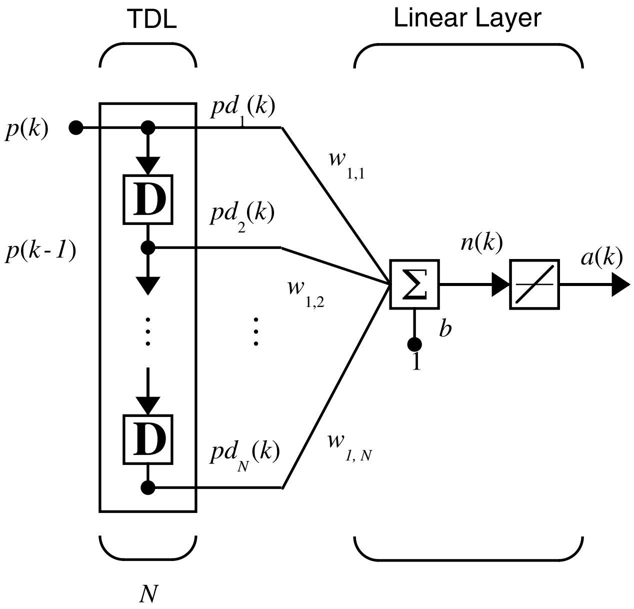
## Tapped Delay Line

We need a new component, the tapped delay line, to make full use of the linear network. Such a delay line is shown below. There the input signal enters from the left, and passes through *N*-1 delays. The output of the tapped delay line (TDL) is an *N*-dimensional vector, made up of the input signal at the current time, the previous input signal, etc.



## Linear Filter

We can combine a tapped delay line with an linear network to create the *linear* filter shown below.



The output of the filter is given by

$$a(k) = purelin(\mathbf{Wp} + b) = \sum_{i=1}^{R} w_{1,\,i}a(k-i+1) + b$$

The network shown above is referred to in the digital signal-processing field as a finite impulse response (FIR) filter [WiSt85]. Let us take a look at the code that we use to generate and simulate such a specific network.

Suppose that we want a linear layer that outputs the sequence `T` given the sequence P and two initial input delay states `Pi`.

```
P = {1 2 1 3 3 2};
Pi = {1 3};
T = {5 6 4 20 7 8};
```

You can use `newlind` to design a network with delays to give the appropriate outputs for the inputs. The delay initial outputs are supplied as a third argument as shown below.

```
net = newlind(P,T,Pi);
```

Now we obtain the output of the designed network with

```
Y = sim(net,P,Pi)
```

to give

```
Y = [2.73]    [10.54]    [5.01]    [14.95]    [10.78]    [5.98]
```

As you can see, the network outputs are not exactly equal to the targets, but they are reasonably close, and in any case, the mean square error is minimized.

# LMS Algorithm (learnwh)

The LMS algorithm or Widrow-Hoff learning algorithm, is based on an approximate steepest descent procedure. Here again, linear networks are trained on examples of correct behavior.

Widrow and Hoff had the insight that they could estimate the mean square error by using the squared error at each iteration. If we take the partial derivative of the squared error with respect to the weights and biases at the $k$th iteration we have

$$\frac{\partial e^2(k)}{\partial w_{1,j}} = 2e(k)\frac{\partial e(k)}{\partial w_{1,j}}$$

for $j = 1, 2, \ldots, R$ and

$$\frac{\partial e^2(k)}{\partial b} = 2e(k)\frac{\partial e(k)}{\partial b}$$

Next look at the partial derivative with respect to the error.

$$\frac{\partial e(k)}{\partial w_{1,j}} = \frac{\partial[t(k) - a(k)]}{\partial w_{1,j}} = \frac{\partial}{\partial w_{1,j}}[t(k) - (\mathbf{Wp}(k) + b)] \quad \text{or}$$

$$\frac{\partial e(k)}{\partial w_{1,j}} = \frac{\partial}{\partial w_{1,j}}\left[t(k) - \left(\sum_{i=1}^{R} w_{1,i}p_i(k) + b\right)\right]$$

Here $p_i$(k) is the $i$th element of the input vector at the $k$th iteration.

Similarly,

$$\frac{\partial e(k)}{\partial w_{1,j}} = -p_j(k)$$

This can be simplified to:

$$\frac{\partial e(k)}{\partial w_{1,j}} = -p_j(k) \quad \text{and}$$

$$\frac{\partial e(k)}{\partial b} = -1$$

Finally, the change to the weight matrix and the bias will be

$2\alpha e(k)\mathbf{p}(k)$ and $2\alpha e(k)$.

These two equations form the basis of the Widrow-Hoff (LMS) learning algorithm.

These results can be extended to the case of multiple neurons, and written in matrix form as

$$\mathbf{W}(k+1) = \mathbf{W}(k) + 2\alpha\mathbf{e}(k)\mathbf{p}^T(k)$$

$$\mathbf{b}(k+1) = \mathbf{b}(k) + 2\alpha\mathbf{e}(k)$$

Here the error **e** and the bias **b** are vectors and $\alpha$ is a *learning rate*. If $\alpha$ is large, learning occurs quickly, but if it is too large it may lead to instability and errors may even increase. To ensure stable learning, the learning rate must be less than the reciprocal of the largest eigenvalue of the correlation matrix $\mathbf{p}^T\mathbf{p}$ of the input vectors.

You might want to read some of Chapter 10 of [HDB96] for more information about the LMS algorithm and its convergence.

Fortunately we have a toolbox function `learnwh` that does all of the calculation for us. It calculates the change in weights as

```
dw = lr*e*p'
```

and the bias change as

```
db = lr*e
```

The constant 2, shown a few lines above, has been absorbed into the code learning rate `lr`. The function `maxlinlr` calculates this maximum stable learning rate `lr` as 0.999 * `P'*P`.

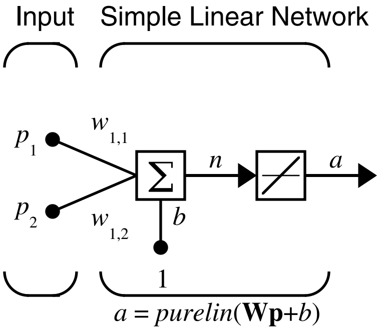
Type `help learnwh` and `help maxlinlr` for more details about these two functions.

# Linear Classification (train)

Linear networks can be trained to perform linear classification with the function `train`. This function applies each vector of a set of input vectors and calculates the network weight and bias increments due to each of the inputs according to `learnp`. Then the network is adjusted with the sum of all these corrections. We will call each pass through the input vectors an epoch. This contrasts with `adapt`, discussed in “Adaptive Filters and Adaptive Training” in Chapter 10, which adjusts weights for each input vector as it is presented.

Finally, `train` applies the inputs to the new network, calculates the outputs, compares them to the associated targets, and calculates a mean square error. If the error goal is met, or if the maximum number of epochs is reached, the training is stopped and `train` returns the new network and a training record. Otherwise `train` goes through another epoch. Fortunately, the LMS algorithm converges when this procedure is executed.

To illustrate this procedure, we will work through a simple problem. Consider the linear network introduced earlier in this chapter.



Next suppose we have the classification problem presented in “Linear Filters” in Chapter 4.

$$\left\{\mathbf{p}_1 = \begin{bmatrix}2\\2\end{bmatrix}, t_1 = 0\right\} \left\{\mathbf{p}_2 = \begin{bmatrix}1\\-2\end{bmatrix}, t_2 = 1\right\} \left\{\mathbf{p}_3 = \begin{bmatrix}-2\\2\end{bmatrix}, t_3 = 0\right\} \left\{\mathbf{p}_4 = \begin{bmatrix}-1\\1\end{bmatrix}, t_4 = 1\right\}$$

Here we have four input vectors, and we would like a network that produces the output corresponding to each input vector when that vector is presented.

We will use train to get the weights and biases for a network that produces the correct targets for each input vector. The initial weights and bias for the new network will be 0 by default. We will set the error goal to 0.1 rather than accept its default of 0.

```
P = [2 1 -2 -1;2 -2 2 1];
t = [0 1 0 1];
net = newlin( [-2 2; -2 2],1);
net.trainParam.goal= 0.1;
[net, tr] = train(net,P,t);
```

The problem runs, producing the following training record.

```
TRAINB, Epoch 0/100, MSE 0.5/0.1.
TRAINB, Epoch 25/100, MSE 0.181122/0.1.
TRAINB, Epoch 50/100, MSE 0.111233/0.1.
TRAINB, Epoch 64/100, MSE 0.0999066/0.1.
TRAINB, Performance goal met.
```

Thus, the performance goal is met in 64 epochs. The new weights and bias are

```
weights = net.iw{1,1}
weights =
   -0.0615   -0.2194
bias = net.b(1)
bias =
    [0.5899]
```

We can simulate the new network as shown below.

```
A = sim(net, p)
A =
    0.0282    0.9672    0.2741    0.4320,
```

We also can calculate the error.

```
err = t - sim(net,P)
err =
   -0.0282    0.0328   -0.2741    0.5680
```

Note that the targets are not realized exactly. The problem would have run longer in an attempt to get perfect results had we chosen a smaller error goal, but in this problem it is not possible to obtain a goal of 0. The network is limited in its capability. See “Limitations and Cautions” at the end of this chapter for examples of various limitations.

This demonstration program `demolin2` shows the training of a linear neuron, and plots the weight trajectory and error during training

You also might try running the demonstration program `nnd10lc`. It addresses a classic and historically interesting problem, shows how a network can be trained to classify various patterns, and how the trained network responds when noisy patterns are presented.

# Limitations and Cautions

Linear networks may only learn linear relationships between input and output vectors. Thus, they cannot find solutions to some problems. However, even if a perfect solution does not exist, the linear network will minimize the sum of squared errors if the learning rate `lr` is sufficiently small. The network will find as close a solution as is possible given the linear nature of the network's architecture. This property holds because the error surface of a linear network is a multidimensional parabola. Since parabolas have only one minimum, a gradient descent algorithm (such as the LMS rule) must produce a solution at that minimum.

Linear networks have other various limitations. Some of them are discussed below.

## Overdetermined Systems

Consider an overdetermined system. Suppose that we have a network to be trained with four 1-element input vectors and four targets. A perfect solution to $wp + b = t$ for each of the inputs may not exist, for there are four constraining equations and only one weight and one bias to adjust. However, the LMS rule will still minimize the error. You might try `demolin4` to see how this is done.

## Underdetermined Systems

Consider a single linear neuron with one input. This time, in `demolin5`, we will train it on only one one-element input vector and its one-element target vector:

```
P = [+1.0];
T = [+0.5];
```

Note that while there is only one constraint arising from the single input/target pair, there are two variables, the weight and the bias. Having more variables than constraints results in an underdetermined problem with an infinite number of solutions. You can try `demoin5` to explore this topic.

## Linearly Dependent Vectors

Normally it is a straightforward job to determine whether or not a linear network can solve a problem. Commonly, if a linear network has at least as many degrees of freedom ($S*R+S$ = number of weights and biases) as constraints ($Q$ = pairs of input/target vectors), then the network can solve the problem. This is true except when the input vectors are linearly dependent and they are applied to a network without biases. In this case, as shown with demonstration script `demolin6`, the network cannot solve the problem with zero error. You might want to try `demolin6`.

## Too Large a Learning Rate

A linear network can always be trained with the Widrow-Hoff rule to find the minimum error solution for its weights and biases, as long as the learning rate is small enough. Demonstration script `demolin7` shows what happens when a neuron with one input and a bias is trained with a learning rate larger than that recommended by `maxlinlr`. The network is trained with two different learning rates to show the results of using too large a learning rate.

## Summary

Single-layer linear networks can perform linear function approximation or pattern association.

Single-layer linear networks can be designed directly or trained with the Widrow-Hoff rule to find a minimum error solution. In addition, linear networks can be trained adaptively allowing the network to track changes in its environment.

The design of a single-layer linear network is constrained completely by the problem to be solved. The number of network inputs and the number of neurons in the layer are determined by the number of inputs and outputs required by the problem.

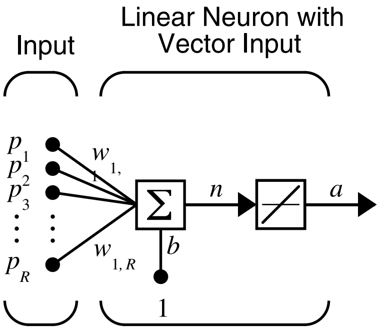
Multiple layers in a linear network do not result in a more powerful network, so the single layer is not a limitation. However, linear networks can solve only linear problems.

Nonlinear relationships between inputs and targets cannot be represented exactly by a linear network. The networks discussed in this chapter make a linear approximation with the minimum sum-squared error.

If the relationship between inputs and targets is linear or a linear approximation is desired, then linear networks are made for the job. Otherwise, backpropagation may be a good alternative.

# Figures and Equations

## Linear Neuron

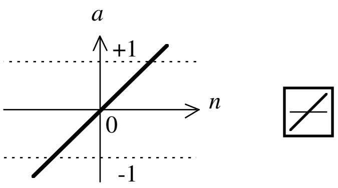


$a = purelin(\mathbf{Wp} + b)$

Where...

*R* = number of elements in input vector

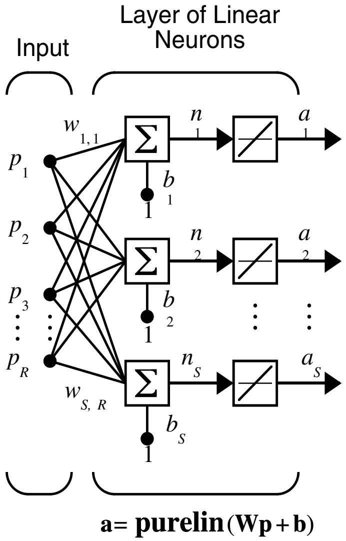
## Purelin Transfer Function

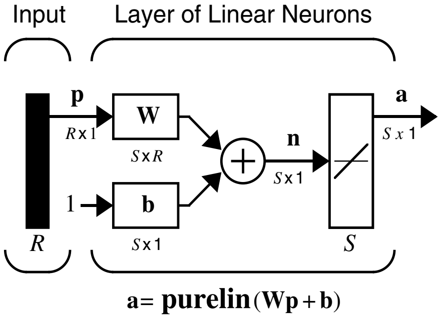


$a = purelin(n)$

Linear Transfer Function

## Linear Network Layer



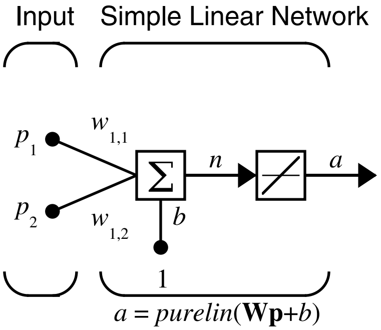


Where...

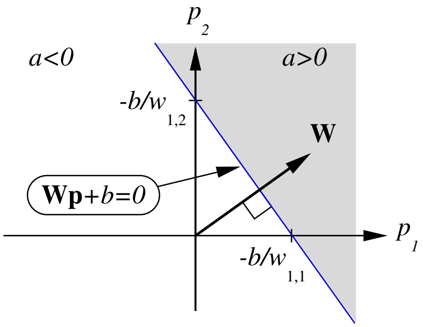
$R$ = numberof elements in input vector

$S$ = numberof neurons in layer

## Simple Linear Network



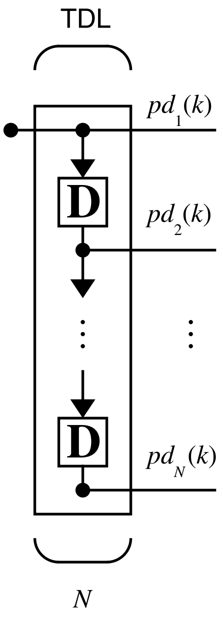
## Decision Boundary



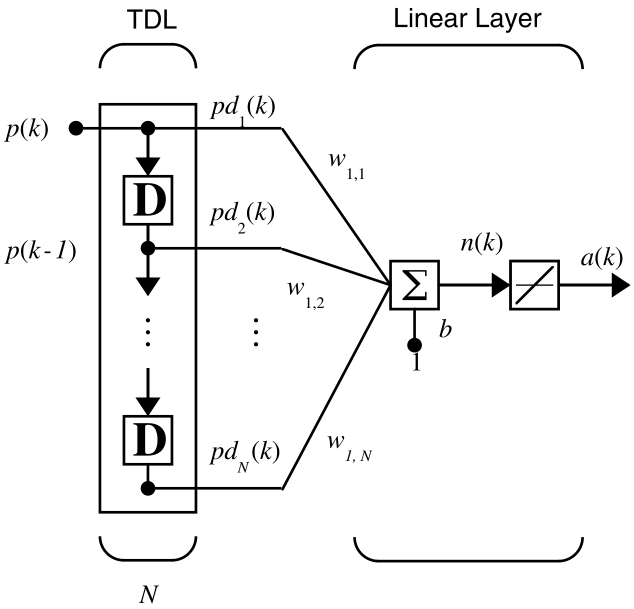
## Mean Square Error

$$mse = \frac{1}{Q}\sum_{k=1}^{Q} e(k)^2 = \frac{1}{Q}\sum_{k=1}^{Q} (t(k)-a(k))^2$$

Tapped Delay Line



### Linear Filter



### LMS (Widrow-Hoff) Algorithm

$$\mathbf{W}(k+1) = \mathbf{W}(k) + 2\alpha\mathbf{e}(k)\mathbf{p}^T(k)$$

$$\mathbf{b}(k+1) = \mathbf{b}(k) + 2\alpha\mathbf{e}(k) .$$

## New Functions

This chapter introduces the following new functions.

| Function | Description |
|---|---|
| `newlin` | Creates a linear layer. |
| `newlind` | Design a linear layer. |

| Function | Description |
|---|---|
| `learnwh` | Widrow-Hoff weight/bias learning rule. |
| `purelin` | A linear transfer function. |

5

# Backpropagation

# Introduction

Backpropagation was created by generalizing the Widrow-Hoff learning rule to multiple-layer networks and nonlinear differentiable transfer functions. Input vectors and the corresponding target vectors are used to train a network until it can approximate a function, associate input vectors with specific output vectors, or classify input vectors in an appropriate way as defined by you. Networks with biases, a sigmoid layer, and a linear output layer are capable of approximating any function with a finite number of discontinuities.

Standard backpropagation is a gradient descent algorithm, as is the Widrow-Hoff learning rule, in which the network weights are moved along the negative of the gradient of the performance function. The term *backpropagation* refers to the manner in which the gradient is computed for nonlinear multilayer networks. There are a number of variations on the basic algorithm that are based on other standard optimization techniques, such as conjugate gradient and Newton methods. The Neural Network Toolbox implements a number of these variations. This chapter explains how to use each of these routines and discusses the advantages and disadvantages of each.

Properly trained backpropagation networks tend to give reasonable answers when presented with inputs that they have never seen. Typically, a new input leads to an output similar to the correct output for input vectors used in training that are similar to the new input being presented. This generalization property makes it possible to train a network on a representative set of input/target pairs and get good results without training the network on all possible input/output pairs. There are two features of the Neural Network Toolbox that are designed to improve network generalization - regularization and early stopping. These features and their use are discussed later in this chapter.

This chapter also discusses preprocessing and postprocessing techniques, which can improve the efficiency of network training.

Before beginning this chapter you may want to read a basic reference on backpropagation, such as D.E Rumelhart, G.E. Hinton, R.J. Williams, “Learning internal representations by error propagation,” D. Rumelhart and J. McClelland, editors. *Parallel Data Processing*, Vol.1, Chapter 8, the M.I.T. Press, Cambridge, MA 1986 pp. 318-362. This subject is also covered in detail in Chapters 11 and 12 of M.T. Hagan, H.B. Demuth, M.H. Beale, *Neural Network Design*, PWS Publishing Company, Boston, MA 1996.

The primary objective of this chapter is to explain how to use the backpropagation training functions in the toolbox to train feedforward neural networks to solve specific problems. There are generally four steps in the training process:

1 Assemble the training data

2 Create the network object

3 Train the network

4 Simulate the network response to new inputs

This chapter discusses a number of different training functions, but in using each function we generally follow these four steps.

The next section, “Fundamentals”, describes the basic feedforward network structure and demonstrates how to create a feedforward network object. Then the simulation and training of the network objects are presented.

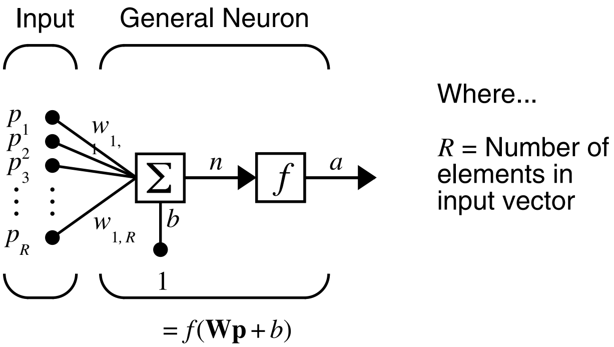
# Fundamentals

## Architecture

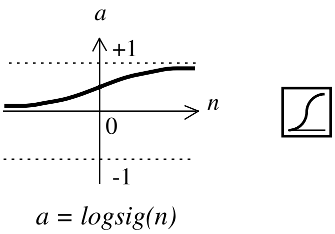
This section presents the architecture of the network that is most commonly used with the backpropagation algorithm - the multilayer feedforward network. The routines in the Neural Network Toolbox can be used to train more general networks; some of these will be briefly discussed in later chapters.

### Neuron Model (tansig, logsig, purelin)

An elementary neuron with $R$ inputs is shown below. Each input is weighted with an appropriate $w$. The sum of the weighted inputs and the bias forms the input to the transfer function $f$. Neurons may use any differentiable transfer function $f$ to generate their output.



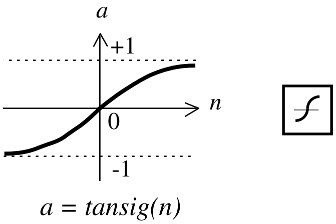
Multilayer networks often use the log-sigmoid transfer function `logsig`.



Log-Sigmoid Transfer Function

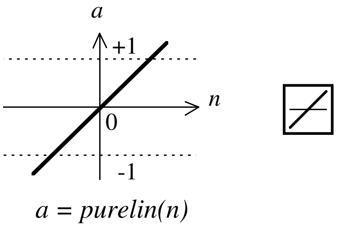
The function logsig generates outputs between 0 and 1 as the neuron's net input goes from negative to positive infinity.

Alternatively, multilayer networks may use the tan-sigmoid transfer function tansig.



Tan-Sigmoid Transfer Function

Occasionally, the linear transfer function purelin is used in backpropagation networks.



Linear Transfer Function

If the last layer of a multilayer network has sigmoid neurons, then the outputs of the network are limited to a small range. If linear output neurons are used the network outputs can take on any value.

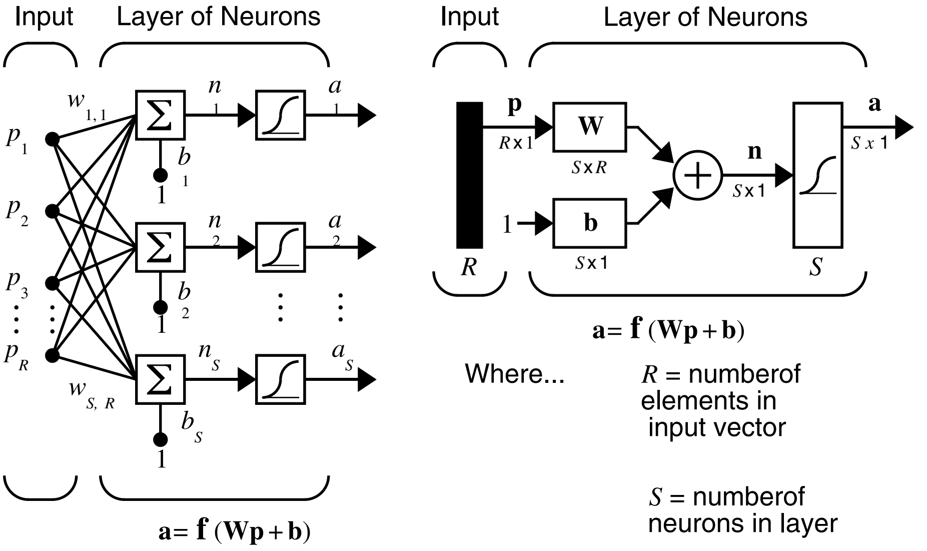
In backpropagation it is important to be able to calculate the derivatives of any transfer functions used. Each of the transfer functions above, tansig, logsig, and purelin, have a corresponding derivative function: dtansig, dlogsig, and dpurelin. To get the name of a transfer function's associated derivative function, call the transfer function with the string 'deriv'.

```
tansig('deriv')
ans = dtansig
```

The three transfer functions described here are the most commonly used transfer functions for backpropagation, but other differentiable transfer functions can be created and used with backpropagation if desired. See Chapter 12, “Advanced Topics.”

### Feedforward Network

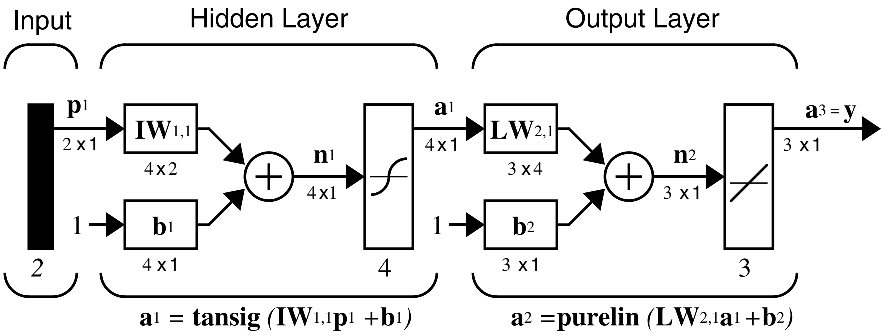
A single-layer network of $S$ `logsig` neurons having $R$ inputs is shown below in full detail on the left and with a layer diagram on the right.



Feedforward networks often have one or more hidden layers of sigmoid neurons followed by an output layer of linear neurons. Multiple layers of neurons with nonlinear transfer functions allow the network to learn nonlinear and linear relationships between input and output vectors. The linear output layer lets the network produce values outside the range –1 to +1.

On the other hand, if you want to constrain the outputs of a network (such as between 0 and 1), then the output layer should use a sigmoid transfer function (such as `logsig`).

As noted in Chapter 2, “Neuron Model and Network Architectures”, for multiple-layer networks we use the number of the layers to determine the superscript on the weight matrices. The appropriate notation is used in the two-layer `tansig`/`purelin` network shown next.



This network can be used as a general function approximator. It can approximate any function with a finite number of discontinuities, arbitrarily well, given sufficient neurons in the hidden layer.

**Creating a Network (newff).** The first step in training a feedforward network is to create the network object. The function `newff` creates a feedforward network. It requires four inputs and returns the network object. The first input is an R by 2 matrix of minimum and maximum values for each of the R elements of the input vector. The second input is an array containing the sizes of each layer. The third input is a cell array containing the names of the transfer functions to be used in each layer. The final input contains the name of the training function to be used.

For example, the following command creates a two-layer network. There is one input vector with two elements. The values for the first element of the input vector range between -1 and 2, the values of the second element of the input vector range between 0 and 5. There are three neurons in the first layer and one neuron in the second (output) layer. The transfer function in the first layer is tan-sigmoid, and the output layer transfer function is linear. The training function is `traingd` (which is described in a later section).

```
net=newff([-1 2; 0 5],[3,1],{'tansig','purelin'},'traingd');
```

This command creates the network object and also initializes the weights and biases of the network; therefore the network is ready for training. There are times when you may want to reinitialize the weights, or to perform a custom initialization. The next section explains the details of the initialization process.

**Initializing Weights (init).** Before training a feedforward network, the weights and biases must be initialized. The `newff` command will automatically initialize the weights, but you may want to reinitialize them. This can be done with the command init. This function takes a network object as input and returns a network object with all weights and biases initialized. Here is how a network is initialized (or reinitialized):

```
net = init(net);
```

For specifics on how the weights are initialized, see Chapter 12, “Advanced Topics.”

## Simulation (sim)

The function sim simulates a network. sim takes the network input p, and the network object net, and returns the network outputs a. Here is how you can use sim to simulate the network we created above for a single input vector:

```
p = [1;2];
a = sim(net,p)
a =
   -0.1011
```

(If you try these commands, your output may be different, depending on the state of your random number generator when the network was initialized.) Below, sim is called to calculate the outputs for a concurrent set of three input vectors. This is the batch mode form of simulation, in which all of the input vectors are place in one matrix. This is much more efficient than presenting the vectors one at a time.

```
p = [1 3 2;2 4 1];
a=sim(net,p)
a =
  -0.1011   -0.2308    0.4955
```

## Training

Once the network weights and biases have been initialized, the network is ready for training. The network can be trained for function approximation (nonlinear regression), pattern association, or pattern classification. The training process requires a set of examples of proper network behavior - network inputs p and target outputs t. During training the weights and biases of the network are iteratively adjusted to minimize the network performance function net.performFcn. The default performance function for feedforward networks is mean square error `mse` - the average squared error between the network outputs `a` and the target outputs `t`.

The remainder of this chapter describes several different training algorithms for feedforward networks. All of these algorithms use the gradient of the performance function to determine how to adjust the weights to minimize performance. The gradient is determined using a technique called backpropagation, which involves performing computations backwards through the network. The backpropagation computation is derived using the chain rule of calculus and is described in Chapter 11 of [HDB96].

The basic backpropagation training algorithm, in which the weights are moved in the direction of the negative gradient, is described in the next section. Later sections describe more complex algorithms that increase the speed of convergence.

### Backpropagation Algorithm

There are many variations of the backpropagation algorithm, several of which we discuss in this chapter. The simplest implementation of backpropagation learning updates the network weights and biases in the direction in which the performance function decreases most rapidly - the negative of the gradient. One iteration of this algorithm can be written

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha_k \mathbf{g}_k$$

where $\mathbf{x}_k$ is a vector of current weights and biases, $\mathbf{g}_k$ is the current gradient, and $\alpha_k$ is the learning rate.

There are two different ways in which this gradient descent algorithm can be implemented: incremental mode and batch mode. In the incremental mode, the gradient is computed and the weights are updated after each input is applied to the network. In the batch mode all of the inputs are applied to the network before the weights are updated. The next section describes the batch mode of training; incremental training will be discussed in a later chapter.

**Batch Training (train).** In batch mode the weights and biases of the network are updated only after the entire training set has been applied to the network. The gradients calculated at each training example are added together to determine the change in the weights and biases. For a discussion of batch training with the backpropagation algorithm see page 12-7 of [HDB96].

**Batch Gradient Descent (traingd).** The batch steepest descent training function is `traingd`. The weights and biases are updated in the direction of the negative gradient of the performance function. If you want to train a network using batch steepest descent, you should set the network `trainFcn` to `traingd`, and then call the function `train`. There is only one training function associated with a given network.

There are seven training parameters associated with `traingd`: `epochs`, `show`, `goal`, `time`, `min_grad`, `max_fail`, and `lr`. The learning rate `lr` is multiplied times the negative of the gradient to determine the changes to the weights and biases. The larger the learning rate, the bigger the step. If the learning rate is made too large, the algorithm becomes unstable. If the learning rate is set too small, the algorithm takes a long time to converge. See page 12-8 of [HDB96] for a discussion of the choice of learning rate.

The training status is displayed for every `show` iteration of the algorithm. (If `show` is set to `NaN`, then the training status never displays.) The other parameters determine when the training stops. The training stops if the number of iterations exceeds `epochs`, if the performance function drops below `goal`, if the magnitude of the gradient is less than `mingrad`, or if the training time is longer than `time` seconds. We discuss `max_fail`, which is associated with the early stopping technique, in the section on improving generalization.

The following code creates a training set of inputs `p` and targets `t`. For batch training, all of the input vectors are placed in one matrix.

```
p = [-1 -1 2 2;0 5 0 5];
t = [-1 -1 1 1];
```

Next, we create the feedforward network. Here we use the function `minmax` to determine the range of the inputs to be used in creating the network.

```
net=newff(minmax(p),[3,1],{'tansig','purelin'},'traingd');
```

At this point, we might want to modify some of the default training parameters.

```
net.trainParam.show = 50;
net.trainParam.lr = 0.05;
net.trainParam.epochs = 300;
net.trainParam.goal = 1e-5;
```

If you want to use the default training parameters, the above commands are not necessary.

Now we are ready to train the network.

```
[net,tr]=train(net,p,t);
   TRAINGD, Epoch 0/300, MSE 1.59423/1e-05, Gradient 
2.76799/1e-10
   TRAINGD, Epoch 50/300, MSE 0.00236382/1e-05, Gradient 
0.0495292/1e-10
   TRAINGD, Epoch 100/300, MSE 0.000435947/1e-05, Gradient 
0.0161202/1e-10
   TRAINGD, Epoch 150/300, MSE 8.68462e-05/1e-05, Gradient 
0.00769588/1e-10
   TRAINGD, Epoch 200/300, MSE 1.45042e-05/1e-05, Gradient 
0.00325667/1e-10
   TRAINGD, Epoch 211/300, MSE 9.64816e-06/1e-05, Gradient 
0.00266775/1e-10
   TRAINGD, Performance goal met.
```

The training record `tr` contains information about the progress of training. An example of its use is given in the Sample Training Session near the end of this chapter.

Now the trained network can be simulated to obtain its response to the inputs in the training set.

```
a = sim(net,p)
a =
   -1.0010   -0.9989    1.0018    0.9985
```

Try the Neural Network Design Demonstration `nnd12sd1`[HDB96] for an illustration of the performance of the batch gradient descent algorithm.

**Batch Gradient Descent with Momentum (traingdm).** In addition to `traingd`, there is another batch algorithm for feedforward networks that often provides faster convergence - `traingdm`, steepest descent with momentum. Momentum allows a network to respond not only to the local gradient, but also to recent trends in the error surface. Acting like a low-pass filter, momentum allows the network to ignore small features in the error surface. Without momentum a network may get stuck in a shallow local minimum. With momentum a network can slide through such a minimum. See page 12-9 of [HDB96] for a discussion of momentum.

Momentum can be added to backpropagation learning by making weight changes equal to the sum of a fraction of the last weight change and the new change suggested by the backpropagation rule. The magnitude of the effect that the last weight change is allowed to have is mediated by a momentum constant, mc, which can be any number between 0 and 1. When the momentum constant is 0, a weight change is based solely on the gradient. When the momentum constant is 1, the new weight change is set to equal the last weight change and the gradient is simply ignored. The gradient is computed by summing the gradients calculated at each training example, and the weights and biases are only updated after all training examples have been presented.

If the new performance function on a given iteration exceeds the performance function on a previous iteration by more than a predefined ratio max_perf_inc (typically 1.04), the new weights and biases are discarded, and the momentum coefficient mc is set to zero.

The batch form of gradient descent with momentum is invoked using the training function traingdm. The traingdm function is invoked using the same steps shown above for the traingd function, except that the mc, lr and max_perf_inc learning parameters can all be set.

In the following code we recreate our previous network and retrain it using gradient descent with momentum. The training parameters for traingdm are the same as those for traingd, with the addition of the momentum factor mc and the maximum performance increase max_perf_inc. (The training parameters are reset to the default values whenever net.trainFcn is set to traingdm.)

```
p = [-1 -1 2 2;0 5 0 5];
t = [-1 -1 1 1];
net=newff(minmax(p),[3,1],{'tansig','purelin'},'traingdm');
net.trainParam.show = 50;
net.trainParam.lr = 0.05;
net.trainParam.mc = 0.9;
net.trainParam.epochs = 300;
net.trainParam.goal = 1e-5;
[net,tr]=train(net,p,t);
   TRAINGDM, Epoch 0/300, MSE 3.6913/1e-05, Gradient
4.54729/1e-10
   TRAINGDM, Epoch 50/300, MSE 0.00532188/1e-05, Gradient
0.213222/1e-10
```

```
   TRAINGDM, Epoch 100/300, MSE 6.34868e-05/1e-05, Gradient
0.0409749/1e-10
   TRAINGDM, Epoch 114/300, MSE 9.06235e-06/1e-05, Gradient
0.00908756/1e-10
   TRAINGDM, Performance goal met.
a = sim(net,p)
a =
    -1.0026   -1.0044    0.9969    0.9992
```

Note that since we reinitialized the weights and biases before training (by calling `newff` again), we obtain a different mean square error than we did using `traingd`. If we were to reinitialize and train again using `traingdm`, we would get yet a different mean square error. The random choice of initial weights and biases will affect the performance of the algorithm. If you want to compare the performance of different algorithms, you should test each using several different sets of initial weights and biases. You may want to use `net=init(net)` to reinitialize the weights, rather than recreating the entire network with `newff`.

Try the Neural Network Design Demonstration `nnd12mo` [HDB96] for an illustration of the performance of the batch momentum algorithm.

# Faster Training

The previous section presented two backpropagation training algorithms: gradient descent, and gradient descent with momentum. These two methods are often too slow for practical problems. In this section we discuss several high performance algorithms that can converge from ten to one hundred times faster than the algorithms discussed previously. All of the algorithms in this section operate in the batch mode and are invoked using `train`.

These faster algorithms fall into two main categories. The first category uses heuristic techniques, which were developed from an analysis of the performance of the standard steepest descent algorithm. One heuristic modification is the momentum technique, which was presented in the previous section. This section discusses two more heuristic techniques: variable learning rate backpropagation, `traingda`; and resilient backpropagation `trainrp`.

The second category of fast algorithms uses standard numerical optimization techniques. (See Chapter 9 of [HDB96] for a review of basic numerical optimization.) Later in this section we present three types of numerical optimization techniques for neural network training: conjugate gradient (`traincgf`, `traincgp`, `traincgb`, `trainscg`), quasi-Newton (`trainbfg`, `trainoss`), and Levenberg-Marquardt (`trainlm`).

## Variable Learning Rate (traingda, traingdx)

With standard steepest descent, the learning rate is held constant throughout training. The performance of the algorithm is very sensitive to the proper setting of the learning rate. If the learning rate is set too high, the algorithm may oscillate and become unstable. If the learning rate is too small, the algorithm will take too long to converge. It is not practical to determine the optimal setting for the learning rate before training, and, in fact, the optimal learning rate changes during the training process, as the algorithm moves across the performance surface.

The performance of the steepest descent algorithm can be improved if we allow the learning rate to change during the training process. An adaptive learning rate will attempt to keep the learning step size as large as possible while keeping learning stable. The learning rate is made responsive to the complexity of the local error surface.

An adaptive learning rate requires some changes in the training procedure used by `traingd`. First, the initial network output and error are calculated. At each epoch new weights and biases are calculated using the current learning rate. New outputs and errors are then calculated.

As with momentum, if the new error exceeds the old error by more than a predefined ratio max_perf_inc (typically 1.04), the new weights and biases are discarded. In addition, the learning rate is decreased (typically by multiplying by lr_dec = 0.7). Otherwise, the new weights, etc., are kept. If the new error is less than the old error, the learning rate is increased (typically by multiplying by lr_inc = 1.05).

This procedure increases the learning rate, but only to the extent that the network can learn without large error increases. Thus, a near-optimal learning rate is obtained for the local terrain. When a larger learning rate could result in stable learning, the learning rate is increased. When the learning rate is too high to guarantee a decrease in error, it gets decreased until stable learning resumes.

Try the Neural Network Design Demonstration nnd12vl [HDB96] for an illustration of the performance of the variable learning rate algorithm.

Backpropagation training with an adaptive learning rate is implemented with the function traingda, which is called just like traingd, except for the additional training parameters max_perf_inc, lr_dec, and lr_inc. Here is how it is called to train our previous two-layer network:

```
p = [-1 -1 2 2;0 5 0 5];
t = [-1 -1 1 1];
net=newff(minmax(p),[3,1],{'tansig','purelin'},'traingda');
net.trainParam.show = 50;
net.trainParam.lr = 0.05;
net.trainParam.lr_inc = 1.05;
net.trainParam.epochs = 300;
net.trainParam.goal = 1e-5;
[net,tr]=train(net,p,t);
   TRAINGDA, Epoch 0/300, MSE 1.71149/1e-05, Gradient
2.6397/1e-06
   TRAINGDA, Epoch 44/300, MSE 7.47952e-06/1e-05, Gradient
0.00251265/1e-06
   TRAINGDA, Performance goal met.
a = sim(net,p)
a =
   -1.0036   -0.9960    1.0008    0.9991
```

The function traingdx combines adaptive learning rate with momentum training. It is invoked in the same way as traingda, except that it has the momentum coefficient mc as an additional training parameter.

## Resilient Backpropagation (trainrp)

Multilayer networks typically use sigmoid transfer functions in the hidden layers. These functions are often called “squashing” functions, since they compress an infinite input range into a finite output range. Sigmoid functions are characterized by the fact that their slope must approach zero as the input gets large. This causes a problem when using steepest descent to train a multilayer network with sigmoid functions, since the gradient can have a very small magnitude; and therefore, cause small changes in the weights and biases, even though the weights and biases are far from their optimal values.

The purpose of the resilient backpropagation (Rprop) training algorithm is to eliminate these harmful effects of the magnitudes of the partial derivatives. Only the sign of the derivative is used to determine the direction of the weight update; the magnitude of the derivative has no effect on the weight update. The size of the weight change is determined by a separate update value. The update value for each weight and bias is increased by a factor delt_inc whenever the derivative of the performance function with respect to that weight has the same sign for two successive iterations. The update value is decreased by a factor delt_dec whenever the derivative with respect that weight changes sign from the previous iteration. If the derivative is zero, then the update value remains the same. Whenever the weights are oscillating the weight change will be reduced. If the weight continues to change in the same direction for several iterations, then the magnitude of the weight change will be increased. A complete description of the Rprop algorithm is given in [ReBr93].

In the following code we recreate our previous network and train it using the Rprop algorithm. The training parameters for trainrp are epochs, show, goal, time, min_grad, max_fail, delt_inc, delt_dec, delta0, deltamax. We have previously discussed the first eight parameters. The last two are the initial step size and the maximum step size, respectively. The performance of Rprop is not very sensitive to the settings of the training parameters. For the example below, we leave most of the training parameters at the default values. We do reduce show below our previous value, because Rprop generally converges much faster than the previous algorithms.

```
p = [-1 -1 2 2;0 5 0 5];
```

```
t = [-1 -1 1 1];
net=newff(minmax(p),[3,1],{'tansig','purelin'},'trainrp');
net.trainParam.show = 10;
net.trainParam.epochs = 300;
net.trainParam.goal = 1e-5;
[net,tr]=train(net,p,t);
   TRAINRP, Epoch 0/300, MSE 0.469151/1e-05, Gradient
1.4258/1e-06
   TRAINRP, Epoch 10/300, MSE 0.000789506/1e-05, Gradient
0.0554529/1e-06
   TRAINRP, Epoch 20/300, MSE 7.13065e-06/1e-05, Gradient
0.00346986/1e-06
   TRAINRP, Performance goal met.
a = sim(net,p)
a =
   -1.0026   -0.9963    0.9978    1.0017
```

Rprop is generally much faster than the standard steepest descent algorithm. It also has the nice property that it requires only a modest increase in memory requirements. We do need to store the update values for each weight and bias, which is equivalent to storage of the gradient.

## Conjugate Gradient Algorithms

The basic backpropagation algorithm adjusts the weights in the steepest descent direction (negative of the gradient). This is the direction in which the performance function is decreasing most rapidly. It turns out that, although the function decreases most rapidly along the negative of the gradient, this does not necessarily produce the fastest convergence. In the conjugate gradient algorithms a search is performed along conjugate directions, which produces generally faster convergence than steepest descent directions. In this section, we present four different variations of conjugate gradient algorithms.

See page 12-14 of [HDB96] for a discussion of conjugate gradient algorithms and their application to neural networks.

In most of the training algorithms that we discussed up to this point, a learning rate is used to determine the length of the weight update (step size). In most of the conjugate gradient algorithms, the step size is adjusted at each iteration. A search is made along the conjugate gradient direction to determine the step size, which minimizes the performance function along that line. There are five different search functions included in the toolbox, and these are discussed at the end of this section. Any of these search functions can be used interchangeably with a variety of the training functions described in the remainder of this chapter. Some search functions are best suited to certain training functions, although the optimum choice can vary according to the specific application. An appropriate default search function is assigned to each training function, but this can be modified by the user.

### Fletcher-Reeves Update (traincgf)

All of the conjugate gradient algorithms start out by searching in the steepest descent direction (negative of the gradient) on the first iteration.

$$\mathbf{p}_0 = -\mathbf{g}_0$$

A line search is then performed to determine the optimal distance to move along the current search direction:

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k$$

Then the next search direction is determined so that it is conjugate to previous search directions. The general procedure for determining the new search direction is to combine the new steepest descent direction with the previous search direction:

$$\mathbf{p}_k = -\mathbf{g}_k + \beta_k \mathbf{p}_{k-1}$$

The various versions of conjugate gradient are distinguished by the manner in which the constant $\beta_k$ is computed. For the Fletcher-Reeves update the procedure is

$$\beta_k = \frac{\mathbf{g}_k^T \mathbf{g}_k}{\mathbf{g}_{k-1}^T \mathbf{g}_{k-1}}$$

This is the ratio of the norm squared of the current gradient to the norm squared of the previous gradient.

See [FlRe64] or [HDB96] for a discussion of the Fletcher-Reeves conjugate gradient algorithm.

In the following code, we reinitialize our previous network and retrain it using the Fletcher-Reeves version of the conjugate gradient algorithm. The training parameters for traincgf are epochs, show, goal, time, min_grad, max_fail, srchFcn, scal_tol, alpha, beta, delta, gama, low_lim, up_lim, maxstep, minstep, bmax. We have previously discussed the first six parameters. The parameter srchFcn is the name of the line search function. It can be any of the functions described later in this section (or a user-supplied function). The remaining parameters are associated with specific line search routines and are described later in this section. The default line search routine srchcha is used in this example. traincgf generally converges in fewer iterations than trainrp (although there is more computation required in each iteration).

```
p = [-1 -1 2 2;0 5 0 5];
t = [-1 -1 1 1];
net=newff(minmax(p),[3,1],{'tansig','purelin'},'traincgf');
net.trainParam.show = 5;
net.trainParam.epochs = 300;
net.trainParam.goal = 1e-5;
[net,tr]=train(net,p,t);
   TRAINCGF-srchcha, Epoch 0/300, MSE 2.15911/1e-05, Gradient
3.17681/1e-06
   TRAINCGF-srchcha, Epoch 5/300, MSE 0.111081/1e-05, Gradient
0.602109/1e-06
   TRAINCGF-srchcha, Epoch 10/300, MSE 0.0095015/1e-05, Gradient
0.197436/1e-06
   TRAINCGF-srchcha, Epoch 15/300, MSE 0.000508668/1e-05,
Gradient 0.0439273/1e-06
   TRAINCGF-srchcha, Epoch 17/300, MSE 1.33611e-06/1e-05,
Gradient 0.00562836/1e-06
   TRAINCGF, Performance goal met.
a = sim(net,p)
a =
   -1.0001   -1.0023    0.9999    1.0002
```

The conjugate gradient algorithms are usually much faster than variable learning rate backpropagation, and are sometimes faster than trainrp, although the results will vary from one problem to another. The conjugate gradient algorithms require only a little more storage than the simpler algorithms, so they are often a good choice for networks with a large number of weights.

Try the Neural Network Design Demonstration nnd12cg [HDB96] for an illustration of the performance of a conjugate gradient algorithm.

### Polak-Ribiére Update (traincgp)

Another version of the conjugate gradient algorithm was proposed by Polak and Ribiére. As with the Fletcher-Reeves algorithm, the search direction at each iteration is determined by

$$\mathbf{p}_k = -\mathbf{g}_k + \beta_k \mathbf{p}_{k-1}$$

For the Polak-Ribiére update, the constant $\beta_k$ is computed by

$$\beta_k = \frac{\Delta\mathbf{g}_{k-1}^T \mathbf{g}_k}{\mathbf{g}_{k-1}^T \mathbf{g}_{k-1}}$$

This is the inner product of the previous change in the gradient with the current gradient divided by the norm squared of the previous gradient. See [FlRe64] or [HDB96] for a discussion of the Polak-Ribiére conjugate gradient algorithm.

In the following code, we recreate our previous network and train it using the Polak-Ribiére version of the conjugate gradient algorithm. The training parameters for traincgp are the same as those for traincgf. The default line search routine srchcha is used in this example. The parameters show and epoch are set to the same values as they were for traincgf.

```
p = [-1 -1 2 2;0 5 0 5];
t = [-1 -1 1 1];
net=newff(minmax(p),[3,1],{'tansig','purelin'},'traincgp');
net.trainParam.show = 5;
net.trainParam.epochs = 300;
net.trainParam.goal = 1e-5;
[net,tr]=train(net,p,t);
   TRAINCGP-srchcha, Epoch 0/300, MSE 1.21966/1e-05, Gradient
1.77008/1e-06
   TRAINCGP-srchcha, Epoch 5/300, MSE 0.227447/1e-05, Gradient
0.86507/1e-06
   TRAINCGP-srchcha, Epoch 10/300, MSE 0.000237395/1e-05,
Gradient 0.0174276/1e-06
   TRAINCGP-srchcha, Epoch 15/300, MSE 9.28243e-05/1e-05,
Gradient 0.00485746/1e-06
   TRAINCGP-srchcha, Epoch 20/300, MSE 1.46146e-05/1e-05,
Gradient 0.000912838/1e-06
```

```
   TRAINCGP-srchcha, Epoch 25/300, MSE 1.05893e-05/1e-05,
Gradient 0.00238173/1e-06
   TRAINCGP-srchcha, Epoch 26/300, MSE 9.10561e-06/1e-05,
Gradient 0.00197441/1e-06
   TRAINCGP, Performance goal met.
a = sim(net,p)
a =
   -0.9967   -1.0018    0.9958    1.0022
```

The traincgp routine has performance similar to traincgf. It is difficult to predict which algorithm will perform best on a given problem. The storage requirements for Polak-Ribiére (four vectors) are slightly larger than for Fletcher-Reeves (three vectors).

## Powell-Beale Restarts (traincgb)

For all conjugate gradient algorithms, the search direction will be periodically reset to the negative of the gradient. The standard reset point occurs when the number of iterations is equal to the number of network parameters (weights and biases), but there are other reset methods that can improve the efficiency of training. One such reset method was proposed by Powell [Powe77], based on an earlier version proposed by Beale [Beal72]. For this technique we will restart if there is very little orthogonality left between the current gradient and the previous gradient. This is tested with the following inequality.

$$\left|\mathbf{g}_{k-1}^T \mathbf{g}_k\right| \geq 0.2\left\|\mathbf{g}_k\right\|^2$$

If this condition is satisfied, the search direction is reset to the negative of the gradient.

In the following code, we recreate our previous network and train it using the Powell-Beale version of the conjugate gradient algorithm. The training parameters for traincgb are the same as those for traincgf. The default line search routine srchcha is used in this example. The parameters show and epoch are set to the same values as they were for traincgf.

```
p = [-1 -1 2 2;0 5 0 5];
t = [-1 -1 1 1];
net=newff(minmax(p),[3,1],{'tansig','purelin'},'traincgb');
net.trainParam.show = 5;
net.trainParam.epochs = 300;
net.trainParam.goal = 1e-5;
```

```
[net,tr]=train(net,p,t);
   TRAINCGB-srchcha, Epoch 0/300, MSE 2.5245/1e-05, Gradient
3.66882/1e-06
   TRAINCGB-srchcha, Epoch 5/300, MSE 4.86255e-07/1e-05, Gradient
0.00145878/1e-06
   TRAINCGB, Performance goal met.
a = sim(net,p)
a =
   -0.9997   -0.9998    1.0000    1.0014
```

The traincgb routine has performance that is somewhat better than traincgp for some problems, although performance on any given problem is difficult to predict. The storage requirements for the Powell-Beale algorithm (six vectors) are slightly larger than for Polak-Ribiére (four vectors).

### Scaled Conjugate Gradient (trainscg)

Each of the conjugate gradient algorithms that we have discussed so far requires a line search at each iteration. This line search is computationally expensive, since it requires that the network response to all training inputs be computed several times for each search. The scaled conjugate gradient algorithm (SCG), developed by Moller [Moll93], was designed to avoid the time-consuming line search. This algorithm is too complex to explain in a few lines, but the basic idea is to combine the model-trust region approach (used in the Levenberg-Marquardt algorithm described later), with the conjugate gradient approach. See {Moll93] for a detailed explanation of the algorithm.

In the following code, we reinitialize our previous network and retrain it using the scaled conjugate gradient algorithm. The training parameters for trainscg are epochs, show, goal, time, min_grad, max_fail, sigma, lambda. We have previously discussed the first six parameters. The parameter sigma determines the change in the weight for the second derivative approximation. The parameter lambda regulates the indefiniteness of the Hessian. The parameters show and epoch are set to 10 and 300, respectively.

```
p = [-1 -1 2 2;0 5 0 5];
t = [-1 -1 1 1];
net=newff(minmax(p),[3,1],{'tansig','purelin'},'trainscg');
net.trainParam.show = 10;
net.trainParam.epochs = 300;
net.trainParam.goal = 1e-5;
[net,tr]=train(net,p,t);
```

```
   TRAINSCG, Epoch 0/300, MSE 4.17697/1e-05, Gradient
5.32455/1e-06
   TRAINSCG, Epoch 10/300, MSE 2.09505e-05/1e-05, Gradient
0.00673703/1e-06
   TRAINSCG, Epoch 11/300, MSE 9.38923e-06/1e-05, Gradient
0.0049926/1e-06
   TRAINSCG, Performance goal met.
a = sim(net,p)
a =
   -1.0057   -1.0008    1.0019    1.0005
```

The `trainscg` routine may require more iterations to converge than the other conjugate gradient algorithms, but the number of computations in each iteration is significantly reduced because no line search is performed. The storage requirements for the scaled conjugate gradient algorithm are about the same as those of Fletcher-Reeves.

## Line Search Routines

Several of the conjugate gradient and quasi-Newton algorithms require that a line search be performed. In this section, we describe five different line searches you can use. To use any of these search routines, you simply set the training parameter `srchFcn` equal to the name of the desired search function, as described in previous sections. It is often difficult to predict which of these routines provide the best results for any given problem, but we set the default search function to an appropriate initial choice for each training function, so you never need to modify this parameter.

### Golden Section Search (srchgol)

The golden section search `srchgol` is a linear search that does not require the calculation of the slope. This routine begins by locating an interval in which the minimum of the performance occurs. This is accomplished by evaluating the performance at a sequence of points, starting at a distance of `delta` and doubling in distance each step, along the search direction. When the performance increases between two successive iterations, a minimum has been bracketed. The next step is to reduce the size of the interval containing the minimum. Two new points are located within the initial interval. The values of the performance at these two points determines a section of the interval that can be discarded, and a new interior point is placed within the new interval.

This procedure is continued until the interval of uncertainty is reduced to a width of tol, which is equal to delta/scale_tol.

See [HDB96], starting on page 12-16, for a complete description of the golden section search. Try the Neural Network Design Demonstration nnd12sd1 [HDB96] for an illustration of the performance of the golden section search in combination with a conjugate gradient algorithm.

### Brent's Search (srchbre)

Brent's search is a linear search, which is a hybrid combination of the golden section search and a quadratic interpolation. Function comparison methods, like the golden section search, have a first-order rate of convergence, while polynomial interpolation methods have an asymptotic rate that is faster than superlinear. On the other hand, the rate of convergence for the golden section search starts when the algorithm is initialized, whereas the asymptotic behavior for the polynomial interpolation methods may take many iterations to become apparent. Brent's search attempts to combine the best features of both approaches.

For Brent's search we begin with the same interval of uncertainty that we used with the golden section search, but some additional points are computed. A quadratic function is then fitted to these points and the minimum of the quadratic function is computed. If this minimum is within the appropriate interval of uncertainty, it is used in the next stage of the search and a new quadratic approximation is performed. If the minimum falls outside the known interval of uncertainty, then a step of the golden section search is performed.

See [Bren73] for a complete description of this algorithm. This algorithm has the advantage that it does not require computation of the derivative. The derivative computation requires a backpropagation through the network, which involves more computation than a forward pass. However, the algorithm may require more performance evaluations than algorithms that use derivative information.

### Hybrid Bisection-Cubic Search (srchhyb)

Like Brent's search, srchhyb is a hybrid algorithm. It is a combination of bisection and cubic interpolation. For the bisection algorithm, one point is located in the interval of uncertainty and the performance and its derivative are computed. Based on this information, half of the interval of uncertainty is discarded. In the hybrid algorithm, a cubic interpolation of the function is obtained by using the value of the performance and its derivative at the two end points. If the minimum of the cubic interpolation falls within the known interval of uncertainty, then it is used to reduce the interval of uncertainty. Otherwise, a step of the bisection algorithm is used.

See [Scal85] for a complete description of the hybrid bisection-cubic search. This algorithm does require derivative information, so it performs more computations at each step of the algorithm than the golden section search or Brent's algorithm.

### Charalambous' Search (srchcha)

The method of Charalambous `srchcha` was designed to be used in combination with a conjugate gradient algorithm for neural network training. Like the previous two methods, it is a hybrid search. It uses a cubic interpolation, together with a type of sectioning.

See [Char92] for a description of Charalambous' search. We have used this routine as the default search for most of the conjugate gradient algorithms, since it appears to produce excellent results for many different problems. It does require the computation of the derivatives (backpropagation) in addition to the computation of performance, but it overcomes this limitation by locating the minimum with fewer steps. This is not true for all problems, and you may want to experiment with other line searches.

### Backtracking (srchbac)

The backtracking search routine `srchbac` is best suited to use with the quasi-Newton optimization algorithms. It begins with a step multiplier of 1 and then backtracks until an acceptable reduction in the performance is obtained. On the first step it uses the value of performance at the current point and at a step multiplier of 1. Also it uses the value of the derivative of performance at the current point, to obtain a quadratic approximation to the performance function along the search direction. The minimum of the quadratic approximation becomes a tentative optimum point (under certain conditions) and the performance at this point is tested. If the performance is not sufficiently reduced, a cubic interpolation is obtained and the minimum of the cubic interpolation becomes the new tentative optimum point. This process is continued until a sufficient reduction in the performance is obtained.

The backtracking algorithm is described in [DeSc83]. It was used as the default line search for the quasi-Newton algorithms, although it may not be the best technique for all problems.

## Quasi-Newton Algorithms

### BFGS Algorithm (trainbgf)

Newton's method is an alternative to the conjugate gradient methods for fast optimization. The basic step of Newton's method is

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \mathbf{A}_k^{-1}\mathbf{g}_k$$

where $\mathbf{A}_k$ is the Hessian matrix (second derivatives) of the performance index at the current values of the weights and biases. Newton's method often converges faster than conjugate gradient methods. Unfortunately, it is complex and expensive to compute the Hessian matrix for feedforward neural networks. There is a class of algorithms that is based on Newton's method, but which doesn't require calculation of second derivatives. These are called quasi-Newton (or secant) methods. They update an approximate Hessian matrix at each iteration of the algorithm. The update is computed as a function of the gradient. The quasi-Newton method that has been most successful in published studies is the Broyden, Fletcher, Goldfarb, and Shanno (BFGS) update. This algorithm has been implemented in the `trainbfg` routine.

In the following code, we reinitialize our previous network and retrain it using the BFGS quasi-Newton algorithm. The training parameters for `trainbfg` are the same as those for `traincgf`. The default line search routine `srchbac` is used in this example. The parameters `show` and `epoch` are set to 5 and 300, respectively.

```
p = [-1 -1 2 2;0 5 0 5];
t = [-1 -1 1 1];
net=newff(minmax(p),[3,1],{'tansig','purelin'},'trainbfg');
net.trainParam.show = 5;
net.trainParam.epochs = 300;
net.trainParam.goal = 1e-5;
[net,tr]=train(net,p,t);
   TRAINBFG-srchbac, Epoch 0/300, MSE 0.492231/1e-05, Gradient
2.16307/1e-06
```

```
   TRAINBFG-srchbac, Epoch 5/300, MSE 0.000744953/1e-05, Gradient
0.0196826/1e-06
   TRAINBFG-srchbac, Epoch 8/300, MSE 7.69867e-06/1e-05, Gradient
0.00497404/1e-06
   TRAINBFG, Performance goal met.
a = sim(net,p)
a =
   -0.9995   -1.0004    1.0008    0.9945
```

The BFGS algorithm is described in [DeSc83]. This algorithm requires more computation in each iteration and more storage than the conjugate gradient methods, although it generally converges in fewer iterations. The approximate Hessian must be stored, and its dimension is $n \times n$, where $n$ is equal to the number of weights and biases in the network. For very large networks it may be better to use Rprop or one of the conjugate gradient algorithms. For smaller networks, however, `trainbfg` can be an efficient training function.

### One Step Secant Algorithm (trainoss)

Since the BFGS algorithm requires more storage and computation in each iteration than the conjugate gradient algorithms, there is need for a secant approximation with smaller storage and computation requirements. The one step secant (OSS) method is an attempt to bridge the gap between the conjugate gradient algorithms and the quasi-Newton (secant) algorithms. This algorithm does not store the complete Hessian matrix; it assumes that at each iteration, the previous Hessian was the identity matrix. This has the additional advantage that the new search direction can be calculated without computing a matrix inverse.

In the following code, we reinitialize our previous network and retrain it using the one-step secant algorithm. The training parameters for `trainoss` are the same as those for `traincgf`. The default line search routine `srchbac` is used in this example. The parameters `show` and `epoch` are set to 5 and 300, respectively.

```
p = [-1 -1 2 2;0 5 0 5];
t = [-1 -1 1 1];
net=newff(minmax(p),[3,1],{'tansig','purelin'},'trainoss');
net.trainParam.show = 5;
net.trainParam.epochs = 300;
net.trainParam.goal = 1e-5;
[net,tr]=train(net,p,t);
```

```
   TRAINOSS-srchbac, Epoch 0/300, MSE 0.665136/1e-05, Gradient
1.61966/1e-06
   TRAINOSS-srchbac, Epoch 5/300, MSE 0.000321921/1e-05, Gradient
0.0261425/1e-06
   TRAINOSS-srchbac, Epoch 7/300, MSE 7.85697e-06/1e-05, Gradient
0.00527342/1e-06
   TRAINOSS, Performance goal met.
a = sim(net,p)
a =
   -1.0035   -0.9958    1.0014    0.9997
```

The one step secant method is described in [Batt92]. This algorithm requires less storage and computation per epoch than the BFGS algorithm. It requires slightly more storage and computation per epoch than the conjugate gradient algorithms. It can be considered a compromise between full quasi-Newton algorithms and conjugate gradient algorithms.

## Levenberg-Marquardt (trainlm)

Like the quasi-Newton methods, the Levenberg-Marquardt algorithm was designed to approach second-order training speed without having to compute the Hessian matrix. When the performance function has the form of a sum of squares (as is typical in training feedforward networks), then the Hessian matrix can be approximated as

$$\mathbf{H} = \mathbf{J}^T\mathbf{J}$$

and the gradient can be computed as

$$\mathbf{g} = \mathbf{J}^T\mathbf{e}$$

where $\mathbf{J}$ is the Jacobian matrix that contains first derivatives of the network errors with respect to the weights and biases, and $\mathbf{e}$ is a vector of network errors. The Jacobian matrix can be computed through a standard backpropagation technique (see [HaMe94]) that is much less complex than computing the Hessian matrix.

The Levenberg-Marquardt algorithm uses this approximation to the Hessian matrix in the following Newton-like update:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - [\mathbf{J}^T\mathbf{J} + \mu\mathbf{I}]^{-1}\mathbf{J}^T\mathbf{e}$$

When the scalar μ is zero, this is just Newton's method, using the approximate Hessian matrix. When μ is large, this becomes gradient descent with a small step size. Newton's method is faster and more accurate near an error minimum, so the aim is to shift towards Newton's method as quickly as possible. Thus, μ is decreased after each successful step (reduction in performance function) and is increased only when a tentative step would increase the performance function. In this way, the performance function will always be reduced at each iteration of the algorithm.

In the following code, we reinitialize our previous network and retrain it using the Levenberg-Marquardt algorithm. The training parameters for `trainlm` are `epochs`, `show`, `goal`, `time`, `min_grad`, `max_fail`, `mu`, `mu_dec`, `mu_inc`, `mu_max`, `mem_reduc`. We have discussed the first six parameters earlier. The parameter `mu` is the initial value for μ. This value is multiplied by `mu_dec` whenever the performance function is reduced by a step. It is multiplied by `mu_inc` whenever a step would increase the performance function. If `mu` becomes larger than `mu_max`, the algorithm is stopped. The parameter `mem_reduc` is used to control the amount of memory used by the algorithm. It is discussed in the next section. The parameters `show` and `epoch` are set to 5 and 300, respectively.

```
p = [-1 -1 2 2;0 5 0 5];
t = [-1 -1 1 1];
net=newff(minmax(p),[3,1],{'tansig','purelin'},'trainlm');
net.trainParam.show = 5;
net.trainParam.epochs = 300;
net.trainParam.goal = 1e-5;
[net,tr]=train(net,p,t);
   TRAINLM, Epoch 0/300, MSE 2.7808/1e-05, Gradient 7.77931/1e-10
   TRAINLM, Epoch 4/300, MSE 3.67935e-08/1e-05, Gradient
0.000808272/1e-10
   TRAINLM, Performance goal met.
a = sim(net,p)
a =
   -1.0000   -1.0000    1.0000    0.9996
```

The original description of the Levenberg-Marquardt algorithm is given in [Marq63]. The application of Levenberg-Marquardt to neural network training is described in [HaMe94] and starting on page 12-19 of [HDB96]. This algorithm appears to be the fastest method for training moderate-sized feedforward neural networks (up to several hundred weights). It also has a very efficient MATLAB® implementation, since the solution of the matrix equation is a built-in function, so its attributes become even more pronounced in a MATLAB setting.

Try the Neural Network Design Demonstration `nnd12m` [HDB96] for an illustration of the performance of the batch Levenberg-Marquardt algorithm.

## Reduced Memory Levenberg-Marquardt (trainlm)

The main drawback of the Levenberg-Marquardt algorithm is that it requires the storage of some matrices that can be quite large for certain problems. The size of the Jacobian matrix is $Q \times n$ , where $Q$ is the number of training sets and $n$ is the number of weights and biases in the network. It turns out that this matrix does not have to be computed and stored as a whole. For example, if we were to divide the Jacobian into two equal submatrices we could compute the approximate Hessian matrix as follows:

$$\mathbf{H} = \mathbf{J}^T\mathbf{J} = \begin{bmatrix} \mathbf{J}_1^T & \mathbf{J}_2^T \end{bmatrix} \begin{bmatrix} \mathbf{J}_1 \\ \mathbf{J}_2 \end{bmatrix} = \mathbf{J}_1^T\mathbf{J}_1 + \mathbf{J}_2^T\mathbf{J}_2$$

Therefore, the full Jacobian does not have to exist at one time. The approximate Hessian can be computed by summing a series of subterms. Once one subterm has been computed, the corresponding submatrix of the Jacobian can be cleared.

When using the training function `trainlm`, the parameter `mem_reduc` is used to determine how many rows of the Jacobian are to be computed in each submatrix. If `mem_reduc` is set to 1, then the full Jacobian is computed, and no memory reduction is achieved. If `mem_reduc` is set to 2, then only half of the Jacobian will be computed at one time. This saves half of the memory used by the calculation of the full Jacobian.

There is a drawback to using memory reduction. A significant computational overhead is associated with computing the Jacobian in submatrices. If you have enough memory available, then it is better to set `mem_reduc` to 1 and to compute the full Jacobian. If you have a large training set, and you are running out of memory, then you should set `mem_reduc` to 2, and try again. If you still run out of memory, continue to increase `mem_reduc`.

Even if you use memory reduction, the Levenberg-Marquardt algorithm will always compute the approximate Hessian matrix, which has dimensions $n \times n$ . If your network is very large, then you may run out of memory. If this is the case, then you will want to try trainscg, trainrp, or one of the conjugate gradient algorithms.

## Speed and Memory Comparison

It is very difficult to know which training algorithm will be the fastest for a given problem. It will depend on many factors, including the complexity of the problem, the number of data points in the training set, the number of weights and biases in the network, the error goal, and whether the network is being used for pattern recognition (discriminant analysis) or function approximation (regression). In this section we perform a number of benchmark comparisons of the various training algorithms. We train feedforward networks on six different problems. Three of the problems fall in the pattern recognition category and the three others fall in the function approximation category. Two of the problems are simple “toy” problems, while the other four are “real world” problems. We use networks with a variety of different architectures and complexities, and we train the networks to a variety of different accuracy levels.

The following table lists the algorithms that are tested and the acronyms we use to identify them.

| **Acronym** | **Algorithm** |
|---|---|
| LM | `trainlm` - Levenberg-Marquardt |
| BFG | `trainbfg` - BFGS Quasi-Newton |
| RP | `trainrp` - Resilient Backpropagation |
| SCG | `trainscg` - Scaled Conjugate Gradient |
| CGB | `traincgb` - Conjugate Gradient with Powell/Beale Restarts |
| CGF | `traincgf` - Fletcher-Powell Conjugate Gradient |
| CGP | `traincgp` - Polak-Ribiére Conjugate Gradient |
| OSS | `trainoss` - One-Step Secant |
| GDX | `traingdx` - Variable Learning Rate Backpropagation |

The following table lists the six benchmark problems and some characteristics of the networks, training processes, and computers used.

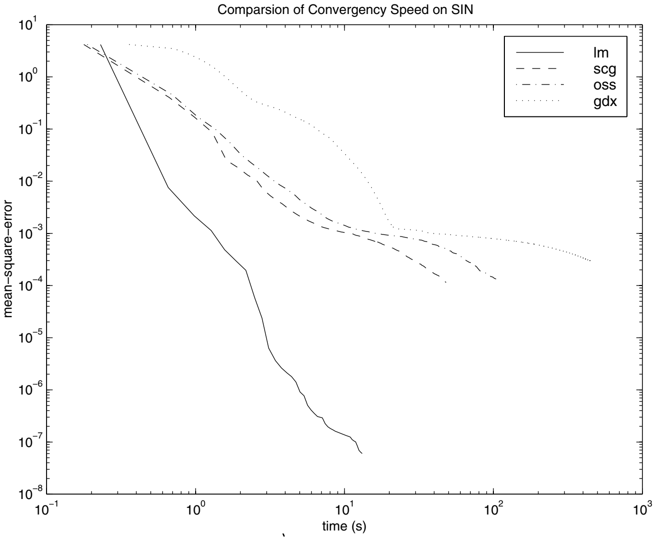
| Problem Title | Problem Type | Network Structure | Error Goal | Computer |
|---|---|---|---|---|
| SIN | Function Approx. | 1-5-1 | 0.002 | Sun Sparc 2 |
| PARITY | Pattern Recog. | 3-10-10-1 | 0.001 | Sun Sparc 2 |
| ENGINE | Function Approx. | 2-30-2 | 0.005 | Sun Enterprise 4000 |
| CANCER | Pattern Recog. | 9-5-5-2 | 0.012 | Sun Sparc 2 |
| CHOLESTEROL | Function Approx. | 21-15-3 | 0.027 | Sun Sparc 20 |
| DIABETES | Pattern Recog. | 8-15-15-2 | 0.05 | Sun Sparc 20 |

### SIN Data Set

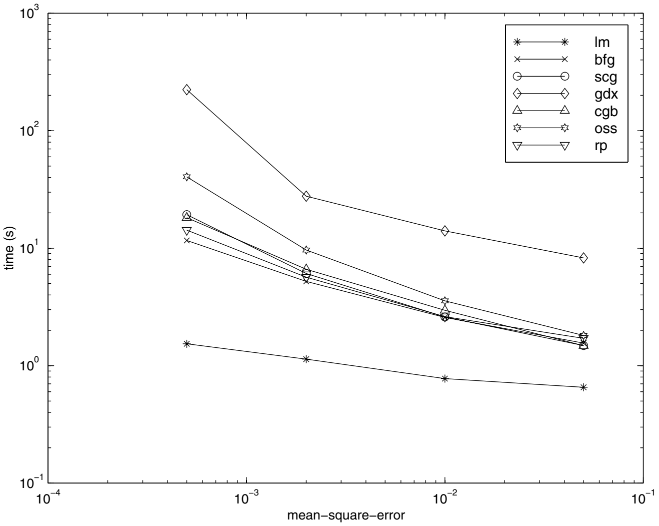
The first benchmark data set is a simple function approximation problem. A 1-5-1 network, with tansig transfer functions in the hidden layer and a linear transfer function in the output layer, is used to approximate a single period of a sine wave. The following table summarizes the results of training the network using nine different training algorithms. Each entry in the table represents 30 different trials, where different random initial weights are used in each trial. In each case, the network is trained until the squared error is less than 0.002. The fastest algorithm for this problem is the Levenberg-Marquardt algorithm. On the average, it is over four times faster than the next fastest algorithm. This is the type of problem for which the LM algorithm is best suited — a function approximation problem where the network has less than one hundred weights and the approximation must be very accurate.

| Algorithm | Mean Time (s) | Ratio | Min. Time (s) | Max. Time (s) | Std. (s) |
|---|---|---|---|---|---|
| LM | 1.14 | 1.00 | 0.65 | 1.83 | 0.38 |
| BFG | 5.22 | 4.58 | 3.17 | 14.38 | 2.08 |
| RP | 5.67 | 4.97 | 2.66 | 17.24 | 3.72 |
| SCG | 6.09 | 5.34 | 3.18 | 23.64 | 3.81 |
| CGB | 6.61 | 5.80 | 2.99 | 23.65 | 3.67 |
| CGF | 7.86 | 6.89 | 3.57 | 31.23 | 4.76 |
| CGP | 8.24 | 7.23 | 4.07 | 32.32 | 5.03 |
| OSS | 9.64 | 8.46 | 3.97 | 59.63 | 9.79 |
| GDX | 27.69 | 24.29 | 17.21 | 258.15 | 43.65 |

The performance of the various algorithms can be affected by the accuracy required of the approximation. This is demonstrated in the following figure, which plots the mean square error versus execution time (averaged over the 30 trials) for several representative algorithms. Here we can see that the error in the LM algorithm decreases much more rapidly with time than the other algorithms shown.



The relationship between the algorithms is further illustrated in the following figure, which plots the time required to converge versus the mean square error convergence goal. Here, we can see that as the error goal is reduced, the improvement provided by the LM algorithm becomes more pronounced. Some algorithms perform better as the error goal is reduced (LM and BFG), and other algorithms degrade as the error goal is reduced (OSS and GDX).

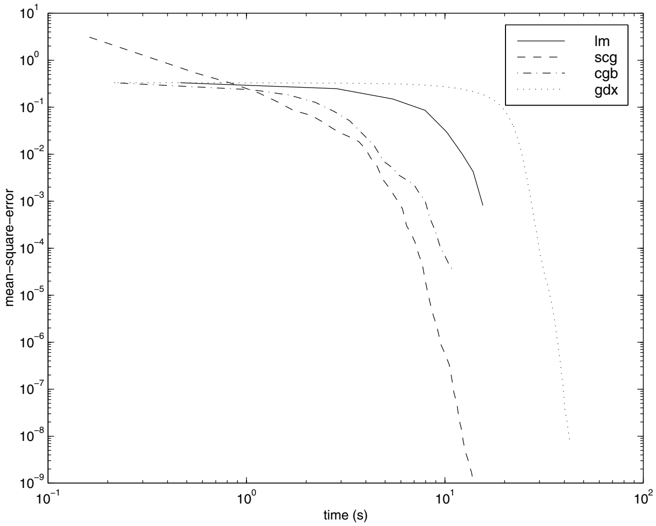


## PARITY Data Set

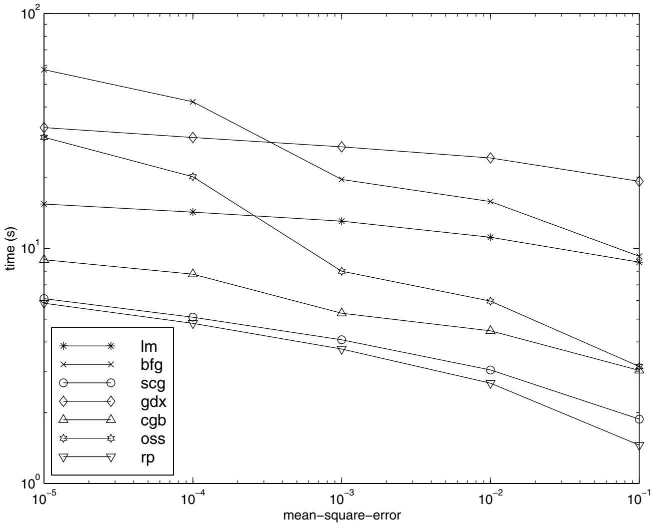
The second benchmark problem is a simple pattern recognition problem — detect the parity of a 3-bit number. If the number of ones in the input pattern is odd, then the network should output a one; otherwise, it should output a minus one. The network used for this problem is a 3-10-10-1 network with tansig neurons in each layer. The following table summarizes the results of training this network with the nine different algorithms. Each entry in the table represents 30 different trials, where different random initial weights are used in each trial. In each case, the network is trained until the squared error is less than 0.001. The fastest algorithm for this problem is the resilient backpropagation algorithm, although the conjugate gradient algorithms (in particular, the scaled conjugate gradient algorithm) are almost as fast. Notice that the LM algorithm does not perform well on this problem. In general, the LM algorithm does not perform as well on pattern recognition problems as it does on function approximation problems. The LM algorithm is designed for least squares problems that are approximately linear. Since the output neurons in pattern recognition problems will generally be saturated, we will not be operating in the linear region.

| Algorithm | Mean Time (s) | Ratio | Min. Time (s) | Max. Time (s) | Std. (s) |
|---|---|---|---|---|---|
| RP | 3.73 | 1.00 | 2.35 | 6.89 | 1.26 |
| SCG | 4.09 | 1.10 | 2.36 | 7.48 | 1.56 |
| CGP | 5.13 | 1.38 | 3.50 | 8.73 | 1.05 |
| CGB | 5.30 | 1.42 | 3.91 | 11.59 | 1.35 |
| CGF | 6.62 | 1.77 | 3.96 | 28.05 | 4.32 |
| OSS | 8.00 | 2.14 | 5.06 | 14.41 | 1.92 |
| LM | 13.07 | 3.50 | 6.48 | 23.78 | 4.96 |
| BFG | 19.68 | 5.28 | 14.19 | 26.64 | 2.85 |
| GDX | 27.07 | 7.26 | 25.21 | 28.52 | 0.86 |

As with function approximation problems, the performance of the various algorithms can be affected by the accuracy required of the network. This is demonstrated in the following figure, which plots the mean square error versus execution time for some typical algorithms. The LM algorithm converges rapidly after some point, but only after the other algorithms have already converged.



The relationship between the algorithms is further illustrated in the following figure, which plots the time required to converge versus the mean square error convergence goal. Again we can see that some algorithms degrade as the error goal is reduced (OSS and BFG).

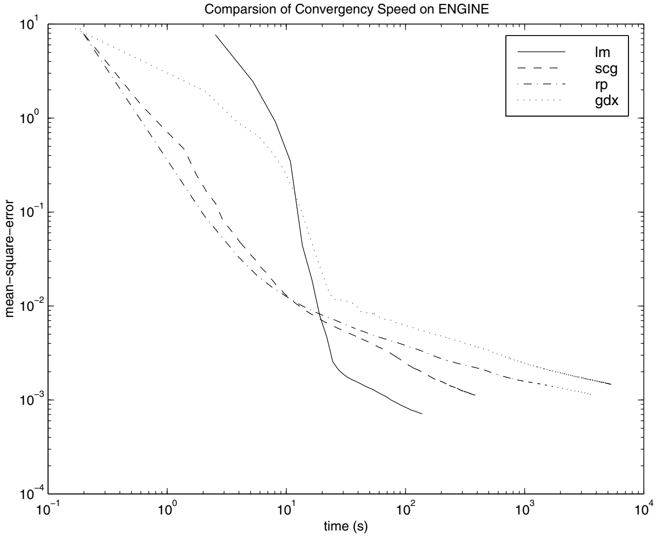


## ENGINE Data Set

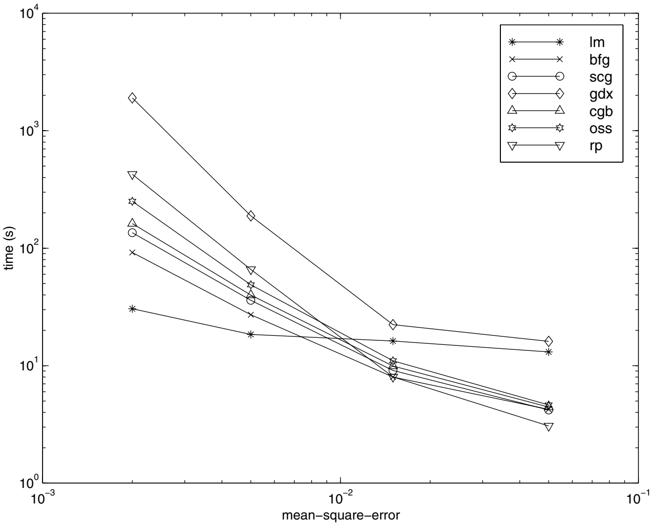
The third benchmark problem is a realistic function approximation (or nonlinear regression) problem. The data is obtained from the operation of an engine. The inputs to the network are engine speed and fueling levels and the network outputs are torque and emission levels. The network used for this problem is a 2-30-2 network with tansig neurons in the hidden layer and linear neurons in the output layer. The following table summarizes the results of training this network with the nine different algorithms. Each entry in the table represents 30 different trials (10 trials for RP and GDX because of time constraints), where different random initial weights are used in each trial. In each case, the network is trained until the squared error is less than 0.005. The fastest algorithm for this problem is the LM algorithm, although the BFGS quasi-Newton algorithm and the conjugate gradient algorithms (the scaled conjugate gradient algorithm in particular) are almost as fast. Although this is a function approximation problem, the LM algorithm is not as clearly superior as it was on the SIN data set. In this case, the number of weights and biases in the network is much larger than the one used on the SIN problem (152 versus. 16), and the advantages of the LM algorithm decrease as the number of network parameters increase.

| Algorithm | Mean Time (s) | Ratio | Min. Time (s) | Max. Time (s) | Std. (s) |
|---|---|---|---|---|---|
| LM | 18.45 | 1.00 | 12.01 | 30.03 | 4.27 |
| BFG | 27.12 | 1.47 | 16.42 | 47.36 | 5.95 |
| SCG | 36.02 | 1.95 | 19.39 | 52.45 | 7.78 |
| CGF | 37.93 | 2.06 | 18.89 | 50.34 | 6.12 |
| CGB | 39.93 | 2.16 | 23.33 | 55.42 | 7.50 |
| CGP | 44.30 | 2.40 | 24.99 | 71.55 | 9.89 |
| OSS | 48.71 | 2.64 | 23.51 | 80.90 | 12.33 |
| RP | 65.91 | 3.57 | 31.83 | 134.31 | 34.24 |
| GDX | 188.50 | 10.22 | 81.59 | 279.90 | 66.67 |

The following figure plots the mean square error versus execution time for some typical algorithms. The performance of the LM algorithm improves over time relative to the other algorithms.



The relationship between the algorithms is further illustrated in the following figure, which plots the time required to converge versus the mean square error convergence goal. Again we can see that some algorithms degrade as the error goal is reduced (GDX and RP), while the LM algorithm improves.



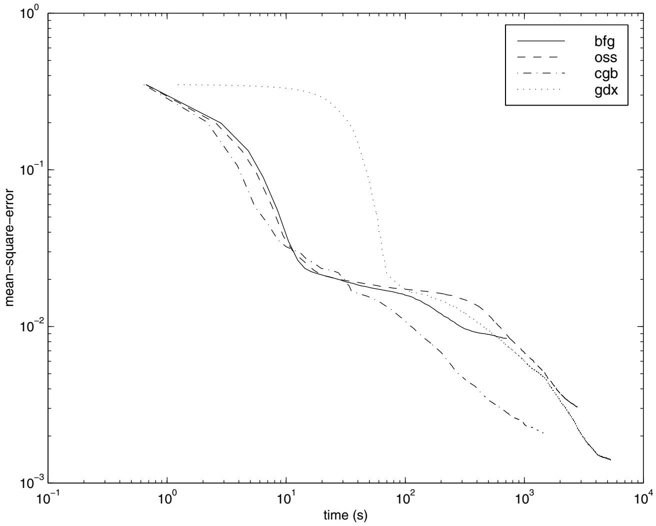
## CANCER Data Set

The fourth benchmark problem is a realistic pattern recognition (or nonlinear discriminant analysis) problem. The objective of the network is to classify a tumor as either benign or malignant based on cell descriptions gathered by microscopic examination. Input attributes include clump thickness, uniformity of cell size and cell shape, the amount of marginal adhesion, and the frequency of bare nuclei. The data was obtained from the University of Wisconsin Hospitals, Madison, from Dr. William H. Wolberg. The network used for this problem is a 9-5-5-2 network with tansig neurons in all layers. The following table summarizes the results of training this network with the nine different algorithms. Each entry in the table represents 30 different trials, where different random initial weights are used in each trial. In each case, the network is trained until the squared error is less than 0.012. A few runs failed to converge for some of the algorithms, so only the top 75% of the runs from each algorithm were used to obtain the statistics.

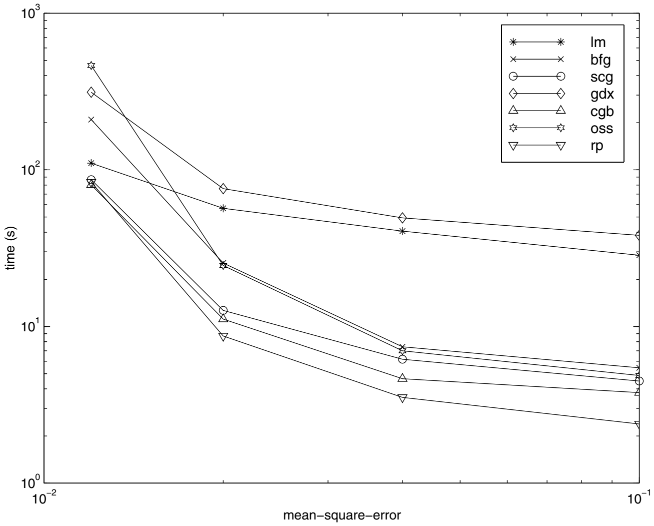
The conjugate gradient algorithms and resilient backpropagation all provide fast convergence, and the LM algorithm is also reasonably fast. As we mentioned with the parity data set, the LM algorithm does not perform as well on pattern recognition problems as it does on function approximation problems.

| Algorithm | Mean Time (s) | Ratio | Min. Time (s) | Max. Time (s) | Std. (s) |
|---|---|---|---|---|---|
| CGB | 80.27 | 1.00 | 55.07 | 102.31 | 13.17 |
| RP | 83.41 | 1.04 | 59.51 | 109.39 | 13.44 |
| SCG | 86.58 | 1.08 | 41.21 | 112.19 | 18.25 |
| CGP | 87.70 | 1.09 | 56.35 | 116.37 | 18.03 |
| CGF | 110.05 | 1.37 | 63.33 | 171.53 | 30.13 |
| LM | 110.33 | 1.37 | 58.94 | 201.07 | 38.20 |
| BFG | 209.60 | 2.61 | 118.92 | 318.18 | 58.44 |
| GDX | 313.22 | 3.90 | 166.48 | 446.43 | 75.44 |
| OSS | 463.87 | 5.78 | 250.62 | 599.99 | 97.35 |

The following figure plots the mean square error versus execution time for some typical algorithms. For this problem we don't see as much variation in performance as we have seen in previous problems.



The relationship between the algorithms is further illustrated in the following figure, which plots the time required to converge versus the mean square error convergence goal. Again we can see that some algorithms degrade as the error goal is reduced (OSS and BFG) while the LM algorithm improves. It is typical of the LM algorithm on any problem that its performance improves relative to other algorithms as the error goal is reduced.



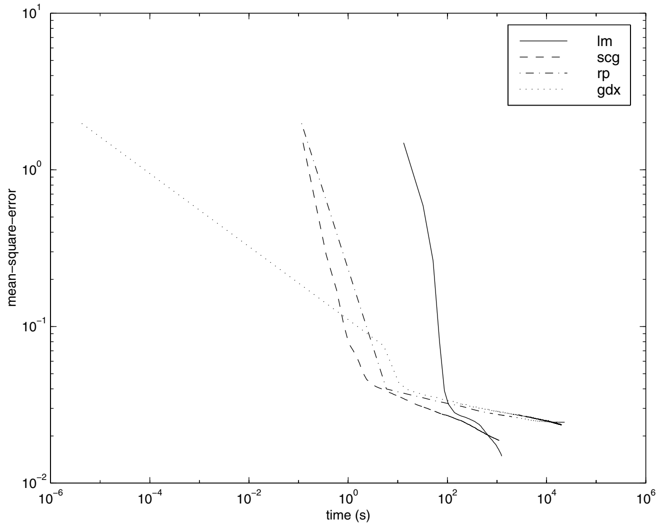
## CHOLESTEROL Data Set

The fifth benchmark problem is a realistic function approximation (or nonlinear regression) problem. The objective of the network is to predict cholesterol levels (ldl, hdl and vldl) based on measurements of 21 spectral components. The data was obtained from Dr. Neil Purdie, Department of Chemistry, Oklahoma State University [PuLu92]. The network used for this problem is a 21-15-3 network with tansig neurons in the hidden layers and linear neurons in the output layer. The following table summarizes the results of training this network with the nine different algorithms. Each entry in the table represents 20 different trials (10 trials for RP and GDX), where different random initial weights are used in each trial. In each case, the network is trained until the squared error is less than 0.027.

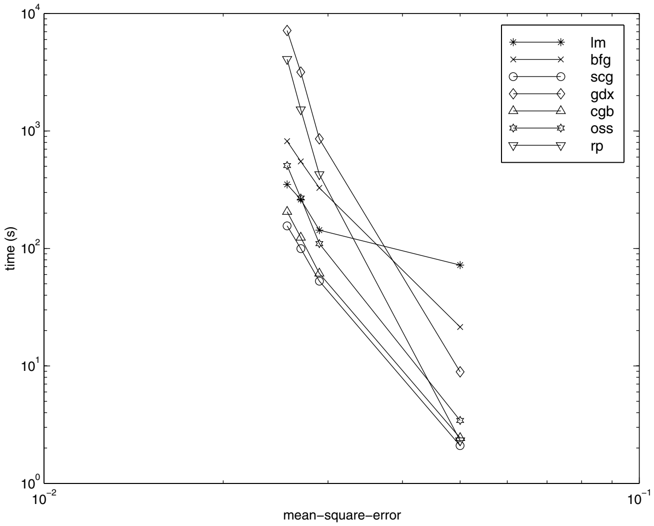
The scaled conjugate gradient algorithm has the best performance on this problem, although all of the conjugate gradient algorithms perform well. The LM algorithm does not perform as well on this function approximation problem as it did on the other two. That is because the number of weights and biases in the network has increased again (378 versus 152 versus 16). As the number of parameters increases, the computation required in the LM algorithm increases geometrically.

| Algorithm | Mean Time (s) | Ratio | Min. Time (s) | Max. Time (s) | Std. (s) |
|---|---|---|---|---|---|
| SCG | 99.73 | 1.00 | 83.10 | 113.40 | 9.93 |
| CGP | 121.54 | 1.22 | 101.76 | 162.49 | 16.34 |
| CGB | 124.06 | 1.24 | 107.64 | 146.90 | 14.62 |
| CGF | 136.04 | 1.36 | 106.46 | 167.28 | 17.67 |
| LM | 261.50 | 2.62 | 103.52 | 398.45 | 102.06 |
| OSS | 268.55 | 2.69 | 197.84 | 372.99 | 56.79 |
| BFG | 550.92 | 5.52 | 471.61 | 676.39 | 46.59 |
| RP | 1519.00 | 15.23 | 581.17 | 2256.10 | 557.34 |
| GDX | 3169.50 | 31.78 | 2514.90 | 4168.20 | 610.52 |

The following figure plots the mean square error versus execution time for some typical algorithms. For this problem, we can see that the LM algorithm is able to drive the mean square error to a lower level than the other algorithms. The SCG and RP algorithms provide the fastest initial convergence.



The relationship between the algorithms is further illustrated in the following figure, which plots the time required to converge versus the mean square error convergence goal. We can see that the LM and BFG algorithms improve relative to the other algorithms as the error goal is reduced.



### DIABETES Data Set

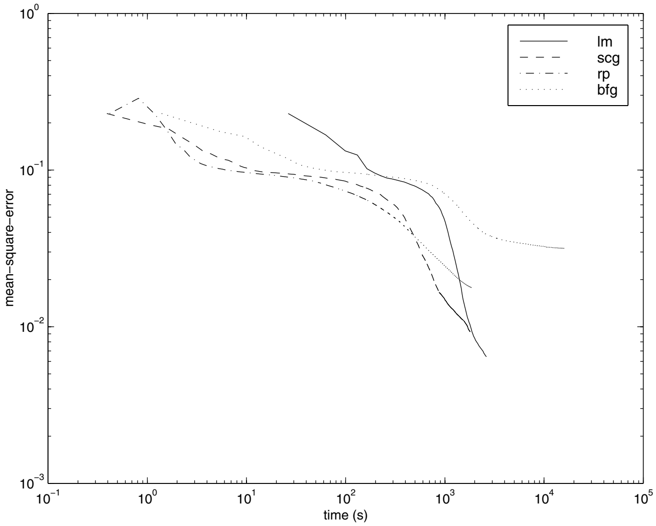
The sixth benchmark problem is a pattern recognition problem. The objective of the network is to decide if an individual has diabetes, based on personal data (age, number of times pregnant) and the results of medical examinations (e.g., blood pressure, body mass index, result of glucose tolerance test, etc.). The data was obtained from the University of California, Irvine, machine learning data base. The network used for this problem is an 8-15-15-2 network with tansig neurons in all layers. The following table summarizes the results of training this network with the nine different algorithms. Each entry in the table represents 10 different trials, where different random initial weights are used in each trial. In each case, the network is trained until the squared error is less than 0.05.

The conjugate gradient algorithms and resilient backpropagation all provide fast convergence. The results on this problem are consistent with the other pattern recognition problems we have considered. The RP algorithm works well on all of the pattern recognition problems. This is reasonable, since that algorithm was designed to overcome the difficulties caused by training with sigmoid functions, which have very small slopes when operating far from the center point. For pattern recognition problems, we use sigmoid transfer functions in the output layer, and we want the network to operate at the tails of the sigmoid function.

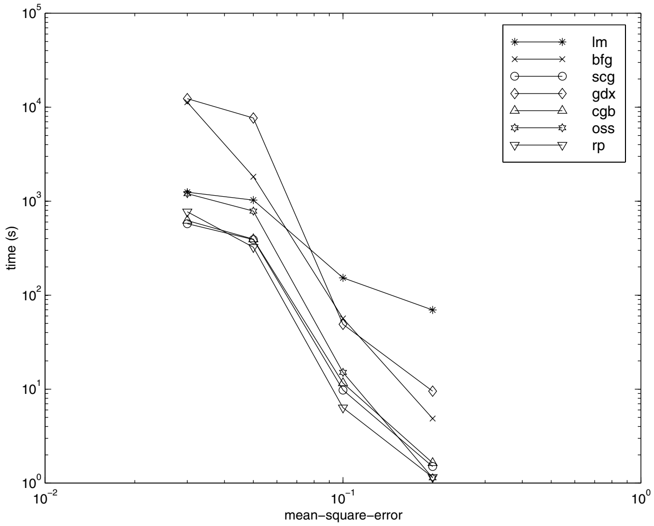
| Algorithm | Mean Time (s) | Ratio | Min. Time (s) | Max. Time (s) | Std. (s) |
|---|---|---|---|---|---|
| RP | 323.90 | 1.00 | 187.43 | 576.90 | 111.37 |
| SCG | 390.53 | 1.21 | 267.99 | 487.17 | 75.07 |
| CGB | 394.67 | 1.22 | 312.25 | 558.21 | 85.38 |
| CGP | 415.90 | 1.28 | 320.62 | 614.62 | 94.77 |
| OSS | 784.00 | 2.42 | 706.89 | 936.52 | 76.37 |
| CGF | 784.50 | 2.42 | 629.42 | 1082.20 | 144.63 |
| LM | 1028.10 | 3.17 | 802.01 | 1269.50 | 166.31 |

| Algorithm | Mean Time (s) | Ratio | Min. Time (s) | Max. Time (s) | Std. (s) |
|---|---|---|---|---|---|
| BFG | 1821.00 | 5.62 | 1415.80 | 3254.50 | 546.36 |
| GDX | 7687.00 | 23.73 | 5169.20 | 10350.00 | 2015.00 |

The following figure plots the mean square error versus execution time for some typical algorithms. As with other problems, we see that the SCG and RP have fast initial convergence, while the LM algorithm is able to provide smaller final error.



The relationship between the algorithms is further illustrated in the following figure, which plots the time required to converge versus the mean square error convergence goal. In this case, we can see that the BFG algorithm degrades as the error goal is reduced, while the LM algorithm improves. The RP algorithm is best, except at the smallest error goal, where SCG is better.



## Summary

There are several algorithm characteristics that we can deduce from the experiments we have described. In general, on function approximation problems, for networks that contain up to a few hundred weights, the Levenberg-Marquardt algorithm will have the fastest convergence. This advantage is especially noticeable if very accurate training is required. In many cases, `trainlm` is able to obtain lower mean square errors than any of the other algorithms tested. However, as the number of weights in the network increases, the advantage of the `trainlm` decreases. In addition, `trainlm` performance is relatively poor on pattern recognition problems. The storage requirements of `trainlm` are larger than the other algorithms tested. By adjusting the `mem_reduc` parameter, discussed earlier, the storage requirements can be reduced, but at a cost of increased execution time.

The `trainrp` function is the fastest algorithm on pattern recognition problems. However, it does not perform well on function approximation problems. Its performance also degrades as the error goal is reduced. The memory requirements for this algorithm are relatively small in comparison to the other algorithms considered.

The conjugate gradient algorithms, in particular `trainscg`, seem to perform well over a wide variety of problems, particularly for networks with a large number of weights. The SCG algorithm is almost as fast as the LM algorithm on function approximation problems (faster for large networks) and is almost as fast as trainrp on pattern recognition problems. Its performance does not degrade as quickly as trainrp performance does when the error is reduced. The conjugate gradient algorithms have relatively modest memory requirements.

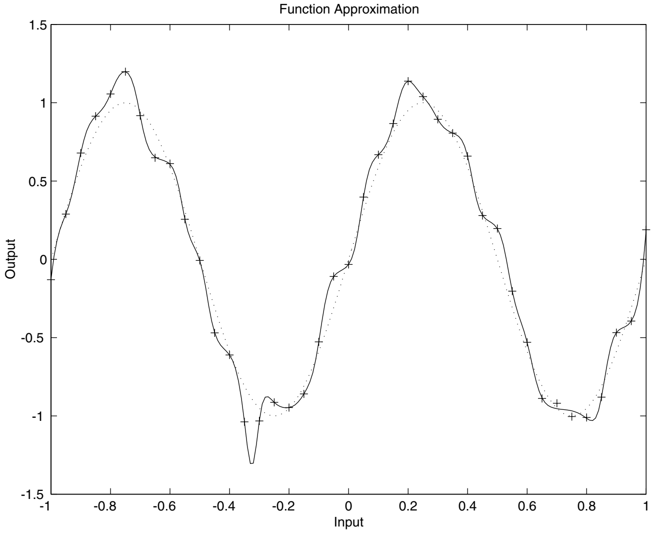
The trainbfg performance is similar to that of trainlm. It does not require as much storage as trainlm, but the computation required does increase geometrically with the size of the network, since the equivalent of a matrix inverse must be computed at each iteration.

The variable learning rate algorithm traingdx is usually much slower than the other methods, and has about the same storage requirements as trainrp, but it can still be useful for some problems. There are certain situations in which it is better to converge more slowly. For example, when using early stopping (as described in the next section) you may have inconsistent results if you use an algorithm that converges too quickly. You may overshoot the point at which the error on the validation set is minimized.

# Improving Generalization

One of the problems that occurs during neural network training is called overfitting. The error on the training set is driven to a very small value, but when new data is presented to the network the error is large. The network has memorized the training examples, but it has not learned to generalize to new situations.

The following figure shows the response of a 1-20-1 neural network that has been trained to approximate a noisy sine function. The underlying sine function is shown by the dotted line, the noisy measurements are given by the '+' symbols, and the neural network response is given by the solid line. Clearly this network has overfit the data and will not generalize well.



One method for improving network generalization is to use a network that is just large enough to provide an adequate fit. The larger a network you use, the more complex the functions the network can create. If we use a small enough network, it will not have enough power to overfit the data. Run the Neural Network Design Demonstration `nnd11gn` [HDB96] to investigate how reducing the size of a network can prevent overfitting.

Unfortunately, it is difficult to know beforehand how large a network should be for a specific application. There are two other methods for improving generalization that are implemented in the Neural Network Toolbox: regularization and early stopping. The next few subsections describe these two techniques, and the routines to implement them.

Note that if the number of parameters in the network is much smaller than the total number of points in the training set, then there is little or no chance of overfitting. If you can easily collect more data and increase the size of the training set, then there is no need to worry about the following techniques to prevent overfitting. The rest of this section only applies to those situations in which you want to make the most of a limited supply of data.

## Regularization

The first method for improving generalization is called regularization. This involves modifying the performance function, which is normally chosen to be the sum of squares of the network errors on the training set. The next subsection explains how the performance function can be modified, and the following subsection describes a routine that automatically sets the optimal performance function to achieve the best generalization.

### Modified Performance Function

The typical performance function that is used for training feedforward neural networks is the mean sum of squares of the network errors.

$$F = mse = \frac{1}{N}\sum_{i=1}^{N}(e_i)^2 = \frac{1}{N}\sum_{i=1}^{N}(t_i - a_i)^2$$

It is possible to improve generalization if we modify the performance function by adding a term that consists of the mean of the sum of squares of the network weights and biases

$$msereg = \gamma mse + (1-\gamma)msw$$

where $\gamma$ is the performance ratio, and

$$msw = \frac{1}{n}\sum_{j=1}^{n} w_j^2$$

Using this performance function will cause the network to have smaller weights and biases, and this will force the network response to be smoother and less likely to overfit.

In the following code we reinitialize our previous network and retrain it using the BFGS algorithm with the regularized performance function. Here we set the performance ratio to 0.5, which gives equal weight to the mean square errors and the mean square weights.

```
p = [-1 -1 2 2;0 5 0 5];
t = [-1 -1 1 1];
net=newff(minmax(p),[3,1],{'tansig','purelin'},'trainbfg');
net.performFcn = 'msereg';
net.performParam.ratio = 0.5;
net.trainParam.show = 5;
net.trainParam.epochs = 300;
net.trainParam.goal = 1e-5;
[net,tr]=train(net,p,t);
```

The problem with regularization is that it is difficult to determine the optimum value for the performance ratio parameter. If we make this parameter too large, we may get overfitting. If the ratio is too small, the network will not adequately fit the training data. In the next section we describe a routine that automatically sets the regularization parameters.

### Automated Regularization (trainbr)

It is desirable to determine the optimal regularization parameters in an automated fashion. One approach to this process is the Bayesian framework of David MacKay [MacK92]. In this framework, the weights and biases of the network are assumed to be random variables with specified distributions. The regularization parameters are related to the unknown variances associated with these distributions. We can then estimate these parameters using statistical techniques.

A detailed discussion of Bayesian regularization is beyond the scope of this users guide. A detailed discussion of the use of Bayesian regularization, in combination with Levenberg-Marquardt training, can be found in [FoHa97].

Bayesian regularization has been implemented in the function `trainbr`. The following code shows how we can train a 1-20-1 network using this function to approximate the noisy sine wave shown earlier in this section.

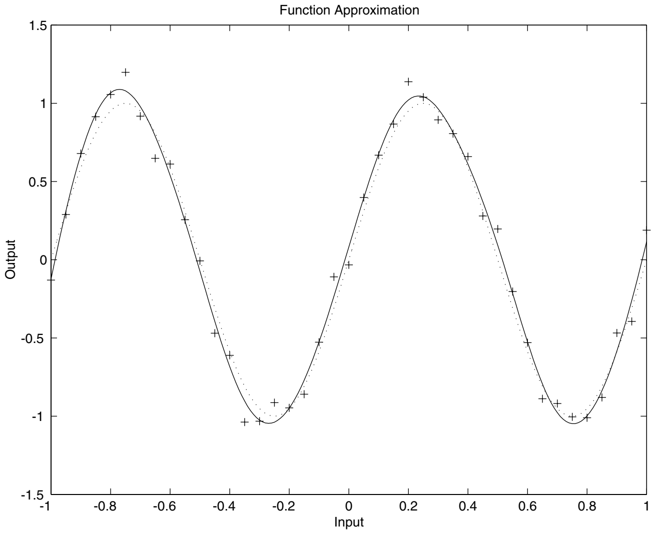
```
p = [-1:.05:1];
t = sin(2*pi*p)+0.1*randn(size(p));
net=newff(minmax(p),[20,1],{'tansig','purelin'},'trainbr');
net.trainParam.show = 10;
net.trainParam.epochs = 50;
randn('seed',192736547);
net = init(net);
[net,tr]=train(net,p,t);
TRAINBR, Epoch 0/200, SSE 273.764/0, SSW 21460.5, Grad
2.96e+02/1.00e-10, #Par 6.10e+01/61
TRAINBR, Epoch 40/200, SSE 0.255652/0, SSW 1164.32, Grad
1.74e-02/1.00e-10, #Par 2.21e+01/61
TRAINBR, Epoch 80/200, SSE 0.317534/0, SSW 464.566, Grad
5.65e-02/1.00e-10, #Par 1.78e+01/61
TRAINBR, Epoch 120/200, SSE 0.379938/0, SSW 123.028, Grad
3.64e-01/1.00e-10, #Par 1.17e+01/61
TRAINBR, Epoch 160/200, SSE 0.380578/0, SSW 108.294, Grad
6.43e-02/1.00e-10, #Par 1.19e+01/61
```

One feature of this algorithm is that it provides a measure of how many network parameters (weights and biases) are being effectively used by the network. In this case, the final trained network uses approximately 12 parameters (indicated by `#Par` in the printout) out of the 61 total weights and biases in the 1-20-1 network. This effective number of parameters should remain approximately the same, no matter how large the total number of parameters in the network becomes. (This assumes that the network has been trained for a sufficient number of iterations to ensure convergence.)

The `trainbr` algorithm generally works best when the network inputs and targets are scaled so that they fall approximately in the range [-1,1]. That is the case for the test problem we have used. If your inputs and targets do not fall in this range, you can use the functions `premnmx`, or `prestd`, to perform the scaling, as described later in this chapter.

The following figure shows the response of the trained network. In contrast to the previous figure, in which a 1-20-1 network overfit the data, here we see that the network response is very close to the underlying sine function (dotted line), and, therefore, the network will generalize well to new inputs. We could have tried an even larger network, but the network response would never overfit the data. This eliminates the guesswork required in determining the optimum network size.

When using `trainbr`, it is important to let the algorithm run until the effective number of parameters has converged. The training may stop with the message "Maximum MU reached." This is typical, and is a good indication that the algorithm has truly converged. You can also tell that the algorithm has converged if the sum squared error (SSE) and sum squared weights (SSW) are relatively constant over several iterations. When this occurs you may want to push the "Stop Training" button in the training window.



## Early Stopping

Another method for improving generalization is called *early stopping*. In this technique the available data is divided into three subsets. The first subset is the training set, which is used for computing the gradient and updating the network weights and biases. The second subset is the validation set. The error on the validation set is monitored during the training process. The validation error will normally decrease during the initial phase of training, as does the training set error. However, when the network begins to overfit the data, the error on the validation set will typically begin to rise. When the validation error increases for a specified number of iterations, the training is stopped, and the weights and biases at the minimum of the validation error are returned.

The test set error is not used during the training, but it is used to compare different models. It is also useful to plot the test set error during the training process. If the error in the test set reaches a minimum at a significantly different iteration number than the validation set error, this may indicate a poor division of the data set.

Early stopping can be used with any of the training functions that were described earlier in this chapter. You simply need to pass the validation data to the training function. The following sequence of commands demonstrates how to use the early stopping function.

First we create a simple test problem. For our training set we generate a noisy sine wave with input points ranging from -1 to 1 at steps of 0.05.

```
p = [-1:0.05:1];
t = sin(2*pi*p)+0.1*randn(size(p));
```

Next we generate the validation set. The inputs range from -1 to 1, as in the test set, but we offset them slightly. To make the problem more realistic, we also add a different noise sequence to the underlying sine wave. Notice that the validation set is contained in a structure that contains both the inputs and the targets.

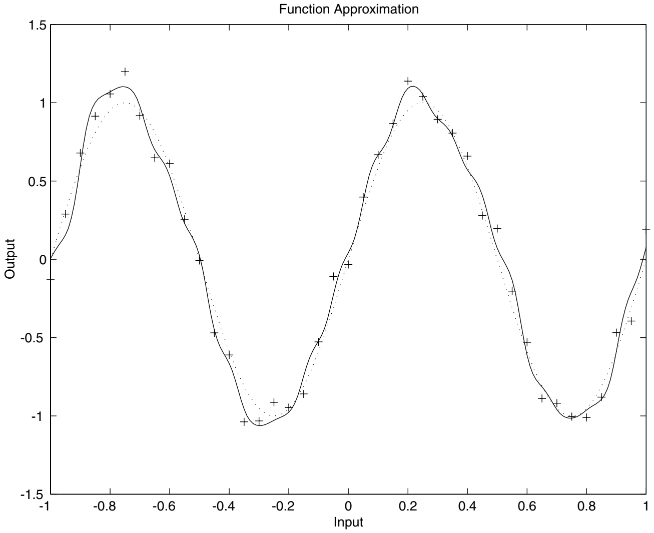
```
val.P = [-0.975:.05:0.975];
val.T = sin(2*pi*v.P)+0.1*randn(size(v.P));
```

We now create a 1-20-1 network, as in our previous example with regularization, and train it. (Notice that the validation structure is passed to `train` after the initial input and layer conditions, which are null vectors in this case since the network contains no delays. Also, in this example we are not using a test set. The test set structure would be the next argument in the call to train.) For this example we use the training function `traingdx`, although early stopping can be used with any of the other training functions we have discussed in this chapter.

```
net=newff([-1 1],[20,1],{'tansig','purelin'},'traingdx');
net.trainParam.show = 25;
net.trainParam.epochs = 300;
net = init(net);
[net,tr]=train(net,p,t,[],[],val);
TRAINGDX, Epoch 0/300, MSE 9.39342/0, Gradient 17.7789/1e-06
TRAINGDX, Epoch 25/300, MSE 0.312465/0, Gradient 0.873551/1e-06
TRAINGDX, Epoch 50/300, MSE 0.102526/0, Gradient 0.206456/1e-06
TRAINGDX, Epoch 75/300, MSE 0.0459503/0, Gradient 0.0954717/1e-06
TRAINGDX, Epoch 100/300, MSE 0.015725/0, Gradient 0.0299898/1e-06
```

```
TRAINGDX, Epoch 125/300, MSE 0.00628898/0, Gradient
0.042467/1e-06
TRAINGDX, Epoch 131/300, MSE 0.00650734/0, Gradient
0.133314/1e-06
TRAINGDX, Validation stop.
```

The following figure shows a graph of the network response. We can see that the network did not overfit the data, as in the earlier example, although the response is not extremely smooth, as when using regularization. This is characteristic of early stopping.



## Summary and Discussion

Both regularization and early stopping can ensure network generalization when properly applied.

When using Bayesian regularization, it is important to train the network until it reaches convergence. The sum squared error, the sum squared weights, and the effective number of parameters should reach constant values when the network has converged.

For early stopping, you must be careful not to use an algorithm that converges too rapidly. If you are using a fast algorithm (like `trainlm`), you want to set the training parameters so that the convergence is relatively slow (e.g., set `mu` to a relatively large value, such as 1, and set `mu_dec` and `mu_inc` to values close to 1, such as 0.8 and 1.5, respectively). The training functions `trainscg` and `trainrp` usually work well with early stopping.

With early stopping, the choice of the validation set is also important. The validation set should be representative of all points in the training set.

With both regularization and early stopping, it is a good idea to train the network starting from several different initial conditions. It is possible for either method to fail in certain circumstances. By testing several different initial conditions, you can verify robust network performance.

Based on our experience, Bayesian regularization generally provides better generalization performance than early stopping, when training function approximation networks. This is because Bayesian regularization does not require that a validation data set be separated out of the training data set. It uses all of the data. This advantage is especially noticeable when the size of the data set is small.

To provide you with some insight into the performance of the algorithms, we tested both early stopping and Bayesian regularization on several benchmark data sets, which are listed in the following table.

| **Data Set Title** | **No. pts.** | **Network** | **Description** |
|---|---|---|---|
| BALL | 67 | 2-10-1 | Dual-sensor calibration for a ball position measurement. |
| SINE (5% N) | 41 | 1-15-1 | Single-cycle sine wave with Gaussian noise at 5% level. |
| SINE (2% N) | 41 | 1-15-1 | Single-cycle sine wave with Gaussian noise at 2% level. |
| ENGINE (ALL) | 1199 | 2-30-2 | Engine sensor - full data set. |
| ENGINE (1/4) | 300 | 2-30-2 | Engine sensor - 1/4 of data set. |

| Data Set Title | No. pts. | Network | Description |
|---|---|---|---|
| CHOLEST (ALL) | 264 | 5-15-3 | Cholesterol measurement - full data set. |
| CHOLEST (1/2) | 132 | 5-15-3 | Cholesterol measurement - 1/2 data set. |

These data sets are of various sizes, with different numbers of inputs and targets. With two of the data sets we trained the networks once using all of the data and then retrained the networks using only a fraction of the data. This illustrates how the advantage of Bayesian regularization becomes more noticeable when the data sets are smaller. All of the data sets are obtained from physical systems, except for the SINE data sets. These two were artificially created by adding various levels of noise to a single cycle of a sine wave. The performance of the algorithms on these two data sets illustrates the effect of noise.

The following table summarizes the performance of Early Stopping (ES) and Bayesian Regularization (BR) on the seven test sets. (The `trainscg` algorithm was used for the early stopping tests. Other algorithms provide similar performance.)

**Mean Squared Test Set Error**

| Method | Ball | Engine (All) | Engine (1/4) | Choles (All) | Choles (1/2) | Sine (5% N) | Sine (2% N) |
|---|---|---|---|---|---|---|---|
| ES | 1.2e-1 | 1.3e-2 | 1.9e-2 | 1.2e-1 | 1.4e-1 | 1.7e-1 | 1.3e-1 |
| BR | 1.3e-3 | 2.6e-3 | 4.7e-3 | 1.2e-1 | 9.3e-2 | 3.0e-2 | 6.3e-3 |
| ES/BR | 92 | 5 | 4 | 1 | 1.5 | 5.7 | 21 |

We can see that Bayesian regularization performs better than early stopping in most cases. The performance improvement is most noticeable when the data set is small, or if there is little noise in the data set. The BALL data set, for example, was obtained from sensors that had very little noise.

Although the generalization performance of Bayesian regularization is often better than early stopping, this is not always the case. In addition, the form of

Bayesian regularization implemented in the toolbox does not perform as well on pattern recognition problems as it does on function approximation problems. This is because the approximation to the Hessian that is used in the Levenberg-Marquardt algorithm is not as accurate when the network output is saturated, as would be the case in pattern recognition problems. Another disadvantage of the Bayesian regularization method is that it generally takes longer to converge than early stopping.

# Preprocessing and Postprocessing

Neural network training can be made more efficient if certain preprocessing steps are performed on the network inputs and targets. In this section, we describe several preprocessing routines that you can use.

## Min and Max (premnmx, postmnmx, tramnmx)

Before training, it is often useful to scale the inputs and targets so that they always fall within a specified range. The function premnmx can be used to scale inputs and targets so that they fall in the range [-1,1]. The following code illustrates the use of this function.

```
[pn,minp,maxp,tn,mint,maxt] = premnmx(p,t);
net=train(net,pn,tn);
```

The original network inputs and targets are given in the matrices p and t. The normalized inputs and targets, pn and tn, that are returned will all fall in the interval [-1,1]. The vectors minp and maxp contain the minimum and maximum values of the original inputs, and the vectors mint and maxt contain the minimum and maximum values of the original targets. After the network has been trained, these vectors should be used to transform any future inputs that are applied to the network. They effectively become a part of the network, just like the network weights and biases.

If premnmx is used to scale both the inputs and targets, then the output of the network will be trained to produce outputs in the range [-1,1]. If you want to convert these outputs back into the same units that were used for the original targets, then you should use the routine postmnmx. In the following code, we simulate the network that was trained in the previous code, and then convert the network output back into the original units.

```
an = sim(net,pn);
a = postmnmx(an,mint,maxt);
```

The network output an will correspond to the normalized targets tn. The un-normalized network output a is in the same units as the original targets t.

If premnmx is used to preprocess the training set data, then whenever the trained network is used with new inputs they should be preprocessed with the minimum and maximums that were computed for the training set. This can be accomplished with the routine tramnmx. In the following code, we apply a new set of inputs to the network we have already trained.

```
pnewn = tramnmx(pnew,minp,maxp);
anewn = sim(net,pnewn);
anew = postmnmx(anewn,mint,maxt);
```

## Mean and Stand. Dev. (prestd, poststd, trastd)

Another approach for scaling network inputs and targets is to normalize the mean and standard deviation of the training set. This procedure is implemented in the function prestd. It normalizes the inputs and targets so that they will have zero mean and unity standard deviation. The following code illustrates the use of prestd.

```
[pn,meanp,stdp,tn,meant,stdt] = prestd(p,t);
```

The original network inputs and targets are given in the matrices p and t. The normalized inputs and targets, pn and tn, that are returned will have zero means and unity standard deviation. The vectors meanp and stdp contain the mean and standard deviations of the original inputs, and the vectors meant and stdt contain the means and standard deviations of the original targets. After the network has been trained, these vectors should be used to transform any future inputs that are applied to the network. They effectively become a part of the network, just like the network weights and biases.

If prestd is used to scale both the inputs and targets, then the output of the network is trained to produce outputs with zero mean and unity standard deviation. If you want to convert these outputs back into the same units that were used for the original targets, then you should use the routine poststd. In the following code we simulate the network that was trained in the previous code, and then convert the network output back into the original units.

```
an = sim(net,pn);
a = poststd(an,meant,stdt);
```

The network output an corresponds to the normalized targets tn. The un-normalized network output a is in the same units as the original targets t.

If prestd is used to preprocess the training set data, then whenever the trained network is used with new inputs, they should be preprocessed with the means and standard deviations that were computed for the training set. This can be accomplished with the routine trastd. In the following code, we apply a new set of inputs to the network we have already trained.

```
pnewn = trastd(pnew,meanp,stdp);
anewn = sim(net,pnewn);
anew = poststd(anewn,meant,stdt);
```

## Principal Component Analysis (prepca, trapca)

In some situations, the dimension of the input vector is large, but the components of the vectors are highly correlated (redundant). It is useful in this situation to reduce the dimension of the input vectors. An effective procedure for performing this operation is principal component analysis. This technique has three effects: it orthogonalizes the components of the input vectors (so that they are uncorrelated with each other); it orders the resulting orthogonal components (principal components) so that those with the largest variation come first; and it eliminates those components that contribute the least to the variation in the data set. The following code illustrates the use of prepca, which performs a principal component analysis.

```
[pn,meanp,stdp] = prestd(p);
[ptrans,transMat] = prepca(pn,0.02);
```

Note that we first normalize the input vectors, using prestd, so that they have zero mean and unity variance. This is a standard procedure when using principal components. In this example, the second argument passed to prepca is 0.02. This means that prepca eliminates those principal components that contribute less than 2% to the total variation in the data set. The matrix ptrans contains the transformed input vectors. The matrix transMat contains the principal component transformation matrix. After the network has been trained, this matrix should be used to transform any future inputs that are applied to the network. It effectively becomes a part of the network, just like the network weights and biases. If you multiply the normalized input vectors pn by the transformation matrix transMat, you obtain the transformed input vectors ptrans.

If prepca is used to preprocess the training set data, then whenever the trained network is used with new inputs they should be preprocessed with the transformation matrix that was computed for the training set. This can be accomplished with the routine trapca. In the following code, we apply a new set of inputs to a network we have already trained.

```
pnewn = trastd(pnew,meanp,stdp);
pnewtrans = trapca(pnewn,transMat);
a = sim(net,pnewtrans);
```

## Post-Training Analysis (postreg)

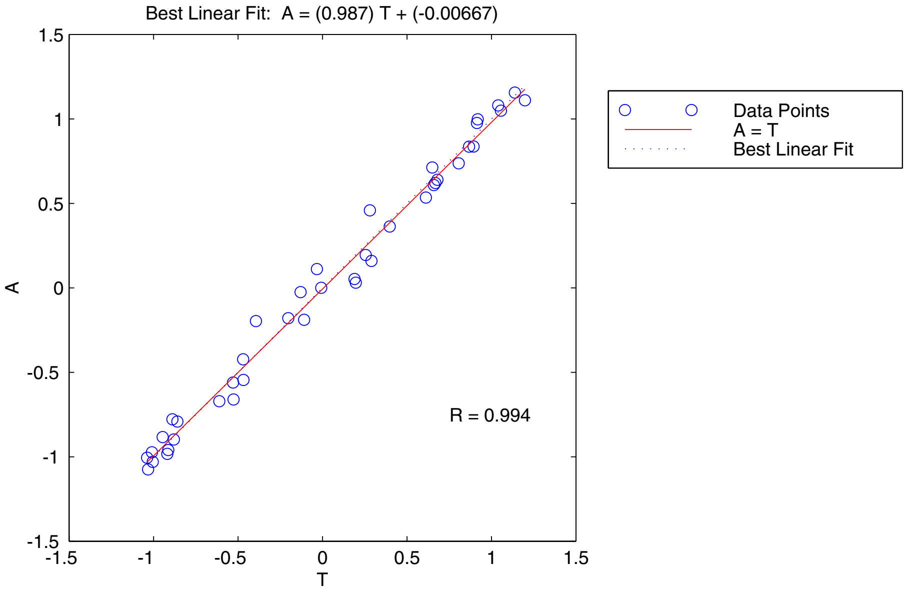
The performance of a trained network can be measured to some extent by the errors on the training, validation and test sets, but it is often useful to investigate the network response in more detail. One option is to perform a regression analysis between the network response and the corresponding targets. The routine postreg is designed to perform this analysis.

The following commands illustrate how we can perform a regression analysis on the network that we previously trained in the early stopping section.

```
a = sim(net,p);
[m,b,r] = postreg(a,t)
m =
    0.9874
b =
   -0.0067
r =
    0.9935
```

Here we pass the network output and the corresponding targets to postreg. It returns three parameters. The first two, m and b, correspond to the slope and the y-intercept of the best linear regression relating targets to network outputs. If we had a perfect fit (outputs exactly equal to targets), the slope would be 1, and the y-intercept would be 0. In this example, we can see that the numbers are very close. The third variable returned by postreg is the correlation coefficient (R-value) between the outputs and targets. It is a measure of how well the variation in the output is explained by the targets. If this number is equal to 1, then there is perfect correlation between targets and outputs. In our example, the number is very close to 1, which indicates a good fit.

The following figure illustrates the graphical output provided by postreg. The network outputs are plotted versus the targets as open circles. The best linear fit is indicated by a dashed line. The perfect fit (output equal to targets) is indicated by the solid line. In this example, it is difficult to distinguish the best linear fit line from the perfect fit line, because the fit is so good.


Best Linear Fit:  A = (0.987) T + (-0.00667)
1.5
1
0.5
0
-0.5
-1
-1.5
A
-1.5
-1
-0.5
0
0.5
1
1.5
T
R = 0.994
○ ○ Data Points
A = T
Best Linear Fit

## Sample Training Session

We have covered a number of different concepts in this chapter. At this point it might be useful to put some of these ideas together with an example of how a typical training session might go.

For this example, we are going to use data from a medical application [PuLu92]. We want to design an instrument that can determine serum cholesterol levels from measurements of spectral content of a blood sample. We have a total of 264 patients for which we have measurements of 21 wavelengths of the spectrum. For the same patients we also have measurements of hdl, ldl, and vldl cholesterol levels, based on serum separation. The first step is to load the data into the workspace and perform a principal component analysis.

```
load choles_all
[pn,meanp,stdp,tn,meant,stdt] = prestd(p,t);
[ptrans,transMat] = prepca(pn,0.001);
```

Here we have conservatively retained those principal components which account for 99.9% of the variation in the data set. Let's check the size of the transformed data.

```
[R,Q] = size(ptrans)
R =
  4
Q =
  264
```

There was apparently significant redundancy in the data set, since the principal component analysis has reduced the size of the input vectors from 21 to 4.

The next step is to divide the data up into training, validation and test subsets. We will take one fourth of the data for the validation set, one fourth for the test set and one half for the training set. We pick the sets as equally spaced points throughout the original data.

```
iitst = 2:4:Q;
iival = 4:4:Q;
iitr = [1:4:Q 3:4:Q];
val.P = ptrans(:,iival); val.T = tn(:,iival);
test.P = ptrans(:,iitst); test.T = tn(:,iitst);
ptr = ptrans(:,iitr); ttr = tn(:,iitr);
```

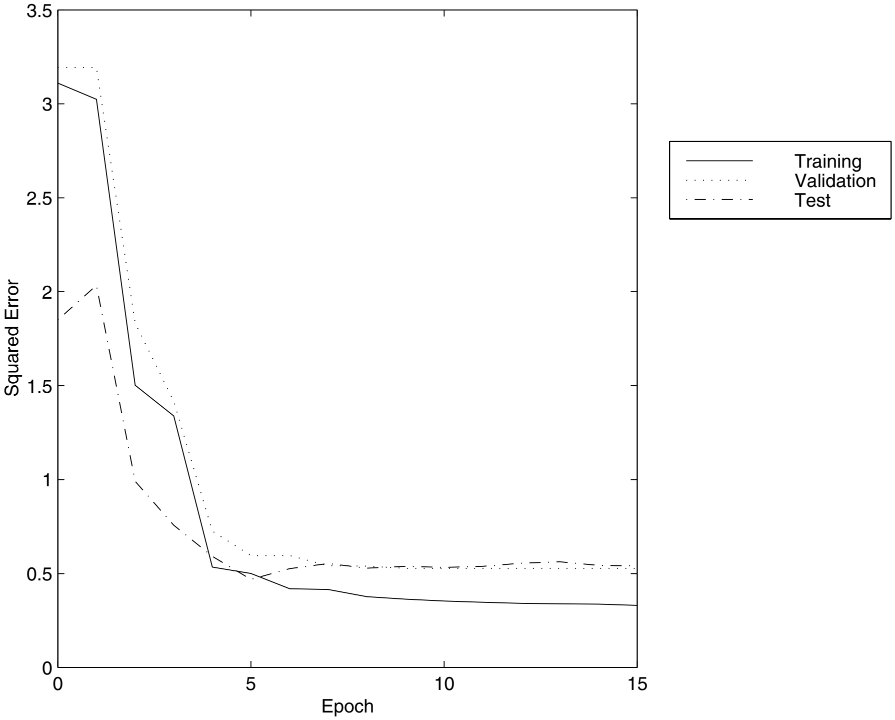
We are now ready to create a network and train it. For this example, we will try a two-layer network, with tan-sigmoid transfer function in the hidden layer and a linear transfer function in the output layer. This is a useful structure for function approximation (or regression) problems. As an initial guess, we use five neurons in the hidden layer. The network should have three output neurons since there are three targets. We will use the Levenberg-Marquardt algorithm for training.

```
net = newff(minmax(ptr),[5 3],{'tansig' 'purelin'},'trainlm');
[net,tr]=train(net,ptr,ttr,[],[],val,test);
TRAINLM, Epoch 0/100, MSE 3.11023/0, Gradient 804.959/1e-10
TRAINLM, Epoch 15/100, MSE 0.330295/0, Gradient 104.219/1e-10
TRAINLM, Validation stop.
```

The training stopped after 15 iterations because the validation error increased. It is a useful diagnostic tool to plot the training, validation and test errors to check the progress of training. We can do that with the following commands.

```
plot(tr.epoch,tr.perf,tr.epoch,tr.vperf,tr.epoch,tr.tperf)
legend('Training','Validation','Test',-1);
ylabel('Squared Error'); xlabel('Epoch')
```

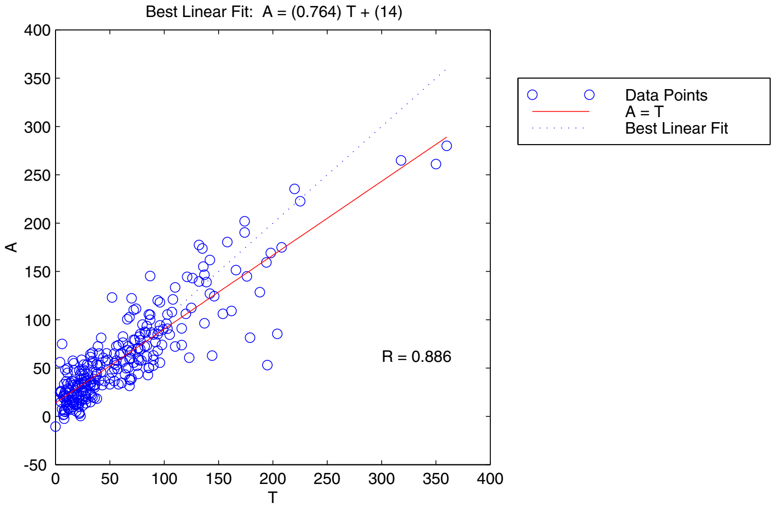
The result is shown in the following figure. The result here is reasonable, since the test set error and the validation set error have similar characteristics, and it doesn’t appear that any significant overfitting has occurred.

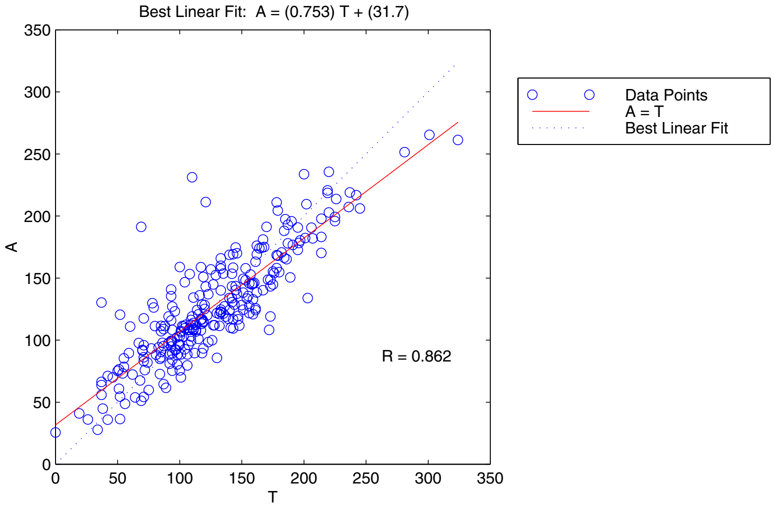


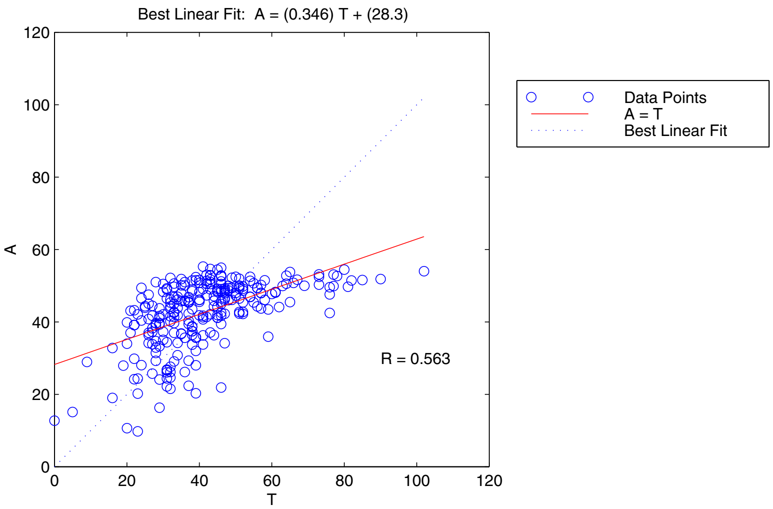
The next step is to perform some analysis of the network response. We will put the entire data set through the network (training, validation and test) and will perform a linear regression between the network outputs and the corresponding targets. First we need to unnormalize the network outputs.

```
an = sim(net,ptrans);
a = poststd(an,meant,stdt);
for i=1:3
  figure(i)
  [m(i),b(i),r(i)] = postreg(a(i,:),t(i,:));
end
```

In this case, we have three outputs, so we perform three regressions. The results are shown in the following figures.


Best Linear Fit: A = (0.764) T + (14)
400
350
300
250
200
150
100
50
0
-50
A
0
50
100
150
200
250
300
350
400
T
R = 0.886
Data Points
A = T
Best Linear Fit


Best Linear Fit: A = (0.753) T + (31.7)
350
300
250
200
150
100
50
0
A
0
50
100
150
200
250
300
350
T
R = 0.862
Data Points
A = T
Best Linear Fit



The first two outputs seem to track the targets reasonably well (this is a difficult problem), and the R-values are almost 0.9. The third output (vldl levels) is not well modeled. We probably need to work more on that problem. We might go on to try other network architectures (more hidden layer neurons), or to try Bayesian regularization instead of early stopping for our training technique. Of course there is also the possibility that vldl levels cannot be accurately computed based on the given spectral components.

The function `demobp1` contains a Slide show demonstration of the sample training session. The function `nnsample1` contains all of the commands that we used in this section. You can use it as a template for your own training sessions.

# Limitations and Cautions

The gradient descent algorithm is generally very slow because it requires small learning rates for stable learning. The momentum variation is usually faster than simple gradient descent, since it allows higher learning rates while maintaining stability, but it is still too slow for many practical applications. These two methods would normally be used only when incremental training is desired. You would normally use Levenberg-Marquardt training for small and medium size networks, if you have enough memory available. If memory is a problem, then there are a variety of other fast algorithms available. For large networks you will probably want to use `trainscg` or `trainrp`.

Multi-layered networks are capable of performing just about any linear or nonlinear computation, and can approximate any reasonable function arbitrarily well. Such networks overcome the problems associated with the perceptron and linear networks. However, while the network being trained may be theoretically capable of performing correctly, backpropagation and its variations may not always find a solution. See page 12-8 of [HDB96] for a discussion of convergence to local minimum points.

Picking the learning rate for a nonlinear network is a challenge. As with linear networks, a learning rate that is too large leads to unstable learning. Conversely, a learning rate that is too small results in incredibly long training times. Unlike linear networks, there is no easy way of picking a good learning rate for nonlinear multilayer networks. See page 12-8 of [HDB96] for examples of choosing the learning rate. With the faster training algorithms, the default parameter values normally perform adequately.

The error surface of a nonlinear network is more complex than the error surface of a linear network. To understand this complexity see the figures on pages 12-5 to 12-7 of [HDB96], which show three different error surfaces for a multilayer network. The problem is that nonlinear transfer functions in multilayer networks introduce many local minima in the error surface. As gradient descent is performed on the error surface it is possible for the network solution to become trapped in one of these local minima. This may happen depending on the initial starting conditions. Settling in a local minimum may be good or bad depending on how close the local minimum is to the global minimum and how low an error is required. In any case, be cautioned that although a multilayer backpropagation network with enough neurons can implement just about any function, backpropagation will not always find the correct weights for the optimum solution. You may want to reinitialize the network and retrain several times to guarantee that you have the best solution.

Networks are also sensitive to the number of neurons in their hidden layers. Too few neurons can lead to underfitting. Too many neurons can contribute to overfitting, in which all training points are well fit, but the fitting curve takes wild oscillations between these points. Ways of dealing with various of these issues are discussed in the section on improving generalization. This topic is also discussed starting on page 11-21 of [HDB96].

# Summary

Backpropagation can train multilayer feed-forward networks with differentiable transfer functions to perform function approximation, pattern association, and pattern classification. (Other types of networks can be trained as well, although the multilayer network is most commonly used.) The term backpropagation refers to the process by which derivatives of network error, with respect to network weights and biases, can be computed. This process can be used with a number of different optimization strategies.

The architecture of a multilayer network is not completely constrained by the problem to be solved. The number of inputs to the network is constrained by the problem, and the number of neurons in the output layer is constrained by the number of outputs required by the problem. However, the number of layers between network inputs and the output layer and the sizes of the layers are up to the designer.

The two-layer sigmoid/linear network can represent any functional relationship between inputs and outputs if the sigmoid layer has enough neurons.

There are several different backpropagation training algorithms. They have a variety of different computation and storage requirements, and no one algorithm is best suited to all locations. The following list summarizes the training algorithms included in the toolbox.

| Function | Description |
|---|---|
| `traingd` | Basic gradient descent. Slow response, can be used in incremental mode training. |
| `traingdm` | Gradient descent with momentum. Generally faster than `traingd`. Can be used in incremental mode training. |
| `traingdx` | Adaptive learning rate. Faster training than `traingd`, but can only be used in batch mode training. |
| `trainrp` | Resilient backpropagation. Simple batch mode training algorithm with fast convergence and minimal storage requirements. |

| Function | Description |
|---|---|
| `traincgf` | Fletcher-Reeves conjugate gradient algorithm. Has smallest storage requirements of the conjugate gradient algorithms. |
| `traincgp` | Polak-Ribiére conjugate gradient algorithm. Slightly larger storage requirements than `traincgf`. Faster convergence on some problems. |
| `traincgb` | Powell-Beale conjugate gradient algorithm. Slightly larger storage requirements than `traincgp`. Generally faster convergence. |
| `trainscg` | Scaled conjugate gradient algorithm. The only conjugate gradient algorithm that requires no line search. A very good general purpose training algorithm. |
| `trainbfg` | BFGS quasi-Newton method. Requires storage of approximate Hessian matrix and has more computation in each iteration than conjugate gradient algorithms, but usually converges in fewer iterations. |
| `trainoss` | One step secant method. Compromise between conjugate gradient methods and quasi-Newton methods. |
| `trainlm` | Levenberg-Marquardt algorithm. Fastest training algorithm for networks of moderate size. Has memory reduction feature for use when the training set is large. |
| `trainbr` | Bayesian regularization. Modification of the Levenberg-Marquardt training algorithm to produce networks that generalize well. Reduces the difficulty of determining the optimum network architecture. |

One problem that can occur when training neural networks is that the network can overfit on the training set and not generalize well to new data outside the training set. This can be prevented by training with `trainbr`, but it can also be prevented by using *early stopping* with any of the other training routines. This requires that the user pass a validation set to the training algorithm, in addition to the standard training set.

To produce the most efficient training, it is often helpful to preprocess the data before training. It is also helpful to analyze the network response after training is complete. The toolbox contains a number of routines for pre- and post-processing. They are summarized in the following table.

| Function | Description |
|---|---|
| `premnmx` | Normalize data to fall in the range [-1,1]. |
| `postmnmx` | Inverse of `premnmx`. Used to convert data back to standard units. |
| `tramnmx` | Normalize data using previously computed minimums and maximums. Used to preprocess new inputs to networks that have been trained with data normalized with `premnmx`. |
| `prestd` | Normalize data to have zero mean and unity standard deviation. |
| `poststd` | Inverse of `prestd`. Used to convert data back to standard units. |
| `trastd` | Normalize data using previously computed means and standard deviations. Used to preprocess new inputs to networks that have been trained with data normalized with `prestd`. |
| `prepca` | Principal component analysis. Reduces dimension of input vector and un-correlates components of input vectors. |
| `trapca` | Preprocess data using previously computed principal component transformation matrix. Used to preprocess new inputs to networks that have been trained with data transformed with `prepca`. |
| `postreg` | Linear regression between network outputs and targets. Used to determine the adequacy of network fit. |

6

# Control Systems

# Introduction

Neural networks have been applied very successfully in the identification and control of dynamic systems. The universal approximation capabilities of the multilayer perceptron make it a popular choice for modeling nonlinear systems and for implementing general-purpose nonlinear controllers [HaDe99]. This chapter introduces three popular neural network architectures for prediction and control that have been implemented in the Neural Network Toolbox:

- Model Predictive Control
- NARMA-L2 (or Feedback Linearization) Control
- Model Reference Control

This chapter presents brief descriptions of each of these architectures and demonstrates how you can use them.

There are typically two steps involved when using neural networks for control:

1. System Identification

2. Control Design

In the system identification stage, you develop a neural network model of the plant that you want to control. In the control design stage, you use the neural network plant model to design (or train) the controller. In each of the three control architectures described in this chapter, the system identification stage is identical. The control design stage, however, is different for each architecture.

- For the model predictive control, the plant model is used to predict future behavior of the plant, and an optimization algorithm is used to select the control input that optimizes future performance.
- For the NARMA-L2 control, the controller is simply a rearrangement of the plant model.
- For the model reference control, the controller is a neural network that is trained to control a plant so that it follows a reference model. The neural network plant model is used to assist in the controller training.

The next three sections of this chapter discuss model predictive control, NARMA-L2 control and model reference control. Each section consists of a brief description of the control concept, followed by a demonstration of the use of the appropriate Neural Network Toolbox function. These three controllers are implemented as Simulink® blocks, which are contained in the Neural Network Toolbox blockset.

To assist you in determining the best controller for your application, the following list summarizes the key controller features. Each controller has its own strengths and weaknesses. No single controller is appropriate for every application.

- **Model Predictive Control**. This controller uses a neural network model to predict future plant responses to potential control signals. An optimization algorithm then computes the control signals that optimize future plant performance. The neural network plant model is trained offline, in batch form, using any of the training algorithms discussed in Chapter 5. (This is true for all three control architectures.) The controller, however, requires a significant amount of on-line computation, since an optimization algorithm is performed at each sample time to compute the optimal control input.
- **NARMA-L2 Control**. This controller requires the least computation of the three architectures described in this chapter. The controller is simply a rearrangement of the neural network plant model, which is trained offline, in batch form. The only online computation is a forward pass through the neural network controller. The drawback of this method is that the plant must either be in companion form, or be capable of approximation by a companion form model. (The companion form model is described later in this chapter.)
- **Model Reference Control**. The online computation of this controller, like NARMA-L2, is minimal. However, unlike NARMA-L2, the model reference architecture requires that a separate neural network controller be trained off-line, in addition to the neural network plant model. The controller training is computationally expensive, since it requires the use of dynamic backpropagation [HaJe99]. On the positive side, model reference control applies to a larger class of plant than does NARMA-L2 control.

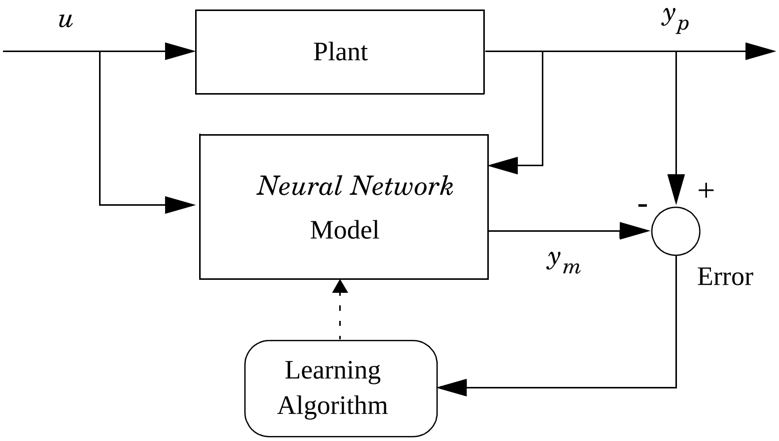
# NN Predictive Control

The neural network predictive controller that is implemented in the Neural Network Toolbox uses a neural network model of a nonlinear plant to predict future plant performance. The controller then calculates the control input that will optimize plant performance over a specified future time horizon. The first step in model predictive control is to determine the neural network plant model (system identification). Next, the plant model is used by the controller to predict future performance. (See the Model Predictive Control Toolbox documentation for a complete coverage of the application of various model predictive control strategies to linear systems.)

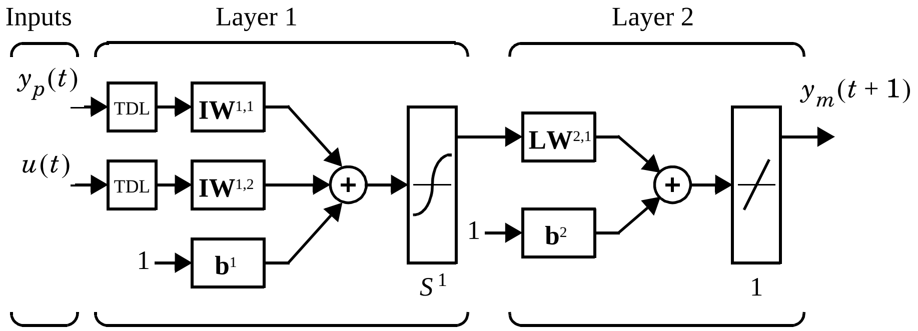
The following section describes the system identification process. This is followed by a description of the optimization process. Finally, it discusses how to use the model predictive controller block that has been implemented in Simulink®.

## System Identification

The first stage of model predictive control is to train a neural network to represent the forward dynamics of the plant. The prediction error between the plant output and the neural network output is used as the neural network training signal. The process is represented by the following figure.



The neural network plant model uses previous inputs and previous plant outputs to predict future values of the plant output. The structure of the neural network plant model is given in the following figure.



This network can be trained offline in batch mode, using data collected from the operation of the plant. Any of the training algorithms discussed in Chapter 5, “Backpropagation”, can be used for network training. This process is discussed in more detail later in this chapter.

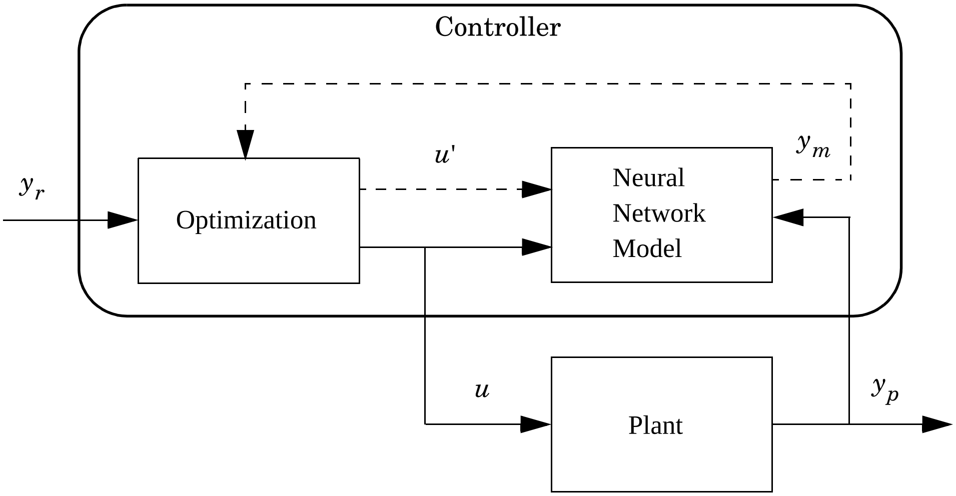
## Predictive Control

The model predictive control method is based on the receding horizon technique [SoHa96]. The neural network model predicts the plant response over a specified time horizon. The predictions are used by a numerical optimization program to determine the control signal that minimizes the following performance criterion over the specified horizon.

$$J = \sum_{j=N_1}^{N_2} (y_r(t+j) - y_m(t+j))^2 + \rho \sum_{j=1}^{N_u} (u'(t+j-1) - u'(t+j-2))^2$$

where $N_1$, $N_2$ and $N_u$ define the horizons over which the tracking error and the control increments are evaluated. The $u'$ variable is the tentative control signal, $y_r$ is the desired response and $y_m$ is the network model response. The $\rho$ value determines the contribution that the sum of the squares of the control increments has on the performance index.

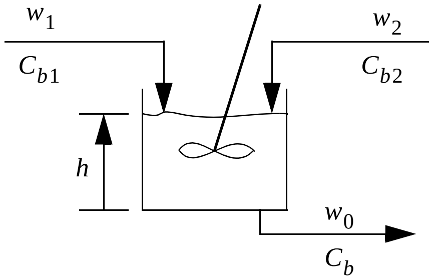
The following block diagram illustrates the model predictive control process. The controller consists of the neural network plant model and the optimization block. The optimization block determines the values of $u'$ that minimize $J$, and then the optimal $u$ is input to the plant. The controller block has been implemented in Simulink, as described in the following section.



## Using the NN Predictive Controller Block

This section demonstrates how the NN Predictive Controller block is used.The first step is to copy the NN Predictive Controller block from the Neural Network Toolbox blockset to your model window. See your Simulink documentation if you are not sure how to do this. This step is skipped in the following demonstration.

A demo model is provided with the Neural Network Toolbox to demonstrate the predictive controller. This demo uses a catalytic Continuous Stirred Tank Reactor (CSTR). A diagram of the process is shown in the following figure.



The dynamic model of the system is

$$\frac{dh(t)}{dt} = w_1(t) + w_2(t) - 0.2\sqrt{h(t)}$$

$$\frac{dC_b(t)}{dt} = (C_{b1} - C_b(t))\frac{w_1(t)}{h(t)} + (C_{b2} - C_b(t))\frac{w_2(t)}{h(t)} - \frac{k_1 C_b(t)}{(1 + k_2 C_b(t))^2}$$

where $h(t)$ is the liquid level, $C_b(t)$ is the product concentration at the output of the process, $w_1(t)$ is the flow rate of the concentrated feed $C_{b1}$, and $w_2(t)$ is the flow rate of the diluted feed $C_{b2}$. The input concentrations are set to $C_{b1} = 24.9$ and $C_{b2} = 0.1$. The constants associated with the rate of consumption are $k_1 = 1$ and $k_2 = 1$.

The objective of the controller is to maintain the product concentration by adjusting the flow $w_2(t)$. To simplify the demonstration, we set $w_1(t) = 0.1$. The level of the tank $h(t)$ is not controlled for this experiment.

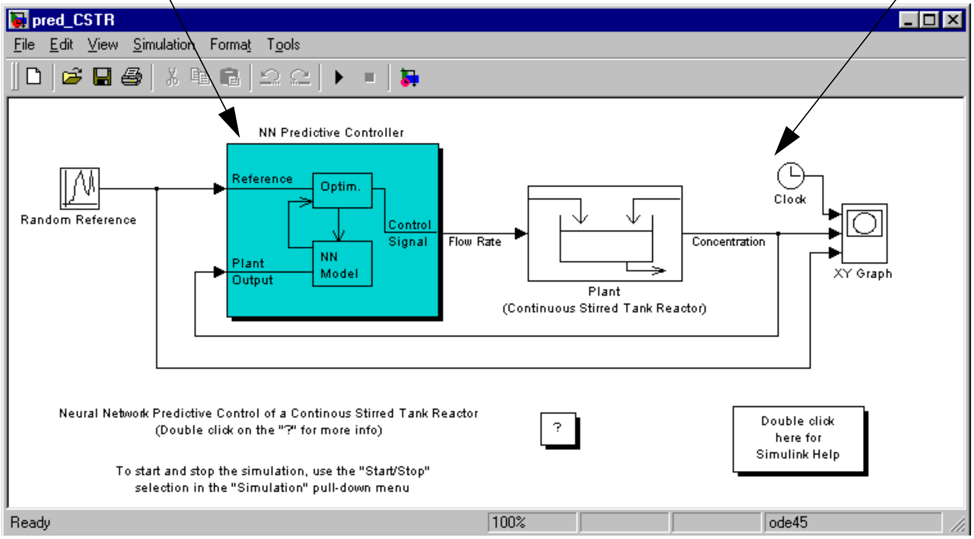
To run this demo, follow these steps.

1. Start MATLAB®.

2. Run the demo model by typing `predcstr` in the MATLAB command window. This command starts Simulink and creates the following model window. The NN Predictive Controller block has already been placed in the model.

This NN Predictive Controller block was copied from the Neural Network Toolbox blockset to this model window. The **Control Signal** was connected to the input of the plant model. The output of the plant model was connected to **Plant Output**. The reference signal was connected to **Reference**.

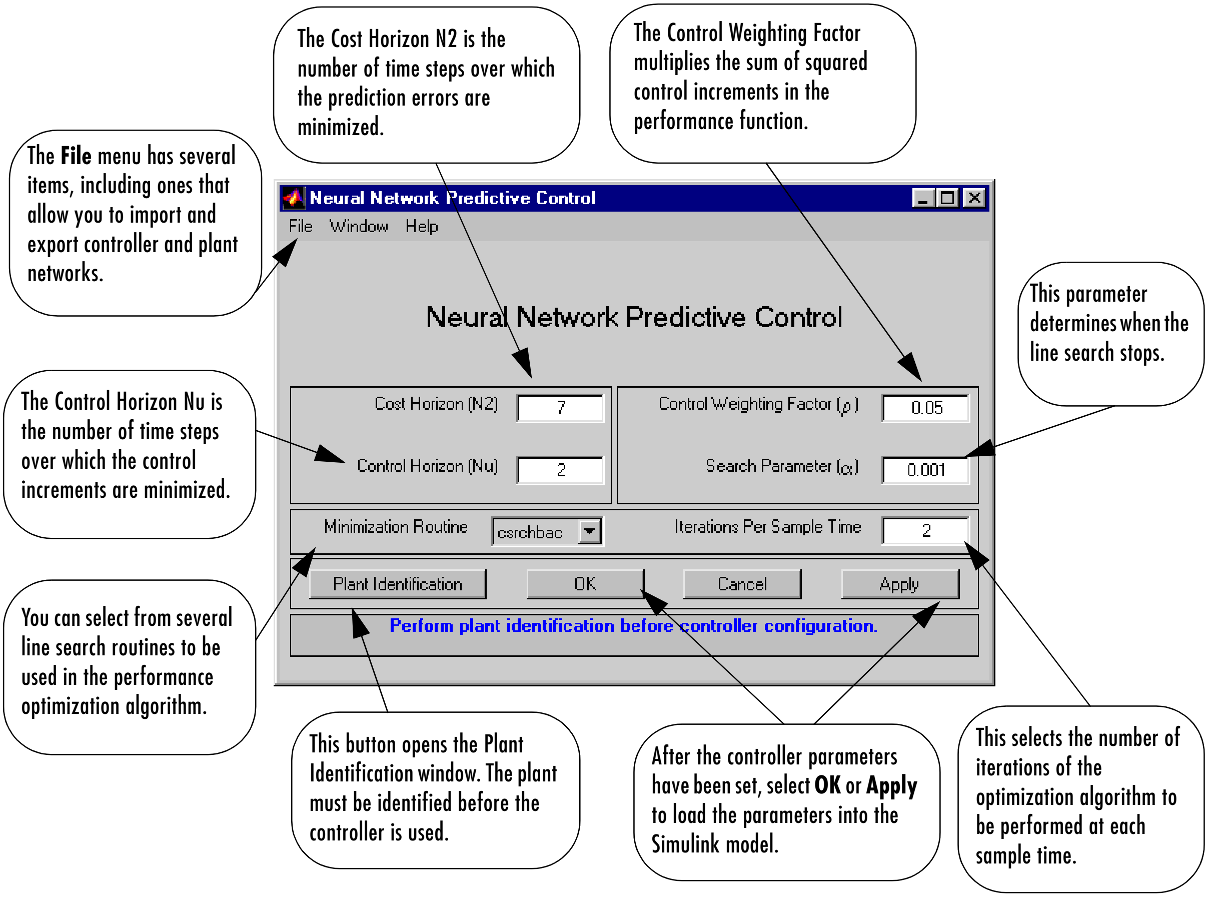
This block contains the Simulink CSTR plant model.



Neural Network Predictive Control of a Continous Stirred Tank Reactor
(Double click on the "?" for more info)

To start and stop the simulation, use the "Start/Stop" selection in the "Simulation" pull-down menu

**3** Double-click the NN Predictive Controller block. This brings up the following window for designing the model predictive controller. This window enables you to change the controller horizons $N_2$ and $N_u$. ($N_1$ is fixed at 1.) The weighting parameter $\rho$, described earlier, is also defined in this window. The parameter $\alpha$ is used to control the optimization. It determines how much reduction in performance is required for a successful optimization step. You can select which linear minimization routine is used by the optimization algorithm, and you can decide how many iterations of the optimization algorithm are performed at each sample time. The linear minimization routines are slight modifications of those discussed in Chapter 5, “Backpropagation.”



The **File** menu has several items, including ones that allow you to import and export controller and plant networks.

The Cost Horizon N2 is the number of time steps over which the prediction errors are minimized.

The Control Weighting Factor multiplies the sum of squared control increments in the performance function.

This parameter determines when the line search stops.

The Control Horizon Nu is the number of time steps over which the control increments are minimized.

You can select from several line search routines to be used in the performance optimization algorithm.

This button opens the Plant Identification window. The plant must be identified before the controller is used.

After the controller parameters have been set, select **OK** or **Apply** to load the parameters into the Simulink model.

This selects the number of iterations of the optimization algorithm to be performed at each sample time.

Neural Network Predictive Control

Cost Horizon (N2): 7

Control Weighting Factor ($\rho$): 0.05

Control Horizon (Nu): 2

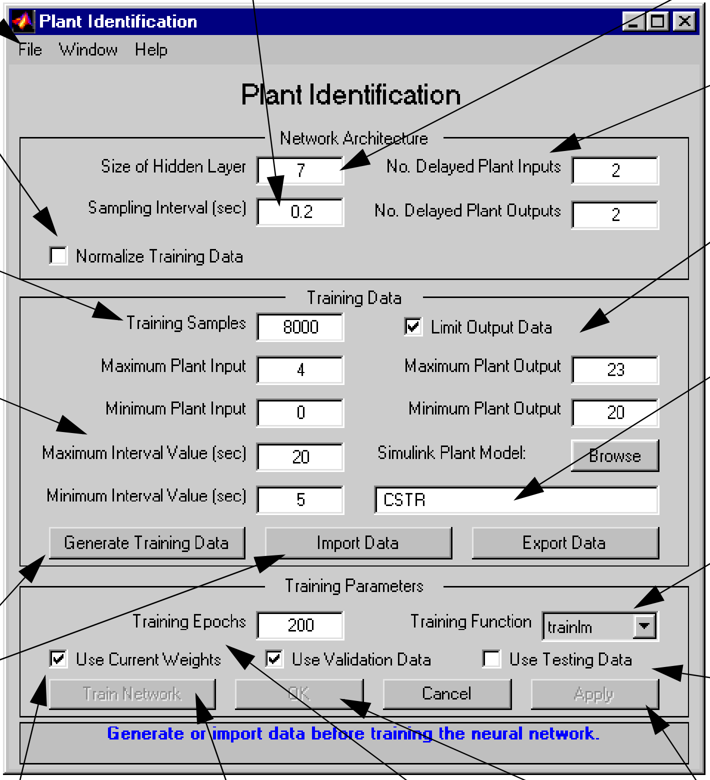
Search Parameter ($\alpha$): 0.001

Minimization Routine: csrchbac

Iterations Per Sample Time: 2

Perform plant identification before controller configuration.

**4** Select **Plant Identification**. This opens the following window. The neural network plant model must be developed before the controller is used. The plant model predicts future plant outputs. The optimization algorithm uses these predictions to determine the control inputs that optimize future performance. The plant model neural network has one hidden layer, as shown earlier. The size of that layer, the number of delayed inputs and delayed outputs, and the training function are selected in this window. You can select any of the training functions described in Chapter 5, “Backpropagation”, to train the neural network plant model.



Plant Identification

Network Architecture

| | | | |
|---|---|---|---|
| Size of Hidden Layer | 7 | No. Delayed Plant Inputs | 2 |
| Sampling Interval (sec) | 0.2 | No. Delayed Plant Outputs | 2 |

□ Normalize Training Data

Training Data

| | | | |
|---|---|---|---|
| Training Samples | 8000 | ☑ Limit Output Data | |
| Maximum Plant Input | 4 | Maximum Plant Output | 23 |
| Minimum Plant Input | 0 | Minimum Plant Output | 20 |
| Maximum Interval Value (sec) | 20 | Simulink Plant Model: | Browse |
| Minimum Interval Value (sec) | 5 | CSTR | |

Training Parameters

| | | | |
|---|---|---|---|
| Training Epochs | 200 | Training Function | trainlm |

☑ Use Current Weights ☑ Use Validation Data □ Use Testing Data

Generate or import data before training the neural network.

The **File** menu has several items, including ones that allow you to import and export plant model networks.

Interval at which the program collects data from the Simulink plant model.

The number of neurons in the first layer of the plant model network.

You can normalize the data using the premnmx function.

You can define the size of the two tapped delay lines coming into the plant model.

Number of data points generated for training, validation, and test sets.

You can select a range on the output data to be used in training.

The random plant input is a series of steps of random height occurring at random intervals. These fields set the minimum and maximum height and interval.

Simulink plant model used to generate training data (file with .mdl extension).

You can use any training function to train the plant model.

This button starts the training data generation.

You can use validation (early stopping) and testing data during training.

You can use existing data to train the network. If you select this, a field will appear for the filename.

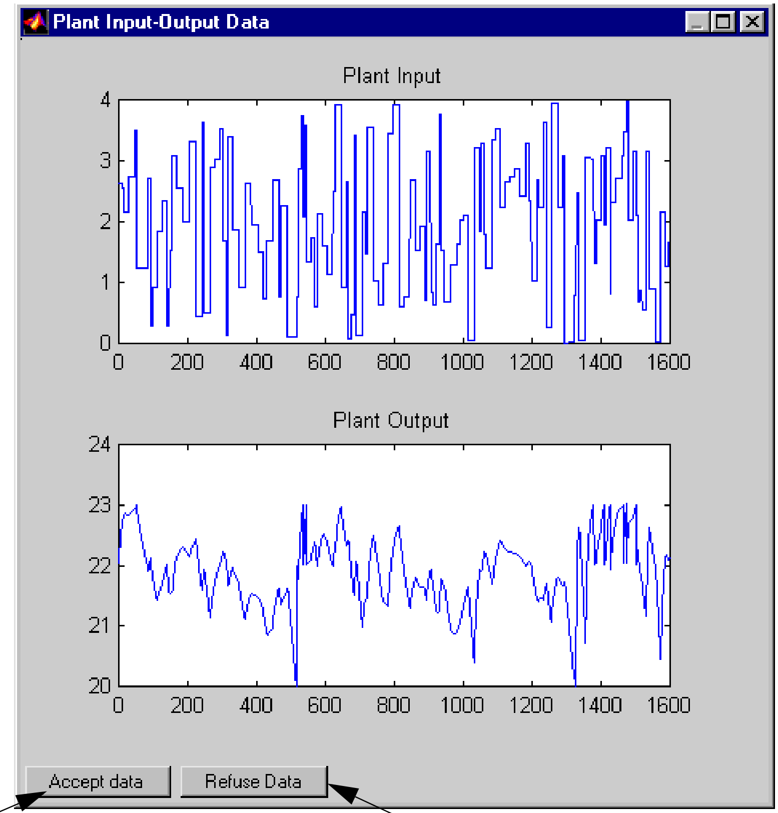
Select this option to continue training with current weights. Otherwise, you use randomly generated weights.

This button begins the plant model training. Generate or import data before training.

Number of iterations of plant training to be performed.

After the plant model has been trained, select **OK** or **Apply** to load the network into the Simulink model.

5 Select the **Generate Training Data** button. The program generates training data by applying a series of random step inputs to the Simulink plant model. The potential training data is then displayed in a figure similar to the following.

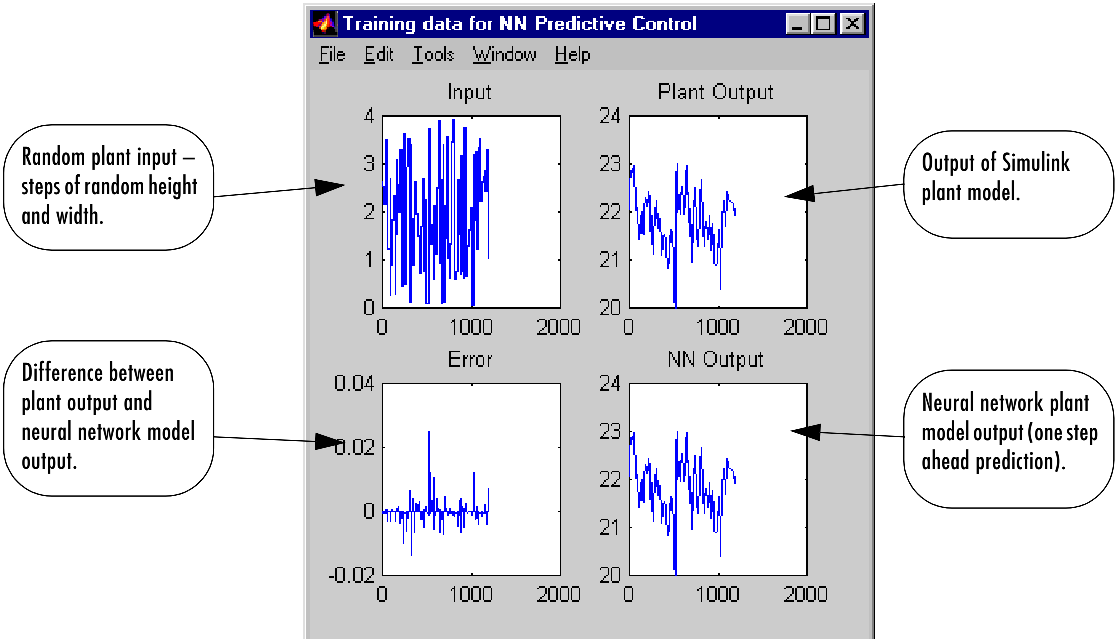


Accept the data if it is sufficiently representative of future plant activity. Then plant training begins.

If you refuse the training data, you return to the Plant Identification window and restart the training.

6 Select **Accept Data**, and then select **Train Network** from the Plant Identification window. Plant model training begins. The training proceeds

according to the selected training algorithm (`trainlm` in this case). This is a straightforward application of batch training, as described in Chapter 5, “Backpropagation.” After the training is complete, the response of the resulting plant model is displayed, as in the following figure. (There are also separate plots for validation and testing data, if they exist.) You can then continue training with the same data set by selecting **Train Network** again, you can **Erase Generated Data** and generate a new data set, or you can accept the current plant model and begin simulating the closed loop system. For this demonstration, begin the simulation, as shown in the following steps.



Random plant input – steps of random height and width.

Output of Simulink plant model.

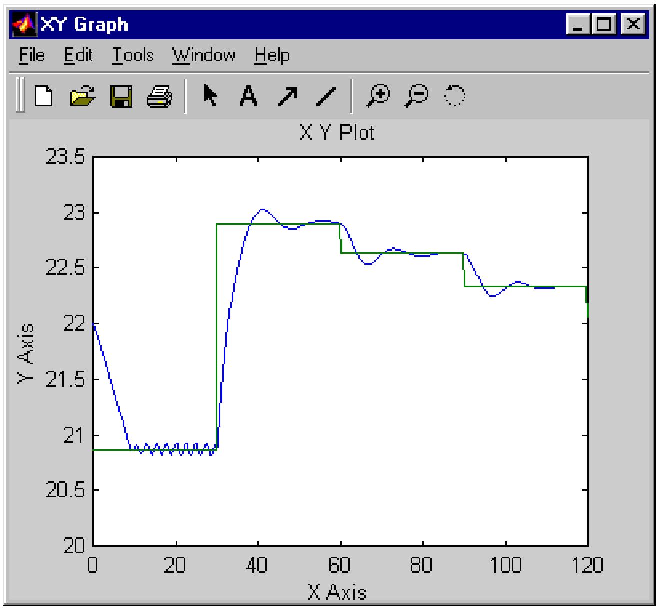
Difference between plant output and neural network model output.

Neural network plant model output (one step ahead prediction).

**7** Select **OK** in the **Plant Identification** window. This loads the trained neural network plant model into the NN Predictive Controller block.

**8** Select **OK** in the **Neural Network Predictive Control** window. This loads the controller parameters into the NN Predictive Controller block.

**9** Return to the Simulink model and start the simulation by choosing the **Start** command from the **Simulation** menu. As the simulation runs, the plant output and the reference signal are displayed, as in the following figure.



# NARMA-L2 (Feedback Linearization) Control

The neurocontroller described in this section is referred to by two different names: feedback linearization control and NARMA-L2 control. It is referred to as feedback linearization when the plant model has a particular form (companion form). It is referred to as NARMA-L2 control when the plant model can be approximated by the same form. The central idea of this type of control is to transform nonlinear system dynamics into linear dynamics by canceling the nonlinearities. This section begins by presenting the companion form system model and demonstrating how you can use a neural network to identify this model. Then it describes how the identified neural network model can be used to develop a controller. This is followed by a demonstration of how to use the NARMA-L2 Control block, which is contained in the Neural Network Toolbox blockset.

## Identification of the NARMA-L2 Model

As with the model predictive control, the first step in using feedback linearization (or NARMA-L2 control) is to identify the system to be controlled. You train a neural network to represent the forward dynamics of the system. The first step is to choose a model structure to use. One standard model that has been used to represent general discrete-time nonlinear systems is the Nonlinear Autoregressive-Moving Average (NARMA) model:

$$y(k+d) = N[y(k), y(k-1), \ldots, y(k-n+1), u(k), u(k-1), \ldots, u(k-n+1)]$$

where $u(k)$ is the system input, and $y(k)$ is the system output. For the identification phase, you could train a neural network to approximate the nonlinear function $N$. This is the identification procedure used for the NN Predictive Controller.

If you want the system output to follow some reference trajectory, $y(k+d) = y_r(k+d)$, the next step is to develop a nonlinear controller of the form:

$$u(k) = G[y(k), y(k-1), \ldots, y(k-n+1), y_r(k+d), u(k-1), \ldots, u(k-m+1)]$$

The problem with using this controller is that if you want to train a neural network to create the function $G$ that will minimize mean square error, you need to use dynamic backpropagation ([NaPa91] or [HaJe99]). This can be quite slow. One solution proposed by Narendra and Mukhopadhyay [NaMu97] is to use approximate models to represent the system. The controller used in this section is based on the NARMA-L2 approximate model:

$$\begin{aligned}\hat{y}(k+d) &= f[y(k), y(k-1), \ldots, y(k-n+1), u(k-1), \ldots, u(k-m+1)] \\ &\quad + g[y(k), y(k-1), \ldots, y(k-n+1), u(k-1), \ldots, u(k-m+1)] \cdot u(k)\end{aligned}$$

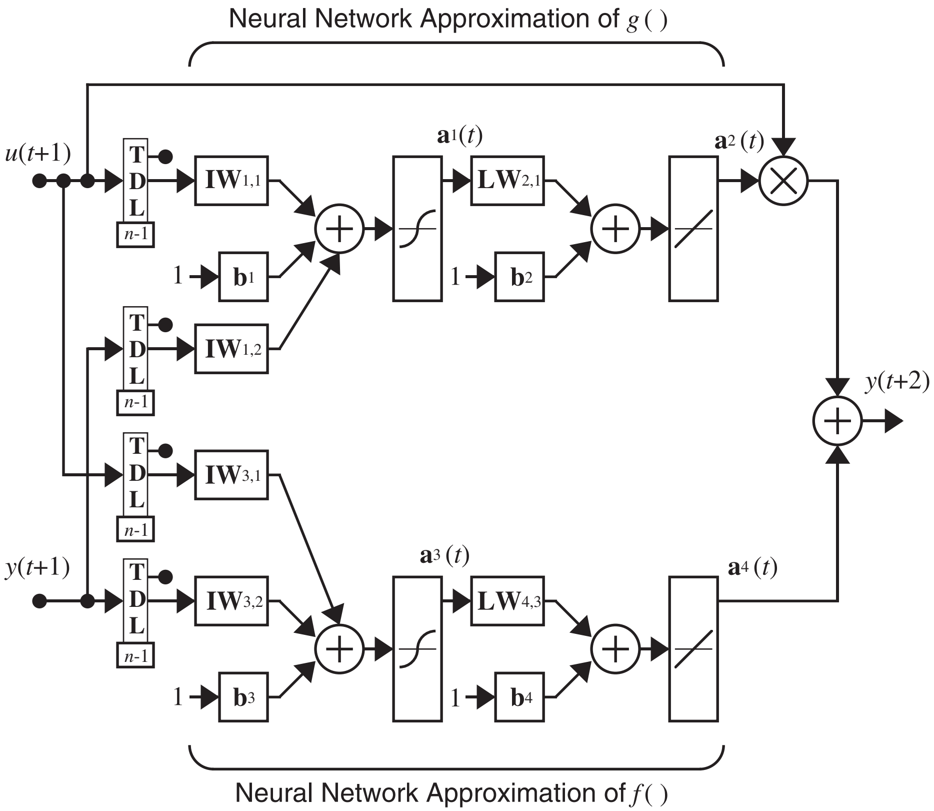
This model is in companion form, where the next controller input $u(k)$ is not contained inside the nonlinearity. The advantage of this form is that you can solve for the control input that causes the system output to follow the reference $y(k+d) = y_r(k+d)$. The resulting controller would have the form

$$u(k) = \frac{y_r(k+d) - f[y(k), y(k-1), \ldots, y(k-n+1), u(k-1), \ldots, u(k-n+1)]}{g[y(k), y(k-1), \ldots, y(k-n+1), u(k-1), \ldots, u(k-n+1)]}$$

Using this equation directly can cause realization problems, because you must determine the control input $u(k)$ based on the output at the same time, $y(k)$. So, instead, use the model

$$\begin{aligned}y(k+d) &= f[y(k), y(k-1), \ldots, y(k-n+1), u(k), u(k-1), \ldots, u(k-n+1)] \\ &\quad + g[y(k), \ldots, y(k-n+1), u(k), \ldots, u(k-n+1)] \cdot u(k+1)\end{aligned}$$

where $d \geq 2$. The following figure shows the structure of a neural network representation.

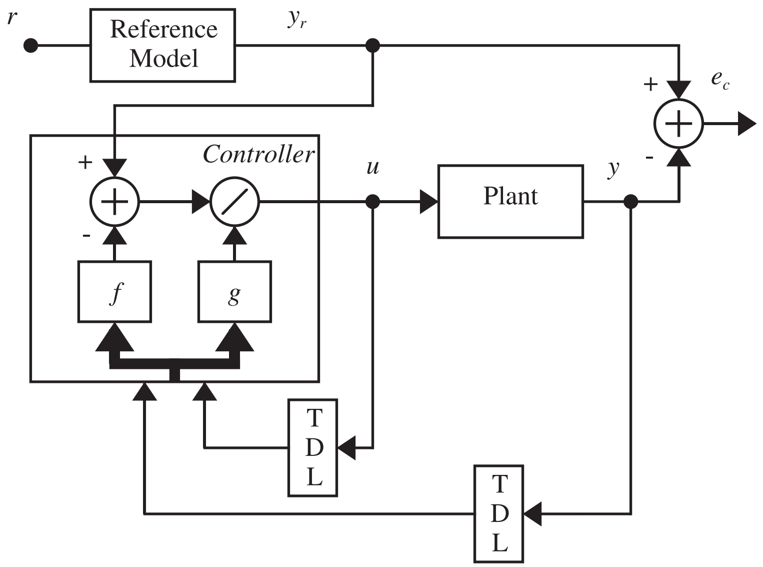


## NARMA-L2 Controller

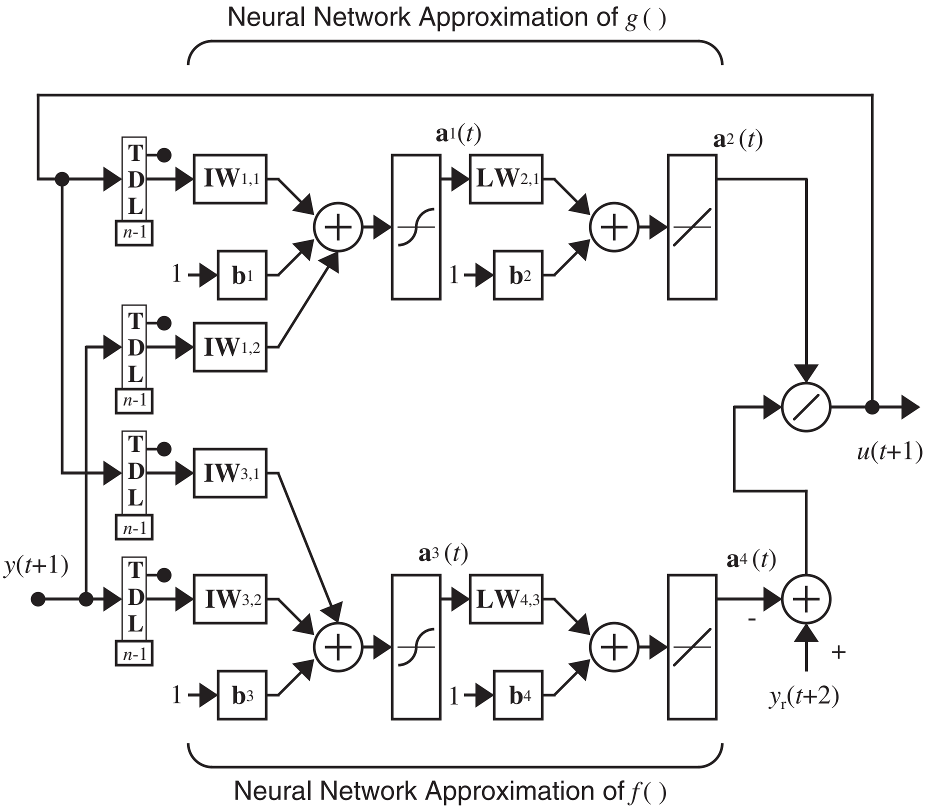
Using the NARMA-L2 model, you can obtain the controller

$$u(k+1) = \frac{y_r(k+d) - f[y(k), \ldots, y(k-n+1), u(k), \ldots, u(k-n+1)]}{g[y(k), \ldots, y(k-n+1), u(k), \ldots, u(k-n+1)]}$$

which is realizable for $d \geq 2$. The following figure is a block diagram of the NARMA-L2 controller.



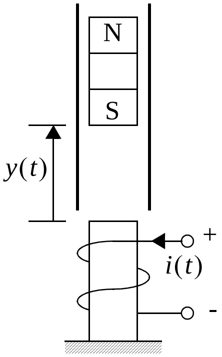
This controller can be implemented with the previously identified NARMA-L2 plant model, as shown in the following figure.



## Using the NARMA-L2 Controller Block

This section demonstrates how the NARMA-L2 controller is trained.The first step is to copy the NARMA-L2 Controller block from the Neural Network Toolbox blockset to your model window. See your Simulink documentation if you are not sure how to do this. This step is skipped in the following demonstration.

A demo model is provided with the Neural Network Toolbox to demonstrate the NARMA-L2 controller. In this demo, the objective is to control the position of a magnet suspended above an electromagnet, where the magnet is constrained so that it can only move in the vertical direction, as in the following figure.



The equation of motion for this system is

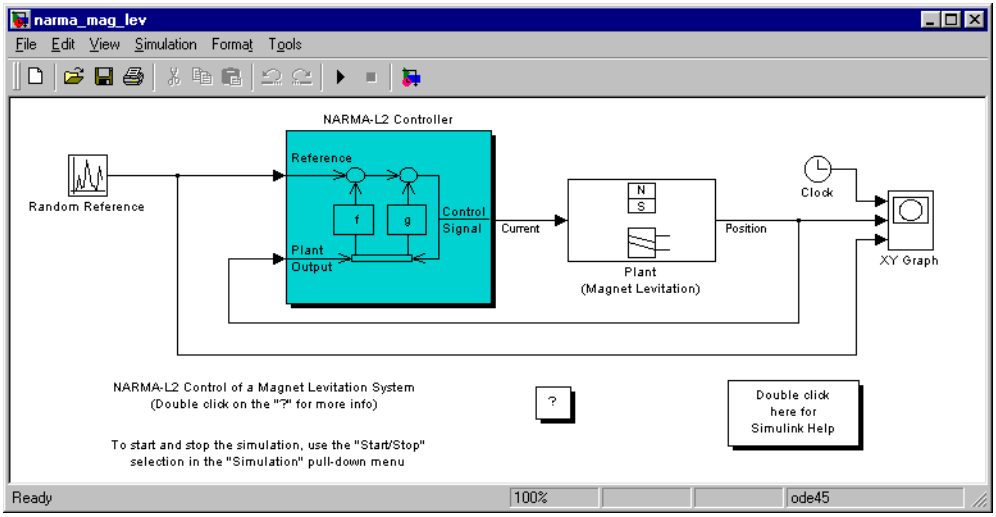
$$\frac{d^2y(t)}{dt^2} = -g + \frac{\alpha}{M}\frac{i^2(t)}{y(t)} - \frac{\beta}{M}\frac{dy(t)}{dt}$$

where $y(t)$ is the distance of the magnet above the electromagnet, $i(t)$ is the current flowing in the electromagnet, $M$ is the mass of the magnet, and $g$ is the gravitational constant. The parameter $\beta$ is a viscous friction coefficient that is determined by the material in which the magnet moves, and $\alpha$ is a field strength constant that is determined by the number of turns of wire on the electromagnet and the strength of the magnet.

To run this demo, follow these steps.

1 Start MATLAB.

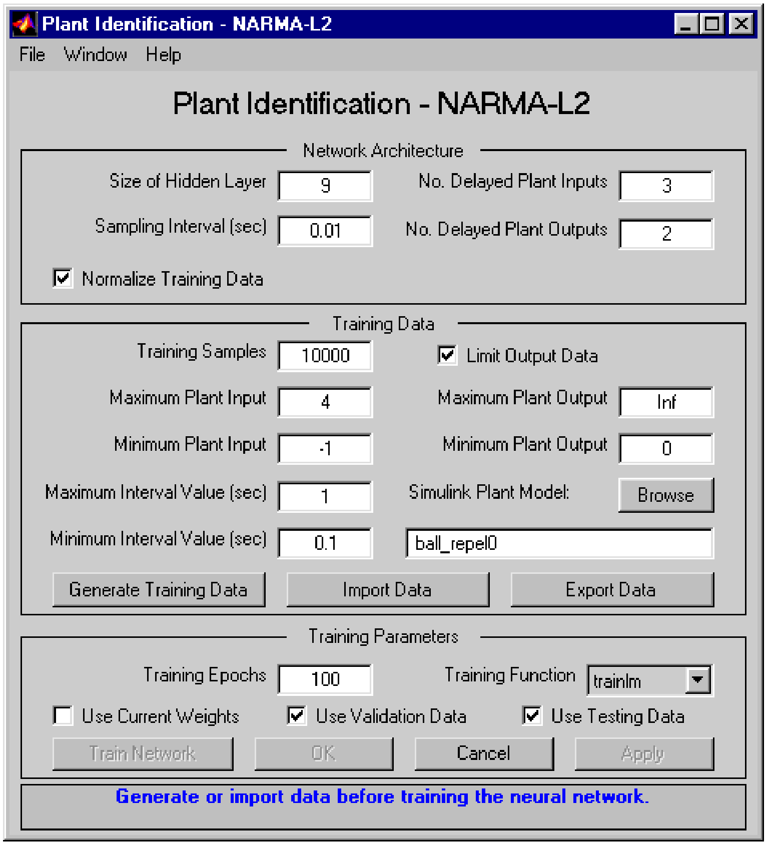
2 Run the demo model by typing `narmamaglev` in the MATLAB command window. This command starts Simulink and creates the following model window. The NARMA-L2 Control block has already been placed in the model.



NARMA-L2 Control of a Magnet Levitation System
(Double click on the "?" for more info)

To start and stop the simulation, use the "Start/Stop" selection in the "Simulation" pull-down menu

3 Double-click the NARMA-L2 Controller block. This brings up the following window. Notice that this window enables you to train the NARMA-L2 model. There is no separate window for the controller, since the controller is determined directly from the model, unlike the model predictive controller.



Plant Identification - NARMA-L2

Network Architecture

| | | | |
|---|---|---|---|
| Size of Hidden Layer | 9 | No. Delayed Plant Inputs | 3 |
| Sampling Interval (sec) | 0.01 | No. Delayed Plant Outputs | 2 |

☑ Normalize Training Data

Training Data

| | | | |
|---|---|---|---|
| Training Samples | 10000 | ☑ Limit Output Data | |
| Maximum Plant Input | 4 | Maximum Plant Output | Inf |
| Minimum Plant Input | -1 | Minimum Plant Output | 0 |
| Maximum Interval Value (sec) | 1 | Simulink Plant Model: | |
| Minimum Interval Value (sec) | 0.1 | ball_repel0 | |

Training Parameters

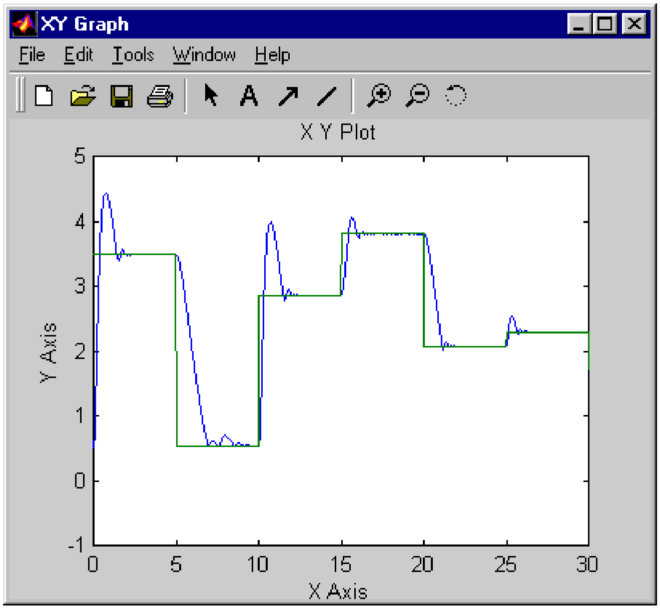
| | | | |
|---|---|---|---|
| Training Epochs | 100 | Training Function | trainlm |

□ Use Current Weights ☑ Use Validation Data ☑ Use Testing Data

Generate or import data before training the neural network.

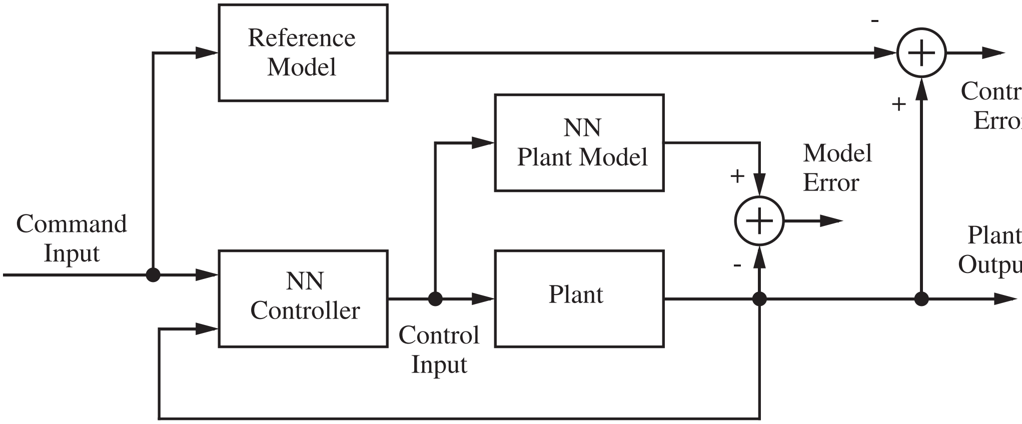
**4** Since this window works the same as the other **Plant Identification** windows, we won't go through the training process again now. Instead, let's simulate the NARMA-L2 controller.

**5** Return to the Simulink model and start the simulation by choosing the **Start** command from the **Simulation** menu. As the simulation runs, the plant output and the reference signal are displayed, as in the following figure.


XY Graph
File Edit Tools Window Help
X Y Plot
Y Axis
X Axis
5
4
3
2
1
0
-1
0
5
10
15
20
25
30

# Model Reference Control

The neural model reference control architecture uses two neural networks: a controller network and a plant model network, as shown in the following figure. The plant model is identified first, and then the controller is trained so that the plant output follows the reference model output.



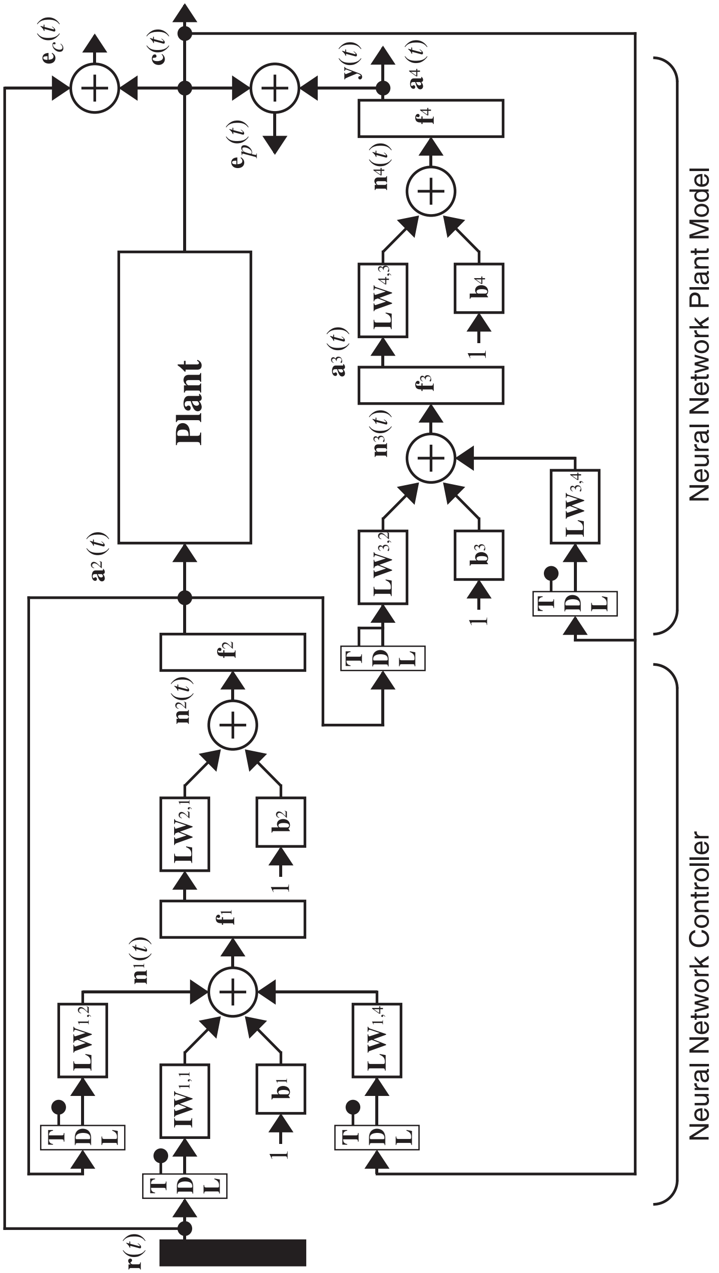
The figure on the following page shows the details of the neural network plant model and the neural network controller, as they are implemented in the Neural Network Toolbox. Each network has two layers, and you can select the number of neurons to use in the hidden layers. There are three sets of controller inputs:

- Delayed reference inputs
- Delayed controller outputs
- Delayed plant outputs

For each of these inputs, you can select the number of delayed values to use. Typically, the number of delays increases with the order of the plant. There are two sets of inputs to the neural network plant model:

- Delayed controller outputs
- Delayed plant outputs

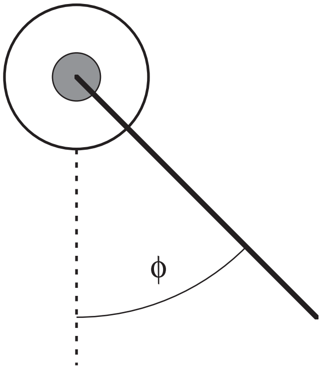
As with the controller, you can set the number of delays. The next section demonstrates how you can set the parameters.


Neural Network Plant Model
Neural Network Controller
Plant
r(t)
c(t)
e_c(t)
e_p(t)
y(t)
a4 (t)
f4
n4(t)
LW4,3
b4
1
a3 (t)
f3
n3(t)
LW3,2
b3
1
LW3,4
T D L
T D L
a2 (t)
f2
n2(t)
LW2,1
b2
1
f1
n1(t)
LW1,2
IW1,1
b1
1
LW1,4
T D L
T D L
T D L

## Using the Model Reference Controller Block

This section demonstrates how the neural network controller is trained. The first step is to copy the Model Reference Control block from the Neural Network Toolbox blockset to your model window. See your Simulink documentation if you are not sure how to do this. This step is skipped in the following demonstration.

A demo model is provided with the Neural Network Toolbox to demonstrate the model reference controller. In this demo, the objective is to control the movement of a simple, single-link robot arm, as shown in the following figure.



The equation of motion for the arm is

$$\frac{d^2\phi}{dt^2} = -10\sin\phi - 2\frac{d\phi}{dt} + u$$

where $\phi$ is the angle of the arm, and $u$ is the torque supplied by the DC motor.

The objective is to train the controller so that the arm tracks the reference model

$$\frac{d^2 y_r}{dt^2} = -9y_r - 6\frac{dy_r}{dt} + 9r$$

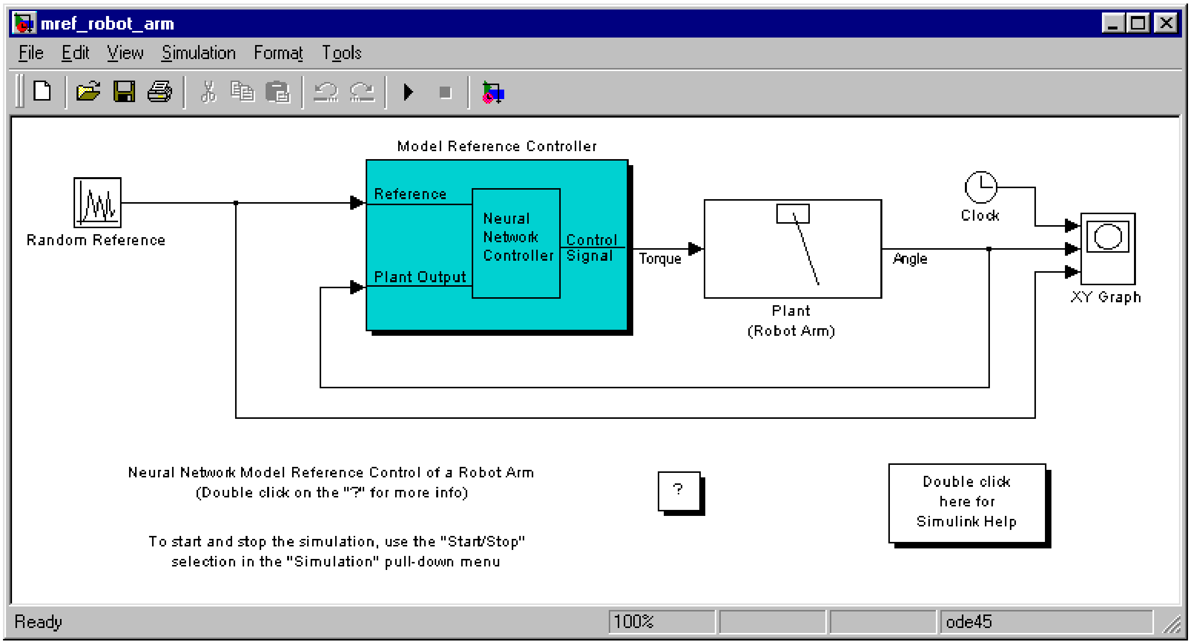
where $y_r$ is the output of the reference model, and $r$ is the input reference signal.

This demo uses a neural network controller with a 5-13-1 architecture. The inputs to the controller consist of two delayed reference inputs, two delayed plant outputs, and one delayed controller output. A sampling interval of 0.05 seconds is used.

To run this demo, follow these steps.

1 Start MATLAB.

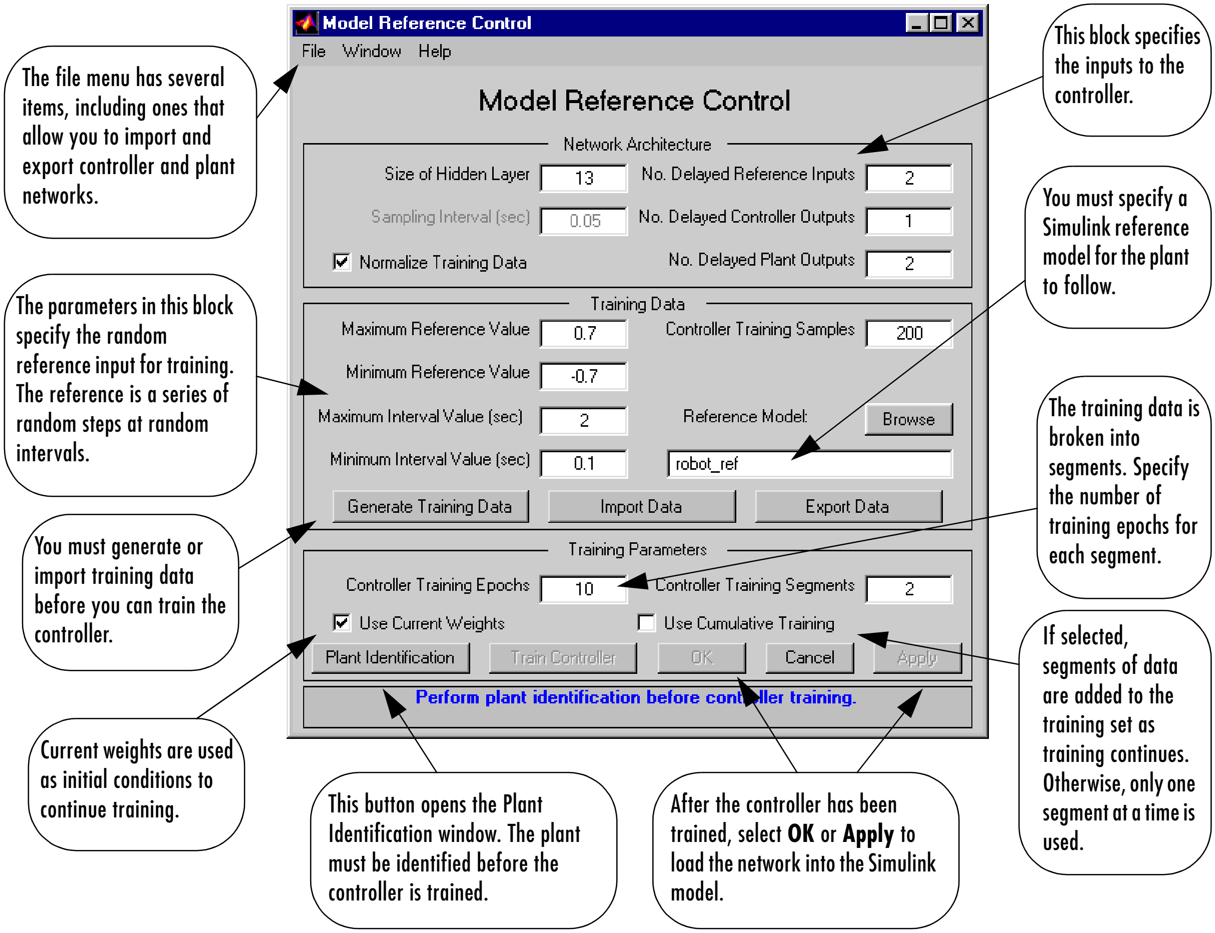
2 Run the demo model by typing `mrefrobotarm` in the MATLAB command window. This command starts Simulink and creates the following model window. The Model Reference Control block has already been placed in the model.



Neural Network Model Reference Control of a Robot Arm
(Double click on the "?" for more info)

To start and stop the simulation, use the "Start/Stop" selection in the "Simulation" pull-down menu

3 Double-click the Model Reference Control block. This brings up the following window for training the model reference controller.



Model Reference Control

Network Architecture

Size of Hidden Layer: 13
No. Delayed Reference Inputs: 2
Sampling Interval (sec): 0.05
No. Delayed Controller Outputs: 1
☑ Normalize Training Data
No. Delayed Plant Outputs: 2

Training Data

Maximum Reference Value: 0.7
Controller Training Samples: 200
Minimum Reference Value: -0.7
Maximum Interval Value (sec): 2
Reference Model: robot_ref
Minimum Interval Value (sec): 0.1

Training Parameters

Controller Training Epochs: 10
Controller Training Segments: 2
☑ Use Current Weights
□ Use Cumulative Training

Perform plant identification before controller training.

The file menu has several items, including ones that allow you to import and export controller and plant networks.

This block specifies the inputs to the controller.

You must specify a Simulink reference model for the plant to follow.

The parameters in this block specify the random reference input for training. The reference is a series of random steps at random intervals.

The training data is broken into segments. Specify the number of training epochs for each segment.

You must generate or import training data before you can train the controller.

If selected, segments of data are added to the training set as training continues. Otherwise, only one segment at a time is used.

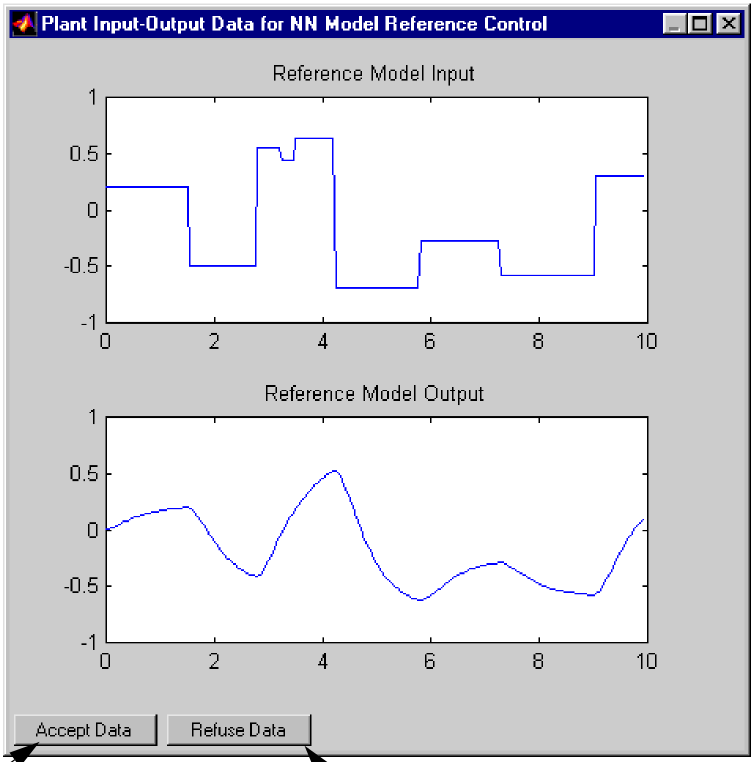
Current weights are used as initial conditions to continue training.

This button opens the Plant Identification window. The plant must be identified before the controller is trained.

After the controller has been trained, select **OK** or **Apply** to load the network into the Simulink model.

**4** The next step would normally be to select **Plant Identification**, which opens the **Plant Identification** window. You would then train the plant model. Since the **Plant Identification** window is identical to the one used with the previous controllers, we won't go through that process here.

5 Select **Generate Data**. The program then starts generating the data for training the controller. After the data is generated, the following window appears.

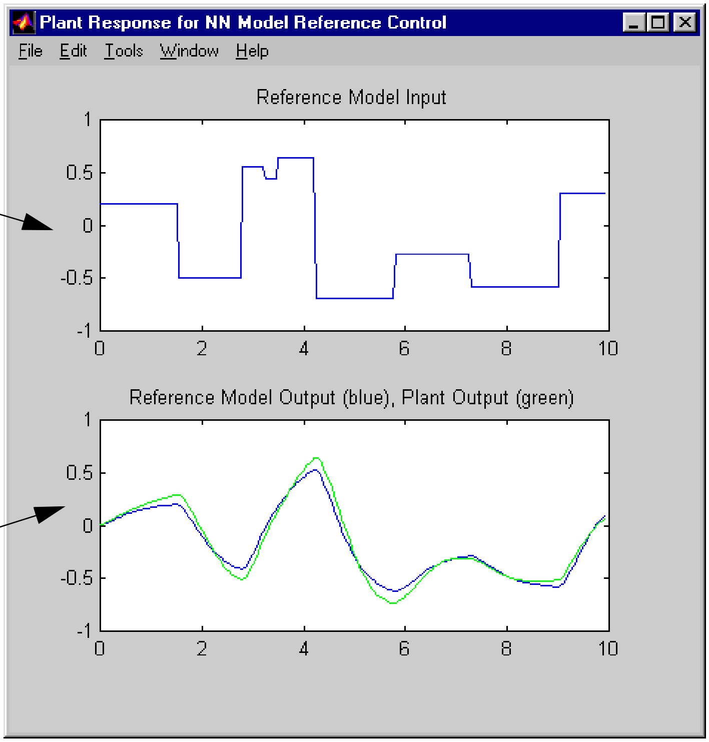


Select this if the training data shows enough variation to adequately train the controller.

If the data is not adequate, select this button and then go back to the controller window and select **Generate Data** again.

6 Select **Accept Data**. Return to the **Model Reference Control** window and select **Train Controller**. The program presents one segment of data to the network and trains the network for a specified number of iterations (five in this case). This process continues one segment at a time until the entire training set has been presented to the network. Controller training can be significantly more time consuming than plant model training. This is

because the controller must be trained using *dynamic* backpropagation (see [HaJe99]). After the training is complete, the response of the resulting closed loop system is displayed, as in the following figure.

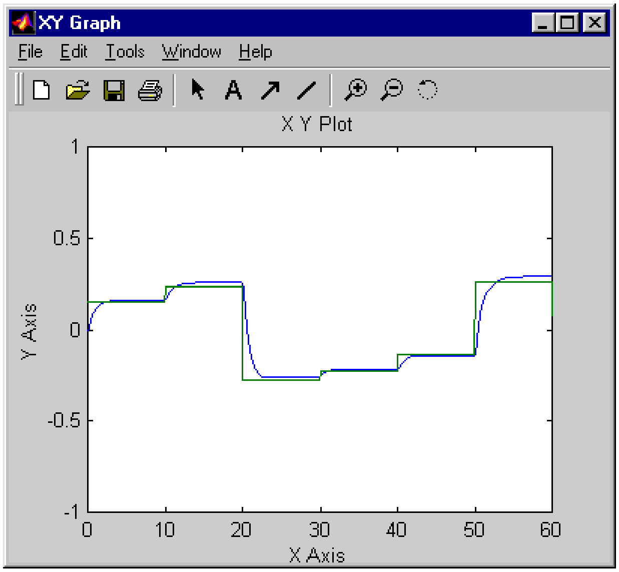

Plant Response for NN Model Reference Control
File Edit Tools Window Help
Reference Model Input
1
0.5
0
-0.5
-1
0
2
4
6
8
10
Reference Model Output (blue), Plant Output (green)
1
0.5
0
-0.5
-1
0
2
4
6
8
10

This axis displays the random reference input that was used for training.

This axis displays the response of the reference model and the response of the closed loop plant. The plant response should follow the reference model.

**7** Go back to the **Model Reference Control** window. If the performance of the controller is not accurate, then you can select **Train Controller** again, which continues the controller training with the same data set. If you would like to use a new data set to continue training, the select **Generate Data** or **Import Data** before you select **Train Controller**. (Be sure that **Use Current Weights** is selected, if you want to continue training with the same weights.) It may also be necessary to retrain the plant model. If the plant model is not accurate, it can affect the controller training. For this demonstration, the controller should be accurate enough, so select **OK**. This loads the controller weights into the Simulink model.

8 Return to the Simulink model and start the simulation by selecting the **Start** command from the **Simulation** menu. As the simulation runs, the plant output and the reference signal are displayed, as in the following figure.



# Importing and Exporting

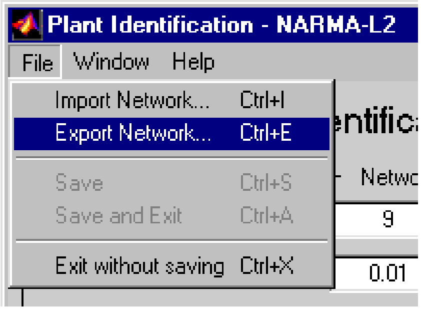
You can save networks and training data to the workspace or to a disk file. The following two sections demonstrate how you can do this.

## Importing and Exporting Networks

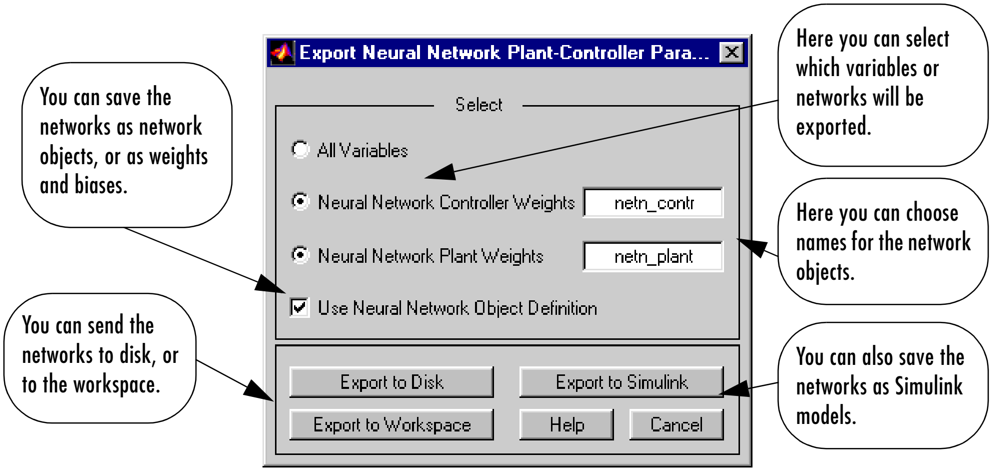
The controller and plant model networks that you develop are stored within Simulink controller blocks. At some point you may want to transfer the networks into other applications, or you may want to transfer a network from one controller block to another. You can do this by using the **Import Network** and **Export Network** menu options. The following demonstration leads you through the export and import processes. (We use the NARMA-L2 window for this demonstration, but the same procedure applies to all of the controllers.)

1 Repeat the first three steps of the NARMA-L2 demonstration. The **NARMA-L2 Plant Identification** window should then be open.

2 Select **Export** from the **File** menu, as shown below.



This causes the following window to open.



○ All Variables

● Neural Network Controller Weights: netn_contr

● Neural Network Plant Weights: netn_plant

☑ Use Neural Network Object Definition

You can save the networks as network objects, or as weights and biases.

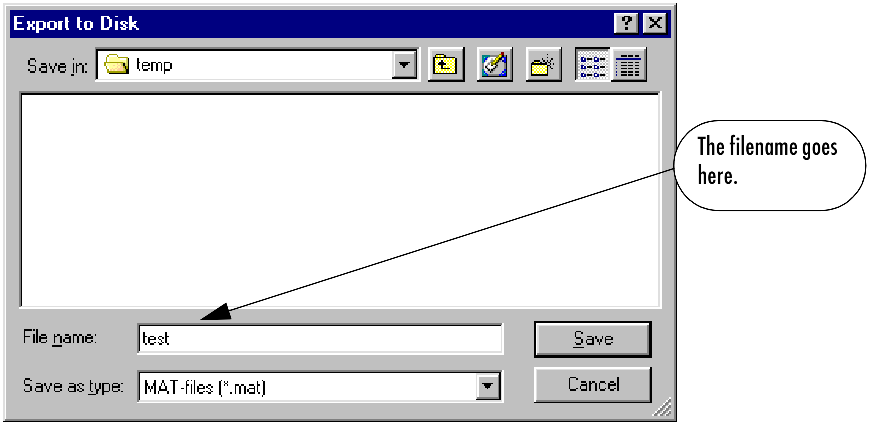
You can send the networks to disk, or to the workspace.

Here you can select which variables or networks will be exported.

Here you can choose names for the network objects.

You can also save the networks as Simulink models.

**3** Select **Export to Disk**. The following window opens. Enter the filename `test` in the box, and select **Save**. This saves the controller and plant networks to disk.



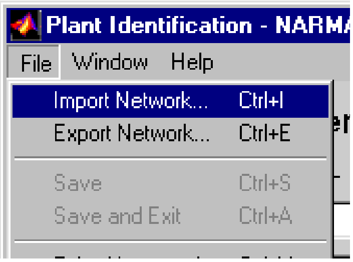
Save in: temp

File name: test

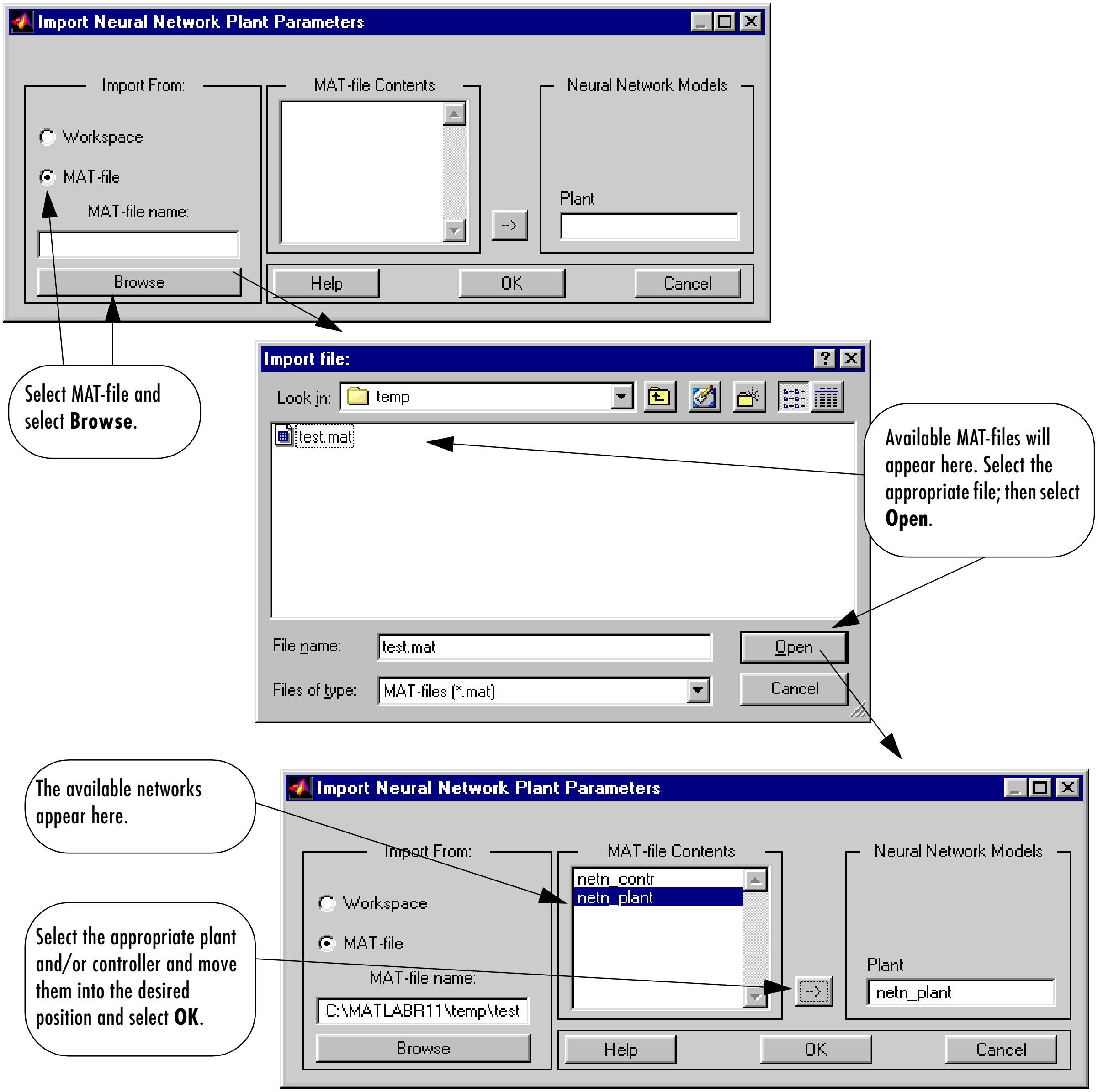
Save as type: MAT-files (*.mat)

The filename goes here.

**4** Retrieve that data with the **Import** menu option. Select **Import Network** from the **File** menu, as in the following figure.



This causes the following window to appear. Follow the steps indicated on the following page to retrieve the data that you previously exported. Once the data is retrieved, you can load it into the controller block by selecting **OK** or **Apply**. Notice that the window only has an entry for the plant model, even though you saved both the plant model and the controller. This is because the NARMA-L2 controller is derived directly from the plant model, so you don't need to import both networks.



File name: test.mat

Files of type: MAT-files (*.mat)

MAT-file name: C:\MATLABR11\temp\test

Plant: netn_plant

Select MAT-file and select **Browse**.

Available MAT-files will appear here. Select the appropriate file; then select **Open**.

The available networks appear here.

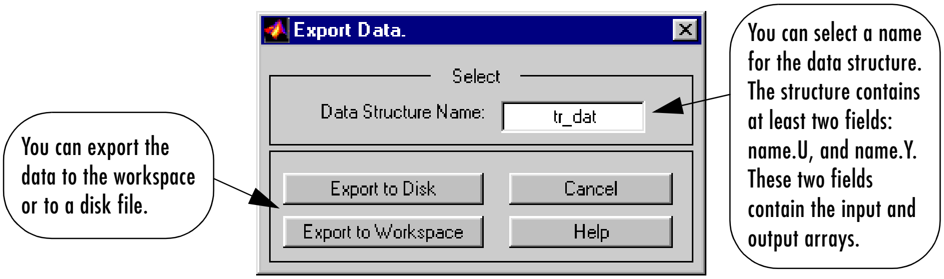
Select the appropriate plant and/or controller and move them into the desired position and select **OK**.

## Importing and Exporting Training Data

The data that you generate to train networks exists only in the corresponding plant identification or controller training window. You may wish to save the training data to the workspace or to a disk file so that you can load it again at a later time. You may also want to combine data sets manually and then load them back into the training window. You can do this by using the Import and Export buttons. The following demonstration leads you through the import and export processes. (We use the **NN Predictive Control** window for this demonstration, but the same procedure applies to all of the controllers.)

1 Repeat the first five steps of the NN Predictive Control demonstration. Then select **Accept Data**. The **Plant Identification** window should then be open, and the Import and Export buttons should be active.

2 Select the **Export** button. This causes the following window to open.

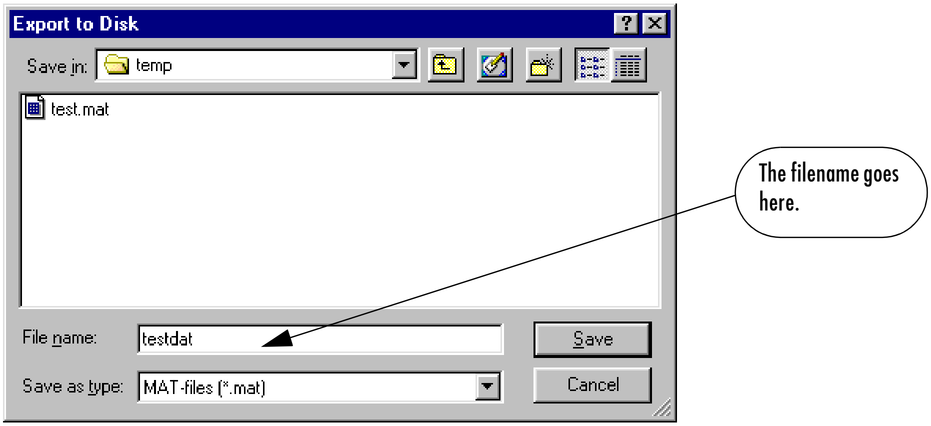


Data Structure Name: tr_dat

You can export the data to the workspace or to a disk file.

You can select a name for the data structure. The structure contains at least two fields: name.U, and name.Y. These two fields contain the input and output arrays.

3 Select **Export to Disk**. The following window opens. Enter the filename `testdat` in the box, and select **Save**. This saves the training data structure to disk.

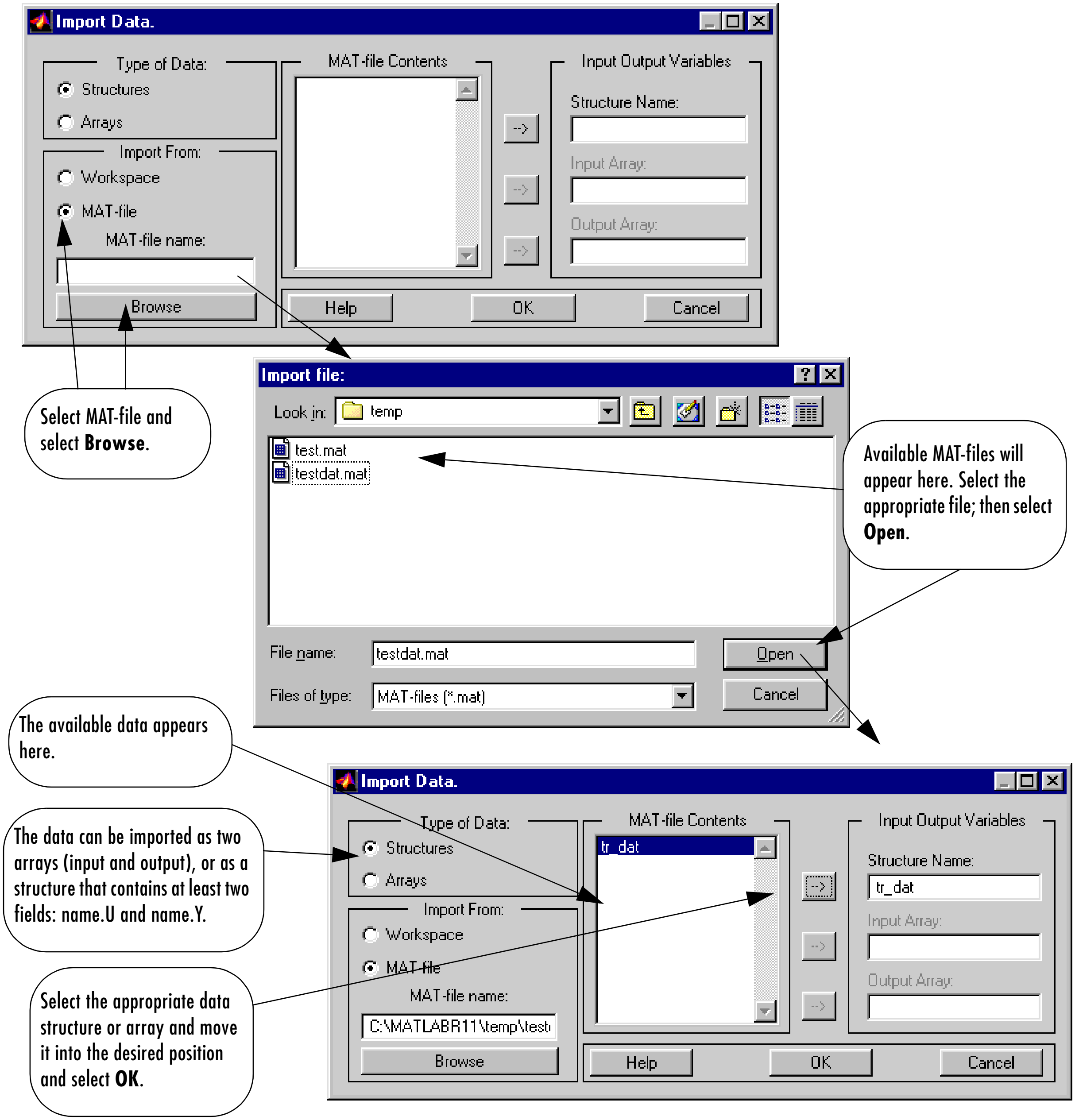


File name: testdat

Save as type: MAT-files (*.mat)

The filename goes here.

4 Now let's retrieve the data with the import command. Select the **Import** button in the **Plant Identification** window.This causes the following window to appear. Follow the steps indicated on the following page to retrieve the data that you previously exported. Once the data is imported, you can train the neural network plant model.



Import Data: Type of Data: ● Structures ○ Arrays; Import From: ○ Workspace ● MAT-file

Import file: Look in: temp; File name: testdat.mat; Files of type: MAT-files (*.mat)

Import Data: Type of Data: ● Structures ○ Arrays; Import From: ○ Workspace ● MAT-file; MAT-file name: C:\MATLABR11\temp\test; MAT-file Contents: tr_dat; Structure Name: tr_dat

Select MAT-file and select **Browse**.

Available MAT-files will appear here. Select the appropriate file; then select **Open**.

The available data appears here.

The data can be imported as two arrays (input and output), or as a structure that contains at least two fields: name.U and name.Y.

Select the appropriate data structure or array and move it into the desired position and select **OK**.

## Summary

The following table summarizes the controllers discussed in this chapter.

| Block | Description |
|---|---|
| NN Predictive Control | Uses a neural network plant model to predict future plant behavior. An optimization algorithm determines the control input that optimizes plant performance over a finite time horizon. The plant training requires only a batch algorithm for static networks and is reasonably fast. The controller requires an online optimization algorithm, which requires more computation than the other controllers. |
| NARMA-L2 Control | An approximate plant model is in companion form. The next control input is computed to force the plant output to follow a reference signal. The neural network plant model is trained with static backpropagation and is reasonably fast. The controller is a rearrangement of the plant model, and requires minimal online computation. |
| Model Reference Control | A neural network plant model is first developed. The plant model is then used to train a neural network controller to force the plant output to follow the output of a reference model. This control architecture requires the use of dynamic backpropagation for training the controller. This generally takes more time than training static networks with the standard backpropagation algorithm. However, this approach applies to a more general class of plant than does the NARMA-L2 control architecture. The controller requires minimal online computation. |

7

# Radial Basis Networks

# Introduction

Radial basis networks may require more neurons than standard feed-forward backpropagation networks, but often they can be designed in a fraction of the time it takes to train standard feed-forward networks. They work best when many training vectors are available.

You may want to consult the following paper on this subject:

Chen, S., C.F.N. Cowan, and P. M. Grant, "Orthogonal Least Squares Learning Algorithm for Radial Basis Function Networks," *IEEE Transactions on Neural Networks*, vol. 2, no. 2, March 1991, pp. 302-309.

This chapter discusses two variants of radial basis networks, Generalized Regression networks (GRNN) and Probabilistic neural networks (PNN). You may want to read about them in P.D. Wasserman, *Advanced Methods in Neural Computing*, New York: Van Nostrand Reinhold, 1993 on pp. 155-61, and pp. 35-55 respectively.

## Important Radial Basis Functions

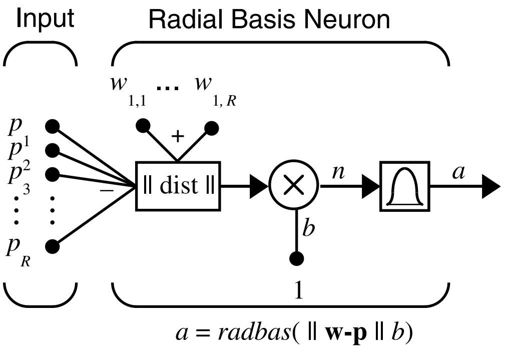
Radial basis networks can be designed with either `newrbe` or `newrb`. GRNN and PNN can be designed with `newgrnn` and `newpnn`, respectively.

Type `help radbasis` to see a listing of all functions and demonstrations related to radial basis networks.

# Radial Basis Functions

## Neuron Model

Here is a radial basis network with $R$ inputs.

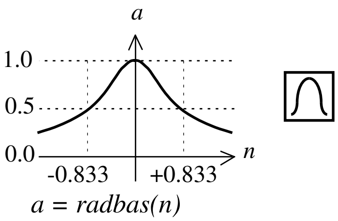


Notice that the expression for the net input of a `radbas` neuron is different from that of neurons in previous chapters. Here the net input to the `radbas` transfer function is the vector distance between its weight vector **w** and the input vector **p**, multiplied by the bias $b$. (The ‖ dist ‖ box in this figure accepts the input vector **p** and the single row input weight matrix, and produces the dot product of the two.)

The transfer function for a radial basis neuron is:

$$radbas(n) = e^{-n^2}$$

Here is a plot of the `radbas` transfer function.



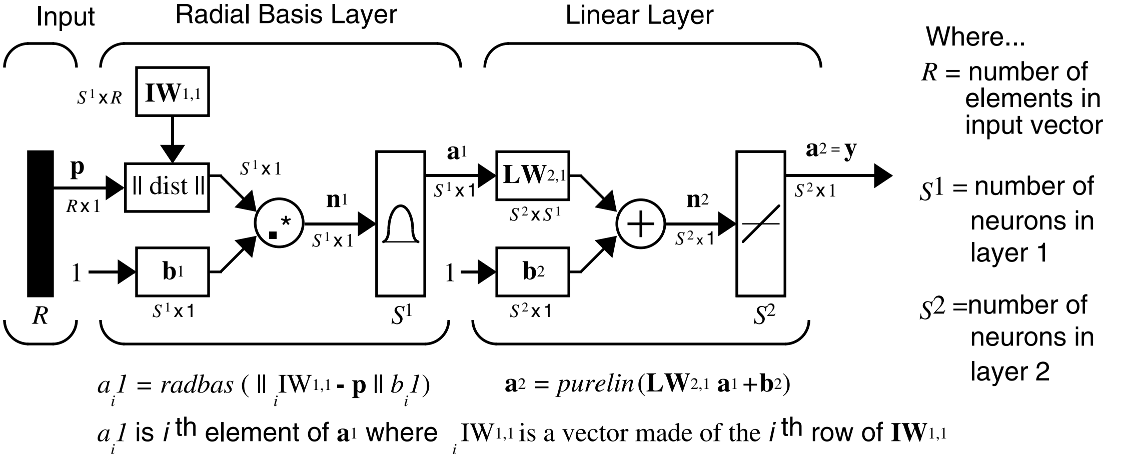
Radial Basis Function

The radial basis function has a maximum of 1 when its input is 0. As the distance between **w** and **p** decreases, the output increases. Thus, a radial basis neuron acts as a detector that produces 1 whenever the input **p** is identical to its weight vector **p**.

The bias $b$ allows the sensitivity of the `radbas` neuron to be adjusted. For example, if a neuron had a bias of 0.1 it would output 0.5 for any input vector **p** at vector distance of 8.326 (0.8326/$b$) from its weight vector **w**.

## Network Architecture

Radial basis networks consist of two layers: a hidden radial basis layer of $S^1$ neurons, and an output linear layer of $S^2$ neurons.


Input
Radial Basis Layer
Linear Layer
Where...
R = number of elements in input vector
S1 = number of neurons in layer 1
S2 =number of neurons in layer 2
S¹xR
IW1,1
p
Rx1
|| dist ||
S¹x1
.*
n1
S¹x1
1
b1
S¹x1
R
S1
a1
S¹x1
LW2,1
S²xS¹
1
b2
S²x1
n2
S²x1
S2
a2 = y
S²x1
a_i1 = radbas ( || _iIW1,1 - p || b_i1)
a2 = purelin (LW2,1 a1 +b2)
a_i1 is i th element of a1 where _iIW1,1 is a vector made of the i th row of IW1,1

The ‖ dist ‖ box in this figure accepts the input vector **p** and the input weight matrix **IW**$^{1,1}$, and produces a vector having $S_1$ elements. The elements are the distances between the input vector and vectors $_i$**IW**$^{1,1}$ formed from the rows of the input weight matrix.

The bias vector **b**$^1$ and the output of ‖ dist ‖ are combined with the MATLAB® operation .* , which does element-by-element multiplication.

The output of the first layer for a feed forward network *net* can be obtained with the following code:

```
a{1} = radbas(netprod(dist(net.IW{1,1},p),net.b{1}))
```

Fortunately, you won't have to write such lines of code. All of the details of designing this network are built into design functions newrbe and newrb, and their outputs can be obtained with sim.

We can understand how this network behaves by following an input vector **p** through the network to the output $\mathbf{a}^2$. If we present an input vector to such a network, each neuron in the radial basis layer will output a value according to how close the input vector is to each neuron's weight vector.

Thus, radial basis neurons with weight vectors quite different from the input vector **p** have outputs near zero. These small outputs have only a negligible effect on the linear output neurons.

In contrast, a radial basis neuron with a weight vector close to the input vector **p** produces a value near 1. If a neuron has an output of 1 its output weights in the second layer pass their values to the linear neurons in the second layer.

In fact, if only one radial basis neuron had an output of 1, and all others had outputs of 0's (or very close to 0), the output of the linear layer would be the active neuron's output weights. This would, however, be an extreme case. Typically several neurons are always firing, to varying degrees.

Now let us look in detail at how the first layer operates. Each neuron's weighted input is the distance between the input vector and its weight vector, calculated with dist. Each neuron's net input is the element-by-element product of its weighted input with its bias, calculated with netprod. Each neuron's output is its net input passed through radbas. If a neuron's weight vector is equal to the input vector (transposed), its weighted input is 0, its net input is 0, and its output is 1. If a neuron's weight vector is a distance of spread from the input vector, its weighted input is spread, its net input is sqrt(-log(.5)) (or 0.8326), therefore its output is 0.5.

## Exact Design (newrbe)

Radial basis networks can be designed with the function newrbe. This function can produce a network with zero error on training vectors. It is called in the following way.

```
net = newrbe(P,T,SPREAD)
```

The function newrbe takes matrices of input vectors P and target vectors T, and a spread constant SPREAD for the radial basis layer, and returns a network with weights and biases such that the outputs are exactly T when the inputs are P.

This function `newrbe` creates as many `radbas` neurons as there are input vectors in P, and sets the first-layer weights to P'. Thus, we have a layer of `radbas` neurons in which each neuron acts as a detector for a different input vector. If there are $Q$ input vectors, then there will be $Q$ neurons.

Each bias in the first layer is set to 0.8326/`SPREAD`. This gives radial basis functions that cross 0.5 at weighted inputs of +/- `SPREAD`. This determines the width of an area in the input space to which each neuron responds. If `SPREAD` is 4, then each `radbas` neuron will respond with 0.5 or more to any input vectors within a vector distance of 4 from their weight vector. As we shall see, `SPREAD` should be large enough that neurons respond strongly to overlapping regions of the input space.

The second-layer weights IW $^{2,1}$ (or in code, IW{2,1}) and biases b$^2$ (or in code, b{2}) are found by simulating the first-layer outputs a$^1$ (A{1}), and then solving the following linear expression.

```
[W{2,1} b{2}] * [A{1}; ones] = T
```

We know the inputs to the second layer (A{1}) and the target (T), and the layer is linear. We can use the following code to calculate the weights and biases of the second layer to minimize the sum-squared error.

```
Wb = T/[P; ones(1,Q)]
```

Here `Wb` contains both weights and biases, with the biases in the last column. The sum-squared error will always be 0, as explained below.

We have a problem with $C$ constraints (input/target pairs) and each neuron has $C$ +1 variables (the $C$ weights from the $C$ `radbas` neurons, and a bias). A linear problem with $C$ constraints and more than $C$ variables has an infinite number of zero error solutions!

Thus, `newrbe` creates a network with zero error on training vectors. The only condition we have to meet is to make sure that `SPREAD` is large enough so that the active input regions of the `radbas` neurons overlap enough so that several `radbas` neurons always have fairly large outputs at any given moment. This makes the network function smoother and results in better generalization for new input vectors occurring between input vectors used in the design. (However, `SPREAD` should not be so large that each neuron is effectively responding in the same, large, area of the input space.)

The drawback to `newrbe` is that it produces a network with as many hidden neurons as there are input vectors. For this reason, `newrbe` does not return an acceptable solution when many input vectors are needed to properly define a network, as is typically the case.

## More Efficient Design (newrb)

The function newrb iteratively creates a radial basis network one neuron at a time. Neurons are added to the network until the sum-squared error falls beneath an error goal or a maximum number of neurons has been reached. The call for this function is:

```
net = newrb(P,T,GOAL,SPREAD)
```

The function newrb takes matrices of input and target vectors, P and T, and design parameters GOAL and, SPREAD, and returns the desired network.

The design method of newrb is similar to that of newrbe. The difference is that newrb creates neurons one at a time. At each iteration the input vector that results in lowering the network error the most, is used to create a radbas neuron. The error of the new network is checked, and if low enough newrb is finished. Otherwise the next neuron is added. This procedure is repeated until the error goal is met, or the maximum number of neurons is reached.

As with newrbe, it is important that the spread parameter be large enough that the radbas neurons respond to overlapping regions of the input space, but not so large that all the neurons respond in essentially the same manner.

Why not always use a radial basis network instead of a standard feed-forward network? Radial basis networks, even when designed efficiently with newrbe, tend to have many times more neurons than a comparable feed-forward network with tansig or logsig neurons in the hidden layer.

This is because sigmoid neurons can have outputs over a large region of the input space, while radbas neurons only respond to relatively small regions of the input space. The result is that the larger the input space (in terms of number of inputs, and the ranges those inputs vary over) the more radbas neurons required.

On the other hand, designing a radial basis network often takes much less time than training a sigmoid/linear network, and can sometimes result in fewer neurons being used, as can be seen in the next demonstration.

## Demonstrations

The demonstration script demorb1 shows how a radial basis network is used to fit a function. Here the problem is solved with only five neurons.

Demonstration scripts demorb3 and demorb4 examine how the spread constant affects the design process for radial basis networks.

In demorb3, a radial basis network is designed to solve the same problem as in demorb1. However, this time the spread constant used is 0.01. Thus, each radial basis neuron returns 0.5 or lower, for any input vectors with a distance of 0.01 or more from its weight vector.

Because the training inputs occur at intervals of 0.1, no two radial basis neurons have a strong output for any given input.

In demorb3, it was demonstrated that having too small a spread constant can result in a solution that does not generalize from the input/target vectors used in the design. This demonstration, demorb4, shows the opposite problem. If the spread constant is large enough, the radial basis neurons will output large values (near 1.0) for all the inputs used to design the network.

If all the radial basis neurons always output 1, any information presented to the network becomes lost. No matter what the input, the second layer outputs 1's. The function newrb will attempt to find a network, but will not be able to do so because to numerical problems that arise in this situation.

The moral of the story is, choose a spread constant larger than the distance between adjacent input vectors, so as to get good generalization, but smaller than the distance across the whole input space.

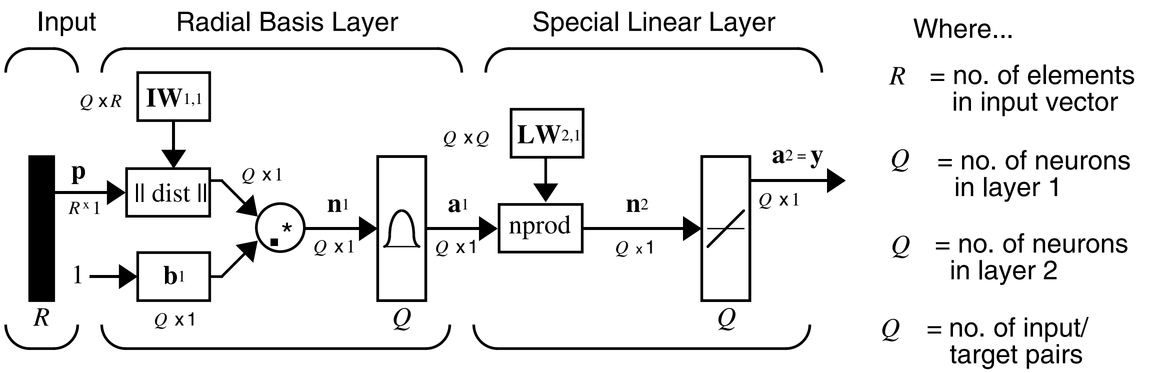
For this problem that would mean picking a spread constant greater than 0.1, the interval between inputs, and less than 2, the distance between the left-most and right-most inputs.

# Generalized Regression Networks

A generalized regression neural network (GRNN) is often used for function approximation. As discussed below, it has a radial basis layer and a special linear layer.

## Network Architecture

The architecture for the GRNN is shown below. It is similar to the radial basis network, but has a slightly different second layer.



$a_i^1 = radbas(\| {}_i\mathrm{IW}^{1,1} - \mathbf{p} \| \, b_i^1)$ $\qquad$ $\mathbf{a}^2 = purelin(\mathbf{n}^2)$

$a_i^1$ is $i$ th element of $\mathbf{a}^1$ where ${}_i\mathrm{IW}^{1,1}$ is a vector made of the $i$ th row of $\mathbf{IW}^{1,1}$

Here the **nprod** box shown above (code function normprod) produces $S^2$ elements in vector $\mathbf{n}^2$. Each element is the dot product of a row of $\mathrm{LW}^{2,1}$ and the input vector $\mathbf{a}^1$, all normalized by the sum of the elements of $\mathbf{a}^1$. For instance, suppose that:

```
LW{1,2}= [1 -2;3 4;5 6];
a{1} = [7; -8;
```

Then

```
aout = normprod(LW{1,2},a{1})
aout =
    -23
     11
     13
```

The first layer is just like that for newrbe networks. It has as many neurons as there are input/ target vectors in **P**. Specifically, the first layer weights are set to **P**'. The bias $\mathbf{b}^1$ is set to a column vector of 0.8326/SPREAD. The user chooses SPREAD, the distance an input vector must be from a neuron's weight vector to be 0.5.

Again, the first layer operates just like the newbe radial basis layer described previously. Each neuron's weighted input is the distance between the input vector and its weight vector, calculated with dist. Each neuron's net input is the product of its weighted input with its bias, calculated with netprod. Each neurons' output is its net input passed through radbas. If a neuron's weight vector is equal to the input vector (transposed), its weighted input will be 0, its net input will be 0, and its output will be 1. If a neuron's weight vector is a distance of spread from the input vector, its weighted input will be spread, and its net input will be sqrt(-log(.5)) (or 0.8326). Therefore its output will be 0.5.

The second layer also has as many neurons as input/target vectors, but here LW{2,1} is set to T.

Suppose we have an input vector **p** close to $\mathbf{p}_i$, one of the input vectors among the input vector/target pairs used in designing layer one weights. This input **p** produces a layer 1 $\mathbf{a}^i$ output close to 1. This leads to a layer 2 output close to $\mathbf{t}_i$, one of the targets used forming layer 2 weights.

A larger spread leads to a large area around the input vector where layer 1 neurons will respond with significant outputs.Therefore if spread is small the radial basis function is very steep so that the neuron with the weight vector closest to the input will have a much larger output than other neurons. The network will tend to respond with the target vector associated with the nearest design input vector.

As spread gets larger the radial basis function's slope gets smoother and several neuron's may respond to an input vector. The network then acts like it is taking a weighted average between target vectors whose design input vectors are closest to the new input vector. As spread gets larger more and more neurons contribute to the average with the result that the network function becomes smoother.

## Design (newgrnn)

You can use the function newgrnn to create a GRNN. For instance, suppose that three input and three target vectors are defined as

```
P = [4 5 6];
T = [1.5 3.6 6.7];
```

We can now obtain a GRNN with

```
net = newgrnn(P,T);
```

and simulate it with

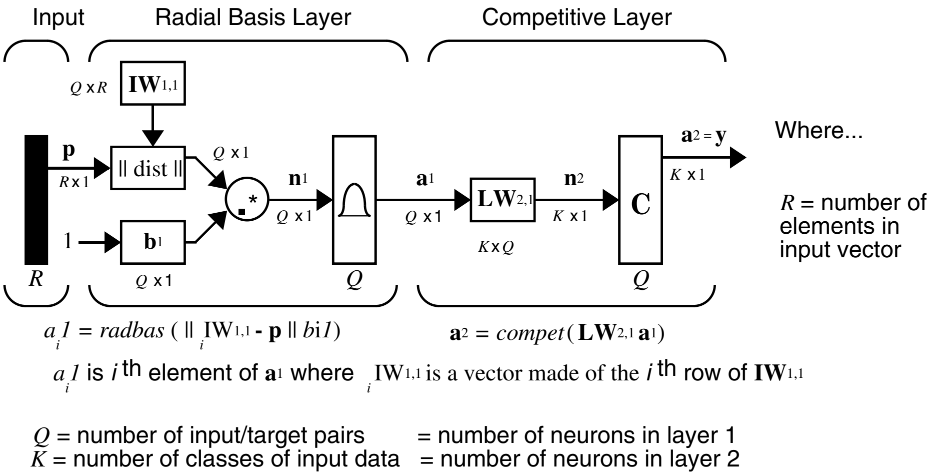
```
P = 4.5;
v = sim(net,P)
```

You might want to try `demogrn1`. It shows how to approximate a function with a GRNN.

# Probabilistic Neural Networks

Probabilistic neural networks can be used for classification problems. When an input is presented, the first layer computes distances from the input vector to the training input vectors, and produces a vector whose elements indicate how close the input is to a training input. The second layer sums these contributions for each class of inputs to produce as its net output a vector of probabilities. Finally, a *compete* transfer function on the output of the second layer picks the maximum of these probabilities, and produces a 1 for that class and a 0 for the other classes. The architecture for this system is shown below.

## Network Architecture



Input
Radial Basis Layer
Competitive Layer
Q x R
IW1,1
p
R x 1
|| dist ||
Q x 1
.*
n1
Q x 1
1
b1
Q x 1
R
Q
a1
Q x 1
LW2,1
K x Q
n2
K x 1
C
Q
a2 = y
K x 1
Where...
R = number of elements in input vector
a_i1 = radbas ( || _iIW1,1 - p || bi1)
a2 = compet(LW2,1 a1)
a_i1 is i th element of a1 where _iIW1,1 is a vector made of the i th row of IW1,1
Q = number of input/target pairs = number of neurons in layer 1
K = number of classes of input data = number of neurons in layer 2

It is assumed that there are Q input vector/target vector pairs. Each target vector has K elements. One of these element is 1 and the rest is 0. Thus, each input vector is associated with one of K classes.

The first-layer input weights, $\mathrm{IW}^{1,1}$ (`net.IW{1,1}`) are set to the transpose of the matrix formed from the Q training pairs, **P**'. When an input is presented the ||dist|| box produces a vector whose elements indicate how close the input is to the vectors of the training set. These elements are multiplied, element by element, by the bias and sent the `radbas` transfer function. An input vector close to a training vector is represented by a number close to 1 in the output vector $\mathbf{a}^1$. If an input is close to several training vectors of a single class, it is represented by several elements of $\mathbf{a}^1$ that are close to 1.

The second-layer weights, $LW^{1,2}$ (`net.LW{2,1}`), are set to the matrix **T** of target vectors. Each vector has a 1 only in the row associated with that particular class of input, and 0's elsewhere. (A function `ind2vec` is used to create the proper vectors.) The multiplication $\mathbf{Ta}^1$ sums the elements of $\mathbf{a}^1$ due to each of the K input classes. Finally, the second-layer transfer function, compete, produces a 1 corresponding to the largest element of $\mathbf{n}^2$, and 0's elsewhere. Thus, the network has classified the input vector into a specific one of K classes because that class had the maximum probability of being correct.

## Design (newpnn)

You can use the function `newpnn` to create a PNN. For instance, suppose that seven input vectors and their corresponding targets are

```
P = [0 0;1 1;0 3;1 4;3 1;4 1;4 3]'
```

which yields

```
P =
     0     1     0     1     3     4     4
     0     1     3     4     1     1     3
Tc = [1 1 2 2 3 3 3];
```

which yields

```
Tc =
     1     1     2     2     3     3     3
```

We need a target matrix with 1's in the right place. We can get it with the function `ind2vec`. It gives a matrix with 0's except at the correct spots. So execute

```
T = ind2vec(Tc)
```

which gives

```
T =
   (1,1)        1
   (1,2)        1
   (2,3)        1
   (2,4)        1
   (3,5)        1
   (3,6)        1
   (3,7)        1
```

Now we can create a network and simulate it, using the input P to make sure that it does produce the correct classifications. We use the function `vec2ind` to convert the output Y into a row Yc to make the classifications clear.

```
net = newpnn(P,T);
Y = sim(net,P)
Yc = vec2ind(Y)
```

Finally we get

```
Yc =
     1     1     2     2     3     3     3
```

We might try classifying vectors other than those that were used to design the network. We will try to classify the vectors shown below in P2.

```
P2 = [1 4;0 1;5 2]'

P2 =
     1     0     5
     4     1     2
```

Can you guess how these vectors will be classified? If we run the simulation and plot the vectors as we did before, we get

```
Yc =
     2     1     3
```

These results look good, for these test vectors were quite close to members of classes 2, 1 and 3 respectively. The network has managed to generalize its operation to properly classify vectors other than those used to design the network.

You might want to try `demopnn1`. It shows how to design a PNN, and how the network can successfully classify a vector not used in the design.

# Summary

Radial basis networks can be designed very quickly in two different ways.

The first design method, newrbe, finds an exact solution. The function newrbe creates radial basis networks with as many radial basis neurons as there are input vectors in the training data.

The second method, newrb, finds the smallest network that can solve the problem within a given error goal. Typically, far fewer neurons are required by newrb than are returned newrbe. However, because the number of radial basis neurons is proportional to the size of the input space, and the complexity of the problem, radial basis networks can still be larger than backpropagation networks.

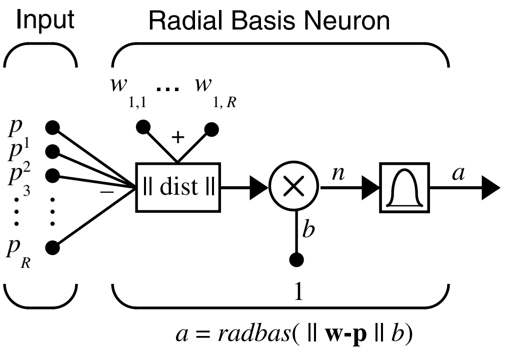
A generalized regression neural network (GRNN) is often used for function approximation. It has been shown that, given a sufficient number of hidden neurons, GRNNs can approximate a continuous function to an arbitrary accuracy.

Probabilistic neural networks (PNN) can be used for classification problems. Their design is straightforward and does not depend on training. A PNN is guaranteed to converge to a Bayesian classifier providing it is given enough training data. These networks generalize well.

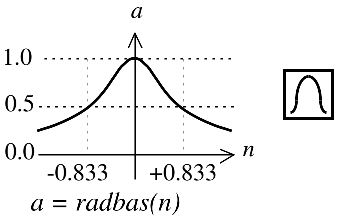
The GRNN and PNN have many advantages, but they both suffer from one major disadvantage. They are slower to operate because they use more computation than other kinds of networks to do their function approximation or classification.

## Figures

### Radial Basis Neuron

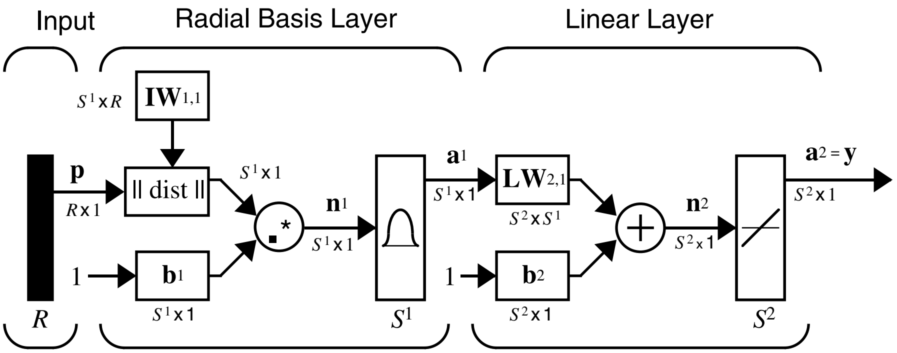


### Radbas Transfer Function



Radial Basis Function

## Radial Basis Network Architecture



Where...

$R$ = number of elements in input vector

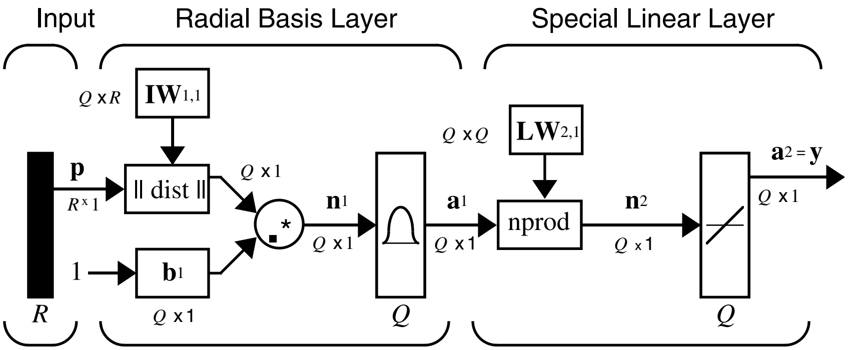
$S1$ = number of neurons in layer 1

$S2$ =number of neurons in layer 2

$a^1_i = radbas\,(\,\|\,{}_i\mathrm{IW}^{1,1} - \mathbf{p}\,\|\,b^1_i)$ $\qquad \mathbf{a}^2 = purelin\,(\mathbf{LW}^{2,1}\,\mathbf{a}^1 + \mathbf{b}^2)$

$a^1_i$ is $i$ th element of $\mathbf{a}^1$ where ${}_i\mathrm{IW}^{1,1}$ is a vector made of the $i$ th row of $\mathbf{IW}^{1,1}$

## Generalized Regression Neural Network Architecture



Where...

$R$ = no. of elements in input vector

$Q$ = no. of neurons in layer 1

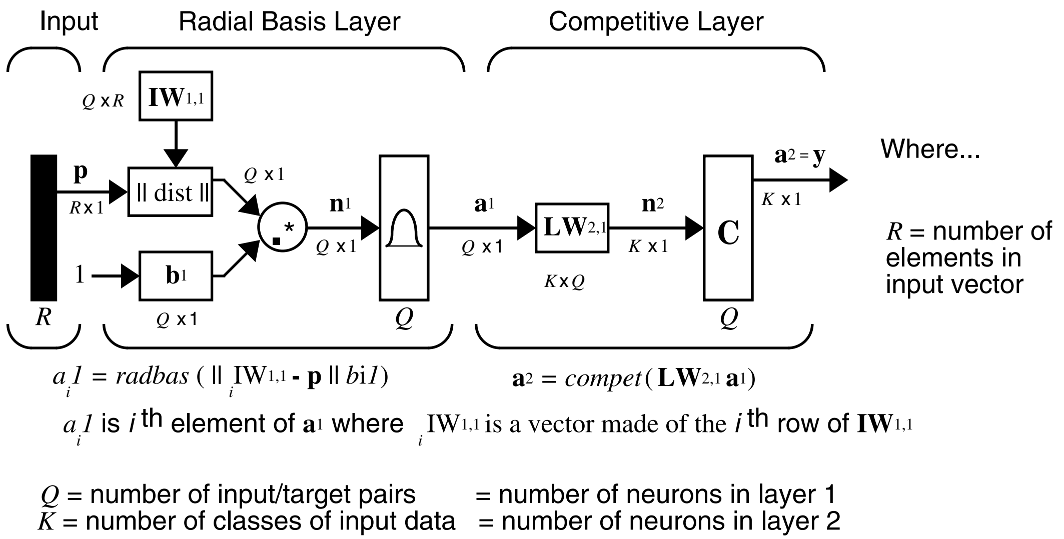
$Q$ = no. of neurons in layer 2

$Q$ = no. of input/ target pairs

$a^1_i = radbas\,(\,\|\,{}_i\mathrm{IW}^{1,1} - \mathbf{p}\,\|\,b^1_i)$ $\qquad \mathbf{a}^2 = purelin\,(\mathbf{n}^2)$

$a^1_i$ is $i$ th element of $\mathbf{a}^1$ where ${}_i\mathrm{IW}^{1,1}$ is a vector made of the $i$ th row of $\mathbf{IW}^{1,1}$

**Probabilistic Neural Network Architecture**


Input
Radial Basis Layer
Competitive Layer
$Q \times R$
$\mathbf{IW}^{1,1}$
$\mathbf{p}$
$R \times 1$
|| dist ||
$Q \times 1$
.*
$\mathbf{n}^1$
$Q \times 1$
1
$\mathbf{b}^1$
$Q \times 1$
$R$
$Q$
$\mathbf{a}^1$
$Q \times 1$
$\mathbf{LW}^{2,1}$
$K \times Q$
$\mathbf{n}^2$
$K \times 1$
C
$Q$
$\mathbf{a}^2 = \mathbf{y}$
$K \times 1$
Where...
$R$ = number of elements in input vector
$a_i1 = radbas(\| {}_i\mathbf{IW}^{1,1} - \mathbf{p} \| b_i1)$
$\mathbf{a}^2 = compet(\mathbf{LW}^{2,1}\mathbf{a}^1)$
$a_i1$ is $i$ th element of $\mathbf{a}^1$ where ${}_i\mathbf{IW}^{1,1}$ is a vector made of the $i$ th row of $\mathbf{IW}^{1,1}$
$Q$ = number of input/target pairs = number of neurons in layer 1
$K$ = number of classes of input data = number of neurons in layer 2

## New Functions

This chapter introduced the following functions.

| Function | Description |
|---|---|
| `compet` | Competitive transfer function. |
| `dist` | Euclidean distance weight function |
| `dotprod` | Dot product weight function. |
| `ind2vec` | Convert indices to vectors. |
| `negdist` | Negative euclidean distance weight function |
| `netprod` | Product net input function. |
| `newgrnn` | Design a generalized regression neural network. |

| Function | Description |
|---|---|
| newpnn | Design a probabilistic neural network. |
| newrb | Design a radial basis network. |
| newrbe | Design an exact radial basis network. |
| normprod | Normalized dot product weight function. |
| radbas | Radial basis transfer function. |
| vec2ind | Convert vectors to indices. |

# 8

# Self-Organizing and Learn. Vector Quant. Nets

# Introduction

Self-organizing in networks is one of the most fascinating topics in the neural network field. Such networks can learn to detect regularities and correlations in their input and adapt their future responses to that input accordingly. The neurons of competitive networks learn to recognize groups of similar input vectors. Self-organizing maps learn to recognize groups of similar input vectors in such a way that neurons physically near each other in the neuron layer respond to similar input vectors. A basic reference is

Kohonen, T. *Self-Organization and Associative Memory, 2nd Edition*, Berlin: Springer-Verlag, 1987.

Learning vector quantization (LVQ) is a method for training competitive layers in a supervised manner. A competitive layer automatically learns to classify input vectors. However, the classes that the competitive layer finds are dependent only on the distance between input vectors. If two input vectors are very similar, the competitive layer probably will put them in the same class. There is no mechanism in a strictly competitive layer design to say whether or not any two input vectors are in the same class or different classes.

LVQ networks, on the other hand, learn to classify input vectors into target classes chosen by the user.

You might consult the following reference:

Kohonen, T. *Self-Organization and Associative Memory, 2nd Edition*, Berlin: Springer-Verlag, 1987.

## Important Self-Organizing and LVQ Functions

Competitive layers and self organizing maps can be created with `newc` and `newsom`, respectively. A listing of all self-organizing functions and demonstrations can be found by typing `help selforg`.

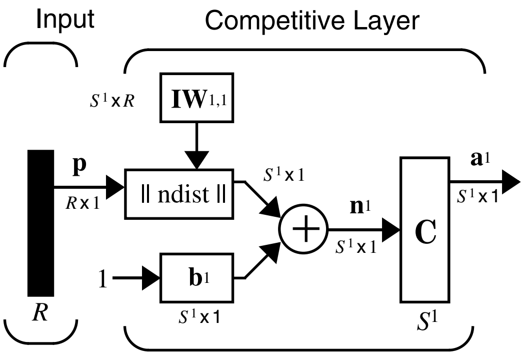
An LVQ network can be created with the function `newlvq`. For a list of all LVQ functions and demonstrations type `help lvq`.

# Competitive Learning

The neurons in a competitive layer distribute themselves to recognize frequently presented input vectors.

## Architecture

The architecture for a competitive network is shown below.



The ‖ dist ‖ box in this figure accepts the input vector **p** and the input weight matrix $\mathbf{IW}^{1,1}$, and produces a vector having $S_1$ elements. The elements are the negative of the distances between the input vector and vectors $_i\mathbf{IW}^{1,1}$ formed from the rows of the input weight matrix.

The net input $\mathbf{n}^1$ of a competitive layer is computed by finding the negative distance between input vector **p** and the weight vectors and adding the biases **b**. If all biases are zero, the maximum net input a neuron can have is 0. This occurs when the input vector **p** equals that neuron’s weight vector.

The competitive transfer function accepts a net input vector for a layer and returns neuron outputs of 0 for all neurons except for the *winner*, the neuron associated with the most positive element of net input $\mathbf{n}^1$. The winner’s output is 1. If all biases are 0, then the neuron whose weight vector is closest to the input vector has the *least* negative net input and, therefore, wins the competition to output a 1.

Reasons for using biases with competitive layers are introduced in a later section on training.

## Creating a Competitive Neural Network (newc)

A competitive neural network can be created with the function `newc`. We show how this works with a simple example.

Suppose we want to divide the following four two-element vectors into two classes.

```
p = [.1 .8  .1 .9; .2 .9 .1 .8]
p =
    0.1000    0.8000    0.1000    0.9000
    0.2000    0.9000    0.1000    0.8000
```

Thus, we have two vectors near the origin and two vectors near (1,1).

First, create a two-neuron layer with two input elements ranging from 0 to 1. The first argument gives the range of the two input vectors and the second argument says that there are to be two neurons.

```
net = newc([0 1; 0 1],2);
```

The weights are initialized to the center of the input ranges with the function `midpoint`. We can check to see these initial values as follows:

```
wts = net.IW{1,1}
wts =
    0.5000    0.5000
    0.5000    0.5000
```

These weights are indeed the values at the midpoint of the range (0 to 1) of the inputs, as we would expect when using `midpoint` for initialization.

The biases are computed by `initcon`, which gives

```
biases =
    5.4366
    5.4366
```

Now we have a network, but we need to train it to do the classification job.

Recall that each neuron competes to respond to an input vector **p**. If the biases are all 0, the neuron whose weight vector is closest to **p** gets the highest net input and, therefore, wins the competition and outputs 1. All other neurons output 0. We would like to adjust the winning neuron so as to move it closer to the input. A learning rule to do this is discussed in the next section.

## Kohonen Learning Rule (learnk)

The weights of the winning neuron (a row of the input weight matrix) are adjusted with the *Kohonen learning* rule. Supposing that the i$^{th}$ neuron wins, the elements of the i$^{th}$ row of the input weight matrix are adjusted as shown below.

$${}_i\mathbf{IW}^{1,1}(q) = {}_i\mathbf{IW}^{1,1}(q-1) + \alpha(\mathbf{p}(q) - {}_i\mathbf{IW}^{1,1}(q-1))$$

The Kohonen rule allows the weights of a neuron to learn an input vector, and because of this it is useful in recognition applications.

Thus, the neuron whose weight vector was closest to the input vector is updated to be even closer. The result is that the winning neuron is more likely to win the competition the next time a similar vector is presented, and less likely to win when a very different input vector is presented. As more and more inputs are presented, each neuron in the layer closest to a group of input vectors soon adjusts its weight vector toward those input vectors. Eventually, if there are enough neurons, every cluster of similar input vectors will have a neuron that outputs 1 when a vector in the cluster is presented, while outputting a 0 at all other times. Thus, the competitive network learns to categorize the input vectors it sees.

The function `learnk` is used to perform the Kohonen learning rule in this toolbox.

## Bias Learning Rule (learncon)

One of the limitations of competitive networks is that some neurons may not always get *allocated*. In other words, some neuron weight vectors may start out far from any input vectors and never win the competition, no matter how long the training is continued. The result is that their weights do not get to learn and they never win. These unfortunate neurons, referred to as *dead neurons*, never perform a useful function.

To stop this from happening, biases are used to give neurons that only win the competition rarely (if ever) an advantage over neurons that win often. A positive bias, added to the negative distance, makes a distant neuron more likely to win.

To do this job a running average of neuron outputs is kept. It is equivalent to the percentages of times each output is 1. This average is used to update the biases with the learning function `learncon` so that the biases of frequently active neurons will get smaller, and biases of infrequently active neurons will get larger.

The learning rates for `learncon` are typically set an order of magnitude or more smaller than for `learnk`. Doing this helps make sure that the running average is accurate.

The result is that biases of neurons that haven't responded very frequently will increase versus biases of neurons that have responded frequently. As the biases of infrequently active neurons increase, the input space to which that neuron responds increases. As that input space increases, the infrequently active neuron responds and moves toward more input vectors. Eventually the neuron will respond to an equal number of vectors as other neurons.

This has two good effects. First, if a neuron never wins a competition because its weights are far from any of the input vectors, its bias will eventually get large enough so that it will be able to win. When this happens, it will move toward some group of input vectors. Once the neuron's weights have moved into a group of input vectors and the neuron is winning consistently, its bias will decrease to 0. Thus, the problem of dead neurons is resolved.

The second advantage of biases is that they force each neuron to classify roughly the same percentage of input vectors. Thus, if a region of the input space is associated with a larger number of input vectors than another region, the more densely filled region will attract more neurons and be classified into smaller subsections.

## Training

Now train the network for 500 epochs. Either `train` or `adapt` can be used.

```
net.trainParam.epochs = 500
net = train(net,p);
```

Note that `train` for competitive networks uses the training function `trainr`. You can verify this by executing the following code after creating the network.

```
net.trainFcn
```

This code produces

```
ans =
trainr
```

Thus, during each epoch, a single vector is chosen randomly and presented to the network and weight and bias values are updated accordingly.

Next, supply the original vectors as input to the network, simulate the network, and finally convert its output vectors to class indices.

```
a = sim(net,p)
ac = vec2ind(a)
```

This yields

```
ac =
     1     2     1     2
```

We see that the network is trained to classify the input vectors into two groups, those near the origin, class 1, and those near (1,1), class 2.

It might be interesting to look at the final weights and biases. They are

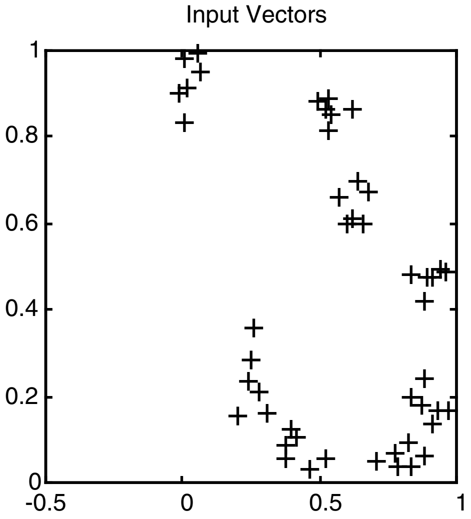
```
wts =
    0.8208    0.8263
    0.1348    0.1787
biases =
    5.3699
    5.5049
```

(You may get different answers if you run this problem, as a random seed is used to pick the order of the vectors presented to the network for training.) Note that the first vector (formed from the first row of the weight matrix) is near the input vectors close to (1,1), while the vector formed from the second row of the weight matrix is close to the input vectors near the origin. Thus, the network has been trained, just by exposing it to the inputs, to classify them.

During training each neuron in the layer closest to a group of input vectors adjusts its weight vector toward those input vectors. Eventually, if there are enough neurons, every cluster of similar input vectors has a neuron that outputs 1 when a vector in the cluster is presented, while outputting a 0 at all other times. Thus, the competitive network learns to categorize the input.

## Graphical Example

Competitive layers can be understood better when their weight vectors and input vectors are shown graphically. The diagram below shows 48 two-element input vectors represented as with ‘+’ markers.



The input vectors above appear to fall into clusters. You can use a competitive network of eight neurons to classify the vectors into such clusters.

Try `democ1` to see a dynamic example of competitive learning.

# Self-Organizing Maps

Self-organizing feature maps (SOFM) learn to classify input vectors according to how they are grouped in the input space. They differ from competitive layers in that neighboring neurons in the self-organizing map learn to recognize neighboring sections of the input space. Thus, self-organizing maps learn both the distribution (as do competitive layers) and topology of the input vectors they are trained on.

The neurons in the layer of an SOFM are arranged originally in physical positions according to a topology function. The functions `gridtop`, `hextop` or `randtop` can arrange the neurons in a grid, hexagonal, or random topology. Distances between neurons are calculated from their positions with a distance function. There are four distance functions, `dist`, `boxdist`, `linkdist` and `mandist`. Link distance is the most common. These topology and distance functions are described in detail later in this section.

Here a self-organizing feature map network identifies a winning neuron $i*$ using the same procedure as employed by a competitive layer. However, instead of updating only the winning neuron, all neurons within a certain neighborhood $N_{i*}(d)$ of the winning neuron are updated using the Kohonen rule. Specifically, we adjust all such neurons $i \in N_{i*}(d)$ as follows.

$$ {}_i\mathbf{w}(q) = {}_i\mathbf{w}(q-1) + \alpha(\mathbf{p}(q) - {}_i\mathbf{w}(q-1)) \text{ or} $$

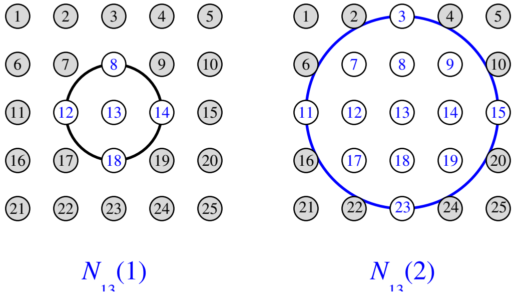
$$ {}_i\mathbf{w}(q) = (1-\alpha)\,{}_i\mathbf{w}(q-1) + \alpha\mathbf{p}(q) $$

Here the *neighborhood* $N_{i*}(d)$ contains the indices for all of the neurons that lie within a radius $d$ of the winning neuron $i*$.

$$ N_i(d) = \{j, d_{ij} \le d\} $$

Thus, when a vector $\mathbf{p}$ is presented, the weights of the winning neuron *and* its close neighbors move toward $\mathbf{p}$. Consequently, after many presentations, neighboring neurons will have learned vectors similar to each other.

To illustrate the concept of neighborhoods, consider the figure given below. The left diagram shows a two-dimensional neighborhood of radius $d = 1$ around neuron $13$. The right diagram shows a neighborhood of radius $d = 2$.



These neighborhoods could be written as

$$N_{13}(1) = \{8, 12, 13, 14, 18\} \text{ and}$$

$$N_{13}(2) = \{3, 7, 8, 9, 11, 12, 13, 14, 15, 17, 18, 19, 23\}$$

Note that the neurons in an SOFM do not have to be arranged in a two-dimensional pattern. You can use a one-dimensional arrangement, or even three or more dimensions. For a one-dimensional SOFM, a neuron has only two neighbors within a radius of 1 (or a single neighbor if the neuron is at the end of the line).You can also define distance in different ways, for instance, by using rectangular and hexagonal arrangements of neurons and neighborhoods. The performance of the network is not sensitive to the exact shape of the neighborhoods.

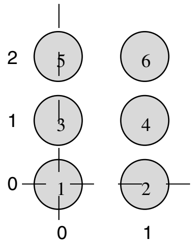
## Topologies (gridtop, hextop, randtop)

You can specify different topologies for the original neuron locations with the functions `gridtop`, `hextop` or `randtop`.

The `gridtop` topology starts with neurons in a rectangular grid similar to that shown in the previous figure. For example, suppose that you want a 2-by-3 array of six neurons You can get this with:

```
pos = gridtop(2,3)
pos =
      0      1      0      1      0      1
      0      0      1      1      2      2
```

Here neuron 1 has the position (0,0); neuron 2 has the position (1,0); neuron 3 had the position (0,1); etc.



gridtop(2,3)

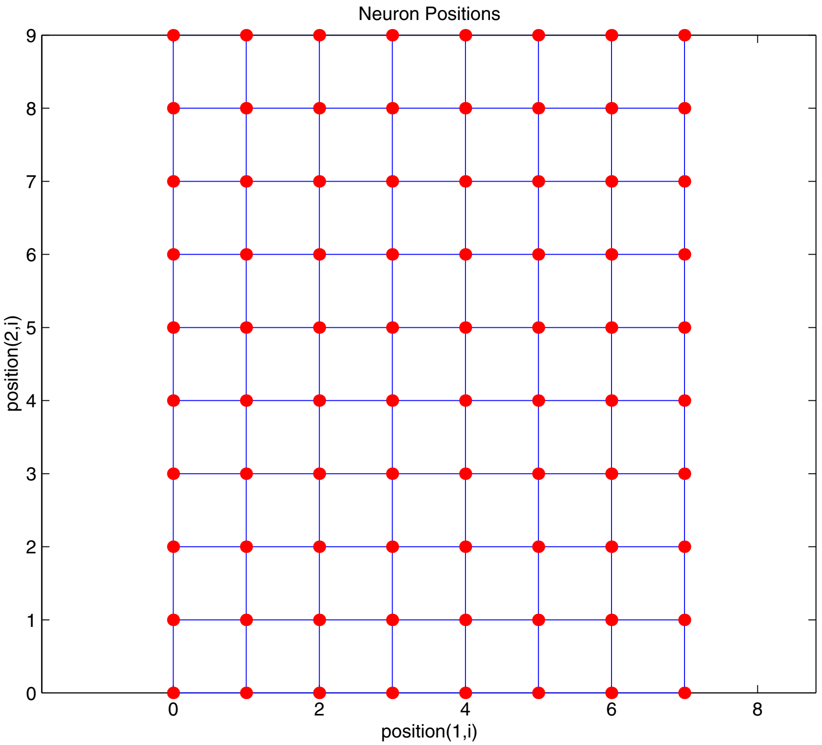
Note that had we asked for a gridtop with the arguments reversed we would have gotten a slightly different arrangement.

```
pos = gridtop(3,2)
pos =
      0     1     2     0     1     2
      0     0     0     1     1     1
```

An 8-by-10 set of neurons in a gridtop topology can be created and plotted with the code shown below

```
pos = gridtop(8,10);
plotsom(pos)
```

to give the following graph.



As shown, the neurons in the gridtop topology do indeed lie on a grid.

The `hextop` function creates a similar set of neurons, but they are in a hexagonal pattern. A 2-by-3 pattern of `hextop` neurons is generated as follows:

```
pos = hextop(2,3)
pos =
              0    1.0000    0.5000    1.5000         0    1.0000
              0         0    0.8660    0.8660    1.7321    1.7321
```

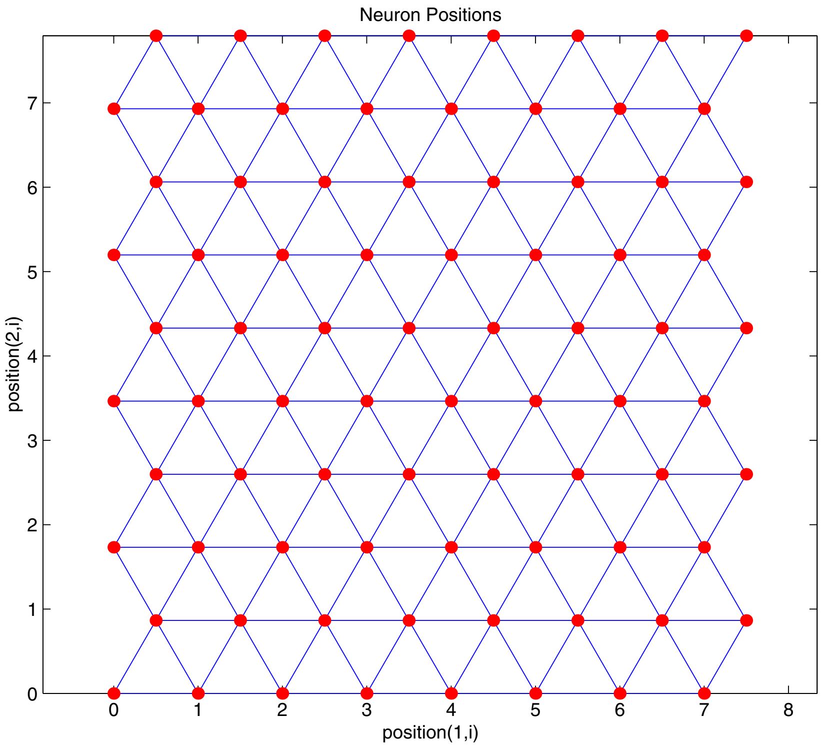
Note that `hextop` is the default pattern for SOFM networks generated with `newsom`.

An 8-by-10 set of neurons in a hextop topology can be created and plotted with the code shown below.

```
pos = hextop(8,10);
```

```
plotsom(pos)
```

to give the following graph.



Note the positions of the neurons in a hexagonal arrangement.

Finally, the randtop function creates neurons in an N dimensional random pattern. The following code generates a random pattern of neurons.

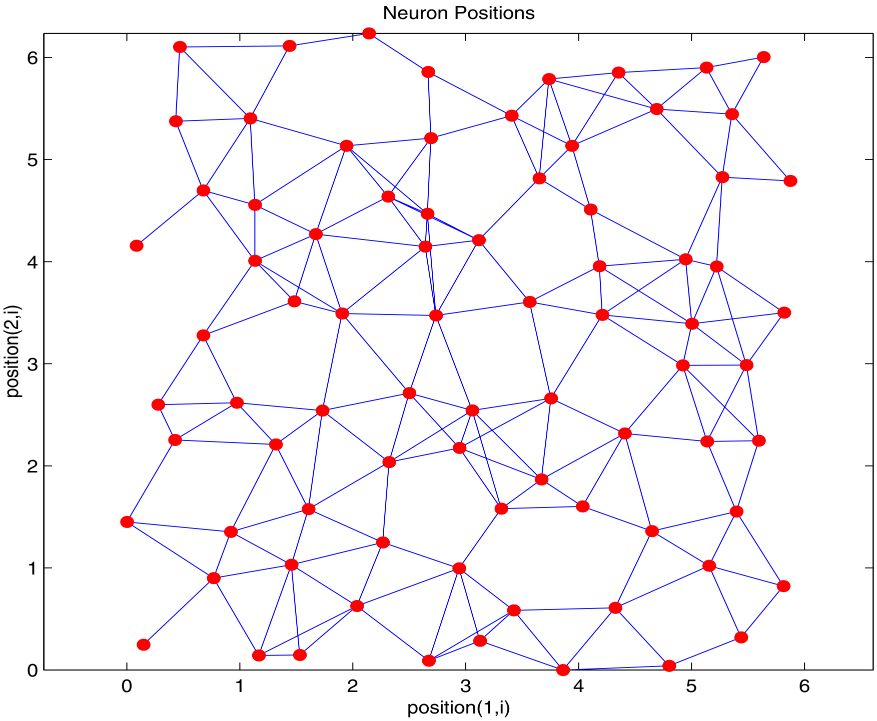
```
pos = randtop(2,3)
pos =
           0    0.7787    0.4390    1.0657    0.1470    0.9070
           0    0.1925    0.6476    0.9106    1.6490    1.4027
```

An 8-by-10 set of neurons in a randtop topology can be created and plotted with the code shown below

```
pos = randtop(8,10);
```

```
plotsom(pos)
```

to give the following graph.



For examples, see the help for these topology functions.

## Distance Funct. (dist, linkdist, mandist, boxdist)

In this toolbox, there are four distinct ways to calculate distances from a particular neuron to its neighbors. Each calculation method is implemented with a special function.

The `dist` function has been discussed before. It calculates the Euclidean distance from a *home* neuron to any other neuron. Suppose we have three neurons:

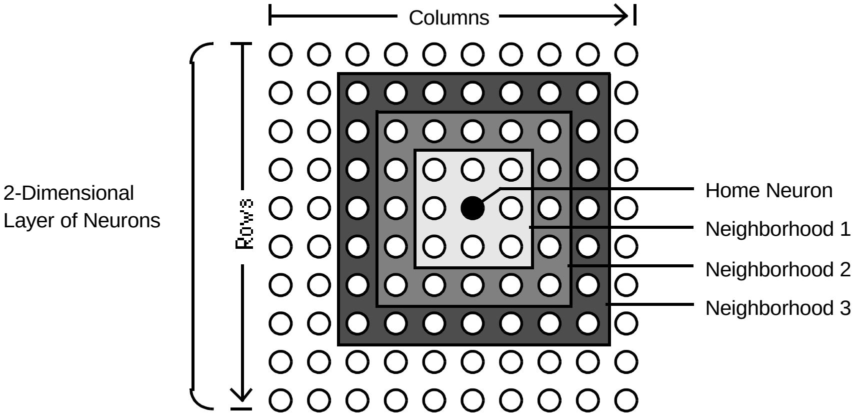
```
pos2 = [ 0 1 2; 0 1 2]
pos2 =
     0     1     2
     0     1     2
```

We find the distance from each neuron to the other with

```
D2 = dist(pos2)
D2 =
          0    1.4142    2.8284
     1.4142         0    1.4142
     2.8284    1.4142         0
```

Thus, the distance from neuron 1 to itself is 0, the distance from neuron 1 to neuron 2 is 1.414, etc. These are indeed the Euclidean distances as we know them.

The graph below shows a home neuron in a two-dimensional (gridtop) layer of neurons. The home neuron has neighborhoods of increasing diameter surrounding it. A neighborhood of diameter 1 includes the home neuron and its immediate neighbors. The neighborhood of diameter 2 includes the diameter 1 neurons and their immediate neighbors.



As for the dist function, all the neighborhoods for an S neuron layer map are represented by an S-by-S matrix of distances. The particular distances shown above (1 in the immediate neighborhood, 2 in neighborhood 2, etc.), are generated by the function boxdist. Suppose that we have six neurons in a gridtop configuration.

```
pos = gridtop(2,3)
pos =
     0     1     0     1     0     1
     0     0     1     1     2     2
```

Then the box distances are

```
d = boxdist(pos)
d =
     0     1     1     1     2     2
     1     0     1     1     2     2
     1     1     0     1     1     1
     1     1     1     0     1     1
     2     2     1     1     0     1
     2     2     1     1     1     0
```

The distance from neuron 1 to 2, 3, and 4 is just 1, for they are in the immediate neighborhood. The distance from neuron 1 to both 5 and 6 is 2. The distance from both 3 and 4 to all other neurons is just 1.

The *link distance* from one neuron is just the number of links, or steps, that must be taken to get to the neuron under consideration. Thus, if we calculate the distances from the same set of neurons with `linkdist` we get

```
dlink =
     0     1     1     2     2     3
     1     0     2     1     3     2
     1     2     0     1     1     2
     2     1     1     0     2     1
     2     3     1     2     0     1
     3     2     2     1     1     0
```

The Manhattan distance between two vectors **x** and **y** is calculated as

```
D = sum(abs(x-y))
```

Thus if we have

```
W1 = [ 1 2; 3 4; 5 6]
W1 =
     1     2
     3     4
     5     6
```

and

```
P1= [1;1]
P1 =
      1
      1
```

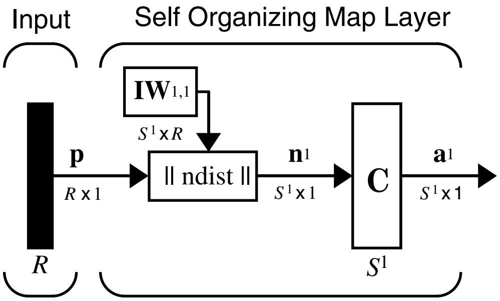
then we get for the distances

```
Z1 = mandist(W1,P1)
Z1 =
      1
      5
      9
```

The distances calculated with `mandist` do indeed follow the mathematical expression given above.

## Architecture

The architecture for this SOFM is shown below.



$$n_i^1 = - \| {}_i\mathbf{IW}^{1,1} - \mathbf{p} \|$$

$$\mathbf{a}^1 = \mathbf{compet}(\mathbf{n}^1)$$

This architecture is like that of a competitive network, except no bias is used here. The competitive transfer function produces a 1 for output element $\mathbf{a}^1{}_i$ corresponding to $i*$ ,the winning neuron. All other output elements in $\mathbf{a}^1$ are 0.

Now, however, as described above, neurons close to the winning neuron are updated along with the winning neuron. As described previously, one can chose from various topologies of neurons. Similarly, one can choose from various distance expressions to calculate neurons that are close to the winning neuron.

## Creating a Self Organizing MAP Neural Network (newsom)

You can create a new SOFM network with the function newsom. This function defines variables used in two phases of learning:

- Ordering-phase learning rate
- Ordering-phase steps
- Tuning-phase learning rate
- Tuning-phase neighborhood distance

These values are used for training and adapting.

Consider the following example.

Suppose that we want to create a network having input vectors with two elements that fall in the range 0 to 2 and 0 to 1 respectively. Further suppose that we want to have six neurons in a hexagonal 2-by-3 network. The code to obtain this network is

```
net = newsom([0 2; 0 1] , [2 3]);
```

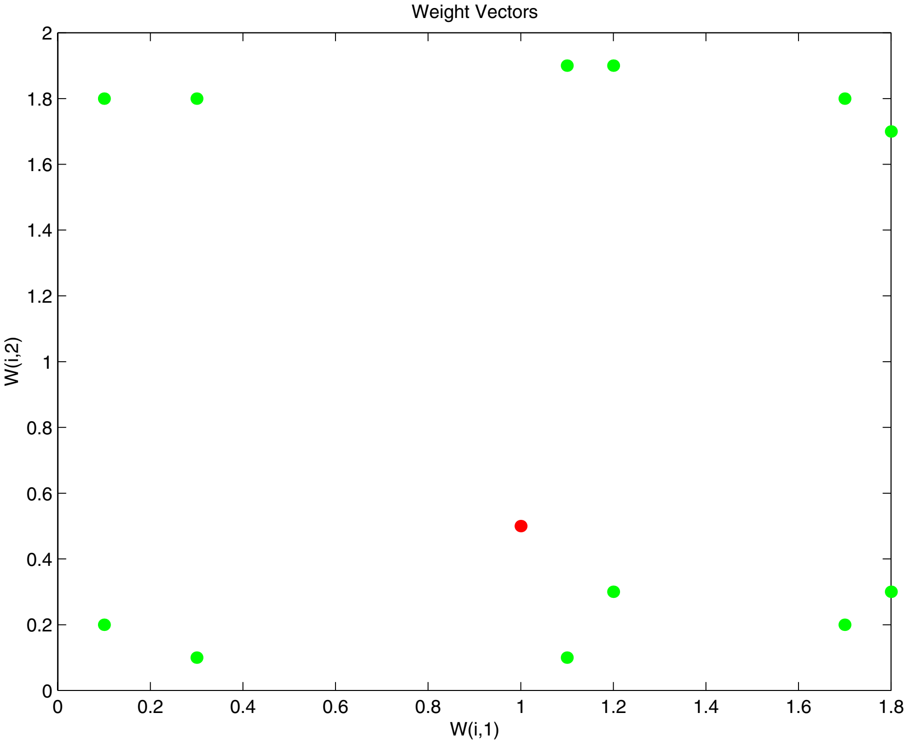
Suppose also that the vectors to train on are

```
P = [.1 .3 1.2 1.1 1.8 1.7 .1 .3 1.2 1.1 1.8 1.7;...
0.2 0.1 0.3 0.1 0.3 0.2 1.8 1.8 1.9 1.9 1.7 1.8]
```

We can plot all of this with

```
plot(P(1,:),P(2,:),'.g','markersize',20)
hold on
plotsom(net.iw{1,1},net.layers{1}.distances)
hold off
```

to give



The various training vectors are seen as fuzzy gray spots around the perimeter of this figure. The initialization for `newsom` is `midpoint`. Thus, the initial network neurons are all concentrated at the black spot at (1, 0.5).

When simulating a network, the negative distances between each neuron's weight vector and the input vector are calculated (`negdist`) to get the weighted inputs. The weighted inputs are also the net inputs (`netsum`). The net inputs compete (`compete`) so that only the neuron with the most positive net input will output a 1.

## Training (learnsom)

Learning in a self-organizing feature map occurs for one vector at a time, independent of whether the network is trained directly (`trainr`) or whether it is trained adaptively (`trains`). In either case, `learnsom` is the self-organizing map weight learning function.

First the network identifies the winning neuron. Then the weights of the winning neuron, and the other neurons in its neighborhood, are moved closer to the input vector at each learning step using the self-organizing map learning function `learnsom`. The winning neuron's weights are altered proportional to the learning rate. The weights of neurons in its neighborhood are altered proportional to half the learning rate. The learning rate and the neighborhood distance used to determine which neurons are in the winning neuron's neighborhood are altered during training through two phases.

### Phase 1: Ordering Phase

This phase lasts for the given number of steps. The neighborhood distance starts as the maximum distance between two neurons, and decreases to the tuning neighborhood distance. The learning rate starts at the ordering-phase learning rate and decreases until it reaches the tuning-phase learning rate. As the neighborhood distance and learning rate decrease over this phase, the neurons of the network typically order themselves in the input space with the same topology in which they are ordered physically.

### Phase 2: Tuning Phase

This phase lasts for the rest of training or adaption. The neighborhood distance stays at the tuning neighborhood distance, (which should include only close neighbors (i.e., typically 1.0). The learning rate continues to decrease from the tuning phase learning rate, but very slowly. The small neighborhood and slowly decreasing learning rate fine tune the network, while keeping the ordering learned in the previous phase stable. The number of epochs for the tuning part of training (or time steps for adaption) should be much larger than the number of steps in the ordering phase, because the tuning phase usually takes much longer.

Now let us take a look at some of the specific values commonly used in these networks.

Learning occurs according to the learnsom learning parameter, shown here with its default value.

| | | |
|---|---|---|
| LP.order_lr | 0.9 | Ordering-phase learning rate. |
| LP.order_steps | 1000 | Ordering-phase steps. |
| LP.tune_lr | 0.02 | Tuning-phase learning rate. |
| LP.tune_nd | 1 | Tuning-phase neighborhood distance. |

learnsom calculates the weight change dW for a given neuron from the neuron's input P, activation A2, and learning rate LR:

```
dw =  lr*a2*(p'-w)
```

where the activation A2 is found from the layer output A and neuron distances D and the current neighborhood size ND:

```
a2(i,q) = 1,   if a(i,q) = 1
        = 0.5, if a(j,q) = 1 and D(i,j) <= nd
        = 0,   otherwise
```

The learning rate LR and neighborhood size NS are altered through two phases: an ordering phase, and a tuning phase.

The ordering phase lasts as many steps as LP.order_steps. During this phase, LR is adjusted from LP.order_lr down to LP.tune_lr, and ND is adjusted from the maximum neuron distance down to 1. It is during this phase that neuron weights are expected to order themselves in the input space consistent with the associated neuron positions.

During the tuning phase LR decreases slowly from LP.tune_lr and ND is always set to LP.tune_nd. During this phase, the weights are expected to spread out relatively evenly over the input space while retaining their topological order found during the ordering phase.

Thus, the neuron’s weight vectors initially take large steps all together toward the area of input space where input vectors are occurring. Then as the neighborhood size decreases to 1, the map tends to order itself topologically over the presented input vectors. Once the neighborhood size is 1, the network should be fairly well ordered and the learning rate is slowly decreased over a longer period to give the neurons time to spread out evenly across the input vectors.

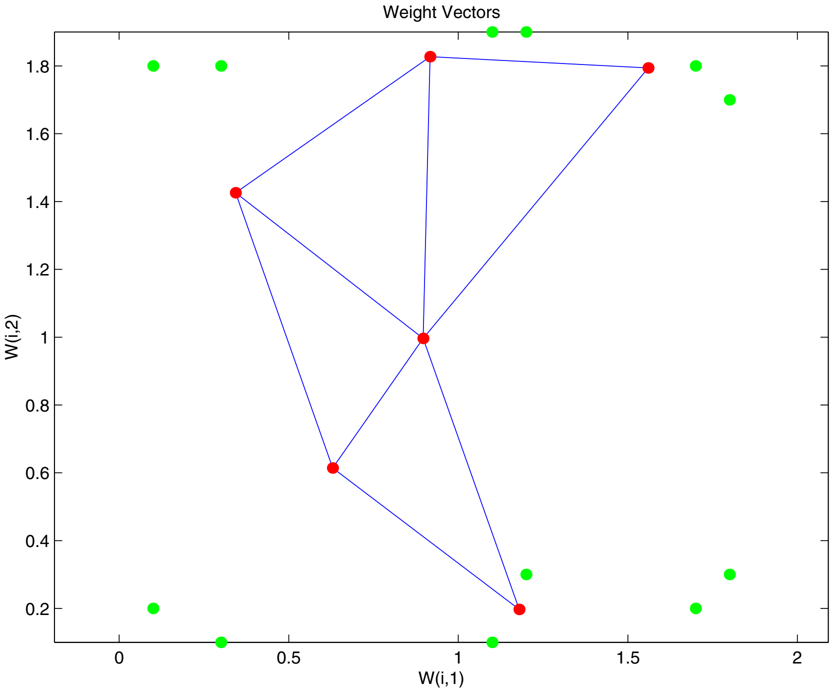
As with competitive layers, the neurons of a self-organizing map will order themselves with approximately equal distances between them if input vectors appear with even probability throughout a section of the input space. Also, if input vectors occur with varying frequency throughout the input space, the feature map layer tends to allocate neurons to an area in proportion to the frequency of input vectors there.

Thus, feature maps, while learning to categorize their input, also learn both the topology and distribution of their input.

We can train the network for 1000 epochs with

```
net.trainParam.epochs = 1000;
net = train(net,P);
```

This training produces the following plot.



We can see that the neurons have started to move toward the various training groups. Additional training is required to get the neurons closer to the various groups.

As noted previously, self-organizing maps differ from conventional competitive learning in terms of which neurons get their weights updated. Instead of updating only the winner, feature maps update the weights of the winner and its neighbors. The result is that neighboring neurons tend to have similar weight vectors and to be responsive to similar input vectors.

## Examples

Two examples are described briefly below. You might try the demonstration scripts `demosm1` and `demosm2` to see similar examples.

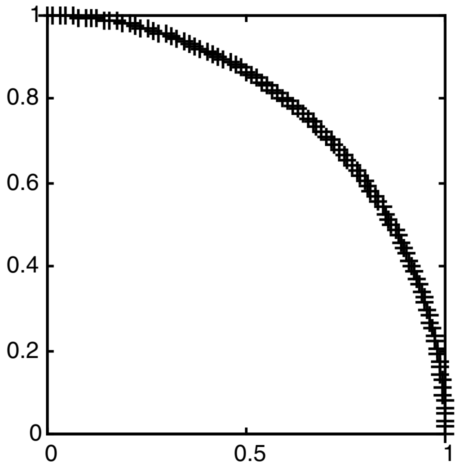
## One-Dimensional Self-Organizing Map

Consider 100 two-element unit input vectors spread evenly between 0° and 90°.

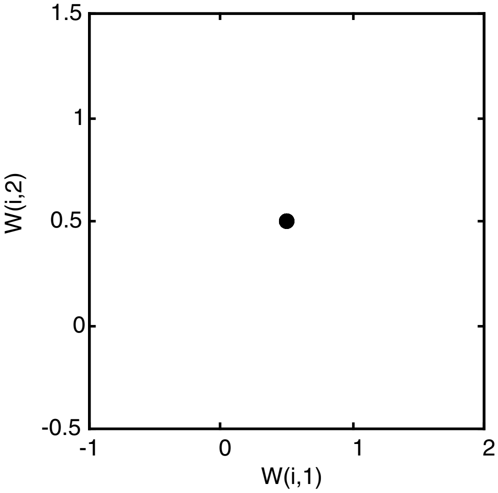
```
angles = 0:0.5*pi/99:0.5*pi;
```

Here is a plot of the data.

```
P = [sin(angles); cos(angles)];
```

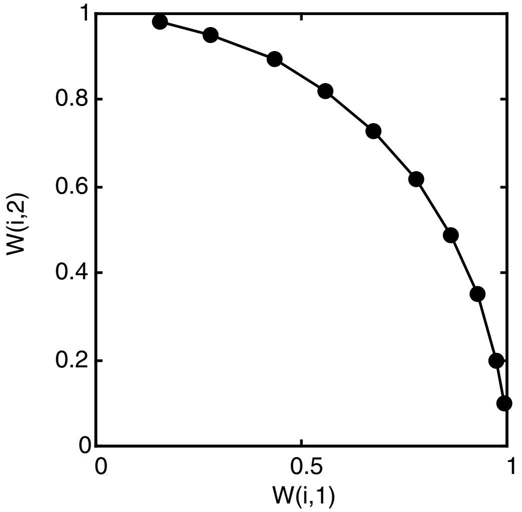


We define a a self-organizing map as a one-dimensional layer of 10 neurons. This map is to be trained on these input vectors shown above. Originally these neurons will be at the center of the figure.



Of course, since all the weight vectors start in the middle of the input vector space, all you see now is a single circle.

As training starts the weight vectors move together toward the input vectors. They also become ordered as the neighborhood size decreases. Finally the layer adjusts its weights so that each neuron responds strongly to a region of the input space occupied by input vectors. The placement of neighboring neuron weight vectors also reflects the topology of the input vectors.



Note that self-organizing maps are trained with input vectors in a random order, so starting with the same initial vectors does not guarantee identical training results.

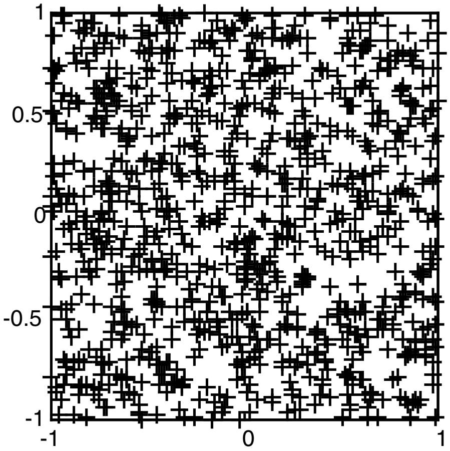
### Two-Dimensional Self-Organizing Map

This example shows how a two-dimensional self-organizing map can be trained.

First some random input data is created with the following code.

```
P = rands(2,1000);
```

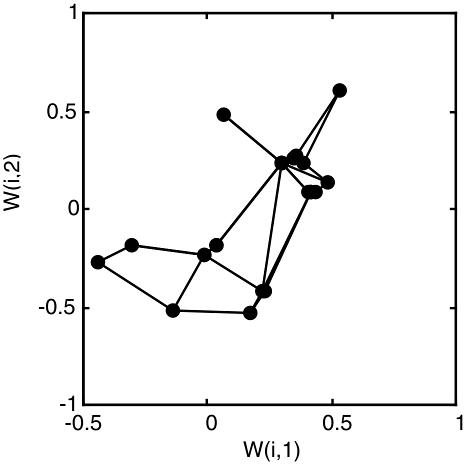
Here is a plot of these 1000 input vectors.



A two-dimensional map of 30 neurons is used to classify these input vectors. The two-dimensional map is five neurons by six neurons, with distances calculated according to the Manhattan distance neighborhood function `mandist`.

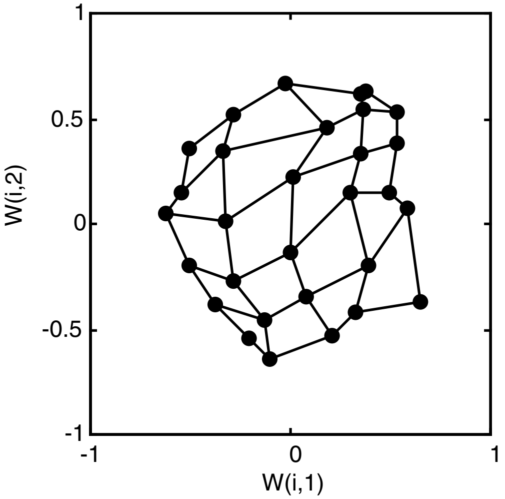
The map is then trained for 5000 presentation cycles, with displays every 20 cycles.

Here is what the self-organizing map looks like after 40 cycles.



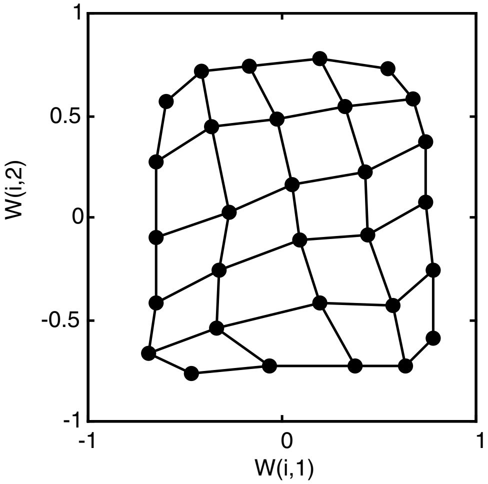
The weight vectors, shown with circles, are almost randomly placed. However, even after only 40 presentation cycles, neighboring neurons, connected by lines, have weight vectors close together.

Here is the map after 120 cycles.

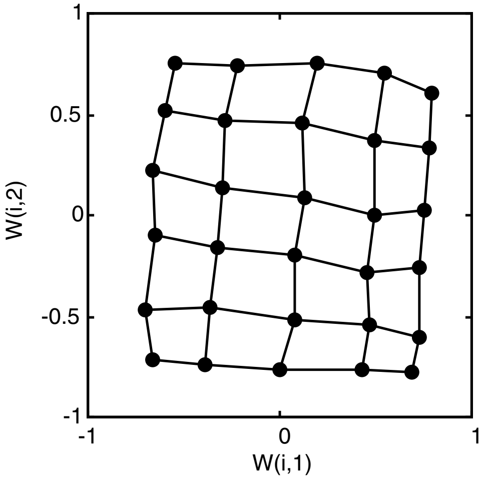


After 120 cycles, the map has begun to organize itself according to the topology of the input space which constrains input vectors.

The following plot, after 500 cycles, shows the map is more evenly distributed across the input space.



Finally, after 5000 cycles, the map is rather evenly spread across the input space. In addition, the neurons are very evenly spaced reflecting the even distribution of input vectors in this problem.



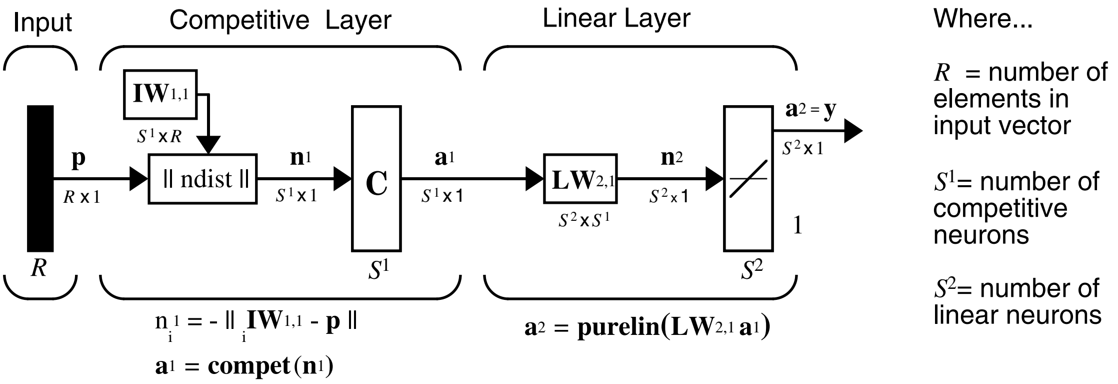
Thus a two-dimensional self-organizing map has learned the topology of its inputs' space.

It is important to note that while a self-organizing map does not take long to organize itself so that neighboring neurons recognize similar inputs, it can take a long time for the map to finally arrange itself according to the distribution of input vectors.

# Learning Vector Quantization Networks

## Architecture

The LVQ network architecture is shown below.



An LVQ network has a first competitive layer and a second linear layer. The competitive layer learns to classify input vectors in much the same way as the competitive layers of “Self-Organizing and Learn. Vector Quant. Nets” described in this chapter. The linear layer transforms the competitive layer’s classes into target classifications defined by the user. We refer to the classes learned by the competitive layer as *subclasses* and the classes of the linear layer as *target classes*.

Both the competitive and linear layers have one neuron per (sub or target) class. Thus, the competitive layer can learn up to $S^1$ subclasses. These, in turn, are combined by the linear layer to form $S^2$ target classes. ($S^1$ is always larger than $S^2$.)

For example, suppose neurons 1, 2, and 3 in the competitive layer all learn subclasses of the input space that belongs to the linear layer target class No. 2. Then competitive neurons 1, 2, and 3, will have $\mathbf{LW}^{2,1}$ weights of 1.0 to neuron $\mathbf{n}^2$ in the linear layer, and weights of 0 to all other linear neurons. Thus, the linear neuron produces a 1 if any of the three competitive neurons (1,2, and 3) win the competition and output a 1. This is how the subclasses of the competitive layer are combined into target classes in the linear layer.

In short, a *1* in the i$^{th}$ row of $\mathbf{a}^1$ (the rest to the elements of $\mathbf{a}^1$ will be zero) effectively picks the i$^{th}$ column of $\mathbf{LW}^{2,1}$ as the network output. Each such column contains a single 1, corresponding to a specific class. Thus, subclass 1s from layer 1 get put into various classes, by the $\mathbf{LW}^{2,1}\mathbf{a}^1$ multiplication in layer 2.

We know ahead of time what fraction of the layer 1 neurons should be classified into the various class outputs of layer 2, so we can specify the elements of $\mathbf{LW}^{2,1}$ at the start. However, we have to go through a training procedure to get the first layer to produce the correct subclass output for each vector of the training set. We discuss this training shortly. First consider how to create the original network.

## Creating an LVQ Network (newlvq)

An LVQ network can be created with the function `newlvq`

```
net = newlvq(PR,S1,PC,LR,LF)
```

where:

- PR is an *R*-by-2 matrix of minimum and maximum values for *R* input elements.
- $S^1$ is the number of first layer hidden neurons.
- PC is an $S^2$ element vector of typical class percentages.
- LR is the learning rate (default 0.01).
- LF is the learning function (default is `learnlv1`).

Suppose we have 10 input vectors. We create a network that assigns each of these input vectors to one of four subclasses. Thus, we have four neurons in the first competitive layer. These subclasses are then assigned to one of two output classes by the two neurons in layer 2. The input vectors and targets are specified by

```
P = [-3 -2 -2  0  0  0  0 +2 + 2 +3; ...
0 +1 -1 +2 +1 -1 -2 +1 -1  0]
```

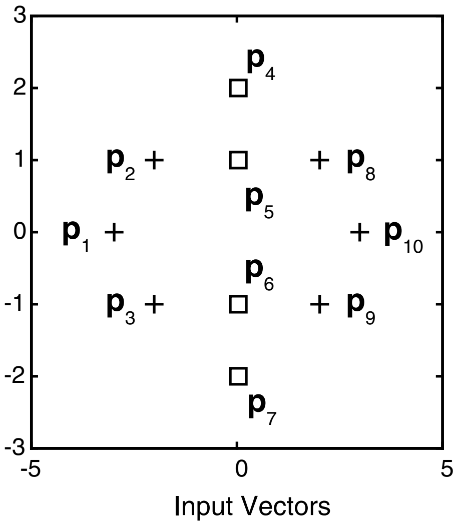
and

```
Tc = [1 1 1 2 2 2 2 1 1 1];
```

It may help to show the details of what we get from these two lines of code.

```
P =
    -3    -2    -2     0     0     0     0     2     2     3
     0     1    -1     2     1    -1    -2     1    -1     0
Tc =
     1     1     1     2     2     2     2     1     1     1
```

A plot of the input vectors follows.



As you can see, there are four subclasses of input vectors. We want a network that classifies $\mathbf{p}_1$, $\mathbf{p}_2$, $\mathbf{p}_3$, $\mathbf{p}_8$, $\mathbf{p}_9$, and $\mathbf{p}_{10}$ to produce an output of 1, and that classifies vectors $\mathbf{p}_4$, $\mathbf{p}_5$, $\mathbf{p}_6$ and $\mathbf{p}_7$ to produce an output of 2. Note that this problem is nonlinearly separable, and so cannot be solved by a perceptron, but an LVQ network has no difficulty.

Next we convert the `Tc` matrix to target vectors.

```
T = ind2vec(Tc)
```

This gives a sparse matrix `T` that can be displayed in full with

```
targets = full(T)
```

which gives

```
targets =
     1     1     1     0     0     0     0     1     1     1
     0     0     0     1     1     1     1     0     0     0
```

This looks right. It says, for instance, that if we have the first column of `P` as input, we should get the first column of `targets` as an output; and that output says the input falls in class 1, which is correct. Now we are ready to call `newlvq`.

We call `newlvq` with the proper arguments so that it creates a network with four neurons in the first layer and two neurons in the second layer. The first-layer weights are initialized to the center of the input ranges with the function `midpoint`. The second-layer weights have 60% (6 of the 10 in `Tc` above) of its columns with a 1 in the first row, (corresponding to class 1), and 40% of its columns will have a 1 in the second row (corresponding to class 2).

```
net = newlvq(minmax(P),4,[.6 .4], 0,1);
```

We can check to see the initial values of the first-layer weight matrix.

```
net.IW{1,1}
ans =
     0     0
     0     0
     0     0
     0     0
```

These zero weights are indeed the values at the midpoint of the range (-3 to +3) of the inputs, as we would expect when using `midpoint` for initialization.

We can look at the second-layer weights with

```
net.LW{2,1}
ans =
     1     1     0     0
     0     0     1     1
```

This makes sense too. It says that if the competitive layer produces a 1 as the first or second element. The input vector is classified as class 1; otherwise it is a class 2.

You may notice that the first two competitive neurons are connected to the first linear neuron (with weights of 1), while the second two competitive neurons are connected to the second linear neuron. All other weights between the competitive neurons and linear neurons have values of 0. Thus, each of the two target classes (the linear neurons) is, in fact, the union of two subclasses (the competitive neurons).

We can simulate the network with `sim`. We use the original `P` matrix as input just to see what we get.

```
Y = sim(net,P);
Y = vec2ind(Yb4t)
Y =
     1     1     1     1     1     1     1     1     1     1
```

The network classifies all inputs into class 1. Since tis not what we want, we have to train the network (adjusting the weights of layer 1 only), before we can expect a good result. First we discuss two LVQ learning rules, and then we look at the training process.

## LVQ1 Learning Rule (learnlv1)

LVQ learning in the competitive layer is based on a set of input/target pairs.

$$\{\mathbf{p}_1, \mathbf{t}_1\}, \{\mathbf{p}_2, \mathbf{t}_2\}, \ldots, \{\mathbf{p}_Q, \mathbf{t}_Q\}$$

Each target vector has a single 1. The rest of its elements are 0. The 1 tells the proper classification of the associated input. For instance, consider the following training pair.

$$\left\{\mathbf{p}_1 = \begin{bmatrix} 2 \\ -1 \\ 0 \end{bmatrix}, \mathbf{t}_1 = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}\right\}$$

Here we have input vectors of three elements, and each input vector is to be assigned to one of four classes. The network is to be trained so that it classifies the input vector shown above into the third of four classes.

To train the network, an input vector $\mathbf{p}$ is presented, and the distance from $\mathbf{p}$ to each row of the input weight matrix $\mathbf{IW}^{1,1}$ is computed with the function `ndist`. The hidden neurons of layer 1 compete. Suppose that the $i^{\text{th}}$ element of $\mathbf{n}^1$ is most positive, and neuron $i^*$ wins the competition. Then the competitive transfer function produces a 1 as the $i^{*\text{th}}$ element of $\mathbf{a}^1$. All other elements of $\mathbf{a}^1$ are 0.

When $\mathbf{a}^1$ is multiplied by the layer 2 weights $\mathbf{LW}^{2,1}$, the single 1 in $\mathbf{a}^1$ selects the class, $k^*$ associated with the input. Thus, the network has assigned the input vector $\mathbf{p}$ to class $k^*$ and $a^2_{k^*}$ will be 1. Of course, this assignment may be a good one or a bad one, for $t_{k^*}$ may be 1 or 0, depending on whether the input belonged to class $k^*$ or not.

We adjust the $i^{*\text{th}}$ row of $\mathbf{IW}^{1,1}$ in such a way as to move this row closer to the input vector $\mathbf{p}$ if the assignment is correct, and to move the row away from $\mathbf{p}$ if the assignment is incorrect. So if $\mathbf{p}$ is classified correctly,

$$(a^2_{k^*} = t_{k^*} = 1)$$

we compute the new value of the $i^{*\text{th}}$ row of $\mathbf{IW}^{1,1}$ as:

$${}_{i^*}\mathbf{IW}^{1,1}(q) = {}_{i^*}\mathbf{IW}^{1,1}(q-1) + \alpha(\mathbf{p}(q) - {}_{i^*}\mathbf{IW}^{1,1}(q-1)) \,.$$

On the other hand, if $\mathbf{p}$ is classified incorrectly,

$$(a^2_{k^*} = 1 \neq t_{k^*} = 0) \,,$$

we compute the new value of the i$^{*\text{th}}$ row of $\mathbf{IW}^{1,1}$ as:

$${}_{i^*}\mathbf{IW}^{1,1}(q) = {}_{i^*}\mathbf{IW}^{1,1}(q-1) - \alpha(\mathbf{p}(q) - {}_{i^*}\mathbf{IW}^{1,1}(q-1))$$

These corrections to the i$^{*\text{th}}$ row of $\mathbf{IW}^{1,1}$ can be made automatically without affecting other rows of $\mathbf{IW}^{1,1}$ by backpropagating the output errors back to layer 1.

Such corrections move the hidden neuron towards vectors that fall into the class for which it forms a subclass, and away from vectors that fall into other classes.

The learning function that implements these changes in the layer 1 weights in LVQ networks is `learnlv1`. It can be applied during training.

## Training

Next we need to train the network to obtain first-layer weights that lead to the correct classification of input vectors. We do this with `train` as shown below. First set the training epochs to 150. Then, use `train`.

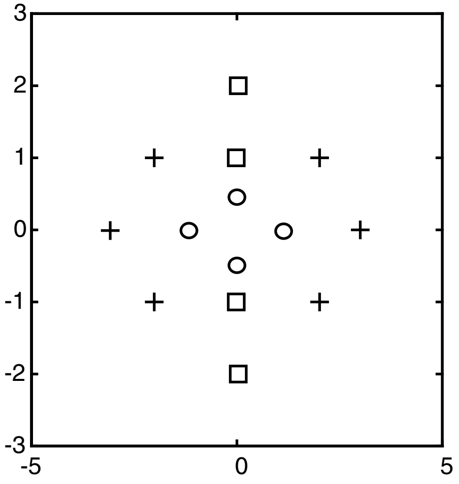
```
net.trainParam.epochs = 150;
```

```
net = train(net,P,T);
```

Now check on the first-layer weights.

```
net.IW{1,1}
ans =
    1.0927    0.0051
   -1.1028   -0.1288
         0   -0.5168
         0    0.3710
```

The following plot shows that these weights have moved toward their respective classification groups.



Weights (circles) after training

To check to see that these weights do indeed lead to the correct classification, take the matrix P as input and simulate the network. Then see what classifications are produced by the network.

```
Y = sim(net,P)
Yc = vec2ind(Y)
```

This gives

```
Yc =
     1     1     1     2     2     2     2     1     1     1
```

which is what we expected. As a last check, try an input close to a vector that was used in training.

```
pchk1 = [0; 0.5];
Y = sim(net,pchk1);
Yc1 = vec2ind(Y)
```

This gives

```
Yc1 =
     2
```

This looks right, for `pchk1` is close to other vectors classified as 2. Similarly,

```
pchk2 = [1; 0];
Y = sim(net,pchk2);
Yc2 = vec2ind(Y)
```

gives

```
Yc2 =
     1
```

This looks right too, for `pchk2` is close to other vectors classified as 1.

You might want to try the demonstration program `demolvq1`. It follows the discussion of training given above.

## Supplemental LVQ2.1 Learning Rule (learnlv2)

The following learning rule is one that might be applied *after* first applying LVQ1. It may improve the result of the first learning. This particular version of LVQ2 (referred to as LVQ2.1 in the literature **[Koho97]**) is embodied in the function `learnlv2`. Note again that LVQ2.1 is to be used only after LVQ1 has been applied

Learning here is similar to that in `learnlv1` except now two vectors of layer 1 that are closest to the input vector may be updated providing that one belongs to the correct class and one belongs to a wrong class and further providing that the input falls into a “window” near the midplane of the two vectors.

The window is defined by

$$\min\left(\frac{d_i}{d_j},\frac{d_j}{d_i}\right) > s \quad \text{where} \quad s \equiv \frac{1-w}{1+w} ,$$

(where $d_i$ and $d_j$ are the Euclidean distances of p from ${}_{i*}\mathbf{IW}^{1,1}$ and ${}_{j*}\mathbf{IW}^{1,1}$ respectively). We take a value for $w$ in the range 0.2 to 0.3. If we pick, for instance, 0.25, then $s = 0.6$. This means that if the minimum of the two distance ratios is greater than 0.6, we adjust the two vectors. i.e., if the input is "near" the midplane, adjust the two vectors providing also that the input vector **p** and ${}_{j*}\mathbf{IW}^{1,1}$ belong to the same class, and **p** and ${}_{i*}\mathbf{IW}^{1,1}$ do not belong in the same class.

The adjustments made are

$${}_{i*}\mathbf{IW}^{1,1}(q) = {}_{i*}\mathbf{IW}^{1,1}(q-1) - \alpha(\mathbf{p}(q) - {}_{i*}\mathbf{IW}^{1,1}(q-1)) \quad \text{and}$$

$${}_{j*}\mathbf{IW}^{1,1}(q) = {}_{j*}\mathbf{IW}^{1,1}(q-1) + \alpha(\mathbf{p}(q) - {}_{j*}\mathbf{IW}^{1,1}(q-1)) .$$

Thus, given two vector closest to the input, as long as one belongs to the wrong class and the other to the correct class, and as long as the input falls in a midplane window, the two vectors will be adjusted. Such a procedure allows a vector that is just barely classified correctly with LVQ1 to be moved even closer to the input, so the results are more robust.

# Summary

## Self-Organizing Maps

A competitive network learns to categorize the input vectors presented to it. If a neural network only needs to learn to categorize its input vectors, then a competitive network will do. Competitive networks also learn the distribution of inputs by dedicating more neurons to classifying parts of the input space with higher densities of input.

A self-organizing map learns to categorize input vectors. It also learns the distribution of input vectors. Feature maps allocate more neurons to recognize parts of the input space where many input vectors occur and allocate fewer neurons to parts of the input space where few input vectors occur.

Self-organizing maps also learn the topology of their input vectors. Neurons next to each other in the network learn to respond to similar vectors. The layer of neurons can be imagined to be a rubber net that is stretched over the regions in the input space where input vectors occur.

Self-organizing maps allow neurons that are neighbors to the winning neuron to output values. Thus the transition of output vectors is much smoother than that obtained with competitive layers, where only one neuron has an output at a time.

## Learning Vector Quantizaton Networks

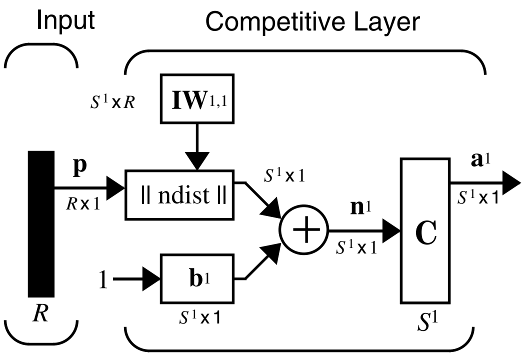
LVQ networks classify input vectors into target classes by using a competitive layer to find subclasses of input vectors, and then combining them into the target classes.

Unlike perceptrons, LVQ networks can classify any set of input vectors, not just linearly separable sets of input vectors. The only requirement is that the competitive layer must have enough neurons, and each class must be assigned enough competitive neurons.

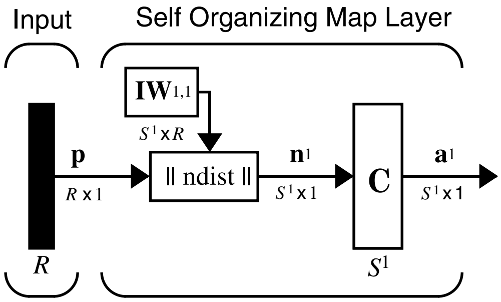
To ensure that each class is assigned an appropriate amount of competitive neurons, it is important that the target vectors used to initialize the LVQ network have the same distributions of targets as the training data the network is trained on. If this is done, target classes with more vectors will be the union of more subclasses.

# Figures

## Competitive Network Architecture



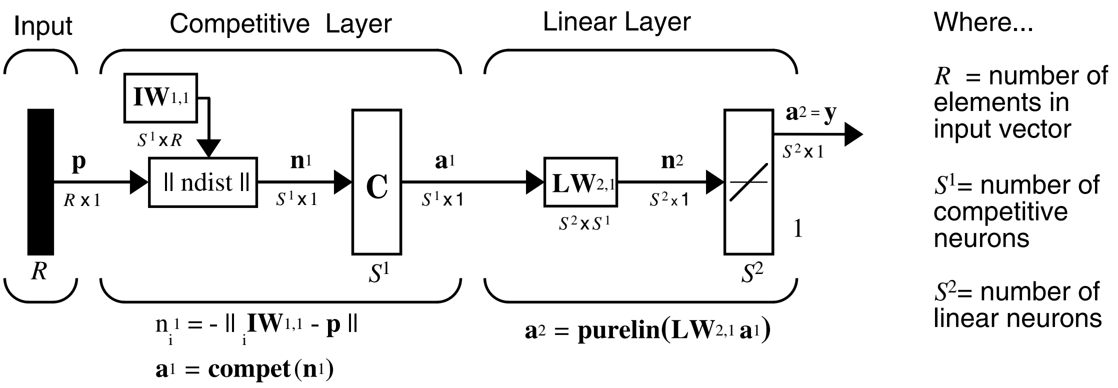
## Self Organizing Feature Map Architecture



$$n_i^1 = - \| {}_i\mathrm{IW}^{1,1} - p \|$$

$$\mathbf{a}^1 = \mathbf{compet}(\mathbf{n}^1)$$

**LVQ Architecture**


Input
Competitive Layer
Linear Layer
Where...
IW1,1
S1 x R
p
R x 1
|| ndist ||
n1
S1 x 1
C
a1
S1 x 1
LW2,1
S2 x S1
n2
S2 x 1
a2 = y
S2 x 1
1
R
S1
S2
R = number of elements in input vector
S1= number of competitive neurons
S2= number of linear neurons
$n^1_i = - \| {}_i\mathbf{IW}^{1,1} - \mathbf{p} \|$
$\mathbf{a}^1 = \mathbf{compet}(\mathbf{n}^1)$
$\mathbf{a}^2 = \mathbf{purelin}(\mathbf{LW}^{2,1}\mathbf{a}^1)$

## New Functions

This chapter introduced the following functions.

| Function | Description |
|---|---|
| `newc` | Create a competitive layer. |
| `learnk` | Kohonen learning rule. |
| `newsom` | Create a self-organizing map. |
| `learncon` | Conscience bias learning function. |
| `boxdist` | Distance between two position vectors. |
| `dist` | Euclidean distance weight function. |
| `linkdist` | Link distance function. |
| `mandist` | Manhattan distance weight function. |
| `gridtop` | Gridtop layer topology function. |
| `hextop` | Hexagonal layer topology function. |
| `randtop` | Random layer topology function. |

| Function | Description |
|---|---|
| `newlvq` | Create a learning vector quantization network. |
| `learnlv1` | LVQ1 weight learning function. |
| `learnlv2` | LVQ2 weight learning function. |

9

# Recurrent Networks

# Introduction

Recurrent networks is a topic of considerable interest. This chapter covers two recurrent networks: Elman, and Hopfield networks.

Elman networks are two-layer backpropagation networks, with the addition of a feedback connection from the output of the hidden layer to its input. This feedback path allows Elman networks to learn to recognize and generate temporal patterns, as well as spatial patterns. The best paper on the Elman network is:

Elman, J. L., "Finding structure in time," *Cognitive Science*, vol. 14, 1990, pp. 179-211.

The Hopfield network is used to store one or more stable target vectors. These stable vectors can be viewed as memories that the network recalls when provided with similar vectors that act as a cue to the network memory. You may want to pursue a basic paper in this field:

Li, J., A. N. Michel, and W. Porod, "Analysis and synthesis of a class of neural networks: linear systems operating on a closed hypercube," *IEEE Transactions on Circuits and Systems*, vol. 36, no. 11, November 1989, pp. 1405-1422.

## Important Recurrent Network Functions

Elman networks can be created with the function `newelm`.

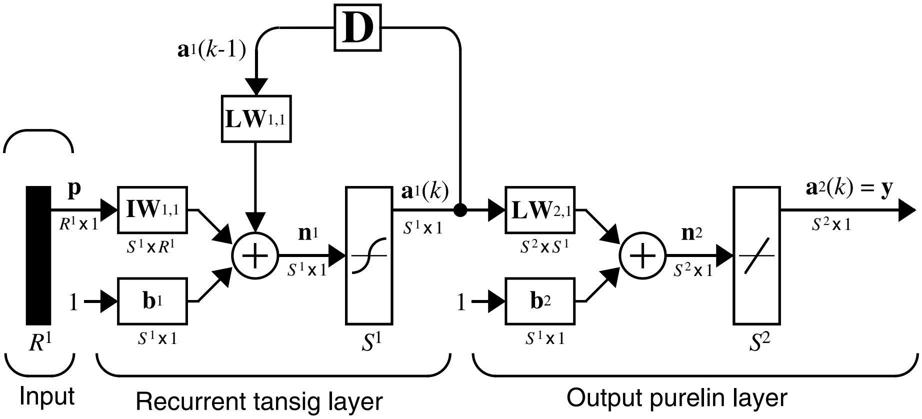
Hopfield networks can be created with the function `newhop`.

Type `help elman` or `help hopfield` to see a list of functions and demonstrations related to either of these networks.

# Elman Networks

## Architecture

The Elman network commonly is a two-layer network with feedback from the first-layer output to the first layer input. This recurrent connection allows the Elman network to both detect and generate time-varying patterns. A two-layer Elman network is shown below.



$$\mathbf{a}^1(k) = \mathbf{tansig}(\mathbf{IW}^{1,1}\mathbf{p} + \mathbf{LW}^{1,1}\mathbf{a}^1(k-1) + \mathbf{b}^1) \qquad \mathbf{a}^2(k) = \mathbf{purelin}(\mathbf{LW}^{2,1}\mathbf{a}^1(k) + \mathbf{b}^2)$$

The Elman network has `tansig` neurons in its hidden (recurrent) layer, and `purelin` neurons in its output layer. This combination is special in that two-layer networks with these transfer functions can approximate any function (with a finite number of discontinuities) with arbitrary accuracy. The only requirement is that the hidden layer must have enough neurons. More hidden neurons are needed as the function being fit increases in complexity.

Note that the Elman network differs from conventional two-layer networks in that the first layer has a recurrent connection. The delay in this connection stores values from the previous time step, which can be used in the current time step.

Thus, even if two Elman networks, with the same weights and biases, are given identical inputs at a given time step, their outputs can be different due to different feedback states.

Because the network can store information for future reference, it is able to learn temporal patterns as well as spatial patterns. The Elman network can be trained to respond to, and to generate, both kinds of patterns.

## Creating an Elman Network (newelm)

An Elman network with two or more layers can be created with the function newelm. The hidden layers commonly have tansig transfer functions, so that is the default for newelm. As shown in the architecture diagram, purelin is commonly the output-layer transfer function.

The default backpropagation training function is trainbfg. One might use trainlm, but it tends to proceed so rapidly that it does not necessarily do well in the Elman network. The backprop weight/bias learning function default is learngdm, and the default performance function is mse.

When the network is created, each layer's weights and biases are initialized with the Nguyen-Widrow layer initialization method implemented in the function initnw.

Now consider an example. Suppose that we have a sequence of single-element input vectors in the range from 0 to 1. Suppose further that we want to have five hidden-layer tansig neurons and a single logsig output layer. The following code creates the desired network.

```
net = newelm([0 1],[5 1],{'tansig','logsig'});
```

### Simulation

Suppose that we want to find the response of this network to an input sequence of eight digits that are either 0 or 1.

```
P = round(rand(1,8))
P =
     0     1     0     1     1     0     0     0
```

Recall that a sequence to be presented to a network is to be in cell array form. We can convert P to this form with

```
Pseq = con2seq(P)
Pseq =
    [0]    [1]    [0]    [1]    [1]    [0]    [0]    [0]
```

Now we can find the output of the network with the function sim.

```
Y = sim(net,Pseq)
Y =
Columns 1 through 5
   [1.9875e-04]   [0.1146]   [5.0677e-05]   [0.0017]   [0.9544]
Columns 6 through 8
    [0.0014]    [5.7241e-05]    [3.6413e-05]
```

We convert this back to concurrent form with

```
z = seq2con(Y);
```

and can finally display the output in concurrent form with

```
z{1,1}
ans =
  Columns 1 through 7
   0.0002    0.1146    0.0001    0.0017    0.9544    0.0014    0.0001
Column 8
    0.0000
```

Thus, once the network is created and the input specified, one need only call `sim`.

## Training an Elman Network

Elman networks can be trained with either of two functions, `train` or `adapt`.

When using the function `train` to train an Elman network the following occurs.

At each epoch:

1. The entire input sequence is presented to the network, and its outputs are calculated and compared with the target sequence to generate an error sequence.

2. For each time step, the error is backpropagated to find *gradients* of errors for each weight and bias. This *gradient* is actually an approximation since the contributions of weights and biases to errors via the delayed recurrent connection are ignored.

3. This gradient is then used to update the weights with the backprop training function chosen by the user. The function `traingdx` is recommended.

When using the function adapt to train an Elman network, the following occurs.

At each time step:

1 Input vectors are presented to the network, and it generates an error.

2 The error is backpropagated to find gradients of errors for each weight and bias. This gradient is actually an approximation since the contributions of weights and biases to the error, via the delayed recurrent connection, are ignored.

3 This approximate gradient is then used to update the weights with the learning function chosen by the user. The function learngdm is recommended.

Elman networks are not as reliable as some other kinds of networks because both training and adaption happen using an approximation of the error gradient.

For an Elman to have the best chance at learning a problem it needs more hidden neurons in its hidden layer than are actually required for a solution by another method. While a solution may be available with fewer neurons, the Elman network is less able to find the most appropriate weights for hidden neurons since the error gradient is approximated. Therefore, having a fair number of neurons to begin with makes it more likely that the hidden neurons will start out dividing up the input space in useful ways.

The function train trains an Elman network to generate a sequence of target vectors when it is presented with a given sequence of input vectors. The input vectors and target vectors are passed to train as matrices P and T. Train takes these vectors and the initial weights and biases of the network, trains the network using backpropagation with momentum and an adaptive learning rate, and returns new weights and biases.

Let us continue with the example of the previous section, and suppose that we want to train a network with an input P and targets T as defined below

```
P = round(rand(1,8))
P =
      1     0     1     1     1     0     1     1
```

and

```
T = [0 (P(1:end-1)+P(2:end) == 2)]
T =
     0     0     0     1     1     0     0     1
```

Here T is defined to be 0, except when two 1's occur in P, in which case T is 1.

As noted previously, our network has five hidden neurons in the first layer.

```
net = newelm([0 1],[5 1],{'tansig','logsig'});
```

We use trainbfg as the training function and train for 100 epochs. After training we simulate the network with the input P and calculate the difference between the target output and the simulated network output.

```
net = train(net,Pseq,Tseq);
Y = sim(net,Pseq);
z = seq2con(Y);
z{1,1};
diff1 = T - z{1,1}
```

Note that the difference between the target and the simulated output of the trained network is very small. Thus, the network is trained to produce the desired output sequence on presentation of the input vector **P**.

See Chapter 11 for an application of the Elman network to the detection of wave amplitudes.

# Hopfield Network

## Fundamentals

The goal here is to design a network that stores a specific set of equilibrium points such that, when an initial condition is provided, the network eventually comes to rest at such a design point. The network is recursive in that the output is fed back as the input, once the network is in operation. Hopefully, the network output will settle on one of the original design points

The design method that we present is not perfect in that the designed network may have undesired spurious equilibrium points in addition to the desired ones. However, the number of these undesired points is made as small as possible by the design method. Further, the domain of attraction of the designed equilibrium points is as large as possible.

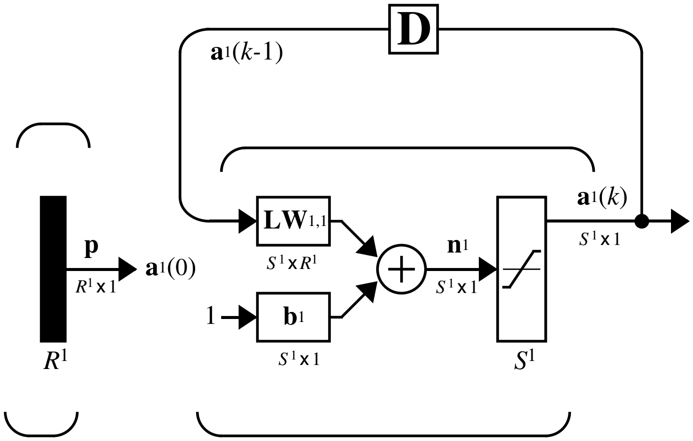
The design method is based on a system of first-order linear ordinary differential equations that are defined on a closed hypercube of the state space. The solutions exist on the boundary of the hypercube. These systems have the basic structure of the Hopfield model, but are easier to understand and design than the Hopfield model.

The material in this section is based on the following paper: Jian-Hua Li, Anthony N. Michel and Wolfgang Porod, "Analysis and synthesis of a class of neural networks: linear systems operating on a closed hypercube," IEEE Trans. on Circuits and Systems vol 36, no. 11, pp. 1405-22, November 1989.

For further information on Hopfield networks, read Chapter 18 of the *Hopfield Network* [HDB96].

## Architecture

The architecture of the network that we are using follows.



Initial conditions

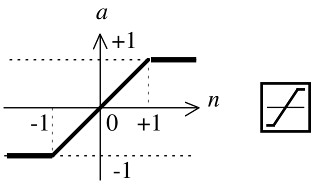
Symmetric saturated linear layer

$\mathbf{a}^1(0) = \mathbf{p}$ **and then for** $k = 1, 2, ...$

$\mathbf{a}^1(k) = \mathbf{satlins}(\mathbf{LW}^{1,1}\mathbf{a}^1(k\text{-}1)) + \mathbf{b}^1)$

As noted, the *input* **p** to this network merely supplies the initial conditions.

The Hopfield network uses the saturated linear transfer function `satlins`.



$a = satlins(n)$

Satlins Transfer Function

For inputs less than -1 `satlins` produces -1. For inputs in the range -1 to +1 it simply returns the input value. For inputs greater than +1 it produces +1.

This network can be tested with one or more input vectors which are presented as initial conditions to the network. After the initial conditions are given, the network produces an output which is then fed back to become the input. This process is repeated over and over until the output stabilizes. Hopefully, each output vector eventually converges to one of the design equilibrium point vectors that is closest to the input that provoked it.

## Design (newhop)

Li et. al. [LiMi89] have studied a system that has the basic structure of the Hopfield network but is, in Li's own words, "easier to analyze, synthesize, and implement than the Hopfield model." The authors are enthusiastic about the reference article, as it has many excellent points and is one of the most readable in the field. However, the design is mathematically complex, and even a short justification of it would burden this guide. Thus, we present the Li design method, with thanks to Li et al., as a recipe that is found in the function `newhop`.

Given a set of target equilibrium points represented as a matrix **T** of vectors, `newhop` returns weights and biases for a recursive network. The network is guaranteed to have stable equilibrium points at the target vectors, but it could contain other spurious equilibrium points as well. The number of these undesired points is made as small as possible by the design method.

Once the network has been designed, it can be tested with one or more input vectors. Hopefully those input vectors close to target equilibrium points will find their targets. As suggested by the network figure, an array of input vectors may be presented at one time or in a batch. The network proceeds to give output vectors that are fed back as inputs. These output vectors can be can be compared to the target vectors to see how the solution is proceeding.

The ability to run batches of trial input vectors quickly allows you to check the design in a relatively short time. First you might check to see that the target equilibrium point vectors are indeed contained in the network. Then you could try other input vectors to determine the domains of attraction of the target equilibrium points and the locations of spurious equilibrium points if they are present.

Consider the following design example. Suppose that we want to design a network with two stable points in a three-dimensional space.

```
T = [-1 -1 1; 1 -1 1]'
T =
    -1     1
    -1    -1
     1     1
```

We can execute the design with

```
net = newhop(T);
```

Next we can check to make sure that the designed network is at these two points. We can do this as follows. (Since Hopfield networks have no inputs, the second argument to `sim` below is Q = 2 when using matrix notation).

```
Ai = T;
[Y,Pf,Af] = sim(net,2,[],Ai);
Y
```

This gives us

```
Y =
    -1     1
    -1    -1
     1     1
```

Thus, the network has indeed been designed to be stable at its design points. Next we can try another input condition that is not a design point, such as:

```
Ai = {[-0.9; -0.8; 0.7]}
```

This point is reasonably close to the first design point, so one might anticipate that the network would converge to that first point. To see if this happens, we run the following code. Note, incidentally, that we specified the original point in cell array form. This allows us to run the network for more than one step.

```
[Y,Pf,Af] = sim(net,{1 5},{},Ai);
Y{1}
```

We get

```
Y =
    -1
    -1
     1
```

Thus, an original condition close to a design point did converge to that point.

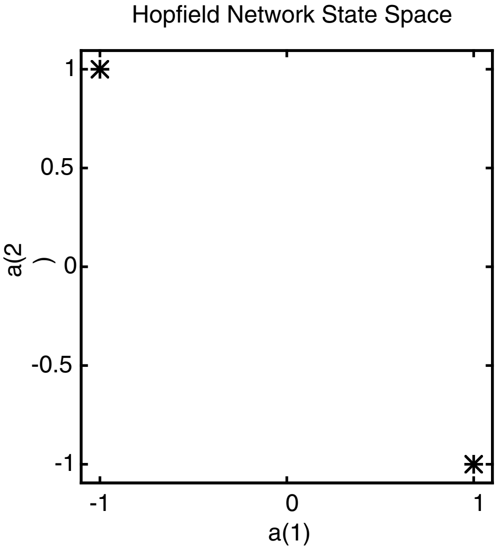
This is, of course, our hope for all such inputs. Unfortunately, even the best known Hopfield designs occasionally include undesired spurious stable points that attract the solution.

### Example

Consider a Hopfield network with just two neurons. Each neuron has a bias and weights to accommodate two-element input vectors weighted. We define the target equilibrium points to be stored in the network as the two columns of the matrix **T**.

```
T = [1 -1; -1 1]'
T =
      1     -1
     -1      1
```

Here is a plot of the Hopfield state space with the two stable points labeled with '*' markers.



These target stable points are given to newhop to obtain weights and biases of a Hopfield network.

```
net = newhop(T);
```

The design  returns a set of weights and a bias for each neuron. The results are obtained from

```
W= net.LW{1,1}
```

which gives

```
W =
    0.6925   -0.4694
   -0.4694    0.6925
```

and from

```
b = net.b{1,1}
```

which gives

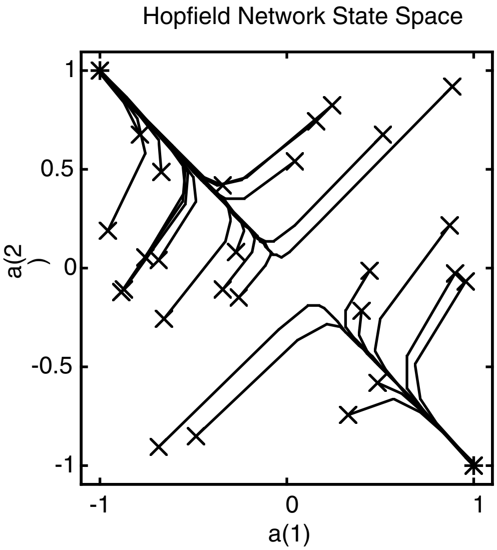
```
b =
   1.0e-16 *
    0.6900
    0.6900
```

Next the design is tested with the target vectors **T** to see if they are stored in the network. The targets are used as inputs for the simulation function sim.

```
Ai = T;
[Y,Pf,Af] = sim(net,2,[],Ai);
Y =
     1    -1
    -1     1
```

As hoped, the new network outputs are the target vectors. The solution stays at its initial conditions after a single update and, therefore, will stay there for any number of updates.

Now you might wonder how the network performs with various random input vectors. Here is a plot showing the paths that the network took through its state space, to arrive at a target point.



This plot show the trajectories of the solution for various starting points. You can try the demonstration `demohop1` to see more of this kind of network behavior.

Hopfield networks can be designed for an arbitrary number of dimensions. You can try `demohop3` to see a three-dimensional design.

Unfortunately, Hopfield networks could have both unstable equilibrium points and spurious stable points. You can try demonstration programs `demohop2` and `demohop4` to investigate these issues.

## Summary

Elman networks, by having an internal feedback loop, are capable of learning to detect and generate temporal patterns. This makes Elman networks useful in such areas as signal processing and prediction where time plays a dominant role.

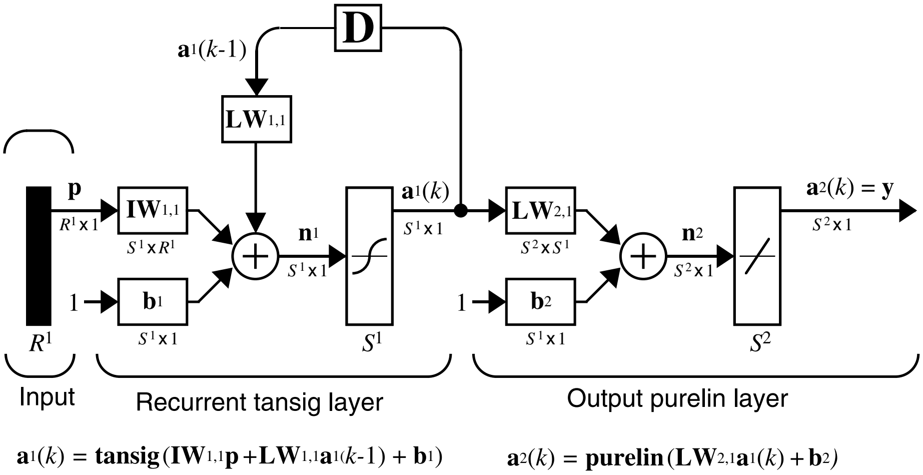
Because Elman networks are an extension of the two-layer sigmoid/linear architecture, they inherit the ability to fit any input/output function with a finite number of discontinuities. They are also able to fit temporal patterns, but may need many neurons in the recurrent layer to fit a complex function.

Hopfield networks can act as error correction or vector categorization networks. Input vectors are used as the initial conditions to the network, which recurrently updates until it reaches a stable output vector.

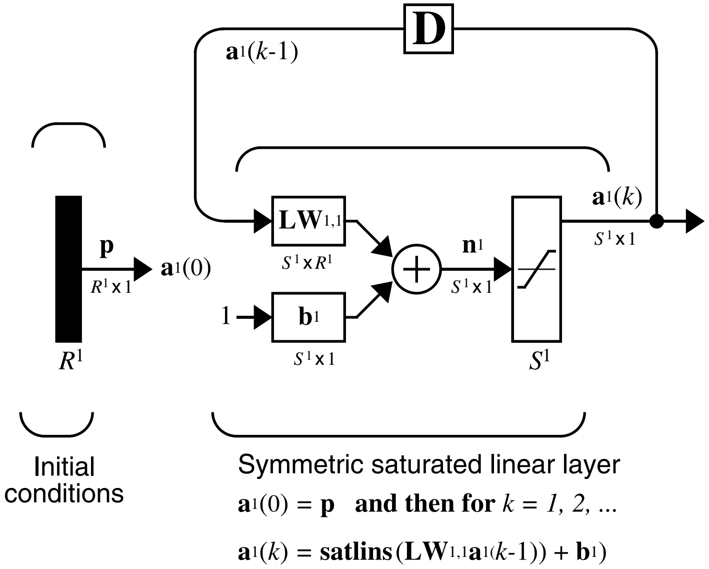
Hopfield networks are interesting from a theoretical standpoint, but are seldom used in practice. Even the best Hopfield designs may have spurious stable points that lead to incorrect answers. More efficient and reliable error correction techniques, such as backpropagation, are available.

# Figures

## Elman Network



## Hopfield Network


D
a1(k-1)
LW1,1
S1 x R1
p
R1 x 1
a1(0)
n1
S1 x 1
a1(k)
S1 x 1
1
b1
S1 x 1
R1
S1
Initial conditions
Symmetric saturated linear layer
a1(0) = p and then for k = 1, 2, ...
a1(k) = satlins (LW1,1a1(k-1)) + b1)

## New Functions

This chapter introduces the following new functions.

| Function | Description |
|---|---|
| `newelm` | Create an Elman backpropagation network. |
| `newhop` | Create a Hopfield recurrent network. |
| `satlins` | Symmetric saturating linear transfer function. |

# 10

# Adaptive Filters and Adaptive Training

# Introduction

The ADALINE (Adaptive Linear Neuron networks) networks discussed in this chapter are similar to the perceptron, but their transfer function is linear rather than hard-limiting. This allows their outputs to take on any value, whereas the perceptron output is limited to either 0 or 1. Both the ADALINE and the perceptron can only solve linearly separable problems. However, here we will make use of the LMS (Least Mean Squares) learning rule, which is much more powerful than the perceptron learning rule. The LMS or Widrow-Hoff learning rule minimizes the mean square error and, thus, moves the decision boundaries as far as it can from the training patterns.

In this chapter, we design an adaptive linear system that responds to changes in its environment as it is operating. Linear networks that are adjusted at each time step based on new input and target vectors can find weights and biases that minimize the network's sum-squared error for recent input and target vectors. Networks of this sort are often used in error cancellation, signal processing, and control systems.

The pioneering work in this field was done by Widrow and Hoff, who gave the name ADALINE to adaptive linear elements. The basic reference on this subject is: Widrow B. and S. D. Sterns, *Adaptive Signal Processing*, New York: Prentice-Hall 1985.

We also consider the adaptive training of self organizing and competitive networks in this chapter.

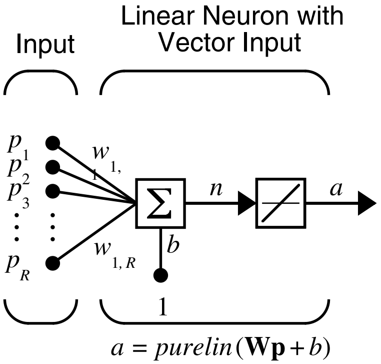
## Important Adaptive Functions

This chapter introduces the function `adapt`, which changes the weights and biases of a network incrementally during training.

You can type `help linnet` to see a list of linear and adaptive network functions, demonstrations, and applications.

# Linear Neuron Model

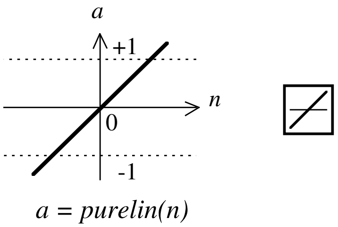
A linear neuron with $R$ inputs is shown below.



Where...

$R$ = number of elements in input vector

This network has the same basic structure as the perceptron. The only difference is that the linear neuron uses a linear transfer function, which we name `purelin`.



Linear Transfer Function

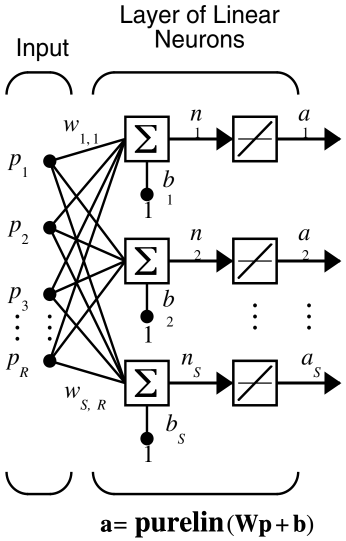
The linear transfer function calculates the neuron's output by simply returning the value passed to it.

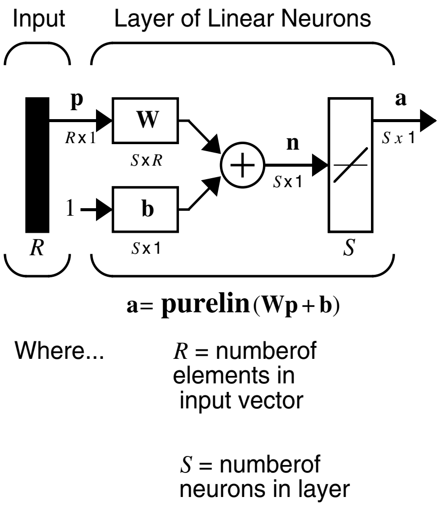
$$a = purelin(n) = purelin(\mathbf{Wp} + b) = \mathbf{Wp} + b$$

This neuron can be trained to learn an affine function of its inputs, or to find a linear approximation to a nonlinear function. A linear network cannot, of course, be made to perform a nonlinear computation.

## Adaptive Linear Network Architecture

The ADALINE network shown below has one layer of $S$ neurons connected to $R$ inputs through a matrix of weights **W**.



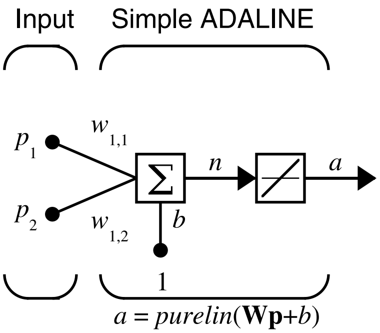


This network is sometimes called a MADALINE for Many ADALINES. Note that the figure on the right defines an $S$-length output vector **a**.

The Widrow-Hoff rule can only train single-layer linear networks. This is not much of a disadvantage, however, as single-layer linear networks are just as capable as multilayer linear networks. For every multilayer linear network, there is an equivalent single-layer linear network.

### Single ADALINE (newlin)

Consider a single ADALINE with two inputs. The diagram for this network is shown below.

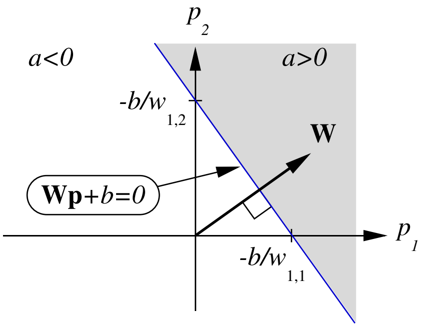


The weight matrix **W** in this case has only one row. The network output is:

$$a = purelin(n) = purelin(\mathbf{Wp} + b) = \mathbf{Wp} + b \qquad \text{or}$$

$$a = w_{1,1}p_1 + w_{1,2}p_2 + b$$

Like the perceptron, the ADALINE has a *decision boundary* that is determined by the input vectors for which the net input $n$ is zero. For $n = 0$ the equation $\mathbf{Wp} + b = 0$ specifies such a decision boundary as shown below (adapted with thanks from [HDB96])



Input vectors in the upper right gray area lead to an output greater than 0. Input vectors in the lower left white area lead to an output less than 0. Thus, the ADALINE can be used to classify objects into two categories. Now you can find the network output with the function `sim`.

```
a = sim(net,p)
a =
    24
```

To summarize, you can create an ADALINE network with `newlin`, adjust its elements as you want and simulate it with `sim`. You can find more about `newlin` by typing `help newlin`.

# Mean Square Error

Like the perceptron learning rule, the least mean square error (LMS) algorithm is an example of supervised training, in which the learning rule is provided with a set of examples of desired network behavior.

$$\{\mathbf{p}_1, \mathbf{t}_1\}, \{\mathbf{p}_2, \mathbf{t}_2\}, \ldots, \{\mathbf{p}_Q, \mathbf{t}_Q\}$$

Here $\mathbf{p}_q$ is an input to the network, and $\mathbf{t}_q$ is the corresponding target output. As each input is applied to the network, the network output is compared to the target. The error is calculated as the difference between the target output and the network output. We want to minimize the average of the sum of these errors.

$$mse = \frac{1}{Q}\sum_{k=1}^{Q} e(k)^2 = \frac{1}{Q}\sum_{k=1}^{Q} (t(k) - a(k))^2$$

The LMS algorithm adjusts the weights and biases of the ADALINE so as to minimize this mean square error.

Fortunately, the mean square error performance index for the ADALINE network is a quadratic function. Thus, the performance index will either have one global minimum, a weak minimum, or no minimum, depending on the characteristics of the input vectors. Specifically, the characteristics of the input vectors determine whether or not a unique solution exists.

You can learn more about this topic in Chapter 10 of [HDB96].

## LMS Algorithm (learnwh)

Adaptive networks will use the The LMS algorithm or Widrow-Hoff learning algorithm based on an approximate steepest descent procedure. Here again, adaptive linear networks are trained on examples of correct behavior.

The LMS algorithm, shown below, is discussed in detail in Chapter 4, “Linear Filters.”

$$\mathbf{W}(k+1) = \mathbf{W}(k) + 2\alpha\mathbf{e}(k)\mathbf{p}^T(k)$$

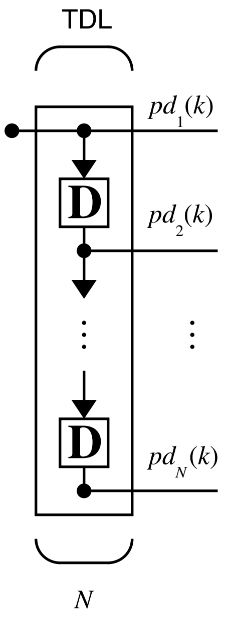
$$\mathbf{b}(k+1) = \mathbf{b}(k) + 2\alpha\mathbf{e}(k) .$$

# Adaptive Filtering (adapt)

The ADALINE network, much like the perceptron, can only solve linearly separable problems. Nevertheless, the ADALINE has been and is today one of the most widely used neural networks found in practical applications. Adaptive filtering is one of its major application areas.

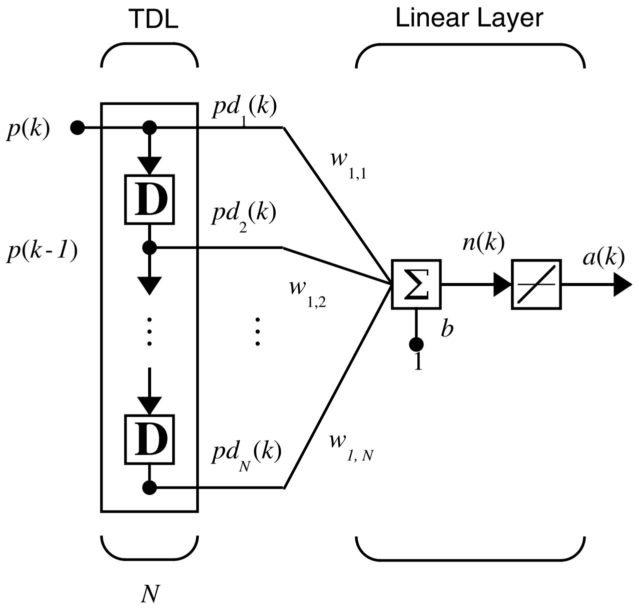
## Tapped Delay Line

We need a new component, the tapped delay line, to make full use of the ADALINE network. Such a delay line is shown below. There the input signal enters from the left, and passes through $N$-1 delays. The output of the tapped delay line (TDL) is an $N$-dimensional vector, made up of the input signal at the current time, the previous input signal, etc.



## Adaptive Filter

We can combine a tapped delay line with an ADALINE network to create the *adaptive filter shown* below.



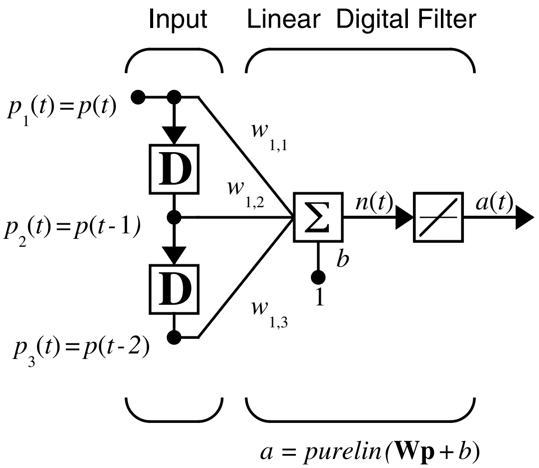
The output of the filter is given by

$$a(k) = purelin(\mathbf{Wp} + b) = \sum_{i=1}^{R} w_{1,i} a(k-i+1) + b$$

The network shown above is referred to in the digital signal processing field as a finite impulse response (FIR) filter [WiSt85]. Let us take a look at the code that we use to generate and simulate such an adaptive network.

## Adaptive Filter Example

First we will define a new linear network using `newlin`.



Assume that the input values have a range from 0 to 10. We can now define our single output network.

```
net = newlin([0,10],1);
```

We can specify the delays in the tapped delay line with

```
net.inputWeights{1,1}.delays = [0 1 2];
```

This says that the delay line is connected to the network weight matrix through delays of 0, 1, and 2 time units. (You can specify as many delays as you want, and can omit some values if you like. They must be in ascending order.)

We can give the various weights and the bias values with

```
net.IW{1,1} = [7 8 9];
net.b{1} = [0];
```

Finally we will define the initial values of the outputs of the delays as

```
pi ={1 2}
```

Note that these are ordered from left to right to correspond to the delays taken from top to bottom in the figure. This concludes the setup of the network. Now how about the input?

We assume that the input scalars arrive in a sequence, first the value 3, then the value 4, next the value 5, and finally the value 6. We can indicate this sequence by defining the values as elements of a cell array. (Note the curly brackets.)

```
p = {3 4 5 6}
```

Now we have a network and a sequence of inputs. We can simulate the network to see what its output is as a function of time.

```
[a,pf] = sim(net,p,pi);
```

This yields an output sequence

```
a =
    [46]    [70]    [94]    [118]
```

and final values for the delay outputs of

```
pf =
    [5]    [6].
```

The example is sufficiently simple that you can check it by hand to make sure that you understand the inputs, initial values of the delays, etc.

The network that we have defined can be trained with the function `adapt` to produce a particular output sequence. Suppose, for instance, we would like the network to produce the sequence of values 10, 20, 30, and 40.

```
T = {10 20 30 40}
```

We can train our defined network to do this, starting from the initial delay conditions that we used above. We specify 10 passes through the input sequence with

```
net.adaptParam.passes = 10;
```

Then we can do the training with

```
[net,y,E pf,af] = adapt(net,p,T,pi);
```

This code returns the final weights, bias, and output sequence shown below.

```
wts = net.IW{1,1}
wts =
    0.5059    3.1053    5.7046
```

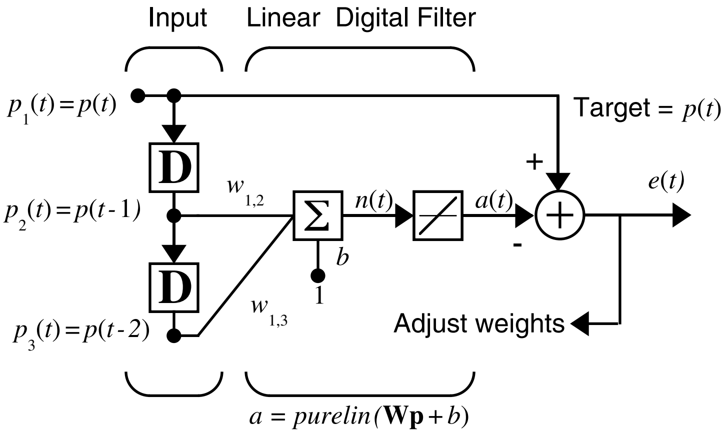
```
bias = net.b{1}
bias =
   -1.5993
y =
    [11.8558]    [20.7735]    [29.6679]    [39.0036]
```

Presumably, if we ran for additional passes the output sequence would have been even closer to the desired values of 10, 20, 30, and 40.

Thus, adaptive networks can be specified, simulated, and finally trained with `adapt`. However, the outstanding value of adaptive networks lies in their use to perform a particular function, such as or prediction or noise cancellation.

## Prediction Example

Suppose that we want to use an adaptive filter to predict the next value of a stationary random process, *p(t)*. We use the network shown below to do this.



Predictive Filter: $a(t)$ is approximation to $p(t)$

The signal to be predicted, *p(t),* enters from the left into a tapped delay line. The previous two values of *p(t)* are available as outputs from the tapped delay line. The network uses `adapt` to change the weights on each time step so as to minimize the error *e(t)* on the far right. If this error is zero, then the network output *a(t)* is exactly equal to *p(t)*, and the network has done its prediction properly.

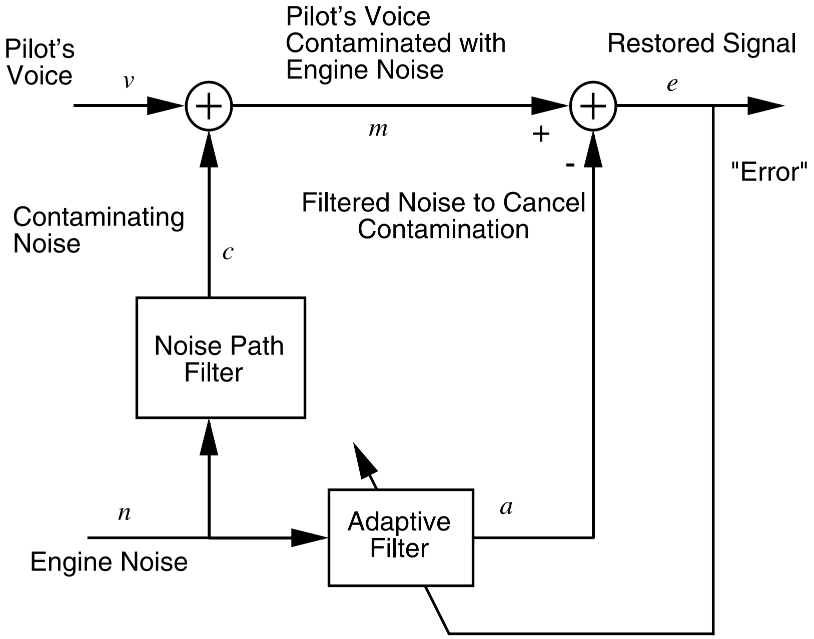
A detailed analysis of this network is not appropriate here, but we can state the main points. Given the autocorrelation function of the stationary random process *p(t)*, the error surface, the maximum learning rate, and the optimum values of the weights can be calculated. Commonly, of course, one does not have detailed information about the random process, so these calculations cannot be performed. But this lack does not matter to the network. The network, once initialized and operating, adapts at each time step to minimize the error and in a relatively short time is able to predict the input p(t).

Chapter 10 of [HDB96] presents this problem, goes through the analysis, and shows the weight trajectory during training. The network finds the optimum weights on its own without any difficulty whatsoever.

You also can try demonstration program `nnd10nc` to see an adaptive noise cancellation program example in action. This demonstration allows you to pick a learning rate and *momentum* (see Chapter 5, “Backpropagation”), and shows the learning trajectory, and the original and cancellation signals verses time.

## Noise Cancellation Example

Consider a pilot in an airplane. When the pilot speaks into a microphone, the engine noise in the cockpit is added to the voice signal, and the resultant signal heard by passengers would be of low quality. We would like to obtain a signal that contains the pilot’s voice, but not the engine noise. We can do this with an adaptive filter if we obtain a sample of the engine noise and apply it as the input to the adaptive filter.



Adaptive Filter Adjusts to Minimize Error.
This removes the engine noise from contaminated signal, leaving the pilot's voice as the "error."

Here we adaptively train the neural linear network to predict the combined pilot/engine signal $m$ from an engine signal $n$. Notice that the engine signal $n$ does not tell the adaptive network anything about the pilot's voice signal contained in $m$. However, the engine signal $n$. does give the network information it can use to predict the engine's contribution to the pilot/engine signal $m$.

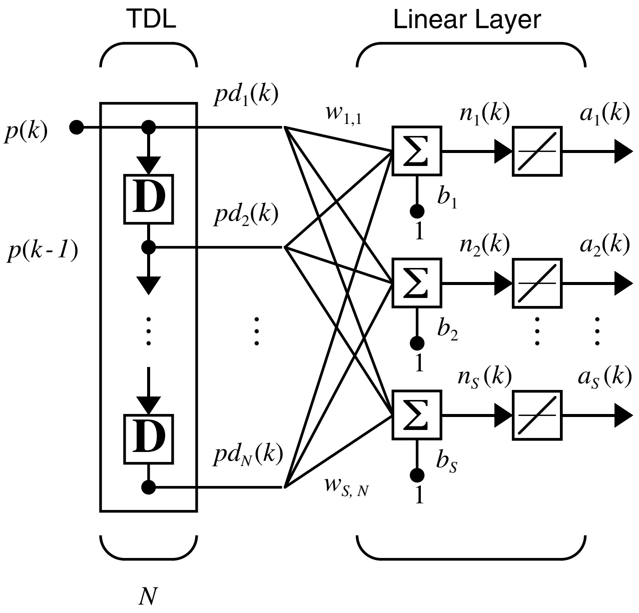
The network will do its best to adaptively output $m$. In this case, the network can only predict the engine interference noise in the pilot/engine signal $m$. The network error $e$ is equal to $m$, the pilot/engine signal, minus the predicted contaminating engine noise signal. Thus, $e$ contains only the pilot's voice! Our linear adaptive network adaptively learns to cancel the engine noise.

Note, in closing, that such adaptive noise canceling generally does a better job than a classical filter because the noise here is subtracted from rather than filtered out of the signal $m$.

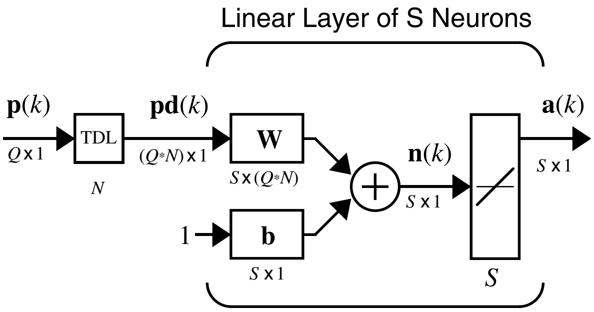
Try `demolin8` for an example of adaptive noise cancellation.

## Multiple Neuron Adaptive Filters

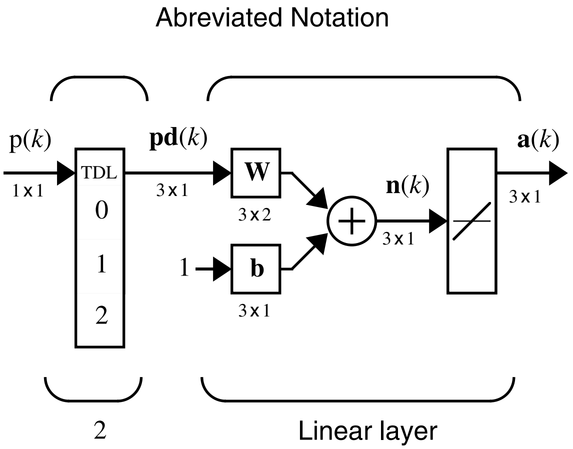
We may want to use more than one neuron in an adaptive system, so we need some additional notation. A tapped delay line can be used with $S$ linear neurons as shown below.



Alternatively, we can show this same network in abbreviated form.



If we want to show more of the detail of the tapped delay line and there are not too many delays, we can use the following notation.



Here we have a tapped delay line that sends the current signal, the previous signal, and the signal delayed before that to the weight matrix. We could have a longer list, and some delay values could be omitted if desired. The only requirement is that the delays are shown in increasing order as they go from top to bottom.

# Summary

The ADALINE (Adaptive Linear Neuron networks) networks discussed in this chapter are similar to the perceptron, but their transfer function is linear rather than hard-limiting. They make use of the LMS (Least Mean Squares) learning rule, which is much more powerful that the perceptron learning rule. The LMS or Widrow-Hoff learning rule minimizes the mean square error and, thus, moves the decision boundaries as far as it can from the training patterns.

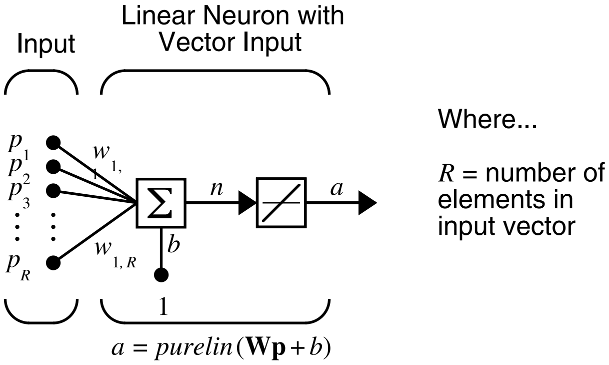
In this chapter, we design an adaptive linear system that responds to changes in its environment as it is operating. Linear networks that are adjusted at each time step based on new input and target vectors can find weights and biases that minimize the network's sum-squared error for recent input and target vectors.

Adaptive linear filters have many practical applications such as noise cancellation, signal processing, and prediction in control and communication systems.

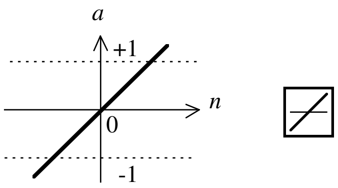
This chapter introduces the function `adapt`, which changes the weights and biases of a network incrementally during training.

## Figures and Equations

### Linear Neuron


Input
Linear Neuron with Vector Input
$p_1$
$p_2$
$p_3$
$p_R$
$w_{1,1}$
$w_{1,R}$
$\Sigma$
$n$
$a$
$b$
1
$a = purelin(\mathbf{Wp} + b)$
Where...
$R$ = number of elements in input vector

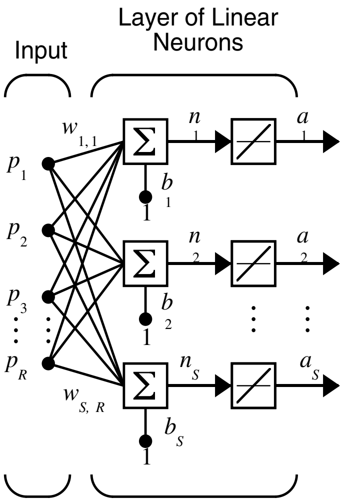
## Purelin Transfer Function



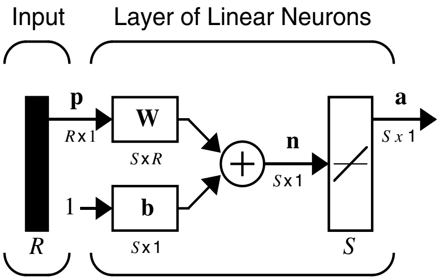
*a = purelin(n)*

Linear Transfer Function

## MADALINE



**a**= **purelin**(**Wp**+**b**)



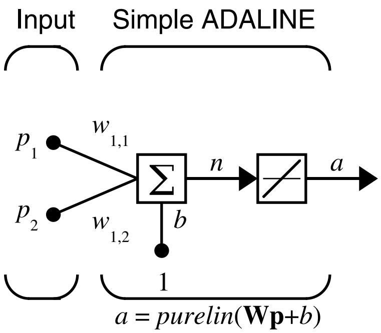
**a**= **purelin**(**Wp**+**b**)

Where...

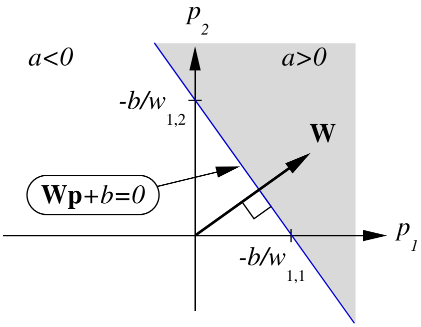
*R* = numberof elements in input vector

*S* = numberof neurons in layer

## ADALINE



## Decision Boundary



## Mean Square Error

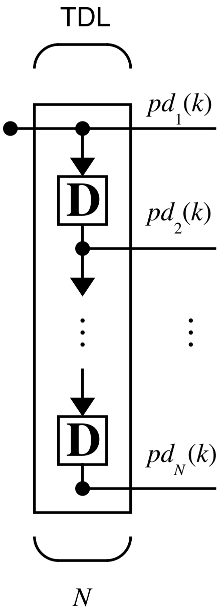
$$mse = \frac{1}{Q}\sum_{k=1}^{Q} e(k)^2 = \frac{1}{Q}\sum_{k=1}^{Q} (t(k)-a(k))^2$$

### LMS (Widrow-Hoff) Algorithm

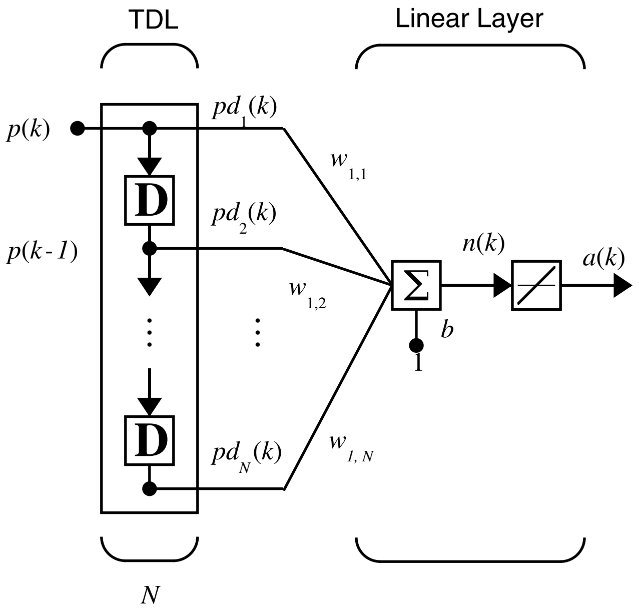
$$\mathbf{W}(k+1) = \mathbf{W}(k) + 2\alpha\mathbf{e}(k)\mathbf{p}^T(k)$$

$$\mathbf{b}(k+1) = \mathbf{b}(k) + 2\alpha\mathbf{e}(k)$$

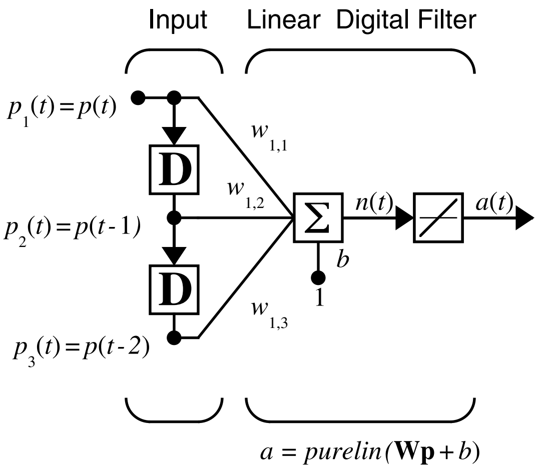
### Tapped Delay Line



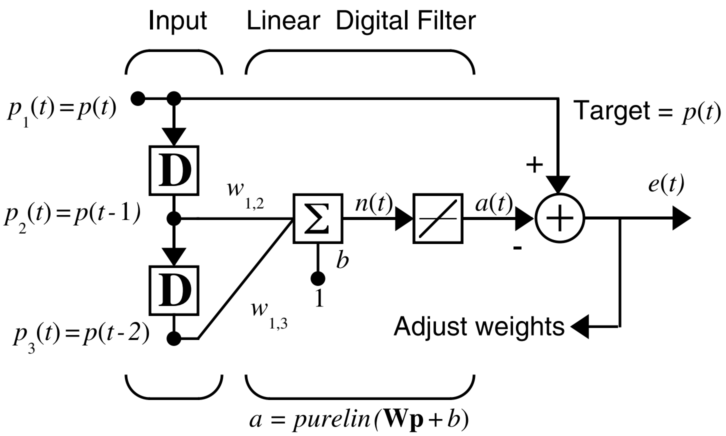
## Adaptive Filter



### Adaptive Filter Example

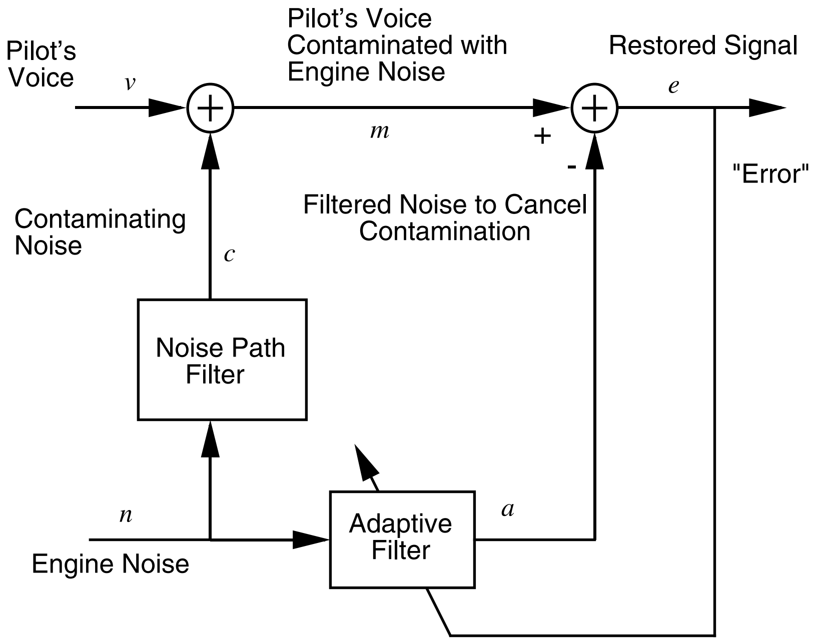


### Prediction Example



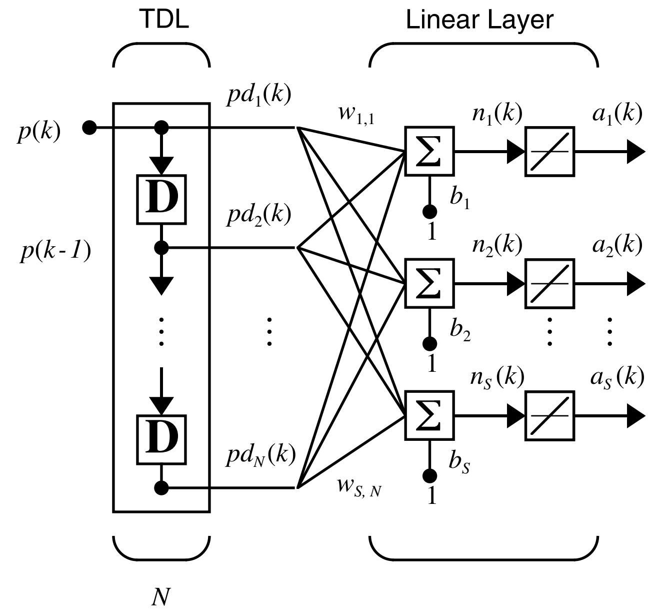
Predictive Filter: $a(t)$ is approximation to $p(t)$

### Noise Cancellation Example

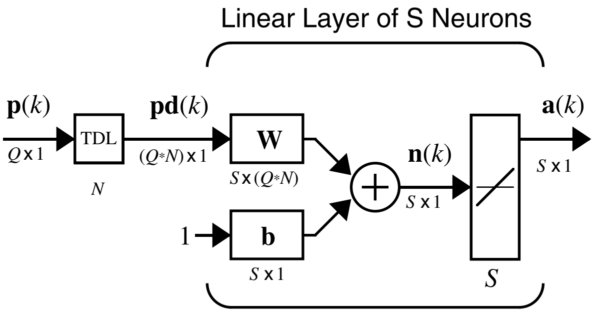


Adaptive Filter Adjusts to Minimize Error.
This removes the engine noise from contaminated signal, leaving the pilot's voice as the "error."

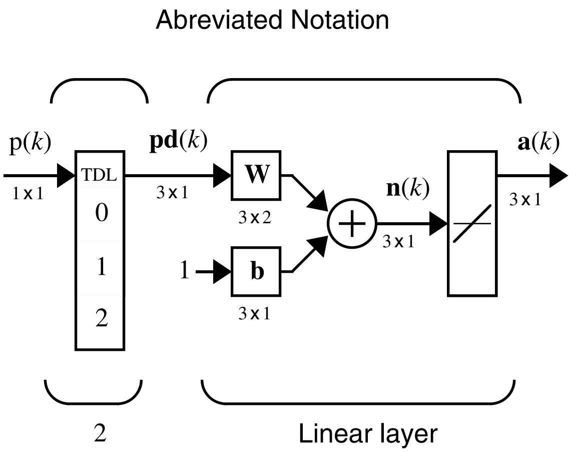
### Multiple Neuron Adaptive Filter



### Abbreviated Form of Adaptive Filter



### Small Specific Adaptive Filter



## New Functions

This chapter introduced the following function.

| Function | Description |
|---|---|
| `adapt` | Trains a network using a sequence of inputs |

11

# Applications

# Introduction

Today, neural networks can solve problems of economic importance that could not be approached previously in any practical way. Some of the recent neural network applications are discussed in this chapter. See Chapter 1, "Introduction" for a list of many areas where neural networks already have been applied.

**Note** The rest of this chapter describes applications that are practical and make extensive use of the neural network functions described throughout this documentation.

## Application Scripts

The linear network applications are contained in scripts `applin1` and `applin2`.

The Elman network amplitude detection application is contained in the script `appelm1`.

The character recognition application is in `appcr1`.

Type `help nndemos` to see a listing of all neural network demonstrations or applications.

# Applin1: Linear Design

## Problem Definition

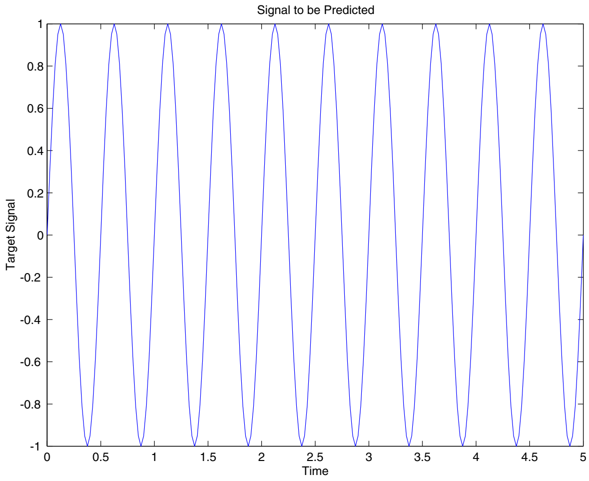
Here is the definition of a signal T, which lasts 5 seconds, and is defined at a sampling rate of 40 samples per second.

```
time = 0:0.025:5;
T = sin(time*4*pi);
Q = length(T);
```

At any given time step, the network is given the last five values of the signal t, and expected to give the next value. The inputs P are found by delaying the signal T from one to five time steps.

```
P = zeros(5,Q);
P(1,2:Q) = T(1,1:(Q-1));
P(2,3:Q) = T(1,1:(Q-2));
P(3,4:Q) = T(1,1:(Q-3));
P(4,5:Q) = T(1,1:(Q-4));
P(5,6:Q) = T(1,1:(Q-5));
```

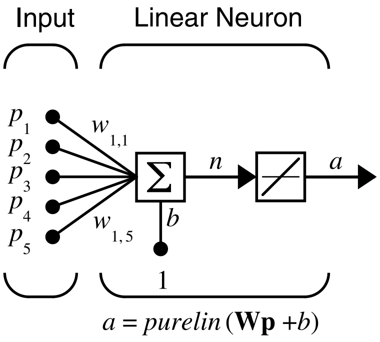
Here is a plot of the signal T.



## Network Design

Because the relationship between past and future values of the signal is not changing, the network can be designed directly from examples using `newlind`.

The problem as defined above has five inputs (the five delayed signal values), and one output (the next signal value). Thus, the network solution must consist of a single neuron with five inputs.



Here `newlind` finds the weights and biases, for the neuron above, that minimize the sum-squared error for this problem.

```
net = newlind(P,T);
```

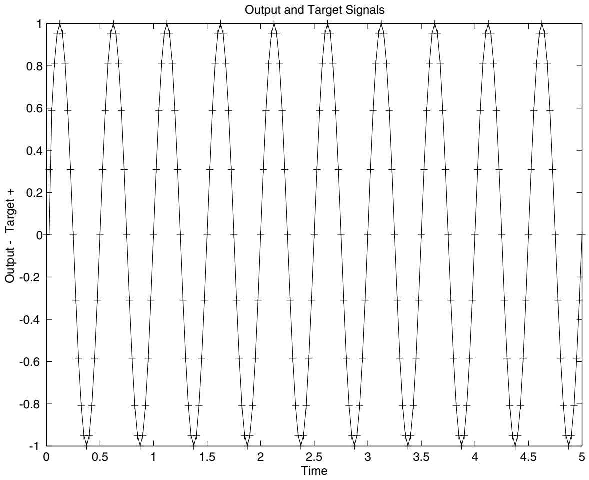
The resulting network can now be tested.

## Network Testing

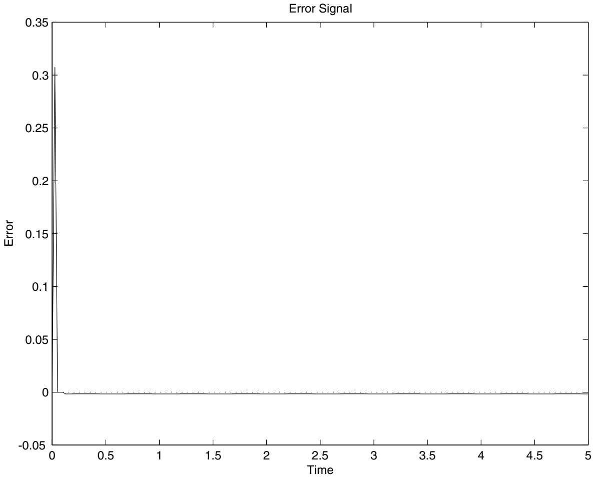
To test the network, its output `a` is computed for the five delayed signals `P` and compared with the actual signal `T`.

```
a = sim(net,P);
```

Here is a plot of `a` compared to `T`.



The network's output a and the actual signal t appear to match up perfectly. Just to be sure, let us plot the error e = T – a.



The network did have some error for the first few time steps. This occurred because the network did not actually have five delayed signal values available until the fifth time step. However, after the fifth time step error was negligible. The linear network did a good job. Run the script applin1 to see these plots.

## Thoughts and Conclusions

While newlind is not able to return a zero error solution for nonlinear problems, it does minimize the sum-squared error. In many cases, the solution, while not perfect, may model a nonlinear relationship well enough to meet the application specifications. Giving the linear network many delayed signal values gives it more information with which to find the lowest error linear fit for a nonlinear problem.

Of course, if the problem is very nonlinear and/or the desired error is very low, backpropagation or radial basis networks would be more appropriate.

# Applin2: Adaptive Prediction

In application script applin2, a linear network is trained incrementally with adapt to predict a time series. Because the linear network is trained incrementally, it can respond to changes in the relationship between past and future values of the signal.

## Problem Definition

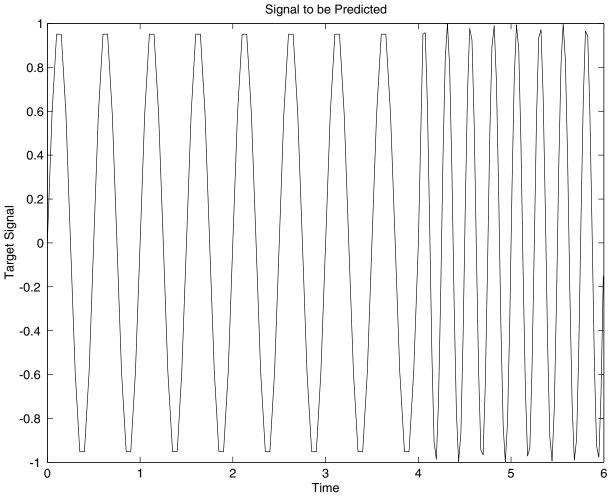
The signal T to be predicted lasts 6 seconds with a sampling rate of 20 samples per second. However, after 4 seconds the signal's frequency suddenly doubles.

```
time1 = 0:0.05:4;
time2 = 4.05:0.024:6;
time = [time1 time2];
T = [sin(time1*4*pi) sin(time2*8*pi)];
```

Since we are training the network incrementally, we change t to a sequence.

```
T = con2seq(T);
```

Here is a plot of this signal.

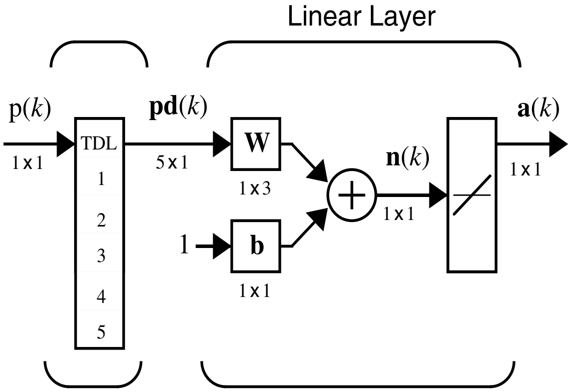


The input to the network is the same signal that makes up the target.

```
P = T;
```

## Network Initialization

The network has only one neuron, as only one output value of the signal T is being generated at each time step. This neuron has five inputs, the five delayed values of the signal T.



The function newlin creates the network shown above. We use a learning rate of 0.1 for incremental training.

```
lr = 0.1;
delays = [1 2 3 4 5];
net = newlin(minmax(cat(2,P{:})),1,delays,lr);
[w,b] = initlin(P,t)
```

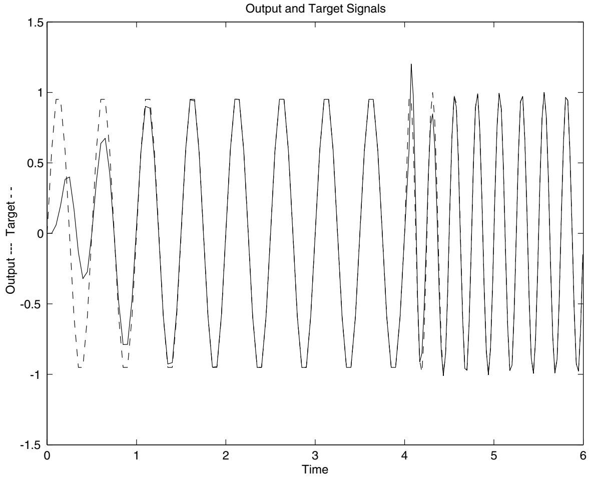
## Network Training

The above neuron is trained incrementally with adapt. Here is the code to train the network on input/target signals P and T.

```
[net,a,e]=adapt(net,P,T);
```

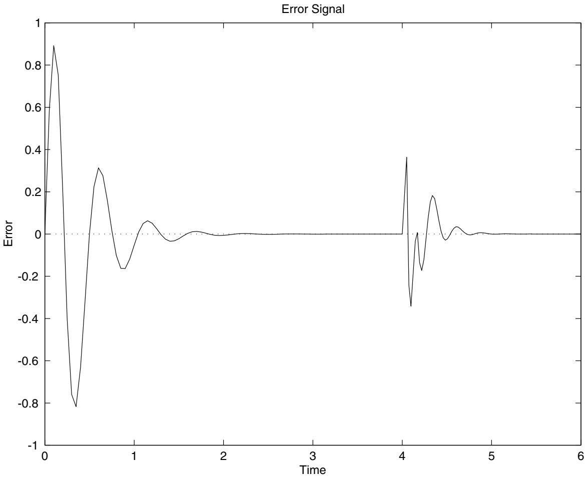
## Network Testing

Once the network is adapted, we can plot its output signal and compare it to the target signal.



Initially, it takes the network 1.5 seconds (30 samples) to track the target signal. Then, the predictions are accurate until the fourth second when the target signal suddenly changes frequency. However, the adaptive network learns to track the new signal in an even shorter interval as it has already learned a behavior (a sine wave) similar to the new signal.

A plot of the error signal makes these effects easier to see.



## Thoughts and Conclusions

The linear network was able to adapt very quickly to the change in the target signal. The 30 samples required to learn the wave form are very impressive when one considers that in a typical signal processing application, a signal may be sampled at 20 kHz. At such a sampling frequency, 30 samples go by in 1.5 milliseconds.

For example, the adaptive network can be monitored so as to give a warning that its constants were nearing values that would result in instability.

Another use for an adaptive linear model is suggested by its ability to find a minimum sum-squared error linear estimate of a nonlinear system's behavior. An adaptive linear model is highly accurate as long as the nonlinear system stays near a given operating point. If the nonlinear system moves to a different operating point, the adaptive linear network changes to model it at the new point.

The sampling rate should be high to obtain the linear model of the nonlinear system at its current operating point in the shortest amount of time. However, there is a minimum amount of time that must occur for the network to see enough of the system's behavior to properly model it. To minimize this time, a small amount of noise can be added to the input signals of the nonlinear system. This allows the network to adapt faster as more of the operating points dynamics are expressed in a shorter amount of time. Of course, this noise should be small enough so it does not affect the system's usefulness.

# Appelm1: Amplitude Detection

Elman networks can be trained to recognize and produce both spatial and temporal patterns. An example of a problem where temporal patterns are recognized and classified with a spatial pattern is amplitude detection.

Amplitude detection requires that a wave form be presented to a network through time, and that the network output the amplitude of the wave form. This is not a difficult problem, but it demonstrates the Elman network design process.

The following material describes code that is contained in the demonstration script `appelm1`.

## Problem Definition

The following code defines two sine wave forms, one with an amplitude of 1.0, the other with an amplitude of 2.0.

```
p1 = sin(1:20);
p2 = sin(1:20)*2;
```

The target outputs for these wave forms is their amplitudes.

```
t1 = ones(1,20);
t2 = ones(1,20)*2;
```

These wave forms can be combined into a sequence where each wave form occurs twice. These longer wave forms are used to train the Elman network.

```
p = [p1 p2 p1 p2];
t = [t1 t2 t1 t2];
```

We want the inputs and targets to be considered a sequence, so we need to make the conversion from the matrix format.

```
Pseq = con2seq(p);
Tseq = con2seq(t);
```

## Network Initialization

This problem requires that the Elman network detect a single value (the signal), and output a single value (the amplitude), at each time step. Therefore the network must have one input element, and one output neuron.

```
R = 1;% 1 input element
S2 = 1;% 1 layer 2 output neuron
```

The recurrent layer can have any number of neurons. However, as the complexity of the problem grows, more neurons are needed in the recurrent layer for the network to do a good job.

This problem is fairly simple, so only 10 recurrent neurons are used in the first layer.

```
S1 = 10;% 10 recurrent neurons in the first layer
```

Now the function newelm can be used to create initial weight matrices and bias vectors for a network with one input that can vary between –2 and +2. We use variable learning rate (traingdx) for this example.

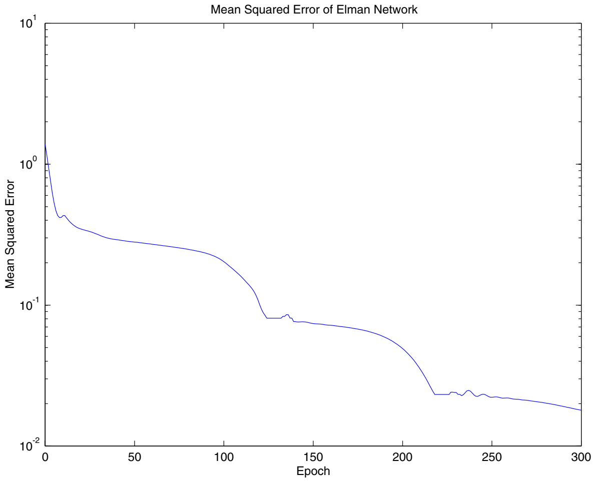
```
net = newelm([-2 2],[S1 S2],{'tansig','purelin'},'traingdx');
```

## Network Training

Now call train.

```
[net,tr] = train(net,Pseq,Tseq);
```

As this function finishes training at 500 epochs, it displays the following plot of errors.



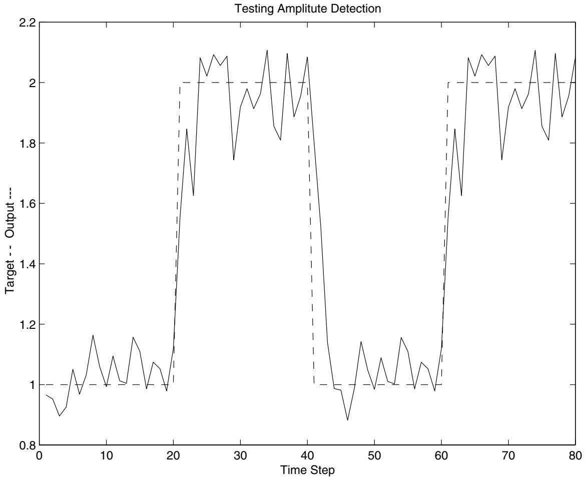
The final mean-squared error was about 1.8e-2. We can test the network to see what this means.

## Network Testing

To test the network, the original inputs are presented, and its outputs are calculated with `simuelm`.

```
a = sim(net,Pseq);
```

Here is the plot.



The network does a good job. New wave amplitudes are detected with a few samples. More neurons in the recurrent layer and longer training times would result in even better performance.

The network has successfully learned to detect the amplitudes of incoming sine waves.

## Network Generalization

Of course, even if the network detects the amplitudes of the training wave forms, it may not detect the amplitude of a sine wave with an amplitude it has not seen before.

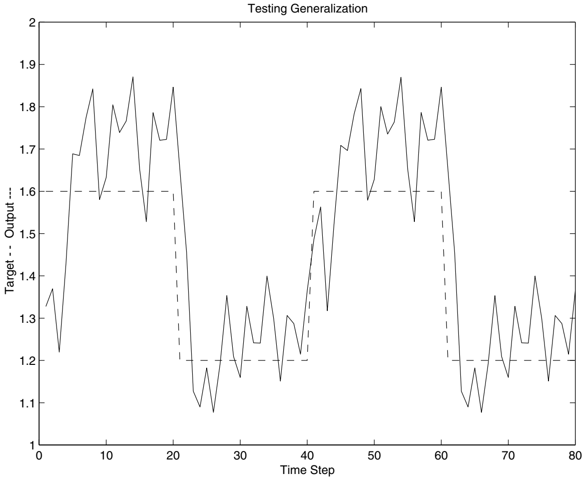
The following code defines a new wave form made up of two repetitions of a sine wave with amplitude 1.6 and another with amplitude 1.2.

```
p3 = sin(1:20)*1.6;
t3 = ones(1,20)*1.6;
p4 = sin(1:20)*1.2;
t4 = ones(1,20)*1.2;
pg = [p3 p4 p3 p4];
tg = [t3 t4 t3 t4];
pgseq = con2seq(pg);
```

The input sequence pg and target sequence tg are used to test the ability of our network to generalize to new amplitudes.

Once again the function sim is used to simulate the Elman network and the results are plotted.

```
a = sim(net,pgseq);
```



This time the network did not do as well. It seems to have a vague idea as to what it should do, but is not very accurate!

Improved generalization could be obtained by training the network on more amplitudes than just 1.0 and 2.0. The use of three or four different wave forms with different amplitudes can result in a much better amplitude detector.

## Improving Performance

Run appelm1 to see plots similar to those above. Then make a copy of this file and try improving the network by adding more neurons to the recurrent layer, using longer training times, and giving the network more examples in its training data.

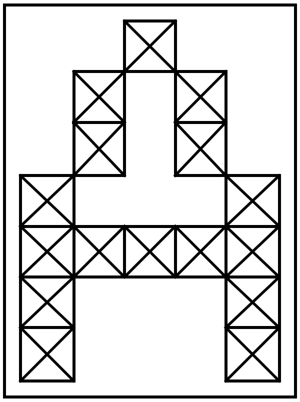
# Appcr1: Character Recognition

It is often useful to have a machine perform pattern recognition. In particular, machines that can read symbols are very cost effective. A machine that reads banking checks can process many more checks than a human being in the same time. This kind of application saves time and money, and eliminates the requirement that a human perform such a repetitive task. The script `appcr1` demonstrates how character recognition can be done with a backpropagation network.

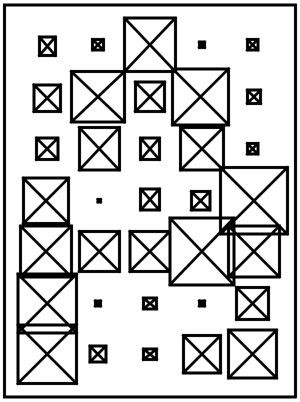
## Problem Statement

A network is to be designed and trained to recognize the 26 letters of the alphabet. An imaging system that digitizes each letter centered in the system's field of vision is available. The result is that each letter is represented as a 5 by 7 grid of boolean values.

For example, here is the letter A.



However, the imaging system is not perfect and the letters may suffer from noise.



Perfect classification of ideal input vectors is required, and reasonably accurate classification of noisy vectors.

The twenty-six 35-element input vectors are defined in the function `prprob` as a matrix of input vectors called `alphabet`. The target vectors are also defined in this file with a variable called `targets`. Each target vector is a 26-element vector with a 1 in the position of the letter it represents, and 0's everywhere else. For example, the letter A is to be represented by a 1 in the first element (as A is the first letter of the alphabet), and 0's in elements two through twenty-six.

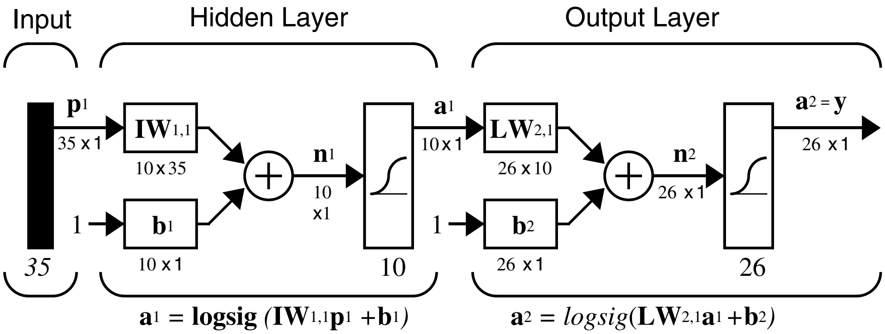
## Neural Network

The network receives the 35 Boolean values as a 35-element input vector. It is then required to identify the letter by responding with a 26-element output vector. The 26 elements of the output vector each represent a letter. To operate correctly, the network should respond with a 1 in the position of the letter being presented to the network. All other values in the output vector should be 0.

In addition, the network should be able to handle noise. In practice, the network does not receive a perfect Boolean vector as input. Specifically, the network should make as few mistakes as possible when classifying vectors with noise of mean 0 and standard deviation of 0.2 or less.

### Architecture

The neural network needs 35 inputs and 26 neurons in its output layer to identify the letters. The network is a two-layer `log-sigmoid/log-sigmoid` network. The log-sigmoid transfer function was picked because its output range (0 to 1) is perfect for learning to output boolean values.



The hidden (first) layer has 10 neurons. This number was picked by guesswork and experience. If the network has trouble learning, then neurons can be added to this layer.

The network is trained to output a 1 in the correct position of the output vector and to fill the rest of the output vector with 0's. However, noisy input vectors may result in the network not creating perfect 1's and 0's. After the network is trained the output is passed through the competitive transfer function `compet`. This makes sure that the output corresponding to the letter most like the noisy input vector takes on a value of 1, and all others have a value of 0. The result of this post-processing is the output that is actually used.

### Initialization

The two-layer network is created with `newff`.

```
S1 = 10;
[R,Q] = size(alphabet);
[S2,Q] = size(targets);
P = alphabet;
net = newff(minmax(P),[S1 S2],{'logsig' 'logsig'},'traingdx');
```

### Training

To create a network that can handle noisy input vectors it is best to train the network on both ideal and noisy vectors. To do this, the network is first trained on ideal vectors until it has a low sum-squared error.

Then, the network is trained on 10 sets of ideal and noisy vectors. The network is trained on two copies of the noise-free alphabet at the same time as it is trained on noisy vectors. The two copies of the noise-free alphabet are used to maintain the network’s ability to classify ideal input vectors.

Unfortunately, after the training described above the network may have learned to classify some difficult noisy vectors at the expense of properly classifying a noise-free vector. Therefore, the network is again trained on just ideal vectors. This ensures that the network responds perfectly when presented with an ideal letter.

All training is done using backpropagation with both adaptive learning rate and momentum with the function `trainbpx`.

### Training Without Noise

The network is initially trained without noise for a maximum of 5000 epochs or until the network sum-squared error falls beneath 0.1.

```
P = alphabet;
T = targets;
net.performFcn = 'sse';
net.trainParam.goal = 0.1;
net.trainParam.show = 20;
net.trainParam.epochs = 5000;
net.trainParam.mc = 0.95;
[net,tr] = train(net,P,T);
```

### Training with Noise

To obtain a network not sensitive to noise, we trained with two ideal copies and two noisy copies of the vectors in `alphabet`. The target vectors consist of four copies of the vectors in `target`. The noisy vectors have noise of mean 0.1 and 0.2 added to them. This forces the neuron to learn how to properly identify noisy letters, while requiring that it can still respond well to ideal vectors.

To train with noise, the maximum number of epochs is reduced to 300 and the error goal is increased to 0.6, reflecting that higher error is expected because more vectors (including some with noise), are being presented.

```
netn = net;
netn.trainParam.goal = 0.6;
netn.trainParam.epochs = 300;
```

```
T = [targets targets targets targets];
for pass = 1:10
P = [alphabet, alphabet, ...
      (alphabet + randn(R,Q)*0.1), ...
      (alphabet + randn(R,Q)*0.2)];
[netn,tr] = train(netn,P,T);
end
```

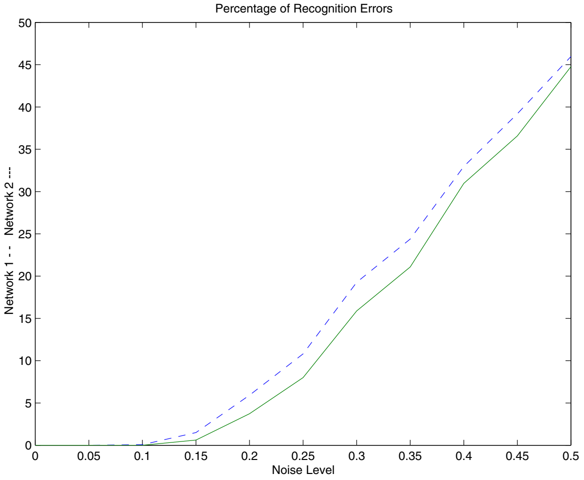
### Training Without Noise Again

Once the network is trained with noise, it makes sense to train it without noise once more to ensure that ideal input vectors are always classified correctly. Therefore, the network is again trained with code identical to the “Training Without Noise” on page 11-19.

## System Performance

The reliability of the neural network pattern recognition system is measured by testing the network with hundreds of input vectors with varying quantities of noise. The script file appcr1 tests the network at various noise levels, and then graphs the percentage of network errors versus noise. Noise with a mean of 0 and a standard deviation from 0 to 0.5 is added to input vectors. At each noise level, 100 presentations of different noisy versions of each letter are made and the network’s output is calculated. The output is then passed through the competitive transfer function so that only one of the 26 outputs (representing the letters of the alphabet), has a value of 1.

The number of erroneous classifications is then added and percentages are obtained.



The solid line on the graph shows the reliability for the network trained with and without noise. The reliability of the same network when it had only been trained without noise is shown with a dashed line. Thus, training the network on noisy input vectors greatly reduces its errors when it has to classify noisy vectors.

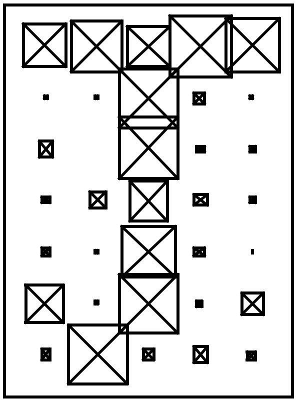
The network did not make any errors for vectors with noise of mean 0.00 or 0.05. When noise of mean 0.2 was added to the vectors both networks began making errors.

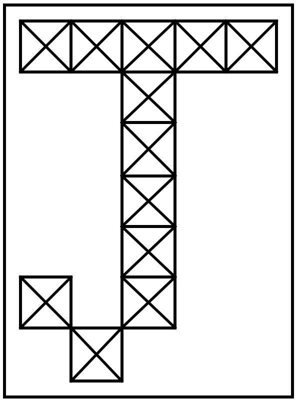
If a higher accuracy is needed, the network can be trained for a longer time or retrained with more neurons in its hidden layer. Also, the resolution of the input vectors can be increased to a 10-by-14 grid. Finally, the network could be trained on input vectors with greater amounts of noise if greater reliability were needed for higher levels of noise.

To test the system, a letter with noise can be created and presented to the network.

```
noisyJ = alphabet(:,10)+randn(35,1) * 0.2;
plotchar(noisyJ);
A2 = sim(net,noisyJ);
A2 = compet(A2);
answer = find(compet(A2) == 1);
plotchar(alphabet(:,answer));
```

Here is the noisy letter and the letter the network picked (correctly).





## Summary

This problem demonstrates how a simple pattern recognition system can be designed. Note that the training process did not consist of a single call to a training function. Instead, the network was trained several times on various input vectors.

In this case, training a network on different sets of noisy vectors forced the network to learn how to deal with noise, a common problem in the real world.

12

# Advanced Topics

## Custom Networks

The Neural Network Toolbox is designed to allow for many kinds of networks. This makes it possible for many functions to use the same network object data type.

Here are all the standard network creation functions in the toolbox.

| New Networks | |
|---|---|
| newc | Create a competitive layer. |
| newcf | Create a cascade-forward backpropagation network. |
| newelm | Create an Elman backpropagation network. |
| newff | Create a feed-forward backpropagation network. |
| newfftd | Create a feed-forward input-delay backprop network. |
| newgrnn | Design a generalized regression neural network. |
| newhop | Create a Hopfield recurrent network. |
| newlin | Create a linear layer. |
| newlind | Design a linear layer. |
| newlvq | Create a learning vector quantization network |
| newp | Create a perceptron. |
| newpnn | Design a probabilistic neural network. |
| newrb | Design a radial basis network. |
| newrbe | Design an exact radial basis network. |
| newsom | Create a self-organizing map. |

This flexibility is possible because we have an object-oriented representation for networks. The representation allows various architectures to be defined and allows various algorithms to be assigned to those architectures.

To create custom networks, start with an empty network (obtained with the `network` function) and set its properties as desired.

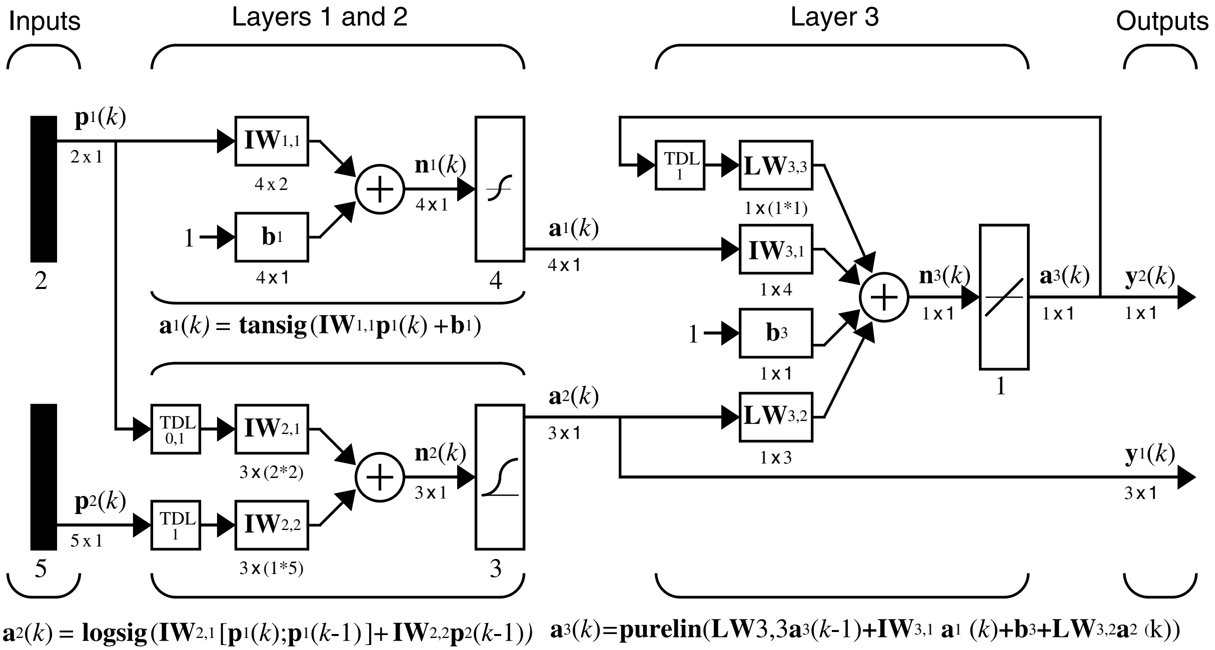
```
   network  - Create a custom neural network.
```

The network object consists of many properties that you can set to specify the structure and behavior of your network. See Chapter 13, “Network Object Reference” for descriptions of all network properties.

The following sections demonstrate how to create a custom network by using these properties.

## Custom Network

Before you can build a network you need to know what it looks like. For dramatic purposes (and to give the toolbox a workout) this section leads you through the creation of the wild and complicated network shown below.



Each of the two elements of the first network input is to accept values ranging between 0 and 10. Each of the five elements of the second network input ranges from -2 to 2.

Before you can complete your design of this network, the algorithms it employs for initialization and training must be specified.

We agree here that each layer's weights and biases are initialized with the Nguyen-Widrow layer initialization method (`initnw`). Also, the network is trained with the Levenberg-Marquardt backpropagation (`trainlm`), so that, given example input vectors, the outputs of the third layer learn to match the associated target vectors with minimal mean squared error (`mse`).

## Network Definition

The first step is to create a new network. Type in the following code to create a network and view its many properties.

```
net = network
```

### Architecture Properties

The first group of properties displayed are labeled `architecture` properties. These properties allow you to select of the number of inputs and layers, and their connections.

**Number of Inputs and Layers.** The first two properties displayed are `numInputs` and `numLayers`. These properties allow us to select how many inputs and layers we want our network to have.

```
net =

    Neural Network object:

    architecture:

         numInputs: 0
         numLayers: 0
         ...
```

Note that the network has no inputs or layers at this time.

Change that by setting these properties to the number of inputs and number of layers in our custom network diagram.

```
net.numInputs = 2;
net.numLayers = 3;
```

Note that `net.numInputs` is the number of input sources, not the number of elements in an input vector (`net.inputs{i}.size`).

**Bias Connections.** Type `net` and press **Return** to view its properties again. The network now has two inputs and three layers.

```
net =

    Neural Network object:

    architecture:

         numInputs: 2
         numLayers: 3
```

Now look at the next five properties.

```
       biasConnect: [0; 0; 0]
      inputConnect: [0 0; 0 0; 0 0]
      layerConnect: [0 0 0; 0 0 0; 0 0 0]
     outputConnect: [0 0 0]
     targetConnect: [0 0 0]
```

These matrices of 1's and 0's represent the presence or absence of bias, input weight, layer weight, output, and target connections. They are currently all zeros, indicating that the network does not have any such connections.

Note that the bias connection matrix is a 3-by-1 vector. To create a bias connection to the *i*th layer you can set `net.biasConnect(i)` to `1`. Specify that the first and third layer's are to have bias connections, as our diagram indicates, by typing in the following code.

```
net.biasConnect(1) = 1;
net.biasConnect(3) = 1;
```

Note that you could also define those connections with a single line of code.

```
net.biasConnect = [1; 0; 1];
```

**Input and Layer Weight Connections.** The input connection matrix is 3-by-2, representing the presence of connections from two sources (the two inputs) to three destinations (the three layers). Thus, `net.inputConnect(i,j)` represents the presence of an input weight connection going to the *i*th layer from the *j*th input.

To connect the first input to the first and second layers, and the second input to the second layer (as is indicated by the custom network diagram), type

```
net.inputConnect(1,1) = 1;
net.inputConnect(2,1) = 1;
net.inputConnect(2,2) = 1;
```

or this single line of code:

```
net.inputConnect = [1 0; 1 1; 0 0];
```

Similarly, `net.layerConnect(i.j)` represents the presence of a layer-weight connection going to the *i*th layer from the *j*th layer. Connect layers 1, 2, and 3 to layer 3 as follows.

```
net.layerConnect = [0 0 0; 0 0 0; 1 1 1];
```

**Output and Target Connections.** Both the output and target connection matrices are 1-by-3 matrices, indicating that they connect to one destination (the external world) from three sources (the three layers).

To connect layers 2 and 3 to network outputs, type

```
net.outputConnect = [0 1 1];
```

To give layer 3 a target connection, type

```
net.targetConnect = [0 0 1];
```

The layer 3 target is compared to the output of layer 3 to generate an error for use when measuring the performance of the network, or when updating the network during training or adaption.

### Number of Outputs and Targets

Type `net` and press **Enter** to view the updated properties. The final four architecture properties are read-only values, which means their values are determined by the choices we make for other properties. The first two read-only properties have the following values.

```
numOutputs: 2  (read-only)
numTargets: 1  (read-only)
```

By defining output connections from layers 2 and 3, and a target connection from layer 3, you specify that the network has two outputs and one target.

### Subobject Properties

The next group of properties is

```
subobject structures:

            inputs: {2x1 cell} of inputs
            layers: {3x1 cell} of layers
           outputs: {1x3 cell} containing 2 outputs
           targets: {1x3 cell} containing 1 target
            biases: {3x1 cell} containing 2 biases
      inputWeights: {3x2 cell} containing 3 input weights
      layerWeights: {3x3 cell} containing 3 layer weights
```

### Inputs

When you set the number of inputs (`net.numInputs`) to 2, the `inputs` property becomes a cell array of two input structures. Each *i*th input structure (`net.inputs{i}`) contains addition properties associated with the *i*th input.

To see how the input structures are arranged, type

```
net.inputs
ans =

    [1x1 struct]
    [1x1 struct]
```

To see the properties associated with the first input, type

```
net.inputs{1}
```

The properties appear as follows.

```
ans =

       range: [0 1]
        size: 1
    userdata: [1x1 struct]
```

Note that the `range` property only has one row. This indicates that the input has only one element, which varies from 0 to 1. The `size` property also indicates that this input has just one element.

The first input vector of the custom network is to have two elements ranging from 0 to 10. Specify this by altering the range property of the first input as follows.

```
net.inputs{1}.range = [0 10; 0 10];
```

If we examine the first input's structure again, we see that it now has the correct size, which was inferred from the new `range` values.

```
ans =

       range: [2x2 double]
        size: 2
    userdata: [1x1 struct]
```

Set the second input vector ranges to be from -2 to 2 for five elements as follows.

```
net.inputs{2}.range = [-2 2; -2 2; -2 2; -2 2; -2 2];
```

**Layers.** When we set the number of layers (`net.numLayers`) to 3, the layers property becomes a cell array of three-layer structures. Type the following line of code to see the properties associated with the first layer.

```
net.layers{1}

ans =

     dimensions: 1
    distanceFcn: 'dist'
      distances: 0
        initFcn: 'initwb'
    netInputFcn: 'netsum'
      positions: 0
           size: 1
    topologyFcn: 'hextop'
    transferFcn: 'purelin'
       userdata: [1x1 struct]
```

Type the following three lines of code to change the first layer's size to 4 neurons, its transfer function to `tansig`, and its initialization function to the Nguyen-Widrow function as required for the custom network diagram.

```
net.layers{1}.size = 4;
net.layers{1}.transferFcn = 'tansig';
net.layers{1}.initFcn = 'initnw';
```

The second layer is to have three neurons, the `logsig` transfer function, and be initialized with `initnw`. Thus, set the second layer's properties to the desired values as follows.

```
net.layers{2}.size = 3;
net.layers{2}.transferFcn = 'logsig';
net.layers{2}.initFcn = 'initnw';
```

The third layer's size and transfer function properties don't need to be changed since the defaults match those shown in the network diagram. You only need to set its initialization function as follows.

```
net.layers{3}.initFcn = 'initnw';
```

**Output and Targets.** Take a look at how the `outputs` property is arranged with this line of code.

```
net.outputs
ans = 

      []    [1x1 struct]    [1x1 struct]
```

Note that `outputs` contains two output structures, one for layer 2 and one for layer 3. This arrangement occurs automatically when `net.outputConnect` was set to `[0 1 1]`.

View the second layer's output structure with the following expression.

```
net.outputs{2}
ans = 

        size: 3
    userdata: [1x1 struct]
```

The `size` is automatically set to 3 when the second layer's size (`net.layers{2}.size`) is set to that value. Take a look at the third layer's output structure if you want to verify that it also has the correct `size`.

Similarly, `targets` contains one structure representing the third layer's target. Type these two lines of code to see how `targets` is arranged and to view the third layer's target properties.

```
net.targets
```

```
ans =
     []    []    [1x1 struct]

net.targets{3}
ans =
        size: 1
    userdata: [1x1 struct]
```

**Biases, Input Weights, and Layer Weights.** Enter the following lines of code to see how bias and weight structures are arranged.

```
net.biases
net.inputWeights
net.layerWeights
```

Here are the results for typing `net.biases`.

```
ans =
    [1x1 struct]
              []
    [1x1 struct]
```

If you examine the results you will note that each contains a structure where the corresponding connections (`net.biasConnect`, `net.inputConnect`, and `net.layerConnect`) contain a 1.

Take a look at their structures with these lines of code.

```
net.biases{1}
net.biases{3}
net.inputWeights{1,1}
net.inputWeights{2,1}
net.inputWeights{2,2}
net.layerWeights{3,1}
net.layerWeights{3,2}
net.layerWeights{3,3}
```

For example, typing `net.biases{1}` results in the following output.

```
ans =
       initFcn: ''
         learn: 1
      learnFcn: ''
    learnParam: ''
```

```
              size: 4
          userdata: [1x1 struct]
```

Specify the weights tap delay lines in accordance with the network diagram, by setting each weights delays property.

```
net.inputWeights{2,1}.delays = [0 1];
net.inputWeights{2,2}.delays = 1;
net.layerWeights{3,3}.delays = 1;
```

### Network Functions

Type net and press return again to see the next set of properties.

```
    functions:

              adaptFcn: (none)
               initFcn: (none)
            performFcn: (none)
              trainFcn: (none)
```

Each of these properties defines a function for a basic network operation.

Set the initialization function to initlay so the network initializes itself according to the layer initialization functions that we have already set to initnw the Nguyen-Widrow initialization function.

```
net.initFcn = 'initlay';
```

This meets the initialization requirement of our network.

Set the performance function to mse (mean squared error) and the training function to trainlm (Levenberg-Marquardt backpropagation) to meet the final requirement of the custom network.

```
net.performFcn = 'mse';
net.trainFcn = 'trainlm';
```

### Weight and Bias Values

Before initializing and training the network, take a look at the final group of network properties (aside from the userdata property).

```
weight and bias values:

               IW: {3x2 cell} containing 3 input weight matrices
```

```
     LW: {3x3 cell} containing 3 layer weight matrices
      b: {3x1 cell} containing 2 bias vectors
```

These cell arrays contain weight matrices and bias vectors in the same positions that the connection properties (`net.inputConnect`, `net.layerConnect`, `net.biasConnect`) contain 1's and the subobject properties (`net.inputWeights`, `net.layerWeights`, `net.biases`) contain structures.

Evaluating each of the following lines of code reveals that all the bias vectors and weight matrices are set to zeros.

```
net.IW{1,1}, net.IW{2,1}, net.IW{2,2}
net.IW{3,1}, net.LW{3,2}, net.LW{3,3}
net.b{1}, net.b{3}
```

Each input weight `net.IW{i,j}`, layer weight `net.LW{i,j}`, and bias vector `net.b{i}` has as many rows as the size of the *i*th layer (`net.layers{i}.size`).

Each input weight `net.IW{i,j}` has as many columns as the size of the *j*th input (`net.inputs{j}.size`) multiplied by the number of its delay values (`length(net.inputWeights{i,j}.delays)`).

Likewise, each layer weight has as many columns as the size of the *j*th layer (`net.layers{j}.size`) multiplied by the number of its delay values (`length(net.layerWeights{i,j}.delays)`).

## Network Behavior

### Initialization

Initialize your network with the following line of code.

```
net = init(net)
```

Reference the network's biases and weights again to see how they have changed.

```
net.IW{1,1}, net.IW{2,1}, net.IW{2,2}
net.IW{3,1}, net.LW{3,2}, net.LW{3,3}
net.b{1}, net.b{3}
```

For example,

```
net.IW{1,1}
```

```
ans =
   -0.3040    0.4703
   -0.5423   -0.1395
    0.5567    0.0604
    0.2667    0.4924
```

### Training

Define the following cell array of two input vectors (one with two elements, one with five) for two time steps (i.e., two columns).

```
P = {[0; 0] [2; 0.5]; [2; -2; 1; 0; 1] [-1; -1; 1; 0; 1]}
```

We want the network to respond with the following target sequence.

```
T = {1 -1}
```

Before training, we can simulate the network to see whether the initial network's response Y is close to the target T.

```
Y = sim(net,P)

Y =

    [3x1 double]    [3x1 double]
    [    0.0456]    [    0.2119]
```

The second row of the cell array `Y` is the output sequence of the second network output, which is also the output sequence of the third layer. The values you got for the second row may differ from those shown due to different initial weights and biases. However, they will almost certainly not be equal to our targets T, which is also true of the values shown.

The next task is to prepare the training parameters. The following line of code displays the default Levenberg-Marquardt training parameters (which were defined when we set `net.trainFcn` to `trainlm`).

```
net.trainParam
```

The following properties should be displayed.

```
ans =

       epochs: 100
         goal: 0
     max_fail: 5
```

```
  mem_reduc: 1
   min_grad: 1.0000e-10
         mu: 1.0000e-03
     mu_dec: 0.1000
     mu_inc: 10
     mu_max: 1.0000e+10
       show: 25
       time:
```

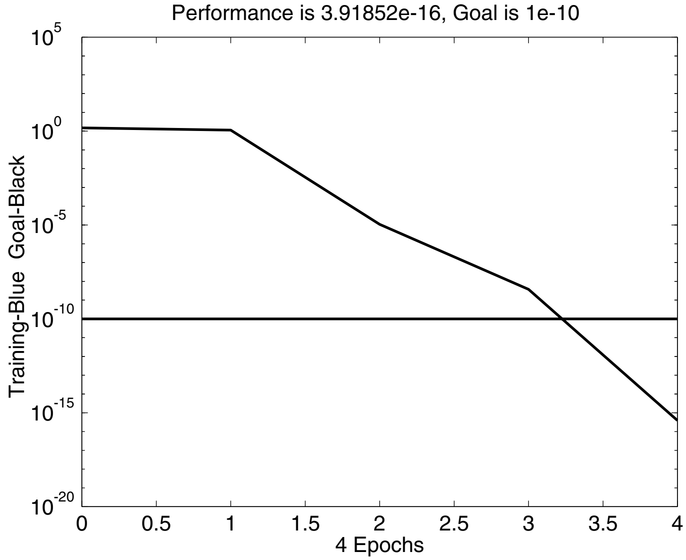
Change the performance goal to `1e-10`.

```
net.trainParam.goal = 1e-10;
```

Next, train the network with the following call.

```
net = train(net,P,T);
```

Below is a typical training plot.



After training you can simulate the network to see if it has learned to respond correctly.

```
Y = sim(net,P)
```

```
Y =

    [3x1 double]    [3x1 double]
    [    1.0000]    [   -1.0000]
```

Note that the second network output (i.e., the second row of the cell array Y), which is also the third layer's output, does match the target sequence T.

# Additional Toolbox Functions

Most toolbox functions are explained in chapters dealing with networks that use them. However, some functions are not used by toolbox networks, but are included as they may be useful to you in creating custom networks.

Each of these is documented in Chapter 14, “Reference.” However, the notes given below may also prove to be helpful.

## Initialization Functions

### randnc

This weight initialization function generates random weight matrices whose columns are normalized to a length of 1.

### randnr

This weight initialization function generates random weight matrices whose rows are normalized to a length of 1.

## Transfer Functions

### satlin

This transfer function is similar to `satlins`, but has a linear region going from 0 to 1 (instead of -1 to 1), and minimum and maximum values of 0 and 1 (instead of -1 and 1).

### softmax

This transfer function is a softer version of the hard competitive transfer function `compet`. The neuron with the largest net input gets an output closest to one, while other neurons have outputs close to zero.

### tribas

The triangular-basis transfer function is similar to the radial-basis transfer function `radbas`, but has a simpler shape.

## Learning Functions

### learnh

The Hebb weight learning function increases weights in proportion to the product, the weights input, and the neuron's output. This allows neurons to learn associations between their inputs and outputs.

### learnhd

The Hebb-with-decay learning function is similar to the Hebb function, but adds a term that decreases weights each time step exponentially. This weight decay allows neurons to forget associations that are not reinforced regularly, and solves the problem that the Hebb function has with weights growing without bounds.

### learnis

The instar weight learning function moves a neuron's weight vector towards the neuron's input vector with steps proportional to the neuron's output. This function allows neurons to learn association between input vectors and their outputs.

### learnos

The outstar weight learning function acts in the opposite way as the instar learning rule. The outstar rule moves the weight vector coming from an input toward the output vector of a layer of neurons with step sizes proportional to the input value. This allows inputs to learn to recall vectors when stimulated.

# Custom Functions

The toolbox allows you to create and use many kinds of functions. This gives you a great deal of control over the algorithms used to initialize, simulate, and train; and allow adaption for your networks.

The following sections describe how to create your own versions of these kinds of functions:

- Simulation functions
  - transfer functions
  - net input functions
  - weight functions
- Initialization functions
  - network initialization functions
  - layer initialization functions
  - weight and bias initialization functions
- Learning functions
  - network training functions
  - network adapt functions
  - network performance functions
  - weight and bias learning functions
- Self-organizing map functions
  - topology functions
  - distance functions

## Simulation Functions

You can create three kinds of simulation functions: transfer, net input, and weight functions. You can also provide associated derivative functions to enable backpropagation learning with your functions.

### Transfer Functions

Transfer functions calculate a layer's output vector (or matrix) `A`, given its net input vector (or matrix) `N`. The only constraint on the relationship between the output and net input is that the output must have the same dimensions as the input.

Once defined, you can assign your transfer function to any layer of a network. For example, the following line of code assigns the transfer function yourtf to the second layer of a network.

```
net.layers{2}.transferFcn = 'yourtf';
```

Your transfer function then is used whenever you simulate your network.

```
[Y,Pf,Af] = sim(net,P,Pi,Ai)
```

To be a valid transfer function, your function must calculate outputs A from net inputs N as follows,

```
A = yourtf(N)
```

where:

- N is an $S$ x $Q$ matrix of $Q$ net input (column) vectors.
- A is an $S$ x $Q$ matrix of $Q$ output (column) vectors.

Your transfer function must also provide information about itself, using this calling format,

```
info = yourtf(code)
```

where the correct information is returned for each of the following string codes:

- 'version' - Returns the Neural Network Toolbox version (3.0).
- 'deriv' - Returns the name of the associated derivative function.
- 'output' - Returns the output range.
- 'active' - Returns the active input range.

The toolbox contains an example custom transfer function called mytf. Enter the following lines of code to see how it is used.

```
help mytf
n = -5:.1:5;
a = mytf(n);
plot(n,a)
mytf('deriv')
```

Enter the following command to see how mytf is implemented.

```
type mytf
```

You can use mytf as a template to create your own transfer function.

**Transfer Derivative Functions.** If you want to use backpropagation with your custom transfer function, you need to create a custom derivative function for it. The function needs to calculate the derivative of the layer's output with respect to its net input,

```
dA_dN = yourdtf(N,A)
```

where:

- N is an $S \times Q$ matrix of $Q$ net input (column) vectors.
- A is an $S \times Q$ matrix of $Q$ output (column) vectors.
- dA_dN is the $S \times Q$ derivative *d*A/*d*N.

This only works for transfer functions whose output elements are independent. In other words, where each A(i) is only a function of N(i). Otherwise, a three-dimensional array is required to store the derivatives in the case of multiple vectors (instead of a matrix as defined above). Such 3-D derivatives are not supported at this time.

To see how the example custom transfer derivative function mydtf works, type

```
help mydtf
da_dn = mydtf(n,a)
subplot(2,1,1), plot(n,a)
subplot(2,1,2), plot(n,dn_da)
```

Use this command to see how mydtf was implemented.

```
type mydtf
```

You can use mydtf as a template to create your own transfer derivative functions.

### Net Input Functions

Net input functions calculate a layer's net input vector (or matrix) N, given its weighted input vectors (or matrices) Zi. The only constraints on the relationship between the net input and the weighted inputs are that the net input must have the same dimensions as the weighted inputs, and that the function cannot be sensitive to the order of the weight inputs.

Once defined, you can assign your net input function to any layer of a network. For example, the following line of code assigns the transfer function `yournif` to the second layer of a network.

```
net.layers{2}.netInputFcn = 'yournif';
```

Your net input function then is used whenever you simulate your network.

```
[Y,Pf,Af] = sim(net,P,Pi,Ai)
```

To be a valid net input function your function must calculate outputs `A` from net inputs `N` as follows,

```
N = yournif(Z1,Z2,...)
```

where

- `Zi` is the $i$th $S \times Q$ matrix of $Q$ weighted input (column) vectors.
- `N` is an $S \times Q$ matrix of $Q$ net input (column) vectors.

Your net input function must also provide information about itself using this calling format,

```
info = yournif(code)
```

where the correct information is returned for each of the following string codes:

- `'version'` - Returns the Neural Network Toolbox version (3.0).
- `'deriv'` - Returns the name of the associated derivative function.

The toolbox contains an example custom net input function called `mynif`. Enter the following lines of code to see how it is used.

```
help mynif
z1 = rand(4,5);
z2 = rand(4,5);
z3 = rand(4,5);
n = mynif(z1,z2,z3)
mynif('deriv')
```

Enter the following command to see how `mynif` is implemented.

```
type mynif
```

You can use mynif as a template to create your own net input function.

**Net Input Derivative Functions.** If you want to use backpropagation with your custom net input function, you need to create a custom derivative function for it. It needs to calculate the derivative of the layer's net input with respect to any of its weighted inputs,

```
dN_dZ = dtansig(Z,N)
```

where:

- Z is one of the $S \times Q$ matrices of $Q$ weighted input (column) vectors.
- N is an $S \times Q$ matrix of $Q$ net input (column) vectors.
- dN_dZ is the $S \times Q$ derivative $d$N/$d$Z.

To see how the example custom net input derivative function mydtf works, type

```
help mydnif
dn_dz1 = mydnif(z1,n)
dn_dz2 = mydnif(z1,n)
dn_dz3 = mydnif(z1,n)
```

Use this command to see how mydtf was implemented.

```
type mydnif
```

You can use mydnif as a template to create your own net input derivative functions.

## Weight Functions

Weight functions calculate a weighted input vector (or matrix) Z, given an input vector (or matrices) P and a weight matrix W.

Once defined, you can assign your weight function to any input weight or layer weight of a network. For example, the following line of code assigns the weight function yourwf to the weight going to the second layer from the first input of a network.

```
net.inputWeights{2,1}.weightFcn = 'yourwf';
```

Your weight function is used whenever you simulate your network.

```
[Y,Pf,Af] = sim(net,P,Pi,Ai)
```

To be a valid weight function your function must calculate weight inputs Z from inputs P and a weight matrix W as follows,

```
Z = yourwf(W,P)
```

where:

- W is an $S \times R$ weight matrix.
- P is an $R \times Q$ matrix of $Q$ input (column) vectors.
- Z is an $S \times Q$ matrix of $Q$ weighted input (column) vectors.

Your net input function must also provide information about itself using this calling format,

```
info = yourwf(code)
```

where the correct information is returned for each of the following string codes:

- 'version' - Returns the Neural Network Toolbox version (3.0).
- 'deriv' - Returns the name of the associated derivative function.

The toolbox contains an example custom weight called mywf. Enter the following lines of code to see how it is used.

```
help mywf
w = rand(1,5);
p = rand(5,1);
z = mywf(w,p);
mywf('deriv')
```

Enter the following command to see how mywf is implemented.

```
type mywf
```

You can use mywf as a template to create your own weight functions.

**Weight Derivative Functions.** If you want to use backpropagation with your custom weight function, you need to create a custom derivative function for it. It needs to calculate the derivative of the weight inputs with respect to both the input and weight,

```
dZ_dP = mydwf('p',W,P,Z)
dZ_dW = mydwf('w',W,P,Z)
```

where:

- W is an $S \times R$ weight matrix.
- P is an $R \times Q$ matrix of $Q$ input (column) vectors.
- Z is an $S \times Q$ matrix of $Q$ weighted input (column) vectors.
- dZ_dP is the $S \times R$ derivative $d$Z/$d$P.
- dZ_dW is the $R \times Q$ derivative $d$Z/$d$W.

This only works for weight functions whose output consists of a sum of $i$ term, where each $i$th term is a function of only W(i) and P(i). Otherwise a three-dimensional array is required to store the derivatives in the case of multiple vectors (instead of a matrix as defined above). Such 3-D derivatives are not supported at this time.

To see how the example custom net input derivative function mydwf works, type

```
help mydwf
dz_dp = mydwf('p',w,p,z)
dz_dw = mydwf('w',w,p,z)
```

Use this command to see how mydwf is implemented.

```
type mydwf
```

You can use mydwf as a template to create your own net input derivative function.

## Initialization Functions

You can create three kinds of initialization functions: network, layer, and weight/bias initialization.

### Network Initialization Functions

The most general kind of initialization function is the network initialization function which sets all the weights and biases of a network to values suitable as a starting point for training or adaption.

Once defined, you can assign your network initialization function to a network.

```
net.initFcn = 'yournif';
```

Your network initialization function is used whenever you initialize your network.

```
net = init(net)
```

To be a valid network initialization function, it must take and return a network.

```
net = yournif(net)
```

Your function can set the network's weight and bias values in any way you want. However, you should be careful not to alter any other properties, or to set the weight matrices and bias vectors of the wrong size. For performance reasons, `init` turns off the normal type checking for network properties before calling your initialization function. So if you set a weight matrix to the wrong size, it won't immediately generate an error, but could cause problems later when you try to simulate or train the network.

You can examine the implementation of the toolbox function `initlay` if you are interested in creating your own network initialization function.

### Layer Initialization Functions

The layer initialization function sets all the weights and biases of a layer to values suitable as a starting point for training or adaption.

Once defined, you can assign your layer initialization function to a layer of a network. For example, the following line of code assigns the layer initialization function `yourlif` to the second layer of a network.

```
net.layers{2}.initFcn = 'yourlif';
```

Layer initialization functions are only called to initialize a layer if the network initialization function (`net.initFcn`) is set to the toolbox function `initlay`. If this is the case, then your function is used to initialize the layer whenever you initialize your network with `init`.

```
net = init(net)
```

To be a valid layer initialization function, it must take a network and a layer index `i`, and return the network after initializing the *i*th layer.

```
net = yournif(net,i)
```

Your function can then set the *i*th layer's weight and bias values in any way you see fit. However, you should be careful not to alter any other properties, or to set the weight matrices and bias vectors to the wrong size.

If you are interested in creating your own layer initialization function, you can examine the implementations of the toolbox functions `initwb` and `initnw`.

### Weight and Bias Initialization Functions

The weight and bias initialization function sets all the weights and biases of a weight or bias to values suitable as a starting point for training or adaption.

Once defined, you can assign your initialization function to any weight and bias in a network. For example, the following lines of code assign the weight and bias initialization function `yourwbif` to the second layer's bias, and the weight coming from the first input to the second layer.

```
net.biases{2}.initFcn = 'yourwbif';
net.inputWeights{2,1}.initFcn = 'yourwbif';
```

Weight and bias initialization functions are only called to initialize a layer if the network initialization function (`net.initFcn`) is set to the toolbox function `initlay`, and the layer's initialization function (`net.layers{i}.initFcn`) is set to the toolbox function `initwb`. If this is the case, then your function is used to initialize the weight and biases it is assigned to whenever you initialize your network with `init`.

```
net = init(net)
```

To be a valid weight and bias initialization function, it must take a the number of neurons in a layer `S`, and a two-column matrix `PR` of $R$ rows defining the minimum and maximum values of $R$ inputs and return a new weight matrix `W`,

```
W = rands(S,PR)
```

where:

- `S` is the number of neurons in the layer.
- `PR` is an $R \times 2$ matrix defining the minimum and maximum values of $R$ inputs.
- `W` is a new $\mathtt{S} \times R$ weight matrix.

Your function also needs to generate a new bias vector as follows,

```
b = rands(S)
```

where:

- S is the number of neurons in the layer.
- b is a new $S \times 1$ bias vector.

To see how an example custom weight and bias initialization function works, type

```
help mywbif
W = mywbif(4,[0 1; -2 2])
b = mywbif(4,[1 1])
```

Use this command to see how mywbif was implemented.

```
type mywbif
```

You can use mywbif as a template to create your own weight and bias initialization function.

## Learning Functions

You can create four kinds of initialization functions: training, adaption, performance, and weight/bias learning.

### Training Functions

One kind of general learning function is a network training function. Training functions repeatedly apply a set of input vectors to a network, updating the network each time, until some stopping criteria is met. Stopping criteria can consists of a maximum number of epochs, a minimum error gradient, an error goal, etc.

Once defined, you can assign your training function to a network.

```
net.trainFcn = 'yourtf';
```

Your network initialization function is used whenever you train your network.

```
[net,tr] = train(NET,P,T,Pi,Ai)
```

To be a valid training function your function must take and return a network,

```
[net,tr] = yourtf(net,Pd,Tl,Ai,Q,TS,VV,TV)
```

where:

- Pd is an $N_l \times N_i \times \mathtt{TS}$ cell array of tap delayed inputs.
  - Each Pd{i,j,ts} is the $R^j \times (D_i^{ij} Q)$ delayed input matrix to the weight going to the $i$th layer from the $j$th input at time step ts. (Pd{i,j,ts} is an empty matrix [] if the $i$th layer doesn't have a weight from the $j$th input.)
- Tl is an $N_l \times \mathtt{TS}$ cell array of layer targets.
  - Each Tl{i,ts} is the $S^i \times Q$ target matrix for the $i$th layer. (Tl{i,ts} is an empty matrix if the $i$th layer doesn't have a target.)
- Ai is an $N_l \times LD$ cell array of initial layer delay states.
  - Each Ai{l,k} is the $S^i \times Q$ delayed $i$th layer output for time step ts = k-$LD$, where ts goes from 0 to $LD$-1.
- Q is the number of concurrent vectors.
- TS is the number of time steps.
- VV and TV are optional structures defining validation and test vectors in the same form as the training vectors defined above: Pd, Tl, Ai, Q, and TS. Note that the validation and testing Q and TS values can be different from each other and from those used by the training vectors.

The dimensions above have the following definitions:

- $N_l$ is the number of network layers (net.numLayers).
- $N_i$ is the number of network inputs (net.numInputs).
- $R^j$ is the size of the $j$th input (net.inputs{j}.size).
- $S^i$ is the size of the $i$th layer (net.layers{i}.size)
- $LD$ is the number of layer delays (net.numLayerDelays).
- $D_i^{ij}$ is the number of delay lines associated with the weight going to the $i$th layer from the $j$th input (length(net.inputWeights{i,j}.delays)).

Your training function must also provide information about itself using this calling format,

```
info = yourtf(code)
```

where the correct information is returned for each of the following string codes:

- 'version' - Returns the Neural Network Toolbox version (3.0).
- 'pdefaults' - Returns a structure of default training parameters.

When you set the network training function (net.trainFcn) to be your function, the network's training parameters (net.trainParam) automatically is set to your default structure. Those values can be altered (or not) before training.

Your function can update the network's weight and bias values in any way you see fit. However, you should be careful not to alter any other properties, or to set the weight matrices and bias vectors to the wrong size. For performance reasons, train turns off the normal type checking for network properties before calling your training function. So if you set a weight matrix to the wrong size, it won't immediately generate an error, but will cause problems later when you try to simulate or adapt the network.

If you are interested in creating your own training function, you can examine the implementations of toolbox functions such as trainc and trainr. The help for each of these utility functions lists the input and output arguments they take.

**Utility Functions.** If you examine training functions such as trainc, traingd, and trainlm, note that they use a set of utility functions found in the nnet/nnutils directory.

These functions are not listed in Chapter 14, "Reference" because they may be altered in the future. However, you can use these functions if you are willing to take the risk that you might have to update your functions for future versions of the toolbox. Use help on each function to view the function's input and output arguments.

These two functions are useful for creating a new training record and truncating it once the final number of epochs is known:

- newtr - New training record with any number of optional fields.
- cliptr - Clip training record to the final number of epochs.

These three functions calculate network signals going forward, errors, and derivatives of performance coming back:

- calca - Calculate network outputs and other signals.
- calcerr - Calculate matrix or cell array errors.
- calcgrad - Calculate bias and weight performance gradients.

These two functions get and set a network's weight and bias values with single vectors. Being able to treat all these adjustable parameters as a single vector is often useful for implementing optimization algorithms:

- `getx` - Get all network weight and bias values as a single vector.
- `setx` - Set all network weight and bias values with a single vector.

These next three functions are also useful for implementing optimization functions. One calculates all network signals going forward, including errors and performance. One backpropagates to find the derivatives of performance as a single vector. The third function backpropagates to find the Jacobian of performance. This latter function is used by advanced optimization techniques like Levenberg-Marquardt:

- `calcperf` - Calculate network outputs, signals, and performance.
- `calcgx` - Calculate weight and bias performance gradient as a single vector.
- `calcjx` - Calculate weight and bias performance Jacobian as a single matrix.

### Adapt Functions

The other kind of the general learning function is a network adapt function. Adapt functions simulate a network, while updating the network for each time step of the input before continuing the simulation to the next input.

Once defined, you can assign your adapt function to a network.

```
net.adaptFcn = 'youraf';
```

Your network initialization function is used whenever you adapt your network.

```
[net,Y,E,Pf,Af] = adapt(NET,P,T,Pi,Ai)
```

To be a valid adapt function, it must take and return a network,

```
[net,Ac,El] = youraf(net,Pd,Tl,Ai,Q,TS)
```

where:

- `Pd` is an $N_l \times N_i \times \mathrm{TS}$ cell array of tap delayed inputs.
  - Each `Pd{i,j,ts}` is the $R^j \times (D_i^{ij} Q)$ delayed input matrix to the weight going to the *i*th layer from the *j*th input at time step `ts`. Note that (`Pd{i,j,ts}` is an empty matrix `[]` if the *i*th layer doesn't have a weight from the *j*th input.)

- `Tl` is an $N_l \times$ `TS` cell array of layer targets.
  - Each `Tl{i,ts}` is the $S^i \times Q$ target matrix for the *i*th layer. Note that (`Tl{i,ts}` is an empty matrix if the *i*th layer doesn't have a target.)
- `Ai` is an $N_l \times LD$ cell array of initial layer delay states.
  - Each `Ai{l,k}` is the $S^i \times Q$ delayed *i*th layer output for time step `ts` = `k`-*LD*, where ts goes from 0 to *LD*-1.
- `Q` is the number of concurrent vectors.
- `TS` is the number of time steps.

The dimensions above have the following definitions:

- $N_l$ is the number of network layers (`net.numLayers`).
- $N_i$ is the number of network inputs (`net.numInputs`).
- $R^j$ is the size of the jth input (`net.inputs{j}.size`).
- $S^i$ is the size of the ith layer (`net.layers{i}.size`)
- *LD* is the number of layer delays (`net.numLayerDelays`).
- $D_i^{ij}$ is the number of delay lines associated with the weight going to the *i*th layer from the *j*th input (`length(net.inputWeights{i,j}.delays)`).

Your adapt function must also provide information about itself using this calling format,

```
info = youraf(code)
```

where the correct information is returned for each of the following string codes:

- `'version'` - Returns the Neural Network Toolbox version (3.0).
- `'pdefaults'` - Returns a structure of default adapt parameters.

When you set the network adapt function (`net.adaptFcn`) to be your function, the network's adapt parameters (`net.adaptParam`) automatically is set to your default structure. Those values can then be altered (or not) before adapting.

Your function can update the network's weight and bias values in any way you see fit. However, you should be careful not to alter any other properties, or to set the weight matrices and bias vectors of the wrong size. For performance reasons, `adapt` turns off the normal type checking for network properties before calling your adapt function. So if you set a weight matrix to the wrong size, it won't immediately generate an error, but will cause problems later when you try to simulate or train the network.

If you are interested in creating your own training function, you can examine the implementation of a toolbox function such as `trains`.

**Utility Functions.** If you examine the toolbox's only adapt function `trains`, note that it uses a set of utility functions found in the `nnet/nnutils` directory. The help for each of these utility functions lists the input and output arguments they take.

These functions are not listed in Chapter 14, "Reference" because they may be altered in the future. However, you can use these functions if you are willing to take the risk that you will have to update your functions for future versions of the toolbox.

These two functions are useful for simulating a network, and calculating its derivatives of performance:

- `calca1` - New training record with any number of optional fields.
- `calce1` - Clip training record to the final number of epochs.
- `calcgrad` - Calculate bias and weight performance gradients.

### Performance Functions

Performance functions allow a network's behavior to be graded. This is useful for many algorithms, such as backpropagation, which operate by adjusting network weights and biases to improve performance.

Once defined you can assign your training function to a network.

```
net.performFcn = 'yourpf';
```

Your network initialization function will then be used whenever you train your adapt your network.

```
[net,tr] = train(NET,P,T,Pi,Ai)
[net,Y,E,Pf,Af] = adapt(NET,P,T,Pi,Ai)
```

To be a valid performance function your function must be called as follows,

```
perf = yourpf(E,X,PP)
```

where:

- `E` is either an S x Q matrix or an $N_l \times \mathrm{TS}$ cell array of layer errors.

  - Each E{i,ts} is the $S^i \times Q$ target matrix for the $i$th layer. (Tl(i,ts) is an empty matrix if the $i$th layer doesn't have a target.)

- X is an $M$ x 1 vector of all the network's weights and biases.
- PP is a structure of network performance parameters.

If E is a cell array you can convert it to a matrix as follows.

```
E = cell2mat(E);
```

Alternatively, your function must also be able to be called as follows,

```
perf = yourpf(E,net)
```

where you can get X and PP (if needed) as follows.

```
X = getx(net);
PP = net.performParam;
```

Your performance function must also provide information about itself using this calling format,

```
info = yourpf(code)
```

where the correct information is returned for each of the following string codes:

- 'version' - Returns the Neural Network Toolbox version (3.0).
- 'deriv' - Returns the name of the associated derivative function.
- 'pdefaults' - Returns a structure of default performance parameters.

When you set the network performance function (net.performFcn) to be your function, the network's adapt parameters (net.performParam) will automatically get set to your default structure. Those values can then be altered or not before training or adaption.

To see how an example custom performance function works type in these lines of code.

```
help mypf
e = rand(4,5);
x = rand(12,1);
pp = mypf('pdefaults')
perf = mypf(e,x,pp)
```

Use this command to see how mypf was implemented.

```
type mypf
```

You can use mypf as a template to create your own weight and bias initialization function.

**Performance Derivative Functions.** If you want to use backpropagation with your performance function, you need to create a custom derivative function for it. It needs to calculate the derivative of the network's errors and combined weight and bias vector, with respect to performance,

```
dPerf_dE = dmsereg('e',E,X,perf,PP)
dPerf_dX = dmsereg('x',E,X,perf,PP)
```

where:

- E is an $N_l \times \mathrm{TS}$ cell array of layer errors.
  - Each E{i,ts} is the $S^i \times Q$ target matrix for the ith layer. Note that (Tl(i,ts) is an empty matrix if the *i*th layer doesn't have a target.)
- X is an $M \times 1$ vector of all the network's weights and biases.
- PP is a structure of network performance parameters.
- dPerf_dE is the $N_l \times \mathrm{TS}$ cell array of derivatives *d*Perf/*d*E.
  - Each E{i,ts} is the $S^i \times Q$ derivative matrix for the *i*th layer. Note that (Tl(i,ts) is an empty matrix if the *i*th layer doesn't have a target.)
- dPerf_dX is the $M \times 1$ derivative *d*Perf/*d*X.

To see how the example custom performance derivative function mydpf works, type

```
help mydpf
e = {e};
dperf_de = mydpf('e',e,x,perf,pp)
dperf_dx = mydpf('x',e,x,perf,pp)
```

Use this command to see how mydpf was implemented.

```
type mydpf
```

You can use mydpf as a template to create your own performance derivative functions.

### Weight and Bias Learning Functions

The most specific kind of learning function is a weight and bias learning function. These functions are used to update individual weights and biases during learning. with some training and adapt functions.

Once defined. you can assign your learning function to any weight and bias in a network. For example, the following lines of code assign the weight and bias learning function `yourwblf` to the second layer's bias, and the weight coming from the first input to the second layer.

```
net.biases{2}.learnFcn = 'yourwblf';
net.inputWeights{2,1}.learnFcn = 'yourwblf';
```

Weight and bias learning functions are only called to update weights and biases if the network training function (`net.trainFcn`) is set to `trainb`, `trainc`, or `trainr`, or if the network adapt function (`net.adaptFcn`) is set to `trains`. If this is the case, then your function is used to update the weight and biases it is assigned to whenever you train or adapt your network with `train` or `adapt`.

```
[net,tr] = train(NET,P,T,Pi,Ai)
[net,Y,E,Pf,Af] = adapt(NET,P,T,Pi,Ai)
```

To be a valid weight and bias learning function, it must be callable as follows,

```
[dW,LS] = yourwblf(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)
```

where:

- `W` is an $S \times R$ weight matrix.
- `P` is an $R \times Q$ matrix of $Q$ input (column) vectors.
- `Z` is an $S \times Q$ matrix of $Q$ weighted input (column) vectors.
- `N` is an $S \times Q$ matrix of $Q$ net input (column) vectors.
- `A` is an $S \times Q$ matrix of $Q$ layer output (column) vectors.
- `T` is an $S \times Q$ matrix of $Q$ target (column) vectors.
- `E` is an $S \times Q$ matrix of $Q$ error (column) vectors.
- `gW` is an $S \times R$ gradient of `W` with respect to performance.
- `gA` is an $S \times Q$ gradient of `A` with respect to performance.
- `D` is an $S \times S$ matrix of neuron distances.
- `LP` is a a structure of learning parameters.

- LS is a structure of the learning state that is updated for each call. (Use a null matrix [] the first time.)
- dW is the resulting $S \times R$ weight change matrix.

Your function is called as follows to update bias vector

```
[db,LS] = yourwblf(b,ones(1,Q),Z,N,A,T,E,gW,gA,D,LP,LS)
```

where:

- S is the number of neurons in the layer.
- b is a new $S \times 1$ bias vector.

Your learning function must also provide information about itself using this calling format,

```
info = yourwblf(code)
```

where the correct information is returned for each of the following string codes:

- 'version' - Returns the Neural Network Toolbox version (3.0).
- 'deriv' - Returns the name of the associated derivative function.
- 'pdefaults' - Returns a structure of default performance parameters.

To see how an example custom weight and bias initialization function works, type

```
help mywblf
```

Use this command to see how mywbif was implemented.

```
type mywblf
```

You can use mywblf as a template to create your own weight and bias learning function.

## Self-Organizing Map Functions

There are two kinds of functions that control how neurons in self-organizing maps respond. They are topology and distance functions.

### Topology Functions

Topology functions calculate the positions of a layer's neurons given its dimensions.

Once defined, you can assign your topology function to any layer of a network. For example, the following line of code assigns the topology function yourtopf to the second layer of a network.

```
net.layers{2}.topologyFcn = 'yourtopf';
```

Your topology function is used whenever your network is trained or adapts.

```
[net,tr] = train(NET,P,T,Pi,Ai)
[net,Y,E,Pf,Af] = adapt(NET,P,T,Pi,Ai)
```

To be a valid topology function your function must calculate positions pos from dimensions dim as follows,

```
pos = yourtopf(dim1,dim2,...,dimN)
```

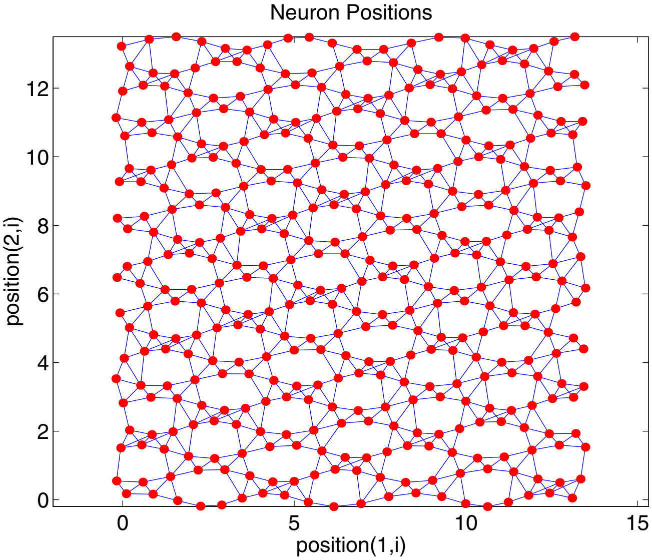
where:

- dimi is the number of neurons along the ith dimension of the layer.
- pos is an $N \times S$ matrix of $S$ position vectors, where $S$ is the total number of neurons that is defined by the product dim1*dim1*...*dimN.

The toolbox contains an example custom topology function called mytopf. Enter the following lines of code to see how it is used.

```
help mytopf
pos = mytopf(20,20);
plotsom(pos)
```

If you type that code, you get the following plot.



Enter the following command to see how mytf is implemented.

```
type mytopf
```

You can use mytopf as a template to create your own topology function.

## Distance Functions

Distance functions calculate the distances of a layer's neurons given their position.

Once defined, you can assign your distance function to any layer of a network. For example, the following line of code assigns the topology function yourdistf to the second layer of a network.

```
net.layers{2}.distanceFcn = 'yourdistf';
```

Your distance function is used whenever your network is trained or adapts.

```
[net,tr] = train(NET,P,T,Pi,Ai)
[net,Y,E,Pf,Af] = adapt(NET,P,T,Pi,Ai)
```

To be a valid distance function, it must calculate distances d from position pos as follows,

```
pos = yourtopf(dim1,dim2,...,dimN)
```

where:

- pos is an $N \times S$ matrix of $S$ neuron position vectors.
- d is an $S \times S$ matrix of neuron distances.

The toolbox contains an example custom distance function called mydistf. Enter the following lines of code to see how it is used.

```
help mydistf
pos = gridtop(4,5);
d = mydistf(pos)
```

Enter the following command to see how mytf is implemented.

```
type mydistf
```

You can use mydistf as a template to create your own distance function.

# Network Object Reference

# Network Properties

The properties define the basic features of a network. “Subobject Properties” on page 13-17 describes properties that define network details.

## Architecture

These properties determine the number of network subobjects (which include inputs, layers, outputs, targets, biases, and weights), and how they are connected.

### numInputs

This property defines the number of inputs a network receives.

```
net.numInputs
```

It can be set to 0 or a positive integer.

**Clarification.** The number of network inputs and the size of a network input are *not* the same thing. The number of inputs defines how many sets of vectors the network receives as input. The size of each input (i.e. the number of elements in each input vector) is determined by the input size (`net.inputs{i}.size`).

Most networks have only one input, whose size is determined by the problem.

**Side Effects.** Any change to this property results in a change in the size of the matrix defining connections to layers from inputs, (`net.inputConnect`) and the size of the cell array of input subobjects (`net.inputs`).

### numLayers

This property defines the number of layers a network has.

```
net.numLayers
```

It can be set to 0 or a positive integer.

**Side Effects.** Any change to this property changes the size of each of these Boolean matrices that define connections to and from layers,

```
net.biasConnect
net.inputConnect
net.layerConnect
net.outputConnect
net.targetConnect
```

and changes the size each cell array of subobject structures whose size depends on the number of layers,

```
net.biases
net.inputWeights
net.layerWeights
net.outputs
net.targets
```

and also changes the size of each of the network's adjustable parameters properties.

```
net.IW
net.LW
net.b
```

### biasConnect

This property defines which layers have biases.

```
net.biasConnect
```

It can be set to any $N$-by-1 matrix of Boolean values, where $N_l$ is the number of network layers (`net.numLayers`). The presence (or absence) of a bias to the $i$th layer is indicated by a 1 (or 0) at:

```
net.biasConnect(i)
```

**Side Effects.** Any change to this property alters the presence or absence of structures in the cell array of biases (`net.biases`) and, in the presence or absence of vectors in the cell array, of bias vectors (`net.b`).

### inputConnect

This property defines which layers have weights coming from inputs.

```
net.inputConnect
```

It can be set to any $N_l \times N_i$ matrix of Boolean values, where $N_l$ is the number of network layers (`net.numLayers`), and $N_i$ is the number of network inputs (`net.numInputs`). The presence (or absence) of a weight going to the $i$th layer from the $j$th input is indicated by a 1 (or 0) at

```
net.inputConnect(i,j)
```

**Side Effects.** Any change to this property will alter the presence or absence of structures in the cell array of input weight subobjects (`net.inputWeights`) and in the presence or absence of matrices in the cell array of input weight matrices (`net.IW`).

### layerConnect

This property defines which layers have weights coming from other layers.

```
net.layerConnect
```

It can be set to any $N_l \times N_l$ matrix of Boolean values, where $N_l$ is the number of network layers (`net.numLayers`). The presence (or absence) of a weight going to the $i$th layer from the $j$th layer is indicated by a 1 (or 0) at

```
net.layerConnect(i,j)
```

**Side Effects.** Any change to this property will alter the presence or absence of structures in the cell array of layer weight subobjects (`net.layerWeights`) and in the presence or absence of matrices in the cell array of layer weight matrices (`net.LW`).

### outputConnect

This property defines which layers generate network outputs.

```
net.outputConnect
```

It can be set to any $1 \times N_l$ matrix of Boolean values, where $N_l$ is the number of network layers (`net.numLayers`). The presence (or absence) of a network output from the $i$th layer is indicated by a 1 (or 0) at

```
net.outputConnect(i)
```

**Side Effects.** Any change to this property will alter the number of network outputs (`net.numOutputs`) and the presence or absence of structures in the cell array of output subobjects (`net.outputs`).

### targetConnect

This property defines which layers have associated targets.

```
net.targetConnect
```

It can be set to any $1 \times N_l$ matrix of Boolean values, where $N_l$ is the number of network layers (net.numLayers). The presence (or absence) of a target associated with the $i$th layer is indicated by a 1 (or 0) at

```
net.targetConnect(i)
```

**Side Effects.** Any change to this property alters the number of network targets (net.numTargets) and the presence or absence of structures in the cell array of target subobjects (net.targets).

### numOutputs (read-only)

This property indicates how many outputs the network has.

```
net.numOutputs
```

It is always set to the number of 1's in the matrix of output connections.

```
numOutputs = sum(net.outputConnect)
```

### numTargets (read-only)

This property indicates how many targets the network has.

```
net.numTargets
```

It is always set to the number of 1's in the matrix of target connections.

```
numTargets = sum(net.targetConnect)
```

### numInputDelays (read-only)

This property indicates the number of time steps of past inputs that must be supplied to simulate the network.

```
net.numInputDelays
```

It is always set to the maximum delay value associated any of the network's input weights.

```
numInputDelays = 0;
for i=1:net.numLayers
  for j=1:net.numInputs
    if net.inputConnect(i,j)
      numInputDelays = max( ...
        [numInputDelays net.inputWeights{i,j}.delays]);
    end
  end
end
```

### numLayerDelays (read-only)

This property indicates the number of time steps of past layer outputs that must be supplied to simulate the network.

```
net.numLayerDelays
```

It is always set to the maximum delay value associated any of the network's layer weights.

```
numLayerDelays = 0;
for i=1:net.numLayers
  for j=1:net.numLayers
    if net.layerConnect(i,j)
      numLayerDelays = max( ...
       [numLayerDelays net.layerWeights{i,j}.delays]);
    end
  end
end
```

## Subobject Structures

These properties consist of cell arrays of structures that define each of the network's inputs, layers, outputs, targets, biases, and weights.

The properties for each kind of subobject are described in "Subobject Properties" on page 13-17.

### inputs

This property holds structures of properties for each of the network's inputs.

```
net.inputs
```

It is always an $N_i \times 1$ cell array of input structures, where $N_i$ is the number of network inputs (`net.numInputs`).

The structure defining the properties of the $i$th network input is located at

```
net.inputs{i}
```

**Input Properties.** See “Inputs” on page 13-17 for descriptions of input properties.

### layers

This property holds structures of properties for each of the network’s layers.

```
net.layers
```

It is always an $N_l \times 1$ cell array of layer structures, where $N_l$ is the number of network layers (`net.numLayers`).

The structure defining the properties of the $i$th layer is located at

```
net.layers{i}
```

**Layer Properties.** See “Layers” on page 13-18 for descriptions of layer properties.

### outputs

This property holds structures of properties for each of the network’s outputs.

```
net.outputs
```

It is always an $1 \times N_l$ cell array, where $N_l$ is the number of network layers (`net.numLayers`).

The structure defining the properties of the output from the $i$th layer (or a null matrix `[]`) is located at

```
net.outputs{i}
```

if the corresponding output connection is 1 (or 0).

```
net.outputConnect(i)
```

**Output Properties.** See “Outputs” on page 13-25 for descriptions of output properties.

### targets

This property holds structures of properties for each of the network’s targets.

```
net.targets
```

It is always an $1 \times N_l$ cell array, where $N_l$ is the number of network layers (`net.numLayers`).

The structure defining the properties of the target associated with the $i$th layer (or a null matrix `[]`) is located at

```
net.targets{i}
```

if the corresponding target connection is 1 (or 0).

```
net.targetConnect(i)
```

**Target Properties.** See “Targets” on page 13-25 for descriptions of target properties.

## biases

This property holds structures of properties for each of the network’s biases.

```
net.biases
```

It is always an $N_l \times 1$ cell array, where $N_l$ is the number of network layers (`net.numLayers`).

The structure defining the properties of the bias associated with the $i$th layer (or a null matrix `[]`) is located at

```
net.biases{i}
```

if the corresponding bias connection is 1 (or 0).

```
net.biasConnect(i)
```

**Bias Properties.** See “Biases” on page 13-26 for descriptions of bias properties.

## inputWeights

This property holds structures of properties for each of the network’s input weights.

```
net.inputWeights
```

It is always an $N_l \times N_i$ cell array, where $N_l$ is the number of network layers (`net.numLayers`), and $N_i$ is the number of network inputs (`net.numInputs`).

The structure defining the properties of the weight going to the $i$th layer from the $j$th input (or a null matrix `[]`) is located at

```
net.inputWeights{i,j}
```

if the corresponding input connection is 1 (or 0).

```
net.inputConnect(i,j)
```

**Input Weight Properties.** See “Input Weights” on page 13-28 for descriptions of input weight properties.

### layerWeights

This property holds structures of properties for each of the network’s layer weights.

```
net.layerWeights
```

It is always an $N_l \times N_l$ cell array, where $N_l$ is the number of network layers (`net.numLayers`).

The structure defining the properties of the weight going to the $i$th layer from the $j$th layer (or a null matrix `[]`) is located at

```
net.layerWeights{i,j}
```

if the corresponding layer connection is 1 (or 0).

```
net.layerConnect(i,j)
```

**Layer Weight Properties.** See “Layer Weights” on page 13-32 for descriptions of layer weight properties.

## Functions

These properties define the algorithms to use when a network is to adapt, is to be initialized, is to have its performance measured, or is to be trained.

### adaptFcn

This property defines the function to be used when the network adapts.

```
net.adaptFcn
```

It can be set to the name of any network adapt function, including this toolbox function:

```
trains - By-weight-and-bias network adaption function.
```

The network adapt function is used to perform adaption whenever adapt is called.

```
[net,Y,E,Pf,Af] = adapt(NET,P,T,Pi,Ai)
```

**Custom Functions.** See Chapter 12, “Advanced Topics” for information on creating custom adapt functions.

**Side Effects.** Whenever this property is altered, the network’s adaption parameters (net.adaptParam) are set to contain the parameters and default values of the new function.

### initFcn

This property defines the function used to initialize the network’s weight matrices and bias vectors.

```
net.initFcn
```

It can be set to the name of any network initialization function, including this toolbox function.

```
initlay  - Layer-by-layer network initialization function.
```

The initialization function is used to initialize the network whenever init is called.

```
net = init(net)
```

**Custom Functions.** See Chapter 12, “Advanced Topics” for information on creating custom initialization functions.

**Side Effects.** Whenever this property is altered, the network’s initialization parameters (net.initParam) are set to contain the parameters and default values of the new function.

### performFcn

This property defines the function used to measure the network’s performance.

```
net.performFcn
```

It can be set to the name of any performance function, including these toolbox functions.

| Performance Functions | |
|---|---|
| `mae` | Mean absolute error-performance function. |
| `mse` | Mean squared error-performance function. |
| `msereg` | Mean squared error w/reg performance function. |
| `sse` | Sum squared error-performance function. |

The performance function is used to calculate network performance during training whenever `train` is called.

```
[net,tr] = train(NET,P,T,Pi,Ai)
```

**Custom functions.** See Chapter 12, "Advanced Topics" for information on creating custom performance functions.

**Side Effects.** Whenever this property is altered, the network's performance parameters (`net.performParam`) are set to contain the parameters and default values of the new function.

### trainFcn

This property defines the function used to train the network.

```
net.trainFcn
```

It can be set to the name of any training function, including these toolbox functions.

| Training Functions | |
|---|---|
| `trainbfg` | BFGS quasi-Newton backpropagation. |
| `trainbr` | Bayesian regularization. |
| `traincgb` | Powell-Beale conjugate gradient backpropagation. |
| `traincgf` | Fletcher-Powell conjugate gradient backpropagation. |

| Training Functions | |
|---|---|
| `traincgp` | Polak-Ribiere conjugate gradient backpropagation. |
| `traingd` | Gradient descent backpropagation. |
| `traingda` | Gradient descent with adaptive lr backpropagation. |
| `traingdm` | Gradient descent with momentum backpropagation. |
| `traingdx` | Gradient descent with momentum and adaptive lr backpropagation |
| `trainlm` | Levenberg-Marquardt backpropagation. |
| `trainoss` | One-step secant backpropagation. |
| `trainrp` | Resilient backpropagation (Rprop). |
| `trainscg` | Scaled conjugate gradient backpropagation. |
| `trainb` | Batch training with weight and bias learning rules. |
| `trainc` | Cyclical order incremental training with learning functions. |
| `trainr` | Random order incremental training with learning functions. |

The training function is used to train the network whenever `train` is called.

```
[net,tr] = train(NET,P,T,Pi,Ai)
```

**Custom Functions.** See Chapter 12, “Advanced Topics” for information on creating custom training functions.

**Side Effects.** Whenever this property is altered, the network’s training parameters (`net.trainParam`) are set to contain the parameters and default values of the new function.

## Parameters

### adaptParam

This property defines the parameters and values of the current adapt function.

```
net.adaptParam
```

The fields of this property depend on the current adapt function (net.adaptFcn). Evaluate the above reference to see the fields of the current adapt function.

Call help on the current adapt function to get a description of what each field means.

```
help(net.adaptFcn)
```

### initParam

This property defines the parameters and values of the current initialization function.

```
net.initParam
```

The fields of this property depend on the current initialization function (net.initFcn). Evaluate the above reference to see the fields of the current initialization function.

Call help on the current initialization function to get a description of what each field means.

```
help(net.initFcn)
```

### performParam

This property defines the parameters and values of the current performance function.

```
net.performParam
```

The fields of this property depend on the current performance function (net.performFcn). Evaluate the above reference to see the fields of the current performance function.

Call help on the current performance function to get a description of what each field means.

```
help(net.performFcn)
```

### trainParam

This property defines the parameters and values of the current training function.

```
net.trainParam
```

The fields of this property depend on the current training function (`net.trainFcn`). Evaluate the above reference to see the fields of the current training function.

Call `help` on the current training function to get a description of what each field means.

```
help(net.trainFcn)
```

## Weight and Bias Values

These properties define the network's adjustable parameters: its weight matrices and bias vectors.

### IW

This property defines the weight matrices of weights going to layers from network inputs.

```
net.IW
```

It is always an $N_l \times N_i$ cell array, where $N_l$ is the number of network layers (`net.numLayers`), and $N_i$ is the number of network inputs (`net.numInputs`).

The weight matrix for the weight going to the $i$th layer from the $j$th input (or a null matrix `[]`) is located at

```
net.IW{i,j}
```

if the corresponding input connection is 1 (or 0).

```
net.inputConnect(i,j)
```

The weight matrix has as many rows as the size of the layer it goes to (`net.layers{i}.size`). It has as many columns as the product of the input size with the number of delays associated with the weight.

```
net.inputs{j}.size * length(net.inputWeights{i,j}.delays)
```

These dimensions can also be obtained from the input weight properties.

```
net.inputWeights{i,j}.size
```

### LW

This property defines the weight matrices of weights going to layers from other layers.

```
net.LW
```

It is always an $N_l \times N_l$ cell array, where $N_l$ is the number of network layers (`net.numLayers`).

The weight matrix for the weight going to the *i*th layer from the *j*th layer (or a null matrix `[]`) is located at

```
net.LW{i,j}
```

if the corresponding layer connection is 1 (or 0).

```
net.layerConnect(i,j)
```

The weight matrix has as many rows as the size of the layer it goes to (`net.layers{i}.size`). It has as many columns as the product of the size of the layer it comes from with the number of delays associated with the weight.

```
net.layers{j}.size * length(net.layerWeights{i,j}.delays)
```

These dimensions can also be obtained from the layer weight properties.

```
net.layerWeights{i,j}.size
```

### b

This property defines the bias vectors for each layer with a bias.

```
net.b
```

It is always an $N_l \times 1$ cell array, where $N_l$ is the number of network layers (`net.numLayers`).

The bias vector for the *i*th layer (or a null matrix `[]`) is located at

```
net.b{i}
```

if the corresponding bias connection is 1 (or 0).

```
net.biasConnect(i)
```

The number of elements in the bias vector is always equal to the size of the layer it is associated with (`net.layers{i}.size`).

This dimension can also be obtained from the bias properties.

```
net.biases{i}.size
```

## Other

The only `other` property is a user data property.

### userdata

This property provides a place for users to add custom information to a network object.

```
net.userdata
```

Only one field is predefined. It contains a *secret* message to all Neural Network Toolbox users.

```
net.userdata.note
```

# Subobject Properties

These properties define the details of a network's inputs, layers, outputs, targets, biases, and weights.

## Inputs

These properties define the details of each *i*th network input.

```
net.inputs{i}
```

### range

This property defines the ranges of each element of the *i*th network input.

```
net.inputs{i}.range
```

It can be set to any $R_i \times 2$ matrix, where $R_i$ is the number of elements in the input (`net.inputs{i}.size`), and each element in column 1 is less than the element next to it in column 2.

Each *j*th row defines the minimum and maximum values of the *j*th input element, in that order

```
net.inputs{i}(j,:)
```

**Uses.** Some initialization functions use input ranges to find appropriate initial values for input weight matrices.

**Side Effects.** Whenever the number of rows in this property is altered, the layers's size (`net.inputs{i}.size`) changes to remain consistent. The size of any weights coming from this input (`net.inputWeights{:,i}.size`) and the dimensions of their weight matrices (`net.IW{:,i}`) also changes size.

### size

This property defines the number of elements in the *i*th network input.

```
net.inputs{i}.size
```

It can be set to 0 or a positive integer.

**Side Effects.** Whenever this property is altered, the input's ranges (`net.inputs{i}.ranges`), any input weights (`net.inputWeights{:,i}.size`) and their weight matrices (`net.IW{:,i}`) change size to remain consistent.

### userdata

This property provides a place for users to add custom information to the *i*th network input.

```
net.inputs{i}.userdata
```

Only one field is predefined. It contains a *secret* message to all Neural Network Toolbox users.

```
net.inputs{i}.userdata.note
```

## Layers

These properties define the details of each *i*th network layer.

```
net.layers{i}
```

### dimensions

This property defines the *physical* dimensions of the *i*th layer's neurons. Being able to arrange a layer's neurons in a multidimensional manner is important for self-organizing maps.

```
net.layers{i}.dimensions
```

It can be set to any row vector of 0 or positive integer elements, where the product of all the elements will becomes the number of neurons in the layer (`net.layers{i}.size`).

**Uses.** Layer dimensions are used to calculate the neuron positions within the layer (`net.layers{i}.positions`) using the layer's topology function (`net.layers{i}.topologyFcn`).

**Side Effects.** Whenever this property is altered, the layers's size (`net.layers{i}.size`) changes to remain consistent. The layer's neuron positions (`net.layers{i}.positions`) and the distances between the neurons (`net.layers{i}.distances`) are also updated.

### distanceFcn

This property defines the function used to calculate distances between neurons in the *i*th layer (`net.layers{i}.distances`) from the neuron positions (`net.layers{i}.positions`). Neuron distances are used by self-organizing maps.

```
net.layers{i}.distanceFcn
```

It can be set to the name of any distance function, including these toolbox functions.

| Distance Functions | |
|---|---|
| boxdist | Distance between two position vectors. |
| dist | Euclidean distance weight function. |
| linkdist | Link distance function. |
| mandist | Manhattan distance weight function. |

**Custom Functions.** See Chapter 12, "Advanced Topics" for information on creating custom distance functions.

**Side Effects.** Whenever this property is altered, the distance between the layer's neurons (`net.layers{i}.distances`) is updated.

### distances (read-only)

This property defines the distances between neurons in the *i*th layer. These distances are used by self-organizing maps.

```
net.layers{i}.distances
```

It is always set to the result of applying the layer's distance function (`net.layers{i}.distanceFcn`) to the positions of the layers neurons (`net.layers{i}.positions`).

### initFcn

This property defines the initialization function used to initialize the *i*th layer, if the network initialization function (`net.initFcn`) is `initlay`.

```
net.layers{i}.initFcn
```

It can be set to the name of any layer initialization function, including these toolbox functions.

| Layer Initialization Functions | |
|---|---|
| initnw | Nguyen-Widrow layer initialization function. |
| initwb | By-weight-and-bias layer initialization function. |

If the network initialization is set to initlay, then the function indicated by this property is used to initialize the layer's weights and biases when init is called.

```
net = init(net)
```

**Custom Functions.** See Chapter 12, "Advanced Topics" for information on creating custom initialization functions.

### netInputFcn

This property defines the net input function use to calculate the *i*th layer's net input, given the layer's weighted inputs and bias.

```
net.layers{i}.netInputFcn
```

It can be set to the name of any net input function, including these toolbox functions.

| Net Input Functions | |
|---|---|
| netprod | Product net input function. |
| netsum | Sum net input function. |

The net input function is used to simulate the network when sim is called.

```
[Y,Pf,Af] = sim(net,P,Pi,Ai)
```

**Custom Functions.** See Chapter 12, "Advanced Topics" for information on creating custom net input functions.

### positions (read-only)

This property defines the positions of neurons in the *i*th layer. These positions are used by self-organizing maps.

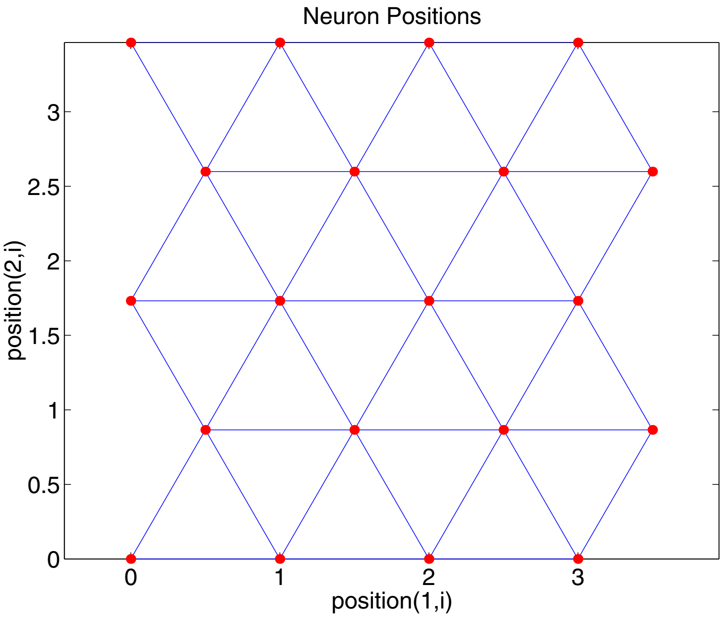
```
net.layers{i}.positions
```

It is always set to the result of applying the layer's topology function (`net.layers{i}.topologyFcn`) to the positions of the layer's dimensions (`net.layers{i}.dimensions`).

**Plotting.** Use `plotsom` to plot the positions of a layer's neurons.

For instance, if the first layer neurons of a network are arranged with dimensions (`net.layers{1}.dimensions`) of [4 5] and the topology function (`net.layers{1}.topologyFcn`) is `hextop`, the neuron's positions can be plotted as shown below.

```
plotsom(net.layers{1}.positions)
```



### size

This property defines the number of neurons in the *i*th layer.

```
net.layers{i}.size
```

It can be set to 0 or a positive integer.

**Side Effects.** Whenever this property is altered, the sizes of any input weights going to the layer (`net.inputWeights{i,:}.size`), and any layer weights going to the layer (`net.layerWeights{i,:}.size`) or coming from the layer (`net.inputWeights{i,:}.size`), and the layer's bias (`net.biases{i}.size`) change.

The dimensions of the corresponding weight matrices (`net.IW{i,:}`, `net.LW{i,:}`, `net.LW{:,i}`) and biases (`net.b{i}`) also change.

Changing this property also changes the size of the layer's output (`net.outputs{i}.size`) and target (`net.targets{i}.size`) if they exist.

Finally, when this property is altered, the dimensions of the layer's neurons (`net.layers{i}.dimension`) are set to the same value. (This results in a one-dimensional arrangement of neurons. If another arrangement is required, set the `dimensions` property directly instead of using `size`).

### topologyFcn

This property defines the function used to calculate the *ith* layer's neuron positions (`net.layers{i}.positions`) from the layer's dimensions (`net.layers{i}.dimensions`).

```
net.topologyFcn
```

It can be set to the name of any topology function, including these toolbox functions.

| **Topology Functions** | |
|---|---|
| `gridtop` | Gridtop layer topology function. |
| `hextop` | Hexagonal layer topology function. |
| `randtop` | Random layer topology function. |

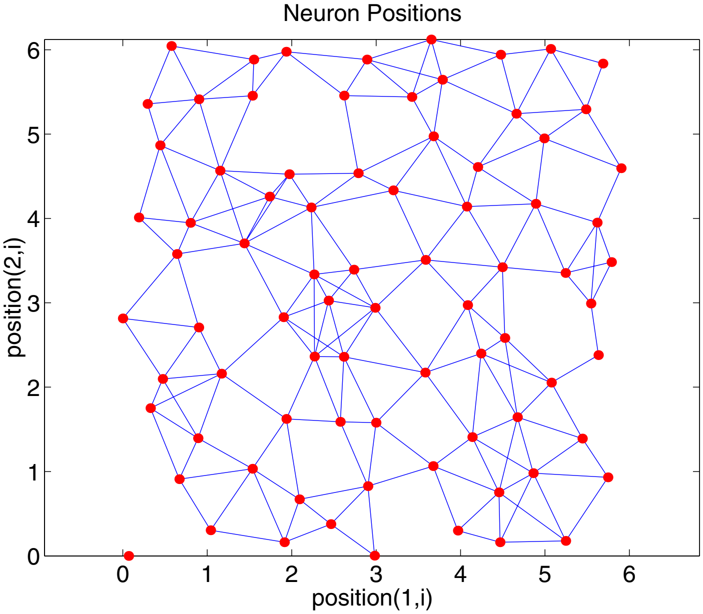
**Custom functions.** See Chapter 12, “Advanced Topics” for information on creating custom topology functions.

**Side Effects.** Whenever this property is altered, the positions of the layer’s neurons (`net.layers{i}.positions`) is updated.

**Plotting.** Use `plotsom` to plot the positions of a layer’s neurons.

For instance, if the first layer neurons of a network are arranged with dimensions (`net.layers{1}.dimensions`) of [8 10] and the topology function (`net.layers{1}.topologyFcn`) is `randtop`, the neuron’s positions are arranged something like those shown in the plot below.

```
plotsom(net.layers{1}.positions)
```



## transferFcn

This function defines the transfer function used to calculate the *i*th layer’s output, given the layer’s net input.

```
net.layers{i}.transferFcn
```

It can be set to the name of any transfer function, including these toolbox functions.

| **Transfer Functions** | |
|---|---|
| `compet` | Competitive transfer function. |
| `hardlim` | Hard-limit transfer function. |
| `hardlims` | Symmetric hard-limit transfer function. |
| `logsig` | Log-sigmoid transfer function. |
| `poslin` | Positive linear transfer function. |
| `purelin` | Hard-limit transfer function. |
| `radbas` | Radial basis transfer function. |
| `satlin` | Saturating linear transfer function. |
| `satlins` | Symmetric saturating linear transfer function. |
| `softmax` | Soft max transfer function. |
| `tansig` | Hyperbolic tangent sigmoid transfer function. |
| `tribas` | Triangular basis transfer function. |

The transfer function is used to simulate the network when `sim` is called.

```
[Y,Pf,Af] = sim(net,P,Pi,Ai)
```

**Custom functions.** See Chapter 12, “Advanced Topics” for information on creating custom transfer functions.

### userdata

This property provides a place for users to add custom information to the *i*th network layer.

```
net.layers{i}.userdata
```

Only one field is predefined. It contains a *secret* message to all Neural Network Toolbox users.

```
net.layers{i}.userdata.note
```

## Outputs

### size (read-only)

This property defines the number of elements in the *i*th layer's output.

```
net.outputs{i}.size
```

It is always set to the size of the *ith* layer (`net.layers{i}.size`).

### userdata

This property provides a place for users to add custom information to the *i*th layer's output.

```
net.outputs{i}.userdata
```

Only one field is predefined. It contains a *secret* message to all Neural Network Toolbox users.

```
net.outputs{i}.userdata.note
```

## Targets

### size (read-only)

This property defines the number of elements in the *i*th layer's target.

```
net.targets{i}.size
```

It is always set to the size of the *ith* layer (`net.layers{i}.size`).

### userdata

This property provides a place for users to add custom information to the *i*th layer's target.

```
net.targets{i}.userdata
```

Only one field is predefined. It contains a *secret* message to all Neural Network Toolbox users.

```
net.targets{i}.userdata.note
```

## Biases

### initFcn

This property defines the function used to initialize the *i*th layer's bias vector, if the network initialization function is `initlay`, and the *i*th layer's initialization function is `initwb`.

```
net.biases{i}.initFcn
```

This function can be set to the name of any bias initialization function, including the toolbox functions.

| Bias Initialization Functions | |
|---|---|
| `initcon` | Conscience bias initialization function. |
| `initzero` | Zero-weight/bias initialization function. |
| `rands` | Symmetric random weight/bias initialization function. |

This function is used to calculate an initial bias vector for the *i*th layer (`net.b{i}`) when `init` is called, if the network initialization function (`net.initFcn`) is `initlay`, and the *ith* layer's initialization function (`net.layers{i}.initFcn`) is `initwb`.

```
net = init(net)
```

**Custom functions.** See Chapter 12, "Advanced Topics" for information on creating custom initialization functions.

### learn

This property defines whether the *i*th bias vector is to be altered during training and adaption.

```
net.biases{i}.learn
```

It can be set to 0 or 1.

It enables or disables the bias' learning during calls to either `adapt` or `train`.

```
[net,Y,E,Pf,Af] = adapt(NET,P,T,Pi,Ai)
[net,tr] = train(NET,P,T,Pi,Ai)
```

### learnFcn

This property defines the function used to update the *i*th layer's bias vector during training, if the network training function is `trainb`, `trainc`, or `trainr`, or during adaption, if the network adapt function is `trains`.

```
net.biases{i}.learnFcn
```

It can be set to the name of any bias learning function, including these toolbox functions.

| Learning Functions | |
|---|---|
| `learncon` | Conscience bias learning function. |
| `learngd` | Gradient descent weight/bias learning function. |
| `learngdm` | Grad. descent w/momentum weight/bias learning function. |
| `learnp` | Perceptron weight/bias learning function. |
| `learnpn` | Normalized perceptron weight/bias learning function. |
| `learnwh` | Widrow-Hoff weight/bias learning rule. |

The learning function updates the *ith* bias vector (`net.b{i}`) during calls to `train`, if the network training function (`net.trainFcn`) is `trainb`, `trainc`, or `trainr`, or during calls to `adapt`, if the network adapt function (`net.adaptFcn`) is `trains`.

```
[net,Y,E,Pf,Af] = adapt(NET,P,T,Pi,Ai)
[net,tr] = train(NET,P,T,Pi,Ai)
```

**Custom functions.** See Chapter 12, "Advanced Topics" for information on creating custom learning functions.

**Side Effects.** Whenever this property is altered, the biases's learning parameters (`net.biases{i}.learnParam`) are set to contain the fields and default values of the new function.

### learnParam

This property defines the learning parameters and values for the current learning function of the *i*th layer's bias.

```
net.biases{i}.learnParam
```

The fields of this property depend on the current learning function (`net.biases{i}.learnFcn`). Evaluate the above reference to see the fields of the current learning function.

Call `help` on the current learning function to get a description of what each field means.

```
help(net.biases{i}.learnFcn)
```

### size (read-only)

This property defines the size of the *i*th layer's bias vector.

```
net.biases{i}.size
```

It is always set to the size of the *i*th layer (`net.layers{i}.size`).

### userdata

This property provides a place for users to add custom information to the *i*th layer's bias.

```
net.biases{i}.userdata
```

Only one field is predefined. It contains a *secret* message to all Neural Network Toolbox users.

```
net.biases{i}.userdata.note
```

## Input Weights

### delays

This property defines a tapped delay line between the *j*th input and its weight to the *i*th layer.

```
net.inputWeights{i,j}.delays
```

It must be set to a row vector of increasing 0 or positive integer values.

**Side Effects.** Whenever this property is altered, the weight's size (`net.inputWeights{i,j}.size`) and the dimensions of its weight matrix (`net.IW{i,j}`) are updated.

### initFcn

This property defines the function used to initialize the weight matrix going to the *i*th layer from the *j*th input, if the network initialization function is `initlay`, and the *i*th layer's initialization function is `initwb`.

```
net.inputWeights{i,j}.initFcn
```

This function can be set to the name of any weight initialization function, including these toolbox functions.

| Weight Initialization Functions | |
|---|---|
| `initzero` | Zero-weight/bias initialization function. |
| `midpoint` | Midpoint-weight initialization function. |
| `randnc` | Normalized column-weight initialization function. |
| `randnr` | Normalized row-weight initialization function. |
| `rands` | Symmetric random-weight/bias initialization function. |

This function is used to calculate an initial weight matrix for the weight going to the *i*th layer from the *j*th input (`net.IW{i,j}`) when `init` is called, if the network initialization function (`net.initFcn`) is `initlay`, and the *i*th layer's initialization function (`net.layers{i}.initFcn`) is `initwb`.

```
net = init(net)
```

**Custom Functions.** See Chapter 12, "Advanced Topics" for information on creating custom initialization functions.

### learn

This property defines whether the weight matrix to the *i*th layer from the *j*th input is to be altered during training and adaption.

```
net.inputWeights{i,j}.learn
```

It can be set to 0 or 1.

It enables or disables the weights learning during calls to either `adapt` or `train`.

```
[net,Y,E,Pf,Af] = adapt(NET,P,T,Pi,Ai)
[net,tr] = train(NET,P,T,Pi,Ai)
```

### learnFcn

This property defines the function used to update the weight matrix going to the *i*th layer from the *j*th input during training, if the network training function is `trainb`, `trainc`, or `trainr`, or during adaption, if the network adapt function is `trains`.

```
net.inputWeights{i,j}.learnFcn
```

It can be set to the name of any weight learning function, including these toolbox functions.

| **Weight Learning Functions** | |
|---|---|
| `learngd` | Gradient descent weight/bias learning function. |
| `learngdm` | Grad. descent w/ momentum weight/bias learning function. |
| `learnh` | Hebb-weight learning function. |
| `learnhd` | Hebb with decay weight learning function. |
| `learnis` | Instar-weight learning function. |
| `learnk` | Kohonen-weight learning function. |
| `learnlv1` | LVQ1-weight learning function. |
| `learnlv2` | LVQ2-weight learning function. |
| `learnos` | Outstar-weight learning function. |
| `learnp` | Perceptron weight/bias learning function. |
| `learnpn` | Normalized perceptron-weight/bias learning function. |
| `learnsom` | Self-organizing map-weight learning function. |
| `learnwh` | Widrow-Hoff weight/bias learning rule. |

The learning function updates the weight matrix of the *i*th layer from the *j*th input (`net.IW{i,j}`) during calls to `train`, if the network training function (net.trainFcn) is trainb, trainc, or trainr, or during calls to adapt, if the network adapt function (net.adaptFcn) is trains.

```
[net,Y,E,Pf,Af] = adapt(NET,P,T,Pi,Ai)
[net,tr] = train(NET,P,T,Pi,Ai)
```

**Custom Functions.** See Chapter 12, "Advanced Topics" for information on creating custom learning functions.

### learnParam

This property defines the learning parameters and values for the current learning function of the *i*th layer's weight coming from the *j*th input.

```
net.inputWeights{i,j}.learnParam
```

The fields of this property depend on the current learning function (net.inputWeights{i,j}.learnFcn). Evaluate the above reference to see the fields of the current learning function.

Call help on the current learning function to get a description of what each field means.

```
help(net.inputWeights{i,j}.learnFcn)
```

### size (read-only)

This property defines the dimensions of the *i*th layer's weight matrix from the *j*th network input.

```
net.inputWeights{i,j}.size
```

It is always set to a two-element row vector indicating the number of rows and columns of the associated weight matrix (net.IW{i,j}). The first element is equal to the size of the *i*th layer (net.layers{i}.size). The second element is equal to the product of the length of the weights delay vectors with the size of the *j*th input:

```
length(net.inputWeights{i,j}.delays) * net.inputs{j}.size
```

### userdata

This property provides a place for users to add custom information to the *(i,j)*th input weight.

```
net.inputWeights{i,j}.userdata
```

Only one field is predefined. It contains a *secret* message to all Neural Network Toolbox users.

```
net.inputWeights{i,j}.userdata.note
```

### weightFcn

This property defines the function used to apply the *i*th layer's weight from the *j*th input to that input.

```
net.inputWeights{i,j}.weightFcn
```

It can be set to the name of any weight function, including these toolbox functions.

| **Weight Functions** | |
|---|---|
| `dist` | Conscience bias initialization function. |
| `dotprod` | Zero-weight/bias initialization function. |
| `mandist` | Manhattan-distance weight function. |
| `negdist` | Normalized column-weight initialization function. |
| `normprod` | Normalized row-weight initialization function. |

The weight function is used when `sim` is called to simulate the network.

```
[Y,Pf,Af] = sim(net,P,Pi,Ai)
```

**Custom functions.** See Chapter 12, "Advanced Topics" for information on creating custom weight functions.

## Layer Weights

### delays

This property defines a tapped delay line between the *j*th layer and its weight to the *ith* layer.

```
net.layerWeights{i,j}.delays
```

It must be set to a row vector of increasing 0 or positive integer values.

### initFcn

This property defines the function used to initialize the weight matrix going to the *i*th layer from the *j*th layer, if the network initialization function is `initlay`, and the *i*th layer's initialization function is `initwb`.

```
net.layerWeights{i,j}.initFcn
```

This function can be set to the name of any weight initialization function, including the toolbox functions.

| Weight and Bias Initialization Functions | |
|---|---|
| `initzero` | Zero-weight/bias initialization function. |
| `midpoint` | Midpoint-weight initialization function. |
| `randnc` | Normalized column-weight initialization function. |
| `randnr` | Normalized row-weight initialization function. |
| `rands` | Symmetric random-weight/bias initialization function. |

This function is used to calculate an initial weight matrix for the weight going to the *i*th layer from the *j*th layer (`net.LW{i,j}`) when `init` is called, if the network initialization function (`net.initFcn`) is `initlay`, and the *i*th layer's initialization function (`net.layers{i}.initFcn`) is `initwb`.

```
net = init(net)
```

**Custom Functions.** See Chapter 12, "Advanced Topics" for information on creating custom initialization functions.

### learn

This property defines whether the weight matrix to the *ith* layer from the *j*th layer is to be altered during training and adaption.

```
net.layerWeights{i,j}.learn
```

It can be set to 0 or 1.

It enables or disables the weights learning during calls to either `adapt` or `train`.

```
[net,Y,E,Pf,Af] = adapt(NET,P,T,Pi,Ai)
[net,tr] = train(NET,P,T,Pi,Ai)
```

### learnFcn

This property defines the function used to update the weight matrix going to the *i*th layer from the *j*th layer during training, if the network training function is `trainb`, `trainc`, or `trainr`, or during adaption, if the network adapt function is `trains`.

```
net.layerWeights{i,j}.learnFcn
```

It can be set to the name of any weight learning function, including these toolbox functions.

| **Learning Functions** | |
|---|---|
| `learngd` | Gradient-descent weight/bias learning function. |
| `learngdm` | Grad. descent w/momentum weight/bias learning function. |
| `learnh` | Hebb-weight learning function. |
| `learnhd` | Hebb with decay weight learning function. |
| `learnis` | Instar-weight learning function. |
| `learnk` | Kohonen-weight learning function. |
| `learnlv1` | LVQ1-weight learning function. |
| `learnlv2` | LVQ2-weight learning function. |
| `learnos` | Outstar-weight learning function. |
| `learnp` | Perceptron-weight/bias learning function. |
| `learnpn` | Normalized perceptron-weight/bias learning function. |
| `learnsom` | Self-organizing map-weight learning function. |
| `learnwh` | Widrow-Hoff weight/bias learning rule. |

The learning function updates the weight matrix of the *i*th layer form the *j*th layer (`net.LW{i,j}`) during calls to `train`, if the network training function (net.trainFcn) is trainb, trainc, or trainr, or during calls to adapt, if the network adapt function (net.adaptFcn) is trains.

```
[net,Y,E,Pf,Af] = adapt(NET,P,T,Pi,Ai)
[net,tr] = train(NET,P,T,Pi,Ai)
```

**Custom Functions.** See Chapter 12, “Advanced Topics” for information on creating custom learning functions.

### learnParam

This property defines the learning parameters fields and values for the current learning function of the *ith* layer’s weight coming from the *j*th layer.

```
net.layerWeights{i,j}.learnParam
```

The subfields of this property depend on the current learning function (net.layerWeights{i,j}.learnFcn). Evaluate the above reference to see the fields of the current learning function.

Get help on the current learning function to get a description of what each field means.

```
help(net.layerWeights{i,j}.learnFcn)
```

### size (read-only)

This property defines the dimensions of the *ith* layer’s weight matrix from the *j*th layer.

```
net.layerWeights{i,j}.size
```

It is always set to a two-element row vector indicating the number of rows and columns of the associated weight matrix (net.LW{i,j}). The first element is equal to the size of the *ith* layer (net.layers{i}.size). The second element is equal to the product of the length of the weights delay vectors with the size of the *j*th layer.

```
length(net.layerWeights{i,j}.delays) * net.layers{j}.size
```

### userdata

This property provides a place for users to add custom information to the *(i,j)th* layer weight.

```
net.layerWeights{i,j}.userdata
```

Only one field is predefined. It contains a *secret* message to all Neural Network Toolbox users.

```
net.layerWeights{i,j}.userdata.note
```

### weightFcn

This property defines the function used to apply the *i*th layer's weight from the *j*th layer to that layer's output.

```
net.layerWeights{i,j}.weightFcn
```

It can be set to the name of any weight function, including these toolbox functions.

| **Weight Functions** | |
|---|---|
| `dist` | Euclidean-distance weight function. |
| `dotprod` | Dot-product weight function. |
| `mandist` | Manhattan-distance weight function. |
| `negdist` | Dot-product weight function. |
| `normprod` | Normalized dot-product weight function. |

The weight function is used when `sim` is called to simulate the network.

```
[Y,Pf,Af] = sim(net,P,Pi,Ai)
```

**Custom Functions.** See Chapter 12, “Advanced Topics” for information on creating custom weight functions.

14

# Reference

# Functions — Categorical List

## Analysis Functions

| | |
|---|---|
| `errsurf` | Error surface of a single input neuron. |
| `maxlinlr` | Maximum learning rate for a linear neuron. |

## Distance Functions

| | |
|---|---|
| `boxdist` | Distance between two position vectors. |
| `dist` | Euclidean distance weight function. |
| `linkdist` | Link distance function. |
| `mandist` | Manhattan distance weight function. |

## Graphical Interface Function

| | |
|---|---|
| `nntool` | Neural Network Tool - Graphical User Interface. |

## Layer Initialization Functions

| | |
|---|---|
| `initnw` | Nguyen-Widrow layer initialization function. |
| `initwb` | By-weight-and-bias layer initialization function. |

## Learning Functions

| | |
|---|---|
| `learncon` | Conscience bias learning function. |
| `learngd` | Gradient descent weight/bias learning function. |
| `learngdm` | Grad. descent w/momentum weight/bias learning function. |
| `learnh` | Hebb weight learning function. |
| `learnhd` | Hebb with decay weight learning rule. |
| `learnis` | Instar weight learning function. |
| `learnk` | Kohonen weight learning function. |
| `learnlv1` | LVQ1 weight learning function. |
| `learnlv2` | LVQ2 weight learning function. |
| `learnos` | Outstar weight learning function. |
| `learnp` | Perceptron weight and bias learning function. |
| `learnpn` | Normalized perceptron weight and bias learning function. |
| `learnsom` | Self-organizing map weight learning function. |
| `learnwh` | Widrow-Hoff weight and bias learning rule. |

## Line Search Functions

| | |
|---|---|
| `srchbac` | One-dim. minimization using backtracking search. |
| `srchbre` | One-dim. interval location using Brent’s method. |
| `srchcha` | One-dim. minimization using Charalambous’ method. |
| `srchgol` | One-dim. minimization using Golden section search. |
| `srchhyb` | One-dim. minimization using Hybrid bisection/cubic search. |

## Net Input Derivative Functions

| | |
|---|---|
| `dnetprod` | Product net input derivative function. |
| `dnetsum` | Sum net input derivative function. |

## Net Input Functions

| | |
|---|---|
| `netprod` | Product net input function. |
| `netsum` | Sum net input function. |

## Network Functions

| | |
|---|---|
| `assoclr` | Associative learning rules |
| `backprop` | Backpropagation networks |
| `elman` | Elman recurrent networks |
| `hopfield` | Hopfield recurrent networks |
| `linnet` | Linear networks |
| `lvq` | Learning vector quantization |
| `percept` | Perceptrons |
| `radbasis` | Radial basis networks |
| `selforg` | Self-organizing networks |

## Network Initialization Function

| | |
|---|---|
| `initlay` | Layer-by-layer network initialization function. |

## Network Use Functions

| | |
|---|---|
| `adapt` | Allow a neural network to adapt. |
| `disp` | Display a neural network's properties. |
| `display` | Display a neural network variable’s name and properties. |
| `init` | Initialize a neural network. |
| `sim` | Simulate a neural network. |
| `train` | Train a neural network. |

## New Networks Functions

| | |
|---|---|
| `network` | Create a custom neural network. |
| `newc` | Create a competitive layer. |
| `newcf` | Create a cascade-forward backpropagation network. |
| `newelm` | Create an Elman backpropagation network. |
| `newff` | Create a feed-forward backpropagation network. |
| `newfftd` | Create a feed-forward input-delay backprop network. |
| `newgrnn` | Design a generalized regression neural network. |
| `newhop` | Create a Hopfield recurrent network. |
| `newlin` | Create a linear layer. |
| `newlind` | Design a linear layer. |
| `newlvq` | Create a learning vector quantization network |
| `newp` | Create a perceptron. |
| `newpnn` | Design a probabilistic neural network. |
| `newrb` | Design a radial basis network. |
| `newrbe` | Design an exact radial basis network. |
| `newsom` | Create a self-organizing map. |

## Performance Derivative Functions

| | |
|---|---|
| `dmae` | Mean absolute error performance derivative function. |
| `dmse` | Mean squared error performance derivatives function. |
| `dmsereg` | Mean squared error w/reg performance derivative function. |
| `dsse` | Sum squared error performance derivative function. |

## Performance Functions

| | |
|---|---|
| `mae` | Mean absolute error performance function. |
| `mse` | Mean squared error performance function. |
| `msereg` | Mean squared error w/reg performance function. |
| `sse` | Sum squared error performance function. |

## Plotting Functions

| | |
|---|---|
| `hintonw` | Hinton graph of weight matrix. |
| `hintonwb` | Hinton graph of weight matrix and bias vector. |
| `plotbr` | Plot network perf. for Bayesian regularization training. |
| `plotep` | Plot weight and bias position on error surface. |
| `plotes` | Plot error surface of single input neuron. |
| `plotpc` | Plot classification line on perceptron vector plot. |
| `plotperf` | Plot network performance. |
| `plotpv` | Plot perceptron input target vectors. |
| `plotsom` | Plot self-organizing map. |
| `plotv` | Plot vectors as lines from the origin. |
| `plotvec` | Plot vectors with different colors. |

## Pre- and Postprocessing Functions

| | |
|---|---|
| `postmnmx` | Unnormalize data which has been norm. by `prenmmx`. |
| `postreg` | Postprocess network response w. linear regression analysis. |
| `poststd` | Unnormalize data which has been normalized by `prestd`. |
| `premnmx` | Normalize data for maximum of 1 and minimum of -1. |
| `prepca` | Principal component analysis on input data. |
| `prestd` | Normalize data for unity standard deviation and zero mean. |
| `tramnmx` | Transform data with precalculated minimum and max. |
| `trapca` | Transform data with PCA matrix computed by `prepca`. |
| `trastd` | Transform data with precalc. mean & standard deviation. |

## Simulink Support Function

| | |
|---|---|
| `gensim` | Generate a Simulink® block for neural network simulation. |

## Topology Functions

| | |
|---|---|
| `gridtop` | Gridtop layer topology function. |
| `hextop` | Hexagonal layer topology function. |
| `randtop` | Random layer topology function. |

## Training Functions

| | |
|---|---|
| trainb | Batch training with weight and bias learning rules. |
| trainbfg | BFGS quasi-Newton backpropagation. |
| trainbr | Bayesian regularization. |
| trainc | Cyclical order incremental update. |
| traincgb | Powell-Beale conjugate gradient backpropagation. |
| traincgf | Fletcher-Powell conjugate gradient backpropagation. |
| traincgp | Polak-Ribiere conjugate gradient backpropagation. |
| traingd | Gradient descent backpropagation. |
| traingda | Gradient descent with adaptive lr backpropagation. |
| traingdm | Gradient descent with momentum backpropagation. |
| traingdx | Gradient descent with momentum & adaptive lr backprop. |
| trainlm | Levenberg-Marquardt backpropagation. |
| trainoss | One step secant backpropagation. |
| trainr | Random order incremental update. |
| trainrp | Resilient backpropagation (Rprop). |
| trains | Sequential order incremental update. |
| trainscg | Scaled conjugate gradient backpropagation. |

## Transfer Derivative Functions

| | |
|---|---|
| `dhardlim` | Hard limit transfer derivative function. |
| `dhardlms` | Symmetric hard limit transfer derivative function. |
| `dlogsig` | Log sigmoid transfer derivative function. |
| `dposlin` | Positive linear transfer derivative function. |
| `dpurelin` | Linear transfer derivative function. |
| `dradbas` | Radial basis transfer derivative function. |
| `dsatlin` | Saturating linear transfer derivative function. |
| `dsatlins` | Symmetric saturating linear transfer derivative function. |
| `dtansig` | Hyperbolic tangent sigmoid transfer derivative function. |
| `dtribas` | Triangular basis transfer derivative function. |

## Transfer Functions

| | | |
|---|---|---|
| `compet` | Competitive transfer function. | C |
| `hardlim` | Hard limit transfer function. | |
| `hardlims` | Symmetric hard limit transfer function | |
| `logsig` | Log sigmoid transfer function. | |
| `poslin` | Positive linear transfer function | |
| `purelin` | Linear transfer function. | |
| `radbas` | Radial basis transfer function. | |
| `satlin` | Saturating linear transfer function. | |
| `satlins` | Symmetric saturating linear transfer function | |
| `softmax` | Softmax transfer function. | S |
| `tansig` | Hyperbolic tangent sigmoid transfer function. | |
| `tribas` | Triangular basis transfer function. | |

## Utility Functions

| | |
|---|---|
| `calca` | Calculate network outputs and other signals. |
| `calca1` | Calculate network signals for one time step. |
| `calce` | Calculate layer errors. |
| `calce1` | Calculate layer errors for one time step. |
| `calcgx` | Calc. weight and bias perform. gradient as a single vector. |
| `calcjejj` | Calculate Jacobian performance vector. |
| `calcjx` | Calculate weight and bias performance Jacobian as a single matrix. |
| `calcpd` | Calculate delayed network inputs. |
| `calcperf` | Calculation network outputs, signals, and performance. |
| `formx` | Form bias and weights into single vector. |
| `getx` | Get all network weight and bias values as a single vector. |
| `setx` | Set all network weight and bias values with a single vector. |

## Vector Functions

| | |
|---|---|
| `cell2mat` | Combine a cell array of matrices into one matrix. |
| `combvec` | Create all combinations of vectors. |
| `con2seq` | Converts concurrent vectors to sequential vectors. |
| `concur` | Create concurrent bias vectors. |
| `ind2vec` | Convert indices to vectors. |
| `mat2cell` | Break matrix up into cell array of matrices. |
| `minmax` | Ranges of matrix rows. |
| `normc` | Normalize columns of matrix. |
| `normr` | Normalize rows of matrix. |
| `pnormc` | Pseudo-normalize columns of matrix. |
| `quant` | Discretize value as multiples of a quantity. |
| `seq2con` | Convert sequential vectors to concurrent vectors. |
| `sumsqr` | Sum squared elements of matrix. |
| `vec2ind` | Convert vectors to indices. |

## Weight and Bias Initialization Functions

| | |
|---|---|
| `initcon` | Conscience bias initialization function. |
| `initzero` | Zero weight and bias initialization function. |
| `midpoint` | Midpoint weight initialization function. |
| `randnc` | Normalized column weight initialization function. |
| `randnr` | Normalized row weight initialization function. |
| `rands` | Symmetric random weight/bias initialization function. |
| `revert` | Change ntwk wts. and biases to prev. initialization values. |

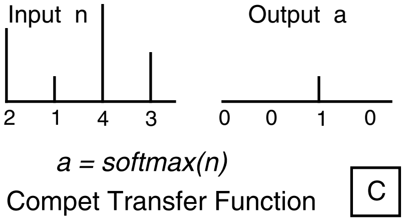
## Weight Derivative Functions

| | |
|---|---|
| ddotprod | Dot product weight derivative function. |

## Weight Functions

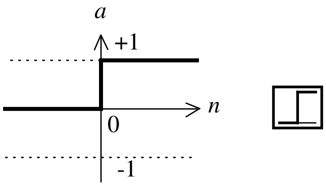
| | |
|---|---|
| dist | Euclidean distance weight function. |
| dotprod | Dot product weight function. |
| mandist | Manhattan distance weight function. |
| negdist | Negative distance weight function. |
| normprod | Normalized dot product weight function. |

# Transfer Function Graphs



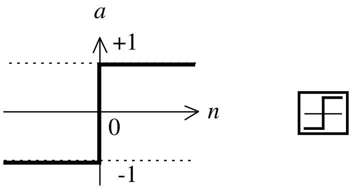
*a = softmax(n)*

Compet Transfer Function



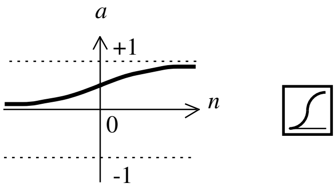
*a = hardlim(n)*

Hard-Limit Transfer Function



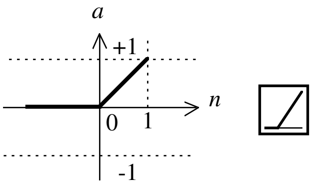
*a = hardlims(n)*

Symmetric Hard-Limit Trans. Funct.



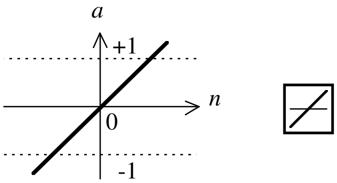
*a = logsig(n)*

Log-Sigmoid Transfer Function



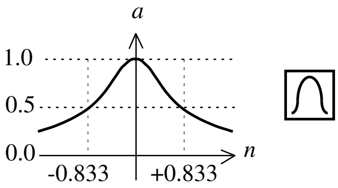
*a = poslin(n)*

Positive Linear Transfer Funct.



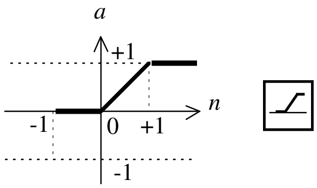
*a = purelin(n)*

Linear Transfer Function



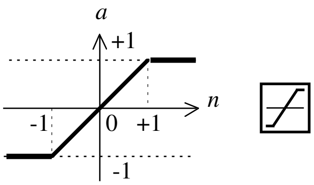
*a = radbas(n)*

Radial Basis Function



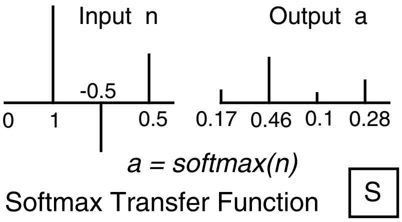
*a = satlin(n)*

Satlin Transfer Function

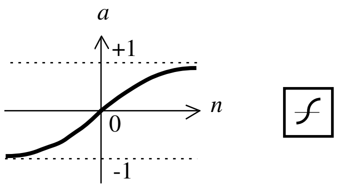


*a = satlins(n)*

Satlins Transfer Function

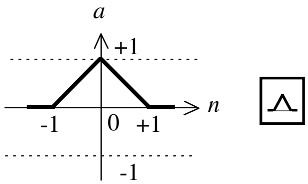


Softmax Transfer Function



$a = tansig(n)$

Tan-Sigmoid Transfer Function



$a = tribas(n)$

Triangular Basis Function

## Functions — Alphabetical List

## Purpose

Allow a neural network to adapt (change weights and biases on each presentation of an input)

## Syntax

```
[net,Y,E,Pf,Af] = adapt(net,P,T,Pi,Ai)
```

## To Get Help

Type `help network/adapt`

## Description

This function calculates network outputs and errors after each presentation of an input.

`[net,Y,E,Pf,Af,tr] = adapt(net,P,T,Pi,Ai)` takes,

- `net` — Network
- `P` — Network inputs
- `T` — Network targets, default = zeros
- `Pi` — Initial input delay conditions, default = zeros
- `Ai` — Initial layer delay conditions, default = zeros

and returns the following after applying the adapt function `net.adaptFcn` with the adaption parameters `net.adaptParam`:

- `net` — Updated network
- `Y` — Network outputs
- `E` — Network errors
- `Pf` — Final input delay conditions
- `Af` — Final layer delay conditions
- `tr` — Training record (epoch and perf)

Note that `T` is optional and only needs to be used for networks that require targets. `Pi` and `Pf` are also optional and only need to be used for networks that have input or layer delays.

`adapt`’s signal arguments can have two formats: cell array or matrix.

The cell array format is easiest to describe. It is most convenient for networks with multiple inputs and outputs, and allows sequences of inputs to be presented:

P — Ni x TS cell array, each element P{i,ts} is an Ri x Q matrix

T — Nt x TS cell array, each element T{i,ts} is a Vi x Q matrix

Pi — Ni x ID cell array, each element Pi{i,k} is an Ri x Q matrix

Ai — Nl x LD cell array, each element Ai{i,k} is an Si x Q matrix

Y — NO x TS cell array, each element Y{i,ts} is a Ui x Q matrix

E — Nt x TS cell array, each element E{i,ts} is a Vi x Q matrix

Pf — Ni x ID cell array, each element Pf{i,k} is an Ri x Q matrix

Af — Nl x LD cell array, each element Af{i,k} is an Si x Q matrix

where

```
Ni = net.numInputs
Nl = net.numLayers
No = net.numOutputs
Nt = net.numTargets
ID = net.numInputDelays
LD = net.numLayerDelays
```

TS = Number of time steps

Q = Batch size

```
Ri = net.inputs{i}.size
Si = net.layers{i}.size
Ui = net.outputs{i}.size
Vi = net.targets{i}.size
```

The columns of Pi, Pf, Ai, and Af are ordered from oldest delay condition to most recent:

Pi{i,k} = input i at time ts = k ID

Pf{i,k} = input i at time ts = TS+k ID

Ai{i,k} = layer output i at time ts = k LD

Af{i,k} = layer output i at time ts = TS+k LD

The matrix format can be used if only one time step is to be simulated (TS = 1). It is convenient for network's with only one input and output, but can be used with networks that have more.

Each matrix argument is found by storing the elements of the corresponding cell array argument in a single matrix:

- P — (sum of Ri) x Q matrix
- T — (sum of Vi) x Q matrix
- Pi — (sum of Ri) x (ID*Q) matrix
- Ai — (sum of Si) x (LD*Q) matrix
- Y — (sum of Ui) x Q matrix
- Pf — (sum of Ri) x (ID*Q) matrix
- Af — (sum of Si) x (LD*Q) matrix

## Examples

Here two sequences of 12 steps (where T1 is known to depend on P1) are used to define the operation of a filter.

```
p1 = {-1  0 1 0 1 1 -1  0 -1 1 0 1};
t1 = {-1 -1 1 1 1 2  0 -1 -1 0 1 1};
```

Here newlin is used to create a layer with an input range of [-1 1]), one neuron, input delays of 0 and 1, and a learning rate of 0.5. The linear layer is then simulated.

```
net = newlin([-1 1],1,[0 1],0.5);
```

Here the network adapts for one pass through the sequence.

The network's mean squared error is displayed. (Since this is the first call of adapt, the default Pi is used.)

```
[net,y,e,pf] = adapt(net,p1,t1);
mse(e)
```

Note the errors are quite large. Here the network adapts to another 12 time steps (using the previous Pf as the new initial delay conditions.)

```
p2 = {1 -1 -1 1 1 -1  0 0 0 1 -1 -1};
t2 = {2  0 -2 0 2  0 -1 0 0 1  0 -1};
[net,y,e,pf] = adapt(net,p2,t2,pf);
mse(e)
```

Here the network adapts for 100 passes through the entire sequence.

```
p3 = [p1 p2];
t3 = [t1 t2];
net.adaptParam.passes = 100;
[net,y,e] = adapt(net,p3,t3);
mse(e)
```

The error after 100 passes through the sequence is very small. The network has adapted to the relationship between the input and target signals.

## Algorithm

adapt calls the function indicated by net.adaptFcn, using the adaption parameter values indicated by net.adaptParam.

Given an input sequence with TS steps, the network is updated as follows. Each step in the sequence of inputs is presented to the network one at a time. The network's weight and bias values are updated after each step, before the next step in the sequence is presented. Thus the network is updated TS times.

## See Also

sim, init, train, revert

### Purpose

Box distance function

### Syntax

```
d = boxdist(pos);
```

### Description

`boxdist` is a layer distance function that is used to find the distances between the layer's neurons, given their positions.

`boxdist`(pos) takes one argument,

| | |
|---|---|
| `pos` | `N` x `S` matrix of neuron positions |

and returns the `S` x `S` matrix of distances

`boxdist` is most commonly used in conjunction with layers whose topology function is `gridtop`.

### Examples

Here we define a random matrix of positions for 10 neurons arranged in three-dimensional space and then find their distances.

```
pos = rand(3,10);
d = boxdist(pos)
```

### Network Use

You can create a standard network that uses `boxdist` as a distance function by calling `newsom`.

To change a network so that a layer's topology uses `boxdist`, set `net.layers{i}.distanceFcn` to '`boxdist`'.

In either case, call `sim` to simulate the network with `boxdist`. See `newsom` for training and adaption examples.

### Algorithm

The box distance `D` between two position vectors `Pi` and `Pj` from a set of `S` vectors is:

```
Dij = max(abs(Pi-Pj))
```

### See Also

`sim`, `dist`, `mandist`, `linkdist`

# calca

## Purpose

Calculate network outputs and other signals

## Syntax

```
[Ac,N,LWZ,IWZ,BZ] = calca(net,Pd,Ai,Q,TS)
```

## Description

This function calculates the outputs of each layer in response to a network's delayed inputs and initial layer delay conditions.

`[Ac,N,LWZ,IWZ,BZ] = calca(net,Pd,Ai,Q,TS)` takes,

`net` — Neural network

`Pd` — Delayed inputs

`Ai` — Initial layer delay conditions

`Q` — Concurrent size

`TS` — Time steps

and returns,

`Ac` — Combined layer outputs = [Ai, calculated layer outputs]

`N` — Net inputs

`LWZ` — Weighted layer outputs

`IWZ` — Weighted inputs

`BZ` — Concurrent biases

## Examples

Here we create a linear network with a single input element ranging from 0 to 1, three neurons, and a tap delay on the input with taps at zero, two, and four time steps. The network is also given a recurrent connection from layer 1 to itself with tap delays of [1 2].

```
net = newlin([0 1],3,[0 2 4]);
net.layerConnect(1,1) = 1;
net.layerWeights{1,1}.delays = [1 2];
```

Here is a single (`Q = 1`) input sequence `P` with eight time steps (`TS = 8`), and the four initial input delay conditions `Pi`, combined inputs `Pc`, and delayed inputs `Pd`.

```
P = {0 0.1 0.3 0.6 0.4 0.7 0.2 0.1};
Pi = {0.2 0.3 0.4 0.1};
Pc = [Pi P];
Pd = calcpd(net,8,1,Pc)
```

Here the two initial layer delay conditions for each of the three neurons are defined:

```
Ai = {[0.5; 0.1; 0.2] [0.6; 0.5; 0.2]};
```

Here we calculate the network's combined outputs Ac, and other signals described above.

```
[Ac,N,LWZ,IWZ,BZ] = calca(net,Pd,Ai,1,8)
```

# calca1

## Purpose

Calculate network signals for one time step

## Syntax

`[Ac,N,LWZ,IWZ,BZ] = calca1(net,Pd,Ai,Q)`

## Description

This function calculates the outputs of each layer in response to a network's delayed inputs and initial layer delay conditions, for a single time step.

Calculating outputs for a single time step is useful for sequential iterative algorithms such as `trains`, which need to calculate the network response for each time step individually.

`[Ac,N,LWZ,IWZ,BZ] = calca1(net,Pd,Ai,Q)` takes,

`net` — Neural network  
`Pd` — Delayed inputs for a single time step  
`Ai` — Initial layer delay conditions for a single time step  
`Q` — Concurrent size

and returns,

`A` — Layer outputs for the time step  
`N` — Net inputs for the time step  
`LWZ` — Weighted layer outputs for the time step  
`IWZ` — Weighted inputs for the time step  
`BZ` — Concurrent biases for the time step

## Examples

Here we create a linear network with a single input element ranging from 0 to 1, three neurons, and a tap delay on the input with taps at zero, two, and four time steps. The network is also given a recurrent connection from layer 1 to itself with tap delays of [1 2].

```
net = newlin([0 1],3,[0 2 4]);
net.layerConnect(1,1) = 1;
net.layerWeights{1,1}.delays = [1 2];
```

Here is a single (`Q = 1`) input sequence `P` with eight time steps (`TS = 8`), and the four initial input delay conditions `Pi`, combined inputs `Pc`, and delayed inputs `Pd`.

```
P = {0 0.1 0.3 0.6 0.4 0.7 0.2 0.1};
Pi = {0.2 0.3 0.4 0.1};
```

```
Pc = [Pi P];
Pd = calcpd(net,8,1,Pc)
```

Here the two initial layer delay conditions for each of the three neurons are defined:

```
Ai = {[0.5; 0.1; 0.2] [0.6; 0.5; 0.2]};
```

Here we calculate the network's combined outputs Ac, and other signals described above.

```
[Ac,N,LWZ,IWZ,BZ] = calca(net,Pd,Ai,1,8)
```

# calce

## Purpose

Calculate layer errors

## Syntax

```
El = calce(net,Ac,Tl,TS)
```

## Description

This function calculates the errors of each layer in response to layer outputs and targets.

`El = calce(net,Ac,Tl,TS)` takes,

`net` — Neural network
`Ac` — Combined layer outputs
`Tl` — Layer targets
`Q` — Concurrent size

and returns,

`El` — Layer errors

## Examples

Here we create a linear network with a single input element ranging from 0 to 1, two neurons, and a tap delay on the input with taps at 0, 2, and 4 time steps. The network is also given a recurrent connection from layer 1 to itself with tap delays of [1 2].

```
net = newlin([0 1],2);
net.layerConnect(1,1) = 1;
net.layerWeights{1,1}.delays = [1 2];
```

Here is a single (`Q = 1`) input sequence `P` with five time steps (`TS = 5`), and the four initial input delay conditions `Pi`, combined inputs `Pc`, and delayed inputs `Pd`.

```
P = {0 0.1 0.3 0.6 0.4};
Pi = {0.2 0.3 0.4 0.1};
Pc = [Pi P];
Pd = calcpd(net,5,1,Pc);
```

Here the two initial layer delay conditions for each of the two neurons are defined, and the networks combined outputs `Ac` and other signals are calculated.

```
Ai = {[0.5; 0.1] [0.6; 0.5]};
[Ac,N,LWZ,IWZ,BZ] = calca(net,Pd,Ai,1,5);
```

Here we define the layer targets for the two neurons for each of the five time steps, and calculate the layer errors.

```
Tl = {[0.1;0.2] [0.3;0.1], [0.5;0.6] [0.8;0.9], [0.5;0.1]};
El = calce(net,Ac,Tl,5)
```

Here we view the network’s error for layer 1 at time step 2.

```
El{1,2}
```

# calce1

## Purpose

Calculate layer errors for one time step

## Syntax

```
El = calce1(net,A,Tl)
```

## Description

This function calculates the errors of each layer in response to layer outputs and targets, for a single time step. Calculating errors for a single time step is useful for sequential iterative algorithms such as `trains` which need to calculate the network response for each time step individually.

`El = calce1(net,A,Tl)` takes,

`net` — Neural network
`A` — Layer outputs, for a single time step
`Tl` — Layer targets, for a single time step

and returns,

`El` — Layer errors, for a single time step

## Examples

Here we create a linear network with a single input element ranging from 0 to 1, two neurons, and a tap delay on the input with taps at zero, two, and four time steps. The network is also given a recurrent connection from layer 1 to itself with tap delays of [1 2].

```
net = newlin([0 1],2);
net.layerConnect(1,1) = 1;
net.layerWeights{1,1}.delays = [1 2];
```

Here is a single (`Q = 1`) input sequence P with five time steps (`TS = 5`), and the four initial input delay conditions `Pi`, combined inputs `Pc`, and delayed inputs `Pd`.

```
P = {0 0.1 0.3 0.6 0.4};
Pi = {0.2 0.3 0.4 0.1};
Pc = [Pi P];
Pd = calcpd(net,5,1,Pc);
```

Here the two initial layer delay conditions for each of the two neurons are defined, and the networks combined outputs `Ac` and other signals are calculated.

```
Ai = {[0.5; 0.1] [0.6; 0.5]};
```

```
[Ac,N,LWZ,IWZ,BZ] = calca(net,Pd,Ai,1,5);
```

Here we define the layer targets for the two neurons for each of the five time steps, and calculate the layer error using the first time step layer output `Ac(:,5)` (The five is found by adding the number of layer delays, 2, to the time step 1.), and the first time step targets `Tl(:,1)`.

```
Tl = {[0.1;0.2] [0.3;0.1], [0.5;0.6] [0.8;0.9], [0.5;0.1]};
El = calce1(net,Ac(:,3),Tl(:,1))
```

Here we view the network's error for layer 1.

```
El{1}
```

# calcgx

## Purpose

Calculate weight and bias performance gradient as a single vector

## Syntax

```
[gX,normgX] = calcgx(net,X,Pd,BZ,IWZ,LWZ,N,Ac,El,perf,Q,TS);
```

## Description

This function calculates the gradient of a network's performance with respect to its vector of weight and bias values X.

If the network has no layer delays with taps greater than 0 the result is the true gradient.

If the network as layer delays greater than 0, the result is the Elman gradient, an approximation of the true gradient.

`[gX,normgX] = calcgx(net,X,Pd,BZ,IWZ,LWZ,N,Ac,El,perf,Q,TS)` takes,

net — Neural network
X — Vector of weight and bias values
Pd — Delayed inputs
BZ — Concurrent biases
IWZ — Weighted inputs
LWZ — Weighted layer outputs
N — Net inputs
Ac — Combined layer outputs
El — Layer errors
perf — Network performance
Q — Concurrent size
TS — Time steps

and returns,

gX — Gradient dPerf/dX
normgX — Norm of gradient

## Examples

Here we create a linear network with a single input element ranging from 0 to 1, two neurons, and a tap delay on the input with taps at zero, two, and four time steps. The network is also given a recurrent connection from layer 1 to itself with tap delays of [1 2].

```
net = newlin([0 1],2);
```

```
net.layerConnect(1,1) = 1;
net.layerWeights{1,1}.delays = [1 2];
```

Here is a single (`Q = 1`) input sequence `P` with five time steps (`TS = 5`), and the four initial input delay conditions `Pi`, combined inputs `Pc`, and delayed inputs `Pd`.

```
P = {0 0.1 0.3 0.6 0.4};
Pi = {0.2 0.3 0.4 0.1};
Pc = [Pi P];
Pd = calcpd(net,5,1,Pc);
```

Here the two initial layer delay conditions for each of the two neurons, and the layer targets for the two neurons over five time steps are defined.

```
Ai = {[0.5; 0.1] [0.6; 0.5]};
Tl = {[0.1;0.2] [0.3;0.1], [0.5;0.6] [0.8;0.9], [0.5;0.1]};
```

Here the network's weight and bias values are extracted, and the network's performance and other signals are calculated.

```
X = getx(net);
[perf,El,Ac,N,BZ,IWZ,LWZ] = calcperf(net,X,Pd,Tl,Ai,1,5);
```

Finally we can use `calcgz` to calculate the gradient of performance with respect to the weight and bias values `X`.

```
[gX,normgX] = calcgx(net,X,Pd,BZ,IWZ,LWZ,N,Ac,El,perf,1,5);
```

## See Also

`calcjx`, `calcjejj`

# calcjejj

**Purpose** Calculate Jacobian performance vector

**Syntax**

```
[je,jj,normje] = calcjejj(net,Pd,BZ,IWZ,LWZ,N,Ac,El,Q,TS,MR)
```

**Description** This function calculates two values (related to the Jacobian of a network) required to calculate the network's Hessian, in a memory efficient way.

Two values needed to calculate the Hessian of a network are J*E (Jacobian times errors) and J'J (Jacobian squared). However the Jacobian J can take up a lot of memory. This function calculates J*E and J'J by dividing up training vectors into groups, calculating partial Jacobians Ji and its associated values Ji*Ei and Ji'Ji, then summing the partial values into the full J*E and J'J values.

This allows the J*E and J'J values to be calculated with a series of smaller Ji matrices, instead of a larger J matrix.

`[je,jj,normgX] = calcjejj(net,PD,BZ,IWZ,LWZ,N,Ac,El,Q,TS,MR)` takes,

`net` — Neural network
`PD` — Delayed inputs
`BZ` — Concurrent biases
`IWZ` — Weighted inputs
`LWZ` — Weighted layer outputs
`N` — Net inputs
`Ac` — Combined layer outputs
`El` — Layer errors
`Q` — Concurrent size
`TS` — Time steps
`MR` — Memory reduction factor

and returns,

`je` — Jacobian times errors
`jj` — Jacobian transposed time the `Jacobian.normgX`
`normgX` — Norm of gradient

## Examples

Here we create a linear network with a single input element ranging from 0 to 1, two neurons, and a tap delay on the input with taps at zero, two, and four time steps. The network is also given a recurrent connection from layer 1 to itself with tap delays of [1 2].

```
net = newlin([0 1],2);
net.layerConnect(1,1) = 1;
net.layerWeights{1,1}.delays = [1 2];
```

Here is a single (`Q = 1`) input sequence `P` with five time steps (`TS = 5`), and the four initial input delay conditions `Pi`, combined inputs `Pc`, and delayed inputs `Pd`.

```
P = {0 0.1 0.3 0.6 0.4};
Pi = {0.2 0.3 0.4 0.1};
Pc = [Pi P];
Pd = calcpd(net,5,1,Pc);
```

Here the two initial layer delay conditions for each of the two neurons, and the layer targets for the two neurons over five time steps are defined.

```
Ai = {[0.5; 0.1] [0.6; 0.5]};
Tl = {[0.1;0.2] [0.3;0.1], [0.5;0.6] [0.8;0.9], [0.5;0.1]};
```

Here the network's weight and bias values are extracted, and the network's performance and other signals are calculated.

```
[perf,El,Ac,N,BZ,IWZ,LWZ] = calcperf(net,X,Pd,Tl,Ai,1,5);
```

Finally we can use `calcgx` to calculate the Jacobian times error, Jacobian squared, and the norm of the Jocobian times error using a memory reduction of 2.

```
[je,jj,normje] = calcjejj(net,Pd,BZ,IWZ,LWZ,N,Ac,El,1,5,2);
```

The results should be the same whatever the memory reduction used. Here a memory reduction of 3 is used.

```
[je,jj,normje] = calcjejj(net,Pd,BZ,IWZ,LWZ,N,Ac,El,1,5,3);
```

## See Also

`calcjx, calcjejj`

# calcjx

## Purpose

Calculate weight and bias performance Jacobian as a single matrix

## Syntax

```
jx = calcjx(net,PD,BZ,IWZ,LWZ,N,Ac,Q,TS)
```

## Description

This function calculates the Jacobian of a network's errors with respect to its vector of weight and bias values `X`.

`[jX] = calcjx(net,PD,BZ,IWZ,LWZ,N,Ac,Q,TS)` takes,

`net` — Neural network

`PD` — Delayed inputs

`BZ` — Concurrent biases

`IWZ` — Weighted inputs

`LWZ` — Weighted layer outputs

`N` — Net inputs

`Ac` — Combined layer outputs

`Q` — Concurrent size

`TS` — Time steps

and returns,

`jX` — Jacobian of network errors with respect to X

## Examples

Here we create a linear network with a single input element ranging from 0 to 1, two neurons, and a tap delay on the input with taps at zero, two, and four time steps. The network is also given a recurrent connection from layer 1 to itself with tap delays of [1 2].

```
net = newlin([0 1],2);
net.layerConnect(1,1) = 1;
net.layerWeights{1,1}.delays = [1 2];
```

Here is a single (`Q = 1`) input sequence `P` with five time steps (`TS = 5`), and the four initial input delay conditions `Pi`, combined inputs `Pc`, and delayed inputs `Pd`.

```
P = {0 0.1 0.3 0.6 0.4};
Pi = {0.2 0.3 0.4 0.1};
Pc = [Pi P];
Pd = calcpd(net,5,1,Pc);
```

Here the two initial layer delay conditions for each of the two neurons, and the layer targets for the two neurons over five time steps are defined.

```
Ai = {[0.5; 0.1] [0.6; 0.5]};
Tl = {[0.1;0.2] [0.3;0.1], [0.5;0.6] [0.8;0.9], [0.5;0.1]};
```

Here the network's weight and bias values are extracted, and the network's performance and other signals are calculated.

```
[perf,El,Ac,N,BZ,IWZ,LWZ] = calcperf(net,X,Pd,Tl,Ai,1,5);
```

Finally we can use `calcjx` to calculate the Jacobian.

```
jX = calcjx(net,Pd,BZ,IWZ,LWZ,N,Ac,1,5);calcpd
```

## See Also

`calcgx`, `calcjejj`

# calcpd

### Purpose

Calculate delayed network inputs

### Syntax

```
Pd = calcpd(net,TS,Q,Pc)
```

### Description

This function calculates the results of passing the network inputs through each input weights tap delay line.

`Pd = calcpd(net,TS,Q,Pc)` takes,

`net` — Neural network

`TS` — Time steps

`Q` — Concurrent size

`Pc` — Combined inputs = [initial delay conditions, network inputs]

and returns,

`Pd` — Delayed inputs

### Examples

Here we create a linear network with a single input element ranging from 0 to 1, three neurons, and a tap delay on the input with taps at zero, two, and four time steps.

```
net = newlin([0 1],3,[0 2 4]);
```

Here is a single (`Q = 1`) input sequence `P` with eight time steps (`TS = 8`).

```
P = {0 0.1 0.3 0.6 0.4 0.7 0.2 0.1};
```

Here we define the four initial input delay conditions `Pi`.

```
Pi = {0.2 0.3 0.4 0.1};
```

The delayed inputs (the inputs after passing through the tap delays) can be calculated with `calcpd`.

```
Pc = [Pi P];
Pd = calcpd(net,8,1,Pc)
```

Here we view the delayed inputs for input weight going to layer 1, from input 1 at time steps 1 and 2.

```
Pd{1,1,1}
Pd{1,1,2}
```

# calcperf

## Purpose

Calculate network outputs, signals, and performance

## Syntax

```
[perf,El,Ac,N,BZ,IWZ,LWZ]=calcperf(net,X,Pd,Tl,Ai,Q,TS)
```

## Description

This function calculates the outputs of each layer in response to a networks delayed inputs and initial layer delay conditions.

`[perf,El,Ac,N,LWZ,IWZ,BZ] = calcperf(net,X,Pd,Tl,Ai,Q,TS)` takes,

`net` — Neural network  
`X` — Network weight and bias values in a single vector  
`Pd` — Delayed inputs  
`Tl` — Layer targets  
`Ai` — Initial layer delay conditions  
`Q` — Concurrent size  
`TS` — Time steps

and returns,

`perf` — Network performance  
`El` — Layer errors  
`Ac` — Combined layer outputs = [Ai, calculated layer outputs]  
`N` — Net inputs  
`LWZ` — Weighted layer outputs  
`IWZ` — Weighted inputs  
`BZ` — Concurrent biases

## Examples

Here we create a linear network with a single input element ranging from 0 to 1, two neurons, and a tap delay on the input with taps at zero, two, and four time steps. The network is also given a recurrent connection from layer 1 to itself with tap delays of [1 2].

```
net = newlin([0 1],2);
net.layerConnect(1,1) = 1;
net.layerWeights{1,1}.delays = [1 2];
```

Here is a single (Q = 1) input sequence P with five time steps (TS = 5),and the four initial input delay conditions Pi, combined inputs Pc, and delayed inputs Pd.

```
P = {0 0.1 0.3 0.6 0.4};
Pi = {0.2 0.3 0.4 0.1};
Pc = [Pi P];
Pd = calcpd(net,5,1,Pc);
```

Here the two initial layer delay conditions for each of the two neurons are defined.

```
Ai = {[0.5; 0.1] [0.6; 0.5]};
```

Here we define the layer targets for the two neurons for each of the five time steps.

```
Tl = {[0.1;0.2] [0.3;0.1], [0.5;0.6] [0.8;0.9], [0.5;0.1]};
```

Here the network's weight and bias values are extracted.

```
X = getx(net);
```

Here we calculate the network's combined outputs Ac, and other signals described above.

```
[perf,El,Ac,N,BZ,IWZ,LWZ] = calcperf(net,X,Pd,Tl,Ai,1,5)
```

## Purpose

Create all combinations of vectors

## Syntax

```
combvec(a1,a2...)
```

## Description

`combvec(A1,A2...)` takes any number of inputs,

`A1` — Matrix of N1 (column) vectors

`A2` — Matrix of N2 (column) vectors

and returns a matrix of `(N1*N2*...)` column vectors, where the columns consist of all possibilities of `A2` vectors, appended to `A1` vectors, etc.

## Examples

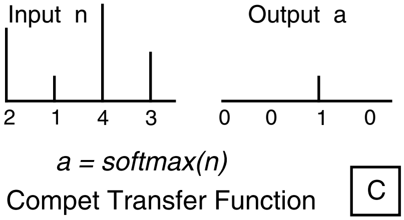
```
a1 = [1 2 3; 4 5 6];
a2 = [7 8; 9 10];
a3 = combvec(a1,a2)
```

# compet

## Purpose

Competitive transfer function

## Graph and Symbol



*a = softmax(n)*
Compet Transfer Function

## Syntax

```
A = compet(N)
info = compet(code)
```

## Description

`compet` is a transfer function. Transfer functions calculate a layer's output from its net input.

`compet(N)` takes one input argument,

`N` - `S x Q` matrix of net input (column) vectors.

and returns output vectors with 1 where each net input vector has its maximum value, and 0 elsewhere.

`compet(code)` returns information about this function.

These codes are defined:

'deriv' — Name of derivative function
'name' — Full name
'output' — Output range
'active' — Active input range

`compet` does not have a derivative function

In many network paradigms it is useful to have a layer whose neurons compete for the ability to output a 1. In biology this is done by strong inhibitory connections between each of the neurons in a layer. The result is that the only neuron that can respond with appreciable output is the neuron whose net input is the highest. All other neurons are inhibited so strongly by the *winning* neuron that their outputs are negligible.

To model this type of layer efficiently on a computer, a competitive transfer function is often used. Such a function transforms the net input vector of a layer of neurons so that the neuron receiving the greatest net input has an output of 1 and all other neurons have outputs of 0.

## Examples

Here we define a net input vector N, calculate the output, and plot both with bar graphs.

```
n = [0; 1; -0.5; 0.5];
a = compet(n);
subplot(2,1,1), bar(n), ylabel('n')
subplot(2,1,2), bar(a), ylabel('a')
```

## Network Use

You can create a standard network that uses compet by calling newc or newpnn.

To change a network so a layer uses compet, set net.layers{i,j}.transferFcn to 'compet'.

In either case, call sim to simulate the network with compet.

See newc or newpnn for simulation examples.

## See Also

sim, softmax

# con2seq

## Purpose

Convert concurrent vectors to sequential vectors

## Syntax

```
s = con2seq(b)
```

## Description

The Neural Network Toolbox arranges concurrent vectors with a matrix, and sequential vectors with a cell array (where the second index is the time step).

con2seq and seq2con allow concurrent vectors to be converted to sequential vectors, and back again.

con2seq(b)takes one input,

b — R x TS matrix

and returns one output,

S — 1 x TS cell array of R x 1 vectors

con2seq(b,TS) can also convert multiple batches,

b — N x 1 cell array of matrices with M*TS columns

TS — Time steps

and will return,

S — N x TS cell array of matrices with M columns

## Examples

Here a batch of three values is converted to a sequence.

```
p1 = [1 4 2]
p2 = con2seq(p1)
```

Here two batches of vectors are converted to two sequences with two time steps.

```
p1 = {[1 3 4 5; 1 1 7 4]; [7 3 4 4; 6 9 4 1]}
p2 = con2seq(p1,2)
```

## See Also

seq2con, concur

## Purpose

Create concurrent bias vectors

## Syntax

```
concur(B,Q)
```

## Description

concur(B,Q)

B — S x 1 bias vector (or Nl x 1 cell array of vectors)

Q — Concurrent size

Returns an S x B matrix of copies of B (or Nl x 1 cell array of matrices).

## Examples

Here concur creates three copies of a bias vector.

```
b = [1; 3; 2; -1];
concur(b,3)
```

## Network Use

To calculate a layer's net input, the layer's weighted inputs must be combined with its biases. The following expression calculates the net input for a layer with the netsum net input function, two input weights, and a bias:

```
n = netsum(z1,z2,b)
```

The above expression works if Z1, Z2, and B are all S x 1 vectors. However, if the network is being simulated by sim (or adapt or train) in response to Q concurrent vectors, then Z1 and Z2 will be S x Q matrices. Before B can be combined with Z1 and Z2, we must make Q copies of it.

```
n = netsum(z1,z2,concur(b,q))
```

## See Also

netsum, netprod, sim, seq2con, con2seq

# ddotprod

## Purpose

Dot product weight derivative function

## Syntax

```
dZ_dP = ddotprod('p',W,P,Z)
dZ_dW = ddotprod('w',W,P,Z)
```

## Description

ddotprod is a weight derivative function.

ddotprod('p',W,P,Z) takes three arguments,

W — S x R weight matrix

P — R x Q inputs

Z — S x Q weighted input

and returns the S x R derivative dZ/dP.

ddotprod('w',W,P,Z) returns the R x Q derivative dZ/dW.

## Examples

Here we define a weight W and input P for an input with three elements and a layer with two neurons.

```
W = [0 -1 0.2; -1.1 1 0];
P = [0.1; 0.6; -0.2];
```

Here we calculate the weighted input with dotprod, then calculate each derivative with ddotprod.

```
Z = dotprod(W,P)
dZ_dP = ddotprod('p',W,P,Z)
dZ_dW = ddotprod('w',W,P,Z)
```

## Algorithm

The derivative of a product of two elements with respect to one element is the other element.

```
dZ/dP = W
dZ/dW = P
```

## See Also

dotprod

## Purpose

Derivative of hard limit transfer function

## Syntax

```
dA_dN = dhardlim(N,A)
```

## Description

`dhardlim` is the derivative function for `hardlim`.

`dhardlim(N,A)` takes two arguments,

N — S x Q net input

A — S x Q output

and returns the S x Q derivative dA/dN.

## Examples

Here we define the net input N for a layer of 3 `hardlim` neurons.

```
N = [0.1; 0.8; -0.7];
```

We calculate the layer's output A with `hardlim` and then the derivative of A with respect to N.

```
A = hardlim(N)
dA_dN = dhardlim(N,A)
```

## Algorithm

The derivative of `hardlim` is calculated as follows:

```
d = 0
```

## See Also

`hardlim`

# dhardlms

## Purpose

Derivative of symmetric hard limit transfer function

## Syntax

```
dA_dN = dhardlms(N,A)
```

## Description

`dhardlms` is the derivative function for `hardlims`.

`dhardlms(N,A)` takes two arguments,

`N` — `S` x `Q` net input

`A` — `S` x `Q` output

and returns the `S` x `Q` derivative `dA/dN`.

## Examples

Here we define the net input `N` for a layer of 3 `hardlims` neurons.

```
N = [0.1; 0.8; -0.7];
```

We calculate the layer's output `A` with `hardlims` and then the derivative of `A` with respect to `N`.

```
A = hardlims(N)
dA_dN = dhardlms(N,A)
```

## Algorithm

The derivative of `hardlims` is calculated as follows:

```
d = 0
```

## See Also

`hardlims`

**Purpose** Display a neural network's properties

**Syntax**

```
disp(net)
```

**To Get Help** Type `help network/disp`

**Description** `disp(net)` displays a network's properties.

**Examples** Here a perceptron is created and displayed.

```
net = newp([-1 1; 0 2],3);
disp(net)
```

**See Also** `display`, `sim`, `init`, `train`, `adapt`

# display

## Purpose

Display the name and properties of a neural network's variables

## Syntax

`display(net)`

## To Get Help

Type `help network/disp`

## Description

`display(net)` displays a network variable's name and properties.

## Examples

Here a perceptron variable is defined and displayed.

```
net = newp([-1 1; 0 2],3);
display(net)
```

`display` is automatically called as follows:

```
net
```

## See Also

`disp, sim, init, train, adapt`

## Purpose

Euclidean distance weight function

## Syntax

```
Z = dist(W,P)
df = dist('deriv')
D = dist(pos)
```

## Description

dist is the Euclidean distance weight function. Weight functions apply weights to an input to get weighted inputs.

dist (W,P) takes these inputs,

W — S x R weight matrix

P — R x Q matrix of Q input (column) vectors

and returns the S x Q matrix of vector distances.

dist('deriv') returns '' because dist does not have a derivative function.

dist is also a layer distance function, which can be used to find the distances between neurons in a layer.

dist(pos) takes one argument,

pos N x S matrix of neuron positions

and returns the S x S matrix of distances.

## Examples

Here we define a random weight matrix W and input vector P and calculate the corresponding weighted input Z.

```
W = rand(4,3);
P = rand(3,1);
Z = dist(W,P)
```

Here we define a random matrix of positions for 10 neurons arranged in three-dimensional space and find their distances.

```
pos = rand(3,10);
D = dist(pos)
```

## Network Use

You can create a standard network that uses dist by calling newpnn or newgrnn.

To change a network so an input weight uses `dist`, set `net.inputWeight{i,j}.weightFcn` to '`dist`'.

For a layer weight set `net.inputWeight{i,j}.weightFcn` to '`dist`'.

To change a network so that a layer's topology uses `dist`, set `net.layers{i}.distanceFcn` to '`dist`'.

In either case, call `sim` to simulate the network with `dist`.

See `newpnn` or `newgrnn` for simulation examples.

## Algorithm

The Euclidean distance d between two vectors `X` and `Y` is:

```
   d = sum((x-y).^2).^0.5
```

## See Also

`sim, dotprod, negdist, normprod, mandist, linkdist`

## Purpose

Log sigmoid transfer derivative function

## Syntax

```
dA_dN = dlogsig(N,A)
```

## Description

dlogsig is the derivative function for logsig.

dlogsig(N,A) takes two arguments,

N — S x Q net input

A — S x Q output

and returns the S x Q derivative dA/dN.

## Examples

Here we define the net input N for a layer of 3 tansig neurons.

```
N = [0.1; 0.8; -0.7];
```

We calculate the layer's output A with logsig and then the derivative of A with respect to N.

```
A = logsig(N)
dA_dN = dlogsig(N,A)
```

## Algorithm

The derivative of logsig is calculated as follows:

```
d = a * (1 - a)
```

## See Also

logsig, tansig, dtansig

# dmae

## Purpose

Mean absolute error performance derivative function

## Syntax

```
dPerf_dE = dmae('e',E,X,PERF,PP)
dPerf_dX = dmae('x',E,X,PERF,PP)
```

## Description

`dmae` is the derivative function for `mae`.

`dmae('d',E,X,PERF,PP)` takes these arguments,

`E` — Matrix or cell array of error vector(s)
`X` — Vector of all weight and bias values
`perf` — Network performance (ignored)
`PP` — Performance parameters (ignored)

and returns the derivative `dPerf/dE`.

`dmae('x',E,X,PERF,PP)` returns the derivative `dPerf/dX`.

## Examples

Here we define `E` and `X` for a network with one 3-element output and six weight and bias values.

```
E = {[1; -2; 0.5]};
X = [0; 0.2; -2.2; 4.1; 0.1; -0.2];
```

Here we calculate the network's mean absolute error performance, and derivatives of performance.

```
perf = mae(E)
dPerf_dE = dmae('e',E,X)
dPerf_dX = dmae('x',E,X)
```

Note that `mae` can be called with only one argument and `dmae` with only three arguments because the other arguments are ignored. The other arguments exist so that `mae` and `dmae` conform to standard performance function argument lists.

## See Also

`mae`

## Purpose

Mean squared error performance derivatives function

## Syntax

```
dPerf_dE = dmse('e',E,X,perf,PP)
dPerf_dX = dmse('x',E,X,perf,PP)
```

## Description

dmse is the derivative function for mse.

dmse('d',E,X,PERF,PP) takes these arguments,

- E — Matrix or cell array of error vector(s)
- X — Vector of all weight and bias values
- perf — Network performance (ignored)
- PP — Performance parameters (ignored)

and returns the derivative dPerf/dE.

dmse('x',E,X,PERF,PP) returns the derivative dPerf/dX.

## Examples

Here we define E and X for a network with one 3-element output and six weight and bias values.

```
E = {[1; -2; 0.5]};
X = [0; 0.2; -2.2; 4.1; 0.1; -0.2];
```

Here we calculate the network's mean squared error performance, and derivatives of performance.

```
perf = mse(E)
dPerf_dE = dmse('e',E,X)
dPerf_dX = dmse('x',E,X)
```

Note that mse can be called with only one argument and dmse with only three arguments because the other arguments are ignored. The other arguments exist so that mse and dmse conform to standard performance function argument lists.

## See Also

mse

# dmsereg

## Purpose

Mean squared error with regularization or performance derivative function

## Syntax

```
dPerf_dE = dmsereg('e',E,X,perf,PP)
dPerf_dX = dmsereg('x',E,X,perf,PP)
```

## Description

`dmsereg` is the derivative function for `msereg`.

`dmsereg('d',E,X,perf,PP)` takes these arguments,

- `E` — Matrix or cell array of error vector(s)
- `X` — Vector of all weight and bias values
- `perf` — Network performance (ignored)
- `PP` — mse performance parameter

where `PP` defines one performance parameters,

- `PP.ratio` — Relative importance of errors vs. weight and bias values

and returns the derivative `dPerf/dE`.

`dmsereg('x',E,X,perf)` returns the derivative `dPerf/dX`.

`mse` has only one performance parameter.

## Examples

Here we define an error `E` and `X` for a network with one 3-element output and six weight and bias values.

```
E = {[1; -2; 0.5]};
X = [0; 0.2; -2.2; 4.1; 0.1; -0.2];
```

Here the ratio performance parameter is defined so that squared errors are 5 times as important as squared weight and bias values.

```
pp.ratio = 5/(5+1);
```

Here we calculate the network's performance, and derivatives of performance.

```
perf = msereg(E,X,pp)
dPerf_dE = dmsereg('e',E,X,perf,pp)
dPerf_dX = dmsereg('x',E,X,perf,pp)
```

## See Also

`msereg`

## Purpose

Derivative of net input product function

## Syntax

```
dN_dZ = dnetprod(Z,N)
```

## Description

dnetprod is the net input derivative function for netprod.

dnetprod takes two arguments,

Z — S x Q weighted input

N — S x Q net input

and returns the S x Q derivative dN/dZ.

## Examples

Here we define two weighted inputs for a layer with three neurons.

```
Z1 = [0; 1; -1];
Z2 = [1; 0.5; 1.2];
```

We calculate the layer's net input N with netprod and then the derivative of N with respect to each weighted input.

```
N = netprod(Z1,Z2)
dN_dZ1 = dnetprod(Z1,N)
dN_dZ2 = dnetprod(Z2,N)
```

## Algorithm

The derivative of a product with respect to any element of that product is the product of the other elements.

## See Also

netsum, netprod, dnetsum

# dnetsum

## Purpose

Sum net input derivative function

## Syntax

```
dN_dZ = dnetsum(Z,N)
```

## Description

`dnetsum` is the net input derivative function for `netsum`.

`dnetsum` takes two arguments,

`Z` — S x Q weighted input<br>
`N` — S x Q net input

and returns the `S` x `Q` derivative `dN/dZ`.

## Examples

Here we define two weighted inputs for a layer with three neurons.

```
Z1 = [0; 1; -1];
Z2 = [1; 0.5; 1.2];
```

We calculate the layer's net input `N` with `netsum` and then the derivative of `N` with respect to each weighted input.

```
N = netsum(Z1,Z2)
dN_dZ1 = dnetsum(Z1,N)
dN_dZ2 = dnetsum(Z2,N)
```

## Algorithm

The derivative of a sum with respect to any element of that sum is always a ones matrix that is the same size as the sum.

## See Also

`netsum, netprod, dnetprod`

## Purpose

Dot product weight function

## Syntax

```
Z = dotprod(W,P)
df = dotprod('deriv')
```

## Description

`dotprod` is the dot product weight function. Weight functions apply weights to an input to get weighted inputs.

`dotprod(W,P)` takes these inputs,

- `W` — S x R weight matrix
- `P` — R x Q matrix of Q input (column) vectors

and returns the `S` x `Q` dot product of `W` and `P`.

## Examples

Here we define a random weight matrix `W` and input vector `P` and calculate the corresponding weighted input `Z`.

```
W = rand(4,3);
P = rand(3,1);
Z = dotprod(W,P)
```

## Network Use

You can create a standard network that uses `dotprod` by calling `newp` or `newlin`.

To change a network so an input weight uses `dotprod`, set `net.inputWeight{i,j}.weightFcn` to 'dotprod'. For a layer weight, set `net.inputWeight{i,j}.weightFcn` to 'dotprod'.

In either case, call `sim` to simulate the network with `dotprod`.

See `newp` and `newlin` for simulation examples.

## See Also

`sim, ddotprod, dist, negdist, normprod`

# dposlin

## Purpose

Derivative of positive linear transfer function

## Syntax

```
dA_dN = dposlin(N,A)
```

## Description

`dposlin` is the derivative function for `poslin`.

`dposlin(N,A)` takes two arguments, and returns the `S` x `Q` derivative `dA/dN`.

## Examples

Here we define the net input `N` for a layer of 3 `poslin` neurons.

```
N = [0.1; 0.8; -0.7];
```

We calculate the layer's output `A` with `poslin` and then the derivative of `A` with respect to `N`.

```
A = poslin(N)
dA_dN = dposlin(N,A)
```

## Algorithm

The derivative of `poslin` is calculated as follows:

d = 1, if 0 <= n; 0, Otherwise.

## See Also

`poslin`

## Purpose

Linear transfer derivative function

## Syntax

```
dA_dN = dpurelin(N,A)
```

## Description

`dpurelin` is the derivative function for `logsig`.

`dpurelin(N,A)` takes two arguments,

- `N` — S x Q net input
- `A` — S x Q output

and returns the `S x Q` derivative `dA_dN`.

## Examples

Here we define the net input `N` for a layer of 3 `purelin` neurons.

```
N = [0.1; 0.8; -0.7];
```

We calculate the layer's output `A` with `purelin` and then the derivative of `A` with respect to `N`.

```
A = purelin(N)
dA_dN = dpurelin(N,A)
```

## Algorithm

The derivative of `purelin` is calculated as follows:

```
D(i,q) = 1
```

## See Also

`purelin`

# dradbas

## Purpose

Derivative of radial basis transfer function

## Syntax

```
dA_dN = dradbas(N,A)
```

## Description

`dradbas` is the derivative function for `radbas`.

`dradbas(N,A)` takes two arguments,

`N` — S x Q net input
`A` — S x Q output

and returns the `S` x `Q` derivative `dA/dN`.

## Examples

Here we define the net input `N` for a layer of 3 `radbas` neurons.

```
N = [0.1; 0.8; -0.7];
```

We calculate the layer's output `A` with `radbas` and then the derivative of `A` with respect to `N`.

```
A = radbas(N)
```

## Algorithm

The derivative of `radbas` is calculated as follows:

```
d = -2*n*a
```

## See Also

`radbas`

## Purpose

Derivative of saturating linear transfer function

## Syntax

```
dA_dN = dsatlin(N,A)
```

## Description

`dsatlin` is the derivative function for `satlin`.

`dsatlin(N,A)` takes two arguments,

`N` — S x Q net input
`A` — S x Q output
and returns the `S x Q` derivative `dA/dN`.

## Examples

Here we define the net input `N` for a layer of 3 `satlin` neurons.

```
N = [0.1; 0.8; -0.7];
```

We calculate the layer's output `A` with `satlin` and then the derivative of `A` with respect to `N`.

```
A = satlin(N)
dA_dN = dsatlin(N,A)
```

## Algorithm

The derivative of `satlin` is calculated as follows:

d = 1, if 0 <= n <= 1; 0, otherwise.

## See Also

`satlin`

# dsatlins

## Purpose

Derivative of symmetric saturating linear transfer function

## Syntax

```
dA_dN = dsatlins(N,A)
```

## Description

`dsatlins` is the derivative function for `satlins`.

`dsatlins(N,A)` takes two arguments,

`N` — S x Q net input
`A` — S x Q output

and returns the `S x Q` derivative `dA/dN`.

## Examples

Here we define the net input `N` for a layer of 3 `satlins` neurons.

```
N = [0.1; 0.8; -0.7];
```

We calculate the layer's output `A` with `satlins` and then the derivative of `A` with respect to `N`.

```
A = satlins(N)
dA_dN = dsatlins(N,A)
```

## Algorithm

The derivative of `satlins` is calculated as follows:

d = 1, if -1 <= n <= 1; 0, otherwise.

## See Also

`satlins`

## Purpose

Sum squared error performance derivative function

## Syntax

```
dPerf_dE = dsse('e',E,X,perf,PP)
dPerf_dX = dsse('x',E,X,perf,PP)
```

## Description

dsse is the derivative function for sse.

dsse('d',E,X,perf,PP) takes these arguments,

| | |
|---|---|
| E | — Matrix or cell array of error vector(s) |
| X | — Vector of all weight and bias values |
| perf | — Network performance (ignored) |
| PP | — Performance parameters (ignored) |

and returns the derivative dPerf_dE.

dsse('x',E,X,perf,PP)returns the derivative dPerf_dX.

## Examples

Here we define an error E and X for a network with one 3-element output and six weight and bias values.

```
E = {[1; -2; 0.5]};
X = [0; 0.2; -2.2; 4.1; 0.1; -0.2];
```

Here we calculate the network’s sum squared error performance, and derivatives of performance.

```
perf = sse(E)
dPerf_dE = dsse('e',E,X)
dPerf_dX = dsse('x',E,X)
```

Note that sse can be called with only one argument and dsse with only three arguments because the other arguments are ignored. The other arguments exist so that sse and dsse conform to standard performance function argument lists.

## See Also

sse

# dtansig

## Purpose

Hyperbolic tangent sigmoid transfer derivative function

## Syntax

```
dA_dN = dtansig(N,A)
```

## Description

`dtansig` is the derivative function for `tansig`.

`dtansig(N,A)` takes two arguments,

`N` — S x Q net input
`A` — S x Q output

and returns the `S x Q` derivative `dA/dN`.

## Examples

Here we define the net input `N` for a layer of 3 `tansig` neurons.

```
N = [0.1; 0.8; -0.7];
```

We calculate the layer's output `A` with `tansig` and then the derivative of `A` with respect to `N`.

```
A = tansig(N)
dA_dN = dtansig(N,A)
```

## Algorithm

The derivative of `tansig` is calculated as follows:

```
d = 1-a^2
```

## See Also

`tansig, logsig, dlogsig`

## Purpose

Derivative of triangular basis transfer function

## Syntax

```
dA_dN = dtribas(N,A)
```

## Description

dtribas is the derivative function for tribas.

dtribas(N,A) takes two arguments,

N — S x Q net input
A — S x Q output

and returns the S x Q derivative dA/dN.

## Examples

Here we define the net input N for a layer of 3 tribas neurons.

```
N = [0.1; 0.8; -0.7];
```

We calculate the layer's output A with tribas and then the derivative of A with respect to N.

```
A = tribas(N)
dA_dN = dtribas(N,A)
```

## Algorithm

The derivative of tribas is calculated as follows:

d = 1, if -1 <= n < 0; -1, if 0 < n <= 1; 0, otherwise.

## See Also

tribas

# errsurf

## Purpose

Error surface of single input neuron

## Syntax

```
errsurf(P,T,WV,BV,F)
```

## Description

`errsurf(P,T,WV,BV,F)` takes these arguments,

- `P` — 1 x Q matrix of input vectors
- `T` — 1 x Q matrix of target vectors
- `WV` — Row vector of values of `W`
- `BV` — Row vector of values of `B`
- `F` — Transfer function (string)

and returns a matrix of error values over `WV` and `BV`.

## Examples

```
p = [-6.0 -6.1 -4.1 -4.0 +4.0 +4.1 +6.0 +6.1];
t = [+0.0 +0.0 +.97 +.99 +.01 +.03 +1.0 +1.0];
wv = -1:.1:1; bv = -2.5:.25:2.5;
es = errsurf(p,t,wv,bv,'logsig');
plotes(wv,bv,ES,[60 30])
```

## See Also

`plotes`

## Purpose

Form bias and weights into single vector

## Syntax

```
X = formx(net,B,IW,LW)
```

## Description

This function takes weight matrices and bias vectors for a network and reshapes them into a single vector.

`X = formx(net,B,IW,LW)` takes these arguments,

- `net` — Neural network
- `B` — Nlx1 cell array of bias vectors
- `IW` — NlxNi cell array of input weight matrices
- `LW` — NlxNl cell array of layer weight matrices

and returns,

- `X` — Vector of weight and bias values

## Examples

Here we create a network with a two-element input, and one layer of three neurons.

```
net = newff([0 1; -1 1],[3]);
```

We can get view its weight matrices and bias vectors as follows:

```
b = net.b
iw = net.iw
lw = net.lw
```

We can put these values into a single vector as follows:

```
x = formx(net,net.b,net.iw,net.lw))
```

## See Also

`getx, setx`

# gensim

## Purpose

Generate a Simulink® block for neural network simulation

## Syntax

```
gensim(net,st)
```

## To Get Help

Type `help network/gensim`

## Description

`gensim(net,st)` creates a Simulink system containing a block that simulates neural network `net`.

`gensim(net,st)` takes these inputs,

`net` — Neural network

`st` — Sample time (default = 1)

and creates a Simulink system containing a block that simulates neural network `net` with a sampling time of `st`.

If `net` has no input or layer delays (`net.numInputDelays` and `net.numLayerDelays` are both 0) then you can use -1 for `st` to get a continuously sampling network.

## Examples

```
net = newff([0 1],[5 1]);
gensim(net)
```

## Purpose

Get all network weight and bias values as a single vector

## Syntax

```
X = getx(net)
```

## Description

This function gets a network's weight and biases as a vector of values.

```
X = getx(NET)
```

NET — Neural network

X — Vector of weight and bias values

## Examples

Here we create a network with a two-element input, and one layer of three neurons.

```
net = newff([0 1; -1 1],[3]);
```

We can get its weight and bias values as follows:

```
net.iw{1,1}
net.b{1}
```

We can get these values as a single vector as follows:

```
x = getx(net);
```

## See Also

setx, formx

# gridtop

**Purpose** Grid layer topology function

**Syntax**

```
pos = gridtop(dim1,dim2,...,dimN)
```

**Description**

`gridtop` calculates neuron positions for layers whose neurons are arranged in an `N` dimensional grid.

`gridtop(dim1,dim2,...,dimN)` takes `N` arguments,

`dimi` — Length of layer in dimension `i`

and returns an `N` x `S` matrix of `N` coordinate vectors where `S` is the product of `dim1*dim2*...*dimN`.

**Examples**

This code creates and displays a two-dimensional layer with 40 neurons arranged in a 8-by-5 grid.

```
pos = gridtop(8,5); plotsom(pos)
```

This code plots the connections between the same neurons, but shows each neuron at the location of its weight vector. The weights are generated randomly so the layer is very disorganized as is evident in the plot generated by the following code.

```
W = rands(40,2); plotsom(W,dist(pos))
```

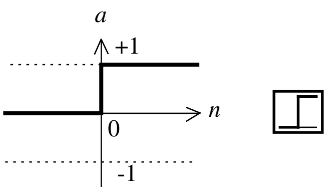
**See Also**

`hextop, randtop`

## Purpose

Hard limit transfer function

## Graph and Symbol



*a = hardlim(n)*

Hard-Limit Transfer Function

## Syntax

```
A = hardlim(N)
info = hardlim(code)
```

## Description

The hard limit transfer function forces a neuron to output a 1 if its net input reaches a threshold, otherwise it outputs 0. This allows a neuron to make a decision or classification. It can say *yes* or *no*. This kind of neuron is often trained with the perceptron learning rule.

`hardlim` is a transfer function. Transfer functions calculate a layer's output from its net input.

`hardlim(N)` takes one input,

`N` — S x Q matrix of net input (column) vectors

and returns 1 where `N` is positive, 0 elsewhere

`hardlim(code)` returns useful information for each `code` string,

`'deriv'` — Name of derivative function

`'name'` — Full name

`'output'` — Output range

`'active'` — Active input range

## Examples

Here is the code to create a plot of the `hardlim` transfer function.

```
n = -5:0.1:5;
a = hardlim(n);
plot(n,a)
```

### Network Use

You can create a standard network that uses `hardlim` by calling `newp`.

To change a network so that a layer uses `hardlim`, set `net.layers{i}.transferFcn` to `'hardlim'`.

In either case call `sim` to simulate the network with `hardlim`.

See `newp` for simulation examples.

### Algorithm

The transfer function output is one is `n` is less than or equal to 0 and zero if `n` is less than 0.

`hardlim(n)` = 1, if `n` >= 0; 0 otherwise.

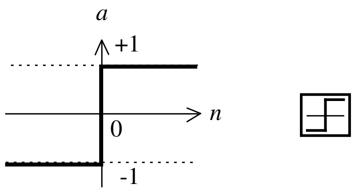
### See Also

`sim, hardlims`

## Purpose

Symmetric hard limit transfer function

## Graph and Symbol



*a = hardlims(n)*

Symmetric Hard-Limit Trans. Funct.

## Syntax

```
A = hardlims(N)
```

```
info = hardlims(code)
```

## Description

The symmetric hard limit transfer function forces a neuron to output a 1 if its net input reaches a threshold. Otherwise it outputs -1. Like the regular hard limit function, this allows a neuron to make a decision or classification. It can say *yes* or *no*.

`hardlims` is a transfer function. Transfer functions calculate a layer's output from its net input.

`hardlims(N)` takes one input,

`N` — S x Q matrix of net input (column) vectors

and returns 1 where `N` is positive, -1 elsewhere.

`hardlims(code)` return useful information for each `code` string:

`'deriv'` — Name of derivative function

`'name'` — Full name

`'output'` — Output range

`'active'` — Active input range

## Examples

Here is the code to create a plot of the `hardlims` transfer function.

```
n = -5:0.1:5;
a = hardlims(n);
plot(n,a)
```

# hardlims

## Network Use

You can create a standard network that uses `hardlims` by calling `newp`.

To change a network so that a layer uses `hardlims`, set `net.layers{i}.transferFcn` to `'hardlims'`.

In either case call `sim` to simulate the network with `hardlims`.

See `newp` for simulation examples.

## Algorithm

The transfer function output is one is `n` is greater than or equal to 0 and -1 otherwise.

`hardlim(n)` = 1, if `n` >= 0; -1 otherwise.

## See Also

`sim, hardlim`

## Purpose

Hexagonal layer topology function

## Syntax

```
pos = hextop(dim1,dim2,...,dimN)
```

## Description

`hextop` calculates the neuron positions for layers whose neurons are arranged in a `N` dimensional hexagonal pattern.

`hextop(dim1,dim2,...,dimN)` takes `N` arguments,

`dimi` — Length of layer in dimension `i`

and returns an `N`-by-`S` matrix of `N` coordinate vectors where `S` is the product of `dim1*dim2*...*dimN`.

## Examples

This code creates and displays a two-dimensional layer with 40 neurons arranged in a 8-by-5 hexagonal pattern.

```
pos = hextop(8,5); plotsom(pos)
```

This code plots the connections between the same neurons, but shows each neuron at the location of its weight vector. The weights are generated randomly so that the layer is very disorganized, as is evident in the fplo generated by the following code.

```
W = rands(40,2); plotsom(W,dist(pos))
```

## See Also

`gridtop`, `randtop`

# hintonw

## Purpose

Hinton graph of weight matrix

## Syntax

```
hintonw(W,maxw,minw)
```

## Description

`hintonw(W,maxw,minw)` takes these inputs,

`W` — S x R weight matrix

`maxw` — Maximum weight, default = max(max(abs(W)))

`minw` — Minimum weight, default = M1/100

and displays a weight matrix represented as a grid of squares.

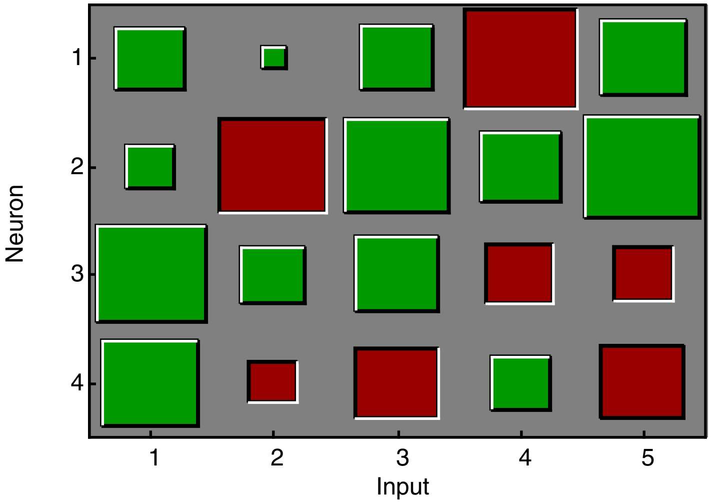
Each square's `area` represents a weight's magnitude. Each square's projection `(color)` represents a weight's sign; inset `(red)` for negative weights, projecting `(green)` for positive.

## Examples

```
W = rands(4,5);
```

The following code displays the matrix graphically.

```
hintonw(W)
```



## See Also

`hintonwb`

## Purpose

Hinton graph of weight matrix and bias vector

## Syntax

```
hintonwb(W,B,maxw,minw)
```

## Description

`hintonwb(W,B,maxw,minw)` takes these inputs,

`W` — S x R weight matrix

`B` — S x 1 bias vector

`maxw` — Maximum weight, default = max(max(abs(W)))

`minw` — Minimum weight, default = M1/100

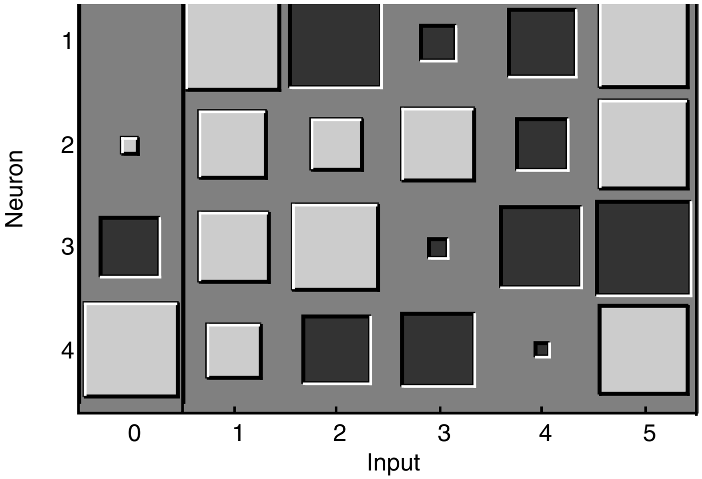
and displays a weight matrix and a bias vector represented as a grid of squares.

Each square's `area` represents a weight's magnitude. Each square's projection `(color)` represents a weight's sign; inset `(red)` for negative weights, projecting `(green)` for positive. The weights are shown on the left.

## Examples

The following code produces the result shown below.

```
W = rands(4,5);
b = rands(4,1);
hintonwb(W,B)
```



## See Also

`hintonw`

# ind2vec

## Purpose

Convert indices to vectors

## Syntax

```
vec = ind2vec(ind)
```

## Description

`ind2vec` and `vec2ind` allow indices to either be represented by themselves, or as vectors containing a 1 in the row of the index they represent.

`ind2vec(ind)` takes one argument,

`ind` — Row vector of indices

and returns a sparse matrix of vectors, with one 1 in each column, as indicated by `ind`.

## Examples

Here four indices are defined and converted to vector representation.

```
ind = [1 3 2 3]
vec = ind2vec(ind)
```

## See Also

`vec2ind`

## Purpose

Initialize a neural network

## Syntax

```
net = init(net)
```

## To Get Help

Type `help network/init`

## Description

`init(net)` returns neural network `net` with weight and bias values updated according to the network initialization function, indicated by `net.initFcn`, and the parameter values, indicated by `net.initParam`.

## Examples

Here a perceptron is created with a two-element input (with ranges of 0 to 1, and -2 to 2) and 1 neuron. Once it is created we can display the neuron's weights and bias.

```
net = newp([0 1;-2 2],1);
net.iw{1,1}
net.b{1}
```

Training the perceptron alters its weight and bias values.

```
P = [0 1 0 1; 0 0 1 1];
T = [0 0 0 1];
net = train(net,P,T);
net.iw{1,1}
net.b{1}
```

`init` reinitializes those weight and bias values.

```
net = init(net);
net.iw{1,1}
net.b{1}
```

The weights and biases are zeros again, which are the initial values used by perceptron networks (see `newp`).

## Algorithm

`init` calls `net.initFcn` to initialize the weight and bias values according to the parameter values `net.initParam`.

Typically, `net.initFcn` is set to `'initlay'` which initializes each layer's weights and biases according to its `net.layers{i}.initFcn`.

Backpropagation networks have `net.layers{i}.initFcn` set to '`initnw`', which calculates the weight and bias values for layer `i` using the Nguyen-Widrow initialization method.

Other networks have `net.layers{i}.initFcn` set to `'initwb'`, which initializes each weight and bias with its own initialization function. The most common weight and bias initialization function is `rands`, which generates random values between -1 and 1.

## See Also

`sim`, `adapt`, `train`, `initlay`, `initnw`, `initwb`, `rands`, `revert`

## Purpose

Conscience bias initialization function

## Syntax

```
b = initcon(s,pr)
```

## Description

`initcon` is a bias initialization function that initializes biases for learning with the `learncon` learning function.

`initcon (S,PR)` takes two arguments,

`S` — Number of rows (neurons)

`PR` — R x 2 matrix of R = [Pmin Pmax], default = [1 1]

and returns an `S x 1` bias vector.

Note that for biases, `R` is always 1. `initcon` could also be used to initialize weights, but it is not recommended for that purpose.

## Examples

Here initial bias values are calculated for a 5 neuron layer.

```
b = initcon(5)
```

## Network Use

You can create a standard network that uses `initcon` to initialize weights by calling `newc`.

To prepare the bias of layer `i` of a custom network to initialize with `initcon`:

1. Set `net.initFcn` to '`initlay`'. (`net.initParam` will automatically become `initlay`'s default parameters.)
2. Set `net.layers{i}.initFcn` to '`initwb`'.
3. Set `net.biases{i}.initFcn` to '`initcon`'.

To initialize the network, call `init`. See `newc` for initialization examples.

## Algorithm

`learncon` updates biases so that each bias value `b(i)` is a function of the average output `c(i)` of the neuron `i` associated with the bias.

`initcon` gets initial bias values by assuming that each neuron has responded to equal numbers of vectors in the "past."

## See Also

`initwb`, `initlay`, `init`, `learncon`

# initlay

## Purpose

Layer-by-layer network initialization function

## Syntax

```
net = initlay(net)
info = initlay(code)
```

## Description

`initlay` is a network initialization function that initializes each layer `i` according to its own initialization function `net.layers{i}.initFcn`.

`initlay(net)` takes,

`net` — Neural network

and returns the network with each layer updated. `initlay(code)` returns useful information for each `code` string:

'`pnames`' — Names of initialization parameters

'`pdefaults`' — Default initialization parameters

`initlay` does not have any initialization parameters

## Network Use

You can create a standard network that uses `initlay` by calling `newp`, `newlin`, `newff`, `newcf`, and many other `new` network functions.

To prepare a custom network to be initialized with `initlay`

1. Set `net.initFcn` to '`initlay`'. (This will set `net.initParam` to the empty matrix [ ] since `initlay` has no initialization parameters.)
2. Set each `net.layers{i}.initFcn` to a layer initialization function. (Examples of such functions are `initwb` and `initnw`).

To initialize the network, call `init`. See `newp` and `newlin` for initialization examples.

## Algorithm

The weights and biases of each layer `i` are initialized according to `net.layers{i}.initFcn`.

## See Also

`initwb, initnw, init`

## Purpose

Nguyen-Widrow layer initialization function

## Syntax

```
net = initnw(net,i)
```

## Description

`initnw` is a layer initialization function that initializes a layer's weights and biases according to the Nguyen-Widrow initialization algorithm. This algorithm chooses values in order to distribute the active region of each neuron in the layer approximately evenly across the layer's input space.

`initnw(net,i)` takes two arguments,

`net` — Neural network

`i` — Index of a layer

and returns the network with layer `i`'s weights and biases updated.

## Network Use

You can create a standard network that uses `initnw` by calling `newff` or `newcf`.

To prepare a custom network to be initialized with `initnw`

1. Set `net.initFcn` to '`initlay`'. (This will set `net.initParam` to the empty matrix [ ] since `initlay` has no initialization parameters.)
2. Set `net.layers{i}.initFcn` to '`initnw`'.

To initialize the network call `init`. See `newff` and `newcf` for training examples.

## Algorithm

The Nguyen-Widrow method generates initial weight and bias values for a layer, so that the active regions of the layer's neurons will be distributed approximately evenly over the input space.

Advantages over purely random weights and biases are

- Few neurons are wasted (since all the neurons are in the input space).
- Training works faster (since each area of the input space has neurons). The Nguyen-Widrow method can only be applied to layers
  - with a bias
  - with weights whose "`weightFcn`" is `dotprod`
  - with "`netInputFcn`" set to `netsum`

If these conditions are not met, then `initnw` uses `rands` to initialize the layer's weights and biases.

**See Also** `initwb`, `initlay`, `init`

## Purpose

By-weight-and-bias layer initialization function

## Syntax

```
net = initwb(net,i)
```

## Description

`initwb` is a layer initialization function that initializes a layer's weights and biases according to their own initialization functions.

`initwb(net,i)` takes two arguments,

`net` — Neural network
`i` — Index of a layer

and returns the network with layer `i`'s weights and biases updated.

## Network Use

You can create a standard network that uses `initwb` by calling `newp` or `newlin`.

To prepare a custom network to be initialized with `initwb`

1. Set `net.initFcn` to `'initlay'`. (This will set `net.initParam` to the empty matrix [ ] since `initlay` has no initialization parameters.)
2. Set `net.layers{i}.initFcn` to `'initwb'`.
3. Set each `net.inputWeights{i,j}.initFcn` to a weight initialization function. Set each `net.layerWeights{i,j}.initFcn` to a weight initialization function. Set each `net.biases{i}.initFcn` to a bias initialization function. (Examples of such functions are `rands` and `midpoint`.)

To initialize the network, call `init`.

See `newp` and `newlin` for training examples.

## Algorithm

Each weight (bias) in layer `i` is set to new values calculated according to its weight (bias) initialization function.

## See Also

`initnw, initlay, init`

# initzero

## Purpose

Zero weight and bias initialization function

## Syntax

```
W = initzero(S,PR)
b = initzero(S,[1 1])
```

## Description

`initzero(S,PR)` takes two arguments,

`S` — Number of rows (neurons)

`PR` — R x 2 matrix of input value ranges = `[Pmin Pmax]`

and returns an `S` x `R` weight matrix of zeros.

`initzero(S,[1 1])` returns `S` x 1 bias vector of zeros.

## Examples

Here initial weights and biases are calculated for a layer with two inputs ranging over [0 1] and [-2 2], and 4 neurons.

```
W = initzero(5,[0 1; -2 2])
b = initzero(5,[1 1])
```

## Network Use

You can create a standard network that uses `initzero` to initialize its weights by calling `newp` or `newlin`.

To prepare the weights and the bias of layer `i` of a custom network to be initialized with `midpoint`

1. Set `net.initFcn` to '`initlay`'. (`net.initParam` will automatically become `initlay`'s default parameters.)
2. Set `net.layers{i}.initFcn` to '`initwb`'.
3. Set each `net.inputWeights{i,j}.initFcn` to '`initzero`'. Set each `net.layerWeights{i,j}.initFcn` to '`initzero`'. Set each `net.biases{i}.initFcn` to '`initzero`'.

To initialize the network, call `init`.

See `newp` or `newlin` for initialization examples.

## See Also

`initwb, initlay, init`

## Purpose

Conscience bias learning function

## Syntax

```
[dB,LS] = learncon(B,P,Z,N,A,T,E,gW,gA,D,LP,LS)
info = learncon(code)
```

## Description

`learncon` is the conscience bias learning function used to increase the net input to neurons that have the lowest average output until each neuron responds approximately an equal percentage of the time.

`learncon(B,P,Z,N,A,T,E,gW,gA,D,LP,LS)` takes several inputs,

`B` — S x 1 bias vector

`P` — 1x Q ones vector

`Z` — S x Q weighted input vectors

`N` — S x Q net input vectors

`A` — S x Q output vectors

`T` — S x Q layer target vectors

`E` — S x Q layer error vectors

`gW` — S x R gradient with respect to performance

`gA` — S x Q output gradient with respect to performance

`D` — S x S neuron distances

`LP` — Learning parameters, none, `LP = []`

`LS` — Learning state, initially should be = `[]`

and returns

`dB` — S x 1 weight (or bias) change matrix

`LS` — New learning state

Learning occurs according to `learncon`'s learning parameter, shown here with its default value.

`LP.lr - 0.001` — Learning rate

`learncon(code)` returns useful information for each `code` string.

`'pnames'` — Names of learning parameters

`'pdefaults'` — Default learning parameters

`'needg'` — Returns 1 if this function uses `gW` or `gA`

Neural Network Toolbox 2.0 compatibility: The LP.lr described above equals 1 minus the bias time constant used by trainc in Neural Network Toolbox 2.0.

## Examples

Here we define a random output A, and bias vector W for a layer with 3 neurons. We also define the learning rate LR.

```
a = rand(3,1);
b = rand(3,1);
lp.lr = 0.5;
```

Since learncon only needs these values to calculate a bias change (see algorithm below), we will use them to do so.

```
dW = learncon(b,[],[],[],a,[],[],[],[],[],lp,[])
```

## Network Use

To prepare the bias of layer i of a custom network to learn with learncon

1. Set net.trainFcn to 'trainr'. (net.trainParam will automatically become trainr's default parameters.)
2. Set net.adaptFcn to 'trains'. (net.adaptParam will automatically become trains's default parameters.)
3. Set net.inputWeights{i}.learnFcn to 'learncon'. Set each net.layerWeights{i,j}.learnFcn to 'learncon'. (Each weight learning parameter property will automatically be set to learncon's default parameters.)

To train the network (or enable it to adapt)

1. Set net.trainParam (or net.adaptParam) properties as desired.
2. Call train (or adapt).

## Algorithm

learncon calculates the bias change db for a given neuron by first updating each neuron's *conscience*, i.e. the running average of its output:

```
c = (1-lr)*c + lr*a
```

The conscience is then used to compute a bias for the neuron that is greatest for smaller conscience values.

```
b = exp(1-log(c)) - b
```

(Note that `learncon` is able to recover `C` each time it is called from the bias values.)

## See Also

`learnk, learnos, adapt, train`

# learngd

## Purpose

Gradient descent weight and bias learning function

## Syntax

```
[dW,LS] = learngd(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)
[db,LS] = learngd(b,ones(1,Q),Z,N,A,T,E,gW,gA,D,LP,LS)
info = learngd(code)
```

## Description

`learngd` is the gradient descent weight and bias learning function.

`learngd(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)` takes several inputs,

`W` — S x R weight matrix (or S x 1 bias vector)<br>
`P` — R x Q input vectors (or ones(1,Q))<br>
`Z` — S x Q weighted input vectors<br>
`N` — S x Q net input vectors<br>
`A` — S x Q output vectors<br>
`T` — S x Q layer target vectors<br>
`E` — S x Q layer error vectors<br>
`gW` — S x R gradient with respect to performance<br>
`gA` — S x Q output gradient with respect to performance<br>
`D` — S x S neuron distances<br>
`LP` — Learning parameters, none, `LP = []`<br>
`LS` — Learning state, initially should be `= []`

and returns,

`dW` — S x R weight (or bias) change matrix<br>
`LS` — New learning state

Learning occurs according to `learngd`'s learning parameter shown here with its default value.

`LP.lr - 0.01` — Learning rate

`learngd(code)` returns useful information for each `code` string:

`'pnames'` -— Names of learning parameters<br>
`'pdefaults'` — Default learning parameters<br>
`'needg'` — Returns 1 if this function uses `gW` or `gA`

## Examples

Here we define a random gradient gW for a weight going to a layer with 3 neurons, from an input with 2 elements. We also define a learning rate of 0.5.

```
gW = rand(3,2);
lp.lr = 0.5;
```

Since learngd only needs these values to calculate a weight change (see algorithm below), we will use them to do so.

```
dW = learngd([],[],[],[],[],[],[],gW,[],[],lp,[])
```

## Network Use

You can create a standard network that uses learngd with newff, newcf, or newelm. To prepare the weights and the bias of layer i of a custom network to adapt with learngd

1. Set net.adaptFcn to 'trains'. net.adaptParam will automatically become trains's default parameters.
2. Set each net.inputWeights{i,j}.learnFcn to 'learngd'. Set each net.layerWeights{i,j}.learnFcn to 'learngd'. Set net.biases{i}.learnFcn to 'learngd'. Each weight and bias learning parameter property will automatically be set to learngd's default parameters.

To allow the network to adapt

1. Set net.adaptParam properties to desired values.
2. Call adapt with the network.

See newff or newcf for examples.

## Algorithm

learngd calculates the weight change dW for a given neuron from the neuron's input P and error E, and the weight (or bias) learning rate LR, according to the gradient descent: dw = lr*gW.

## See Also

learngdm, newff, newcf, adapt, train

# learngdm

## Purpose

Gradient descent with momentum weight and bias learning function

## Syntax

```
[dW,LS] = learngdm(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)
[db,LS] = learngdm(b,ones(1,Q),Z,N,A,T,E,gW,gA,D,LP,LS)
info = learngdm(code)
```

## Description

`learngdm` is the gradient descent with momentum weight and bias learning function.

`learngdm(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)` takes several inputs,

`W` — S x R weight matrix (or S x 1 bias vector)

`P` — R x Q input vectors (or ones(1,Q))

`Z` — S x Q weighted input vectors

`N` — S x Q net input vectors

`A` — S x Q output vectors

`T` — S x Q layer target vectors

`E` — S x Q layer error vectors

`gW` — S x R gradient with respect to performance

`gA` — S x Q output gradient with respect to performance

`D` — S x S neuron distances

`LP` — Learning parameters, none, `LP = []`

`LS` — Learning state, initially should be `= []`

and returns,

`dW` — S x R weight (or bias) change matrix

`LS` — New learning state

Learning occurs according to `learngdm`'s learning parameters, shown here with their default values.

`LP.lr - 0.01` — Learning rate

`LP.mc - 0.9` — Momentum constant

learngdm(code) returns useful information for each code string:

'pnames' Names of learning parameters

'pdefaults' Default learning parameters

'needg' Returns 1 if this function uses gW or gA

## Examples

Here we define a random gradient G for a weight going to a layer with 3 neurons, from an input with 2 elements. We also define a learning rate of 0.5 and momentum constant of 0.8;

```
gW = rand(3,2);
lp.lr = 0.5;
lp.mc = 0.8;
```

Since learngdm only needs these values to calculate a weight change (see algorithm below), we will use them to do so. We will use the default initial learning state.

```
ls = [];
[dW,ls] = learngdm([],[],[],[],[],[],[],gW,[],[],lp,ls)
```

learngdm returns the weight change and a new learning state.

## Network Use

You can create a standard network that uses learngdm with newff, newcf, or newelm.

To prepare the weights and the bias of layer i of a custom network to adapt with learngdm

1. Set net.adaptFcn to 'trains'. net.adaptParam will automatically become trains's default parameters.
2. Set each net.inputWeights{i,j}.learnFcn to 'learngdm'. Set each net.layerWeights{i,j}.learnFcn to 'learngdm'. Set net.biases{i}.learnFcn to 'learngdm'. Each weight and bias learning parameter property will automatically be set to learngdm's default parameters.

To allow the network to adapt

1. Set net.adaptParam properties to desired values.
2. Call adapt with the network.

See newff or newcf for examples.

## Algorithm

learngdm calculates the weight change dW for a given neuron from the neuron's input P and error E, the weight (or bias) W, learning rate LR, and momentum constant MC, according to gradient descent with momentum:

```
dW = mc*dWprev + (1-mc)*lr*gW
```

The previous weight change dWprev is stored and read from the learning state LS.

## See Also

learngd, newff, newcf, adapt, train

## Purpose

Hebb weight learning rule

## Syntax

```
[dW,LS] = learnh(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)
info = learnh(code)
```

## Description

`learnh` is the Hebb weight learning function.

`learnh(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)` takes several inputs,

`W` — S x R weight matrix (or S x 1 bias vector)

`P` — R x Q input vectors (or ones(1,Q))

`Z` — S x Q weighted input vectors

`N` — S x Q net input vectors

`A` — S x Q output vectors

`T` — S x Q layer target vectors

`E` — S x Q layer error vectors

`gW` — S x R gradient with respect to performance

`gA` — S x Q output gradient with respect to performance

`D` — S x S neuron distances

`LP` — Learning parameters, none, `LP = []`

`LS` — Learning state, initially should be = `[]`

and returns,

`dW` — S x R weight (or bias) change matrix

`LS` — New learning state

Learning occurs according to `learnh`'s learning parameter, shown here with its default value.

`LP.lr - 0.01` — Learning rate

`learnh(code)` returns useful information for each `code` string:

`'pnames'` — Names of learning parameters

`'pdefaults'` — Default learning parameters

`'needg'` — Returns 1 if this function uses `gW` or `gA`

## Examples

Here we define a random input P and output A for a layer with a two-element input and three neurons. We also define the learning rate LR.

```
p = rand(2,1);
a = rand(3,1);
lp.lr = 0.5;
```

Since learnh only needs these values to calculate a weight change (see algorithm below), we will use them to do so.

```
dW = learnh([],p,[],[],a,[],[],[],[],[],lp,[])
```

## Network Use

To prepare the weights and the bias of layer i of a custom network to learn with learnh

1. Set net.trainFcn to 'trainr'. (net.trainParam will automatically become trainr's default parameters.)
2. Set net.adaptFcn to 'trains'. (net.adaptParam will automatically become trains's default parameters.)
3. Set each net.inputWeights{i,j}.learnFcn to 'learnh'. Set each net.layerWeights{i,j}.learnFcn to 'learnh'. Each weight learning parameter property will automatically be set to learnh's default parameters.)

To train the network (or enable it to adapt)

1. Set net.trainParam (net.adaptParam) properties to desired values.
2. Call train (adapt).

## Algorithm

learnh calculates the weight change dW for a given neuron from the neuron's input P, output A, and learning rate LR according to the Hebb learning rule:

```
dw =  lr*a*p'
```

## See Also

learnhd, adapt, train

## References

Hebb, D.O., *The Organization of Behavior,* New York: Wiley, 1949.

## Purpose

Hebb with decay weight learning rule

## Syntax

```
[dW,LS] = learnhd(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)
info = learnhd(code)
```

## Description

`learnhd` is the Hebb weight learning function.

`learnhd(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)` takes several inputs,

`W` — S x R weight matrix (or S x 1 bias vector)

`P` — R x Q input vectors (or ones(1,Q))

`Z` — S x Q weighted input vectors

`N` — S x Q net input vectors

`A` — S x Q output vectors

`T` — S x Q layer target vectors

`E` — S x Q layer error vectors

`gW` — S x R gradient with respect to performance

`gA` — S x Q output gradient with respect to performance

`D` — S x S neuron distances

`LP` — Learning parameters, none, `LP = []`

`LS` — Learning state, initially should be `= []`

and returns,

`dW` — S x R weight (or bias) change matrix

`LS` — New learning state

Learning occurs according to `learnhd`'s learning parameters shown here with default values.

`LP.dr - 0.01` — Decay rate

`LP.lr - 0.1` — Learning rate

`learnhd(code)` returns useful information for each `code` string:

`'pnames'` - — Names of learning parameters

`'pdefaults'` — Default learning parameters

`'needg'` — Returns 1 if this function uses `gW` or `gA`

# learnhd

## Examples

Here we define a random input P, output A, and weights W for a layer with a two-element input and three neurons. We also define the decay and learning rates.

```
p = rand(2,1);
a = rand(3,1);
w = rand(3,2);
lp.dr = 0.05;
lp.lr = 0.5;
```

Since learnhd only needs these values to calculate a weight change (see algorithm below), we will use them to do so.

```
dW = learnhd(w,p,[],[],a,[],[],[],[],[],lp,[])
```

## Network Use

To prepare the weights and the bias of layer i of a custom network to learn with learnhd

1. Set net.trainFcn to 'trainr'. (net.trainParam will automatically become trainr's default parameters.)
2. Set net.adaptFcn to 'trains'. (net.adaptParam will automatically become trains's default parameters.)
3. Set each net.inputWeights{i,j}.learnFcn to 'learnhd'. Set each net.layerWeights{i,j}.learnFcn to 'learnhd'. (Each weight learning parameter property will automatically be set to learnhd's default parameters.)

To train the network (or enable it to adapt)

1. Set net.trainParam (net.adaptParam) properties to desired values.
2. Call train (adapt).

## Algorithm

learnhd calculates the weight change dW for a given neuron from the neuron's input P, output A, decay rate DR, and learning rate LR according to the Hebb with decay learning rule:

```
dw =  lr*a*p' - dr*w
```

## See Also

learnh, adapt, train

## Purpose

Instar weight learning function

## Syntax

```
[dW,LS] = learnis(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)
info = learnis(code)
```

## Description

`learnis` is the instar weight learning function.

`learnis(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)` takes several inputs,

- `W` — S x R weight matrix (or S x 1 bias vector)
- `P` — R x Q input vectors (or ones(1,Q))
- `Z` — S x Q weighted input vectors
- `N` — S x Q net input vectors
- `A` — S x Q output vectors
- `T` — S x Q layer target vectors
- `E` — S x Q layer error vectors
- `gW` — S x R gradient with respect to performance
- `gA` — S x Q output gradient with respect to performance
- `D` — S x S neuron distances
- `LP` — Learning parameters, none, `LP = []`
- `LS` — Learning state, initially should be `= []`

and returns,

- `dW` — S x R weight (or bias) change matrix
- `LS` — New learning state

Learning occurs according to `learnis`'s learning parameter, shown here with its default value.

- `LP.lr - 0.01` — Learning rate

`learnis(code)` return useful information for each `code` string:

- `'pnames'` — Names of learning parameters
- `'pdefaults'` — Default learning parameters
- `'needg'` — Returns 1 if this function uses `gW` or `gA`

## Examples

Here we define a random input P, output A, and weight matrix W for a layer with a two-element input and three neurons. We also define the learning rate LR.

```
p = rand(2,1);
a = rand(3,1);
w = rand(3,2);
lp.lr = 0.5;
```

Since learnis only needs these values to calculate a weight change (see algorithm below), we will use them to do so.

```
dW = learnis(w,p,[],[],a,[],[],[],[],[],lp,[])
```

## Network Use

To prepare the weights and the bias of layer i of a custom network so that it can learn with learnis

1. Set net.trainFcn to 'trainr'. (net.trainParam will automatically become trainr's default parameters.)
2. Set net.adaptFcn to 'trains'. (net.adaptParam will automatically become trains's default parameters.)
3. Set each net.inputWeights{i,j}.learnFcn to 'learnis'. Set each net.layerWeights{i,j}.learnFcn to 'learnis'. (Each weight learning parameter property will automatically be set to learnis's default parameters.)

To train the network (or enable it to adapt)

1. Set net.trainParam (net.adaptParam) properties to desired values.
2. Call train (adapt).

## Algorithm

learnis calculates the weight change dW for a given neuron from the neuron's input P, output A, and learning rate LR according to the instar learning rule:

```
dw =  lr*a*(p'-w)
```

## See Also

learnk, learnos, adapt, train

## References

Grossberg, S., *Studies of the Mind and Brain*, Drodrecht, Holland: Reidel Press, 1982.

## Purpose

Kohonen weight learning function

## Syntax

```
[dW,LS] = learnk(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)
info = learnk(code)
```

## Description

`learnk` is the Kohonen weight learning function.

`learnk(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)` takes several inputs,

`W` — S x R weight matrix (or S x 1 bias vector)

`P` — R x Q input vectors (or ones(1,Q))

`Z` — S x Q weighted input vectors

`N` — S x Q net input vectors

`A` — S x Q output vectors

`T` — S x Q layer target vectors

`E` — S x Q layer error vectors

`gW` — S x R gradient with respect to performance

`gA` — S x Q output gradient with respect to performance

`D` — S x S neuron distances

`LP` — Learning parameters, none, `LP = []`

`LS` — Learning state, initially should be = `[]`

and returns,

`dW` — S x R weight (or bias) change matrix

`LS` — New learning state

Learning occurs according to `learnk`'s learning parameter, shown here with its default value.

`LP.lr - 0.01` — Learning rate

`learnk(code)` returns useful information for each `code` string:

`'pnames'` — Names of learning parameters

`'pdefaults'` — Default learning parameters

`'needg'` — Returns 1 if this function uses `gW` or `gA`

## Examples

Here we define a random input P, output A, and weight matrix W for a layer with a two-element input and three neurons. We also define the learning rate LR.

```
p = rand(2,1);
a = rand(3,1);
w = rand(3,2);
lp.lr = 0.5;
```

Since learnk only needs these values to calculate a weight change (see algorithm below), we will use them to do so.

```
dW = learnk(w,p,[],[],a,[],[],[],[],[],lp,[])
```

## Network Use

To prepare the weights of layer i of a custom network to learn with learnk

1. Set net.trainFcn to 'trainr'. (net.trainParam will automatically become trainr's default parameters.)
2. Set net.adaptFcn to 'trains'. (net.adaptParam will automatically become trains's default parameters.)
3. Set each net.inputWeights{i,j}.learnFcn to 'learnk'. Set each net.layerWeights{i,j}.learnFcn to 'learnk'. (Each weight learning parameter property will automatically be set to learnk's default parameters.)

To train the network (or enable it to adapt)

1. Set net.trainParam (or net.adaptParam) properties as desired.
2. Call train (or adapt).

## Algorithm

learnk calculates the weight change dW for a given neuron from the neuron's input P, output A, and learning rate LR according to the Kohonen learning rule:

dw = lr*(p'-w), if a ~= 0; = 0, otherwise.

## See Also

learnis, learnos, adapt, train

## References

Kohonen, T., *Self-Organizing and Associative Memory*, New York: Springer-Verlag, 1984.

## Purpose

LVQ1 weight learning function

## Syntax

```
[dW,LS] = learnlv1(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)
info = learnlv1(code)
```

## Description

`learnlv1` is the LVQ1 weight learning function.

`learnlv1(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)` takes several inputs,

`W` — S x R weight matrix (or S x 1 bias vector)

`P` — R x Q input vectors (or ones(1,Q))

`Z` — S x Q weighted input vectors

`N` — S x Q net input vectors

`A` — S x Q output vectors

`T` — S x Q layer target vectors

`E` — S x Q layer error vectors

`gW` — S x R weight gradient with respect to performance

`gA` — S x Q output gradient with respect to performance

`D` — S x R neuron distances

`LP` — Learning parameters, none, `LP = []`

`LS` — Learning state, initially should be = `[]`

and returns,

`dW` — S x R weight (or bias) change matrix

`LS` — New learning state

Learning occurs according to `learnlv1`'s learning parameter shown here with its default value.

`LP.lr - 0.01` — Learning rate

`learnlv1(code)` returns useful information for each `code` string:

`'pnames'` — Names of learning parameters

`'pdefaults'` — Default learning parameters

`needg'` — Returns 1 if this function uses `gW` or `gA`

## Examples

Here we define a random input P, output A, weight matrix W, and output gradient gA for a layer with a two-element input and three neurons.

We also define the learning rate LR.

```
p = rand(2,1);
w = rand(3,2);
a = compet(negdist(w,p));
gA = [-1;1; 1];
lp.lr = 0.5;
```

Since learnlv1 only needs these values to calculate a weight change (see algorithm below), we will use them to do so.

```
dW = learnlv1(w,p,[],[],a,[],[],[],gA,[],lp,[])
```

## Network Use

You can create a standard network that uses learnlv1 with newlvq. To prepare the weights of layer i of a custom network to learn with learnlv1

1. Set net.trainFcn to ‘trainr'. (net.trainParam will automatically become trainr’s default parameters.)
2. Set net.adaptFcn to 'trains'. (net.adaptParam will automatically become trains’s default parameters.)
3. Set each net.inputWeights{i,j}.learnFcn to 'learnlv1'. Set each net.layerWeights{i,j}.learnFcn to 'learnlv1'. (Each weight learning parameter property will automatically be set to learnlv1’s default parameters.)

To train the network (or enable it to adapt)

1. Set net.trainParam (or net.adaptParam) properties as desired.
2. Call train (or adapt).

## Algorithm

learnlv1 calculates the weight change dW for a given neuron from the neuron’s input P, output A, output gradient gA and learning rate LR, according to the LVQ1 rule, given i the index of the neuron whose output a(i) is 1:

```
dw(i,:) = +lr*(p-w(i,:)) if gA(i) = 0;= -lr*(p-w(i,:)) if gA(i) = -1
```

## See Also

learnlv2, adapt, train

## Purpose

LVQ2.1 weight learning function

## Syntax

```
[dW,LS] = learnlv2(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)
info = learnlv2(code)
```

## Description

learnlv2 is the LVQ2 weight learning function.

learnlv2(W,P,Z,N,A,T,E,gW,gA,D,LP,LS) takes several inputs,

W — S x R weight matrix (or S x 1 bias vector)

P — R x Q input vectors (or ones(1,Q))

Z — S x Q weighted input vectors

N — S x Q net input vectors

A — S x Q output vectors

T — S x Q layer target vectors

E — S x Q layer error vectors

gW — S x R weight gradient with respect to performance

gA — S x Q output gradient with respect to performance

D — S x S neuron distances

LP — Learning parameters, none, LP = []

LS — Learning state, initially should be = []

and returns,

dW — S x R weight (or bias) change matrix

LS — New learning state

Learning occurs according to learnlv1's learning parameter, shown here with its default value.

LP.lr - 0.01 — Learning rate

LP.window - 0.25 — Window size (0 to 1, typically 0.2 to 0.3)

learnlv2(code) returns useful information for each code string:

'pnames' — Names of learning parameters

'pdefaults' — Default learning parameters

'needg' — Returns 1 if this function uses gW or gA

## Examples

Here we define a sample input P, output A, weight matrix W, and output gradient gA for a layer with a two-element input and three neurons.

We also define the learning rate LR.

```
p = rand(2,1);
w = rand(3,2);
n = negdist(w,p);
a = compet(n);
gA = [-1;1; 1];
lp.lr = 0.5;
```

Since learnlv2 only needs these values to calculate a weight change (see algorithm below), we will use them to do so.

```
dW = learnlv2(w,p,[],n,a,[],[],[],gA,[],lp,[])
```

## Network Use

You can create a standard network that uses learnlv2 with newlvq.

To prepare the weights of layer i of a custom network to learn with learnlv2

1. Set net.trainFcn to 'trainr'. (net.trainParam will automatically become trainr's default parameters.)
2. Set net.adaptFcn to 'trains'. (net.adaptParam will automatically become trains's default parameters.)
3. Set each net.inputWeights{i,j}.learnFcn to 'learnlv2'. Set each net.layerWeights{i,j}.learnFcn to 'learnlv2'. (Each weight learning parameter property will automatically be set to learnlv2's default parameters.)

To train the network (or enable it to adapt)

1. Set net.trainParam (or net.adaptParam) properties as desired.
2. Call train (or adapt).

## Algorithm

learnlv2 implements Learning Vector Quantization 2.1, which works as follows:

For each presentation, if the winning neuron i should not have won, and the runner up j should have, and the distance di between the winning neuron and the input p is roughly equal to the distance dj from the runner up neuron to the input p according to the given window,

```
min(di/dj, dj/di) > (1-window)/(1+window)
```

then move the winning neuron i weights away from the input vector, and move the runner up neuron j weights toward the input according to:

```
dw(i,:) = - lp.lr*(p'-w(i,:))
dw(j,:) = + lp.lr*(p'-w(j,:))
```

## See Also

learnlv1, adapt, train

# learnos

## Purpose

Outstar weight learning function

## Syntax

```
[dW,LS] = learnos(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)
info = learnos(code)
```

## Description

`learnos` is the outstar weight learning function.

`learnos(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)` takes several inputs,

`W` — S x R weight matrix (or S x 1 bias vector)

`P` — R x Q input vectors (or ones(1,Q))

`Z` — S x Q weighted input vectors

`N` — S x Q net input vectors

`A` — S x Q output vectors

`T` — S x Q layer target vectors

`E` — S x Q layer error vectors

`gW` — S x R weight gradient with respect to performance

`gA` — S x Q output gradient with respect to performance

`D` — S x S neuron distances

`LP` — Learning parameters, none, `LP = []`

`LS` — Learning state, initially should be = `[]`

and returns

`dW` — S x R weight (or bias) change matrix

`LS` — New learning state

Learning occurs according to `learnos`'s learning parameter, shown here with its default value.

`LP.lr - 0.01` — Learning rate

`learnos(code)` returns useful information for each `code` string:

`'pnames'` — Names of learning parameters

`'pdefaults'` — Default learning parameters

`'needg'` — Returns 1 if this function uses `gW` or `gA`

## Examples

Here we define a random input P, output A, and weight matrix W for a layer with a two-element input and three neurons. We also define the learning rate LR.

```
p = rand(2,1);
a = rand(3,1);
w = rand(3,2);
lp.lr = 0.5;
```

Since learnos only needs these values to calculate a weight change (see algorithm below), we will use them to do so.

```
dW = learnos(w,p,[],[],a,[],[],[],[],[],lp,[])
```

## Network Use

To prepare the weights and the bias of layer i of a custom network to learn with learnos

1. Set net.trainFcn to 'trainr'. (net.trainParam will automatically become trainr's default parameters.)
2. Set net.adaptFcn to 'trains'. (net.adaptParam will automatically become trains's default parameters.)
3. Set each net.inputWeights{i,j}.learnFcn to 'learnos'. Set each net.layerWeights{i,j}.learnFcn to 'learnos'. (Each weight learning parameter property will automatically be set to learnos's default parameters.)

To train the network (or enable it to adapt)

1. Set net.trainParam (net.adaptParam) properties to desired values.
2. Call train (adapt).

## Algorithm

learnos calculates the weight change dW for a given neuron from the neuron's input P, output A, and learning rate LR according to the outstar learning rule:

```
dw =  lr*(a-w)*p'
```

## See Also

learnis, learnk, adapt, train

## References

Grossberg, S., *Studies of the Mind and Brain*, Drodrecht, Holland: Reidel Press, 1982.

# learnp

## Purpose

Perceptron weight and bias learning function

## Syntax

```
[dW,LS] = learnp(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)
[db,LS] = learnp(b,ones(1,Q),Z,N,A,T,E,gW,gA,D,LP,LS)
info = learnp(code)
```

## Description

`learnp` is the perceptron weight/bias learning function.

`learnp(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)` takes several inputs,

- `W` — S x R weight matrix (or b, and S x 1 bias vector)
- `P` — R x Q input vectors (or ones(1,Q))
- `Z` — S x Q weighted input vectors
- `N` — S x Q net input vectors
- `A` — S x Q output vectors
- `T` — S x Q layer target vectors
- `E` — S x Q layer error vectors
- `gW` — S x R weight gradient with respect to performance
- `gA` — S x Q output gradient with respect to performance
- `D` — S x S neuron distances
- `LP` — Learning parameters, none, `LP = []`
- `LS` — Learning state, initially should be = `[]`

and returns,

- `dW` — S x R weight (or bias) change matrix
- `LS` — New learning state

`learnp(code)` returns useful information for each `code` string:

- `'pnames'` — Names of learning parameters
- `'pdefaults'` — Default learning parameters
- `'needg'` — Returns 1 if this function uses `gW` or `gA`

## Examples

Here we define a random input `P` and error `E` to a layer with a two-element input and three neurons.

```
p = rand(2,1);
```

```
e = rand(3,1);
```

Since learnp only needs these values to calculate a weight change (see algorithm below), we will use them to do so.

```
dW = learnp([],p,[],[],[],[],e,[],[],[],[],[])
```

## Network Use

You can create a standard network that uses learnp with newp.

To prepare the weights and the bias of layer i of a custom network to learn with learnp

1. Set net.trainFcn to 'trainb'. (net.trainParam will automatically become trainb's default parameters.)
2. Set net.adaptFcn to 'trains'. (net.adaptParam will automatically become trains's default parameters.)
3. Set each net.inputWeights{i,j}.learnFcn to 'learnp'. Set each net.layerWeights{i,j}.learnFcn to 'learnp'. Set net.biases{i}.learnFcn to 'learnp'. (Each weight and bias learning parameter property will automatically become the empty matrix since learnp has no learning parameters.)

To train the network (or enable it to adapt)

1. Set net.trainParam (net.adaptParam) properties to desired values.
2. Call train (adapt).

See newp for adaption and training examples.

## Algorithm

learnp calculates the weight change dW for a given neuron from the neuron's input P and error E according to the perceptron learning rule:

```
dw =  0,  if e =  0
=  p', if e =  1
= -p', if e = -1
```

This can be summarized as:

```
dw = e*p'
```

## See Also

learnpn, newp, adapt, train

**References**

Rosenblatt, F., *Principles of Neurodynamics*, Washington D.C.: Spartan Press, 1961.

## Purpose

Normalized perceptron weight and bias learning function

## Syntax

```
[dW,LS] = learnpn(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)
info = learnpn(code)
```

## Description

`learnpn` is a weight and bias learning function. It can result in faster learning than `learnp` when input vectors have widely varying magnitudes.

`learnpn(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)` takes several inputs,

`W` — S x R weight matrix (or S x 1 bias vector)

`P` — R x Q input vectors (or ones(1,Q))

`Z` — S x Q weighted input vectors

`N` — S x Q net input vectors

`A` — S x Q output vectors

`T` — S x Q layer target vectors

`E` — S x Q layer error vectors

`gW` — S x R weight gradient with respect to performance

`gA` — S x Q output gradient with respect to performance

`D` — S x S neuron distances

`LP` — Learning parameters, none, `LP = []`

`LS` — Learning state, initially should be = `[]`

and returns,

`dW` — S x R weight (or bias) change matrix

`LS` — New learning state

`learnpn(code)` returns useful information for each `code` string:

`'pnames'` — Names of learning parameters

`'pdefaults'` — Default learning parameters

`'needg'` — Returns 1 if this function uses `gW` or `gA`

## Examples

Here we define a random input `P` and error `E` to a layer with a two-element input and three neurons.

```
p = rand(2,1);
e = rand(3,1);
```

Since learnpn only needs these values to calculate a weight change (see algorithm below), we will use them to do so.

```
dW = learnpn([],p,[],[],[],[],e,[],[],[],[],[])
```

## Network Use

You can create a standard network that uses learnpn with newp.

To prepare the weights and the bias of layer i of a custom network to learn with learnpn

1. Set net.trainFcn to 'trainb'. (net.trainParam will automatically become trainb's default parameters.)
2. Set net.adaptFcn to 'trains'. (net.adaptParam will automatically become trains's default parameters.)
3. Set each net.inputWeights{i,j}.learnFcn to 'learnpn'. Set each net.layerWeights{i,j}.learnFcn to 'learnpn'. Set net.biases{i}.learnFcn to 'learnpn'. (Each weight and bias learning parameter property will automatically become the empty matrix since learnpn has no learning parameters.)

To train the network (or enable it to adapt):

1. Set net.trainParam (net.adaptParam) properties to desired values.
2. Call train (adapt).

See newp for adaption and training examples.

## Algorithm

learnpn calculates the weight change dW for a given neuron from the neuron's input P and error E according to the normalized perceptron learning rule

```
pn = p / sqrt(1 + p(1)^2 + p(2)^2) + ... + p(R)^2)
dw =  0,   if e =  0
=  pn', if e =  1
= -pn', if e = -1
```

The expression for dW can be summarized as:

```
dw = e*pn'
```

## Limitations

Perceptrons do have one real limitation. The set of input vectors must be linearly separable if a solution is to be found. That is, if the input vectors with targets of 1 cannot be separated by a line or hyperplane from the input vectors associated with values of 0, the perceptron will never be able to classify them correctly.

## See Also

`learnp, newp, adapt, train`

# learnsom

## Purpose

Self-organizing map weight learning function

## Syntax

```
[dW,LS] = learnsom(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)
info = learnsom(code)
```

## Description

`learnsom` is the self-organizing map weight learning function.

`learnsom(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)` takes several inputs,

`W` — S x R weight matrix (or S x 1 bias vector)
`P` — R x Q input vectors (or ones(1,Q))
`Z` — S x Q weighted input vectors
`N` — S x Q net input vectors
`A` — S x Q output vectors
`T` — S x Q layer target vectors
`E` — S x Q layer error vectors
`gW` — S x R weight gradient with respect to performance
`gA` — S x Q output gradient with respect to performance
`D` — S x S neuron distances
`LP` — Learning parameters, none, `LP = []`
`LS` — Learning state, initially should be = `[]`

and returns,

`dW` — S x R weight (or bias) change matrix
`LS` — New learning state

Learning occurs according to `learnsom`'s learning parameter, shown here with its default value.

| | | |
|---|---|---|
| `LP.order_lr` | `0.9` | Ordering phase learning rate. |
| `LP.order_steps` | `1000` | Ordering phase steps. |
| `LP.tune_lr` | `0.02` | Tuning phase learning rate. |
| `LP.tune_nd` | `1` | Tuning phase neighborhood distance. |

learnpn(code) returns useful information for each code string:

'pnames' — Names of learning parameters

'pdefaults' — Default learning parameters

'needg' — Returns 1 if this function uses gW or gA

## Examples

Here we define a random input P, output A, and weight matrix W, for a layer with a two-element input and six neurons. We also calculate positions and distances for the neurons, which are arranged in a 2-by-3 hexagonal pattern. Then we define the four learning parameters.

```
p = rand(2,1);
a = rand(6,1);
w = rand(6,2);
pos = hextop(2,3);
d = linkdist(pos);
lp.order_lr = 0.9;
lp.order_steps = 1000;
lp.tune_lr = 0.02;
lp.tune_nd = 1;
```

Since learnsom only needs these values to calculate a weight change (see algorithm below), we will use them to do so.

```
ls = [];
[dW,ls] = learnsom(w,p,[],[],a,[],[],[],[],d,lp,ls)
```

## Network Use

You can create a standard network that uses learnsom with newsom.

1 Set net.trainFcn to 'trainr'. (net.trainParam will automatically become trainr's default parameters.)

2 Set net.adaptFcn to 'trains'. (net.adaptParam will automatically become trains's default parameters.)

3 Set each net.inputWeights{i,j}.learnFcn to 'learnsom'. Set each net.layerWeights{i,j}.learnFcn to 'learnsom'. Set net.biases{i}.learnFcn to 'learnsom'. (Each weight learning parameter property will automatically be set to learnsom's default parameters.)

To train the network (or enable it to adapt)

1 Set net.trainParam (net.adaptParam) properties to desired values.

**2** Call train (adapt).

## Algorithm

learnsom calculates the weight change dW for a given neuron from the neuron's input P, activation A2, and learning rate LR:

```
dw =  lr*a2*(p'-w)
```

where the activation A2 is found from the layer output A and neuron distances D and the current neighborhood size ND:

```
a2(i,q) = 1,   if a(i,q) = 1
        = 0.5, if a(j,q) = 1 and D(i,j) <= nd
        = 0,   otherwise
```

The learning rate LR and neighborhood size NS are altered through two phases: an ordering phase and a tuning phase.

The ordering phases lasts as many steps as LP.order_steps. During this phase LR is adjusted from LP.order_lr down to LP.tune_lr, and ND is adjusted from the maximum neuron distance down to 1. It is during this phase that neuron weights are expected to order themselves in the input space consistent with the associated neuron positions.

During the tuning phase LR decreases slowly from LP.tune_lr and ND is always set to LP.tune_nd. During this phase the weights are expected to spread out relatively evenly over the input space while retaining their topological order found during the ordering phase.

## See Also

adapt, train

## Purpose

Widrow-Hoff weight/bias learning function

## Syntax

```
[dW,LS] = learnwh(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)
[db,LS] = learnwh(b,ones(1,Q),Z,N,A,T,E,gW,gA,D,LP,LS)
info = learnwh(code)
```

## Description

`learnwh` is the Widrow-Hoff weight/bias learning function, and is also known as the delta or least mean squared (LMS) rule.

`learnwh(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)` takes several inputs,

`W` — S x R weight matrix (or b, and S x 1 bias vector)

`P` — R x Q input vectors (or ones(1,Q))

`Z` — S x Q weighted input vectors

`N` — S x Q net input vectors

`A` — S x Q output vectors

`T` — S x Q layer target vectors

`E` — S x Q layer error vectors

`gW` — S x R weight gradient with respect to performance

`gA` — S x Q output gradient with respect to performance

`D` — S x S neuron distances

`LP` — Learning parameters, none, `LP = []`

`LS` — Learning state, initially should be = `[]`

and returns,

`dW` — S x R weight (or bias) change matrix

`LS` — New learning state

Learning occurs according to `learnwh`'s learning parameter shown here with its default value.

`LP.lr` `0.01` — Learning rate

`learnwh(code)` returns useful information for each `code` string:

`'pnames'` — Names of learning parameters

`'pdefaults'` — Default learning parameters

`'needg'` — Returns 1 if this function uses `gW` or `gA`

## Examples

Here we define a random input P and error E to a layer with a two-element input and three neurons. We also define the learning rate LR learning parameter.

```
p = rand(2,1);
e = rand(3,1);
lp.lr = 0.5;
```

Since learnwh only needs these values to calculate a weight change (see algorithm below), we will use them to do so.

```
dW = learnwh([],p,[],[],[],[],e,[],[],[],lp,[])
```

## Network Use

You can create a standard network that uses learnwh with newlin.

To prepare the weights and the bias of layer i of a custom network to learn with learnwh

1. Set net.trainFcn to 'trainb'. net.trainParam will automatically become trainb's default parameters.
2. Set net.adaptFcn to 'trains'. net.adaptParam will automatically become trains's default parameters.
3. Set each net.inputWeights{i,j}.learnFcn to 'learnwh'. Set each net.layerWeights{i,j}.learnFcn to 'learnwh'. Set net.biases{i}.learnFcn to 'learnwh'.

Each weight and bias learning parameter property will automatically be set to learnwh's default parameters.

To train the network (or enable it to adapt)

1. Set net.trainParam (net.adaptParam) properties to desired values.
2. Call train(adapt).

See newlin for adaption and training examples.

## Algorithm

learnwh calculates the weight change dW for a given neuron from the neuron's input P and error E, and the weight (or bias) learning rate LR, according to the Widrow-Hoff learning rule:

```
dw = lr*e*pn'
```

## See Also

`newlin`, `adapt`, `train`

## References

Widrow, B., and M. E. Hoff, “Adaptive switching circuits,” *1960 IRE WESCON Convention Record, New York IRE*, pp. 96-104, 1960.

Widrow B. and S. D. Sterns, *Adaptive Signal Processing*, New York: Prentice-Hall, 1985.

# linkdist

### Purpose

Link distance function

### Syntax

```
d = linkdist(pos)
```

### Description

`linkdist` is a layer distance function used to find the distances between the layer's neurons given their positions.

`linkdist`(pos) takes one argument,

`pos` — N x S matrix of neuron positions

and returns the `S` x `S` matrix of distances.

### Examples

Here we define a random matrix of positions for 10 neurons arranged in three-dimensional space and find their distances.

```
pos = rand(3,10);
D = linkdist(pos)
```

### Network Use

You can create a standard network that uses `linkdist` as a distance function by calling `newsom`.

To change a network so that a layer's topology uses `linkdist`, set `net.layers{i}.distanceFcn` to '`linkdist`'.

In either case, call `sim` to simulate the network with `dist`. See `newsom` for training and adaption examples.

### Algorithm

The link distance `D` between two position vectors `Pi` and `Pj` from a set of `S` vectors is

```
Dij = 0, if i==j
= 1, if (sum((Pi-Pj).^2)).^0.5 is <= 1
= 2, if k exists, Dik = Dkj = 1
= 3, if k1, k2 exist, Dik1 = Dk1k2 = Dk2j = 1
= N, if k1..kN exist, Dik1 = Dk1k2 = ...= DkNj = 1
= S, if none of the above conditions apply.
```

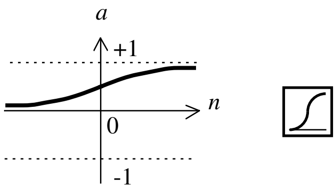
### See Also

`sim`, `dist`, `mandist`

## Purpose

Log sigmoid transfer function

## Graph and Symbol



*a = logsig(n)*

Log-Sigmoid Transfer Function

## Syntax

```
A = logsig(N)
info = logsig(code)
```

## Description

`logsig` is a transfer function. Transfer functions calculate a layer's output from its net input.

`logsig(N)` takes one input,

`N` — S x Q matrix of net input (column) vectors

and returns each element of `N` squashed between 0 and 1.

`logsig(code)` returns useful information for each `code` string:

`'deriv'` — Name of derivative function

`'name'` — Full name

`'output'` — Output range

`'active'` — Active input range

## Examples

Here is the code to create a plot of the `logsig` transfer function.

```
n = -5:0.1:5;
a = logsig(n);
plot(n,a)
```

## Network Use

You can create a standard network that uses `logsig` by calling `newff` or `newcf`.

To change a network so a layer uses `logsig`, set `net.layers{i}.transferFcn` to '`logsig`'.

In either case, call `sim` to simulate the network with `purelin`.

See `newff` or `newcf` for simulation examples.

## Algorithm

```
logsig(n) = 1 / (1 + exp(-n))
```

## See Also

`sim, dlogsig, tansig`

### Purpose

Mean absolute error performance function

### Syntax

```
perf = mae(E,X,PP)
perf = mae(E,net,PP)
info = mae(code)
```

### Description

`mae` is a network performance function.

`mae(E,X,PP)` takes from one to three arguments,

`E` — Matrix or cell array of error vector(s)
`X` — Vector of all weight and bias values (ignored)
`PP` — Performance parameters (ignored)

and returns the mean absolute error.

The errors `E` can be given in cell array form,

`E` — Nt x TS cell array, each element `E{i,ts}` is a `Vi x Q` matrix or`[]`

or as a matrix,

`E` — `(sum of Vi)` x `Q` matrix

where

`Nt` = `net.numTargets`
`TS` = Number of time steps
`Q` = Batch size
`Vi` = `net.targets{i}.size`

`mae(E,net,PP)` can take an alternate argument to `X`,

`net` - Neural network from which `X` can be obtained (ignored).

`mae(code)` returns useful information for each `code` string:

`'deriv'` Name of derivative function
`'name'` Full name
`'pnames'` Names of training parameters
`'pdefaults'` — Default training parameters

## Examples

Here a perceptron is created with a 1-element input ranging from -10 to 10, and one neuron.

```
net = newp([-10 10],1);
```

Here the network is given a batch of inputs P. The error is calculated by subtracting the output A from target T. Then the mean absolute error is calculated.

```
p = [-10 -5 0 5 10];
t = [0 0 1 1 1];
y = sim(net,p)
e = t-y
perf = mae(e)
```

Note that mae can be called with only one argument because the other arguments are ignored. mae supports those arguments to conform to the standard performance function argument list.

## Network Use

You can create a standard network that uses mae with newp.

To prepare a custom network to be trained with mae, set net.performFcn to 'mae'. This will automatically set net.performParam to the empty matrix [], as mae has no performance parameters.

In either case, calling train or adapt will result in mae being used to calculate performance.

See newp for examples.

## See Also

mse, msereg, dmae

## Purpose

Manhattan distance weight function

## Syntax

```
Z = mandist(W,P)
df = mandist('deriv')
D = mandist(pos);
```

## Description

mandist is the Manhattan distance weight function. Weight functions apply weights to an input to get weighted inputs.

mandist(W,P) takes these inputs,

W — S x R weight matrix
P — R x Q matrix of Q input (column) vectors

and returns the S x Q matrix of vector distances.

mandist('deriv') returns '' because mandist does not have a derivative function.

mandist is also a layer distance function, which can be used to find the distances between neurons in a layer.

mandist(pos) takes one argument,

pos — An S row matrix of neuron positions

and returns the S x S matrix of distances.

## Examples

Here we define a random weight matrix W and input vector P and calculate the corresponding weighted input Z.

```
W = rand(4,3);
P = rand(3,1);
Z = mandist(W,P)
```

Here we define a random matrix of positions for 10 neurons arranged in three-dimensional space and then find their distances.

```
pos = rand(3,10);
D = mandist(pos)
```

## Network Use

You can create a standard network that uses mandist as a distance function by calling newsom.

To change a network so an input weight uses `mandist`, set `net.inputWeight{i,j}.weightFcn` to '`mandist`'. For a layer weight, set `net.inputWeight{i,j}.weightFcn` to '`mandist`'.

To change a network so a layer's topology uses `mandist`, set `net.layers{i}.distanceFcn` to '`mandist`'.

In either case, call `sim` to simulate the network with `dist`. See `newpnn` or `newgrnn` for simulation examples.

## Algorithm

The Manhattan distance `D` between two vectors `X` and `Y` is:

```
D = sum(abs(x-y))
```

## See Also

`sim, dist, linkdist`

## Purpose

Maximum learning rate for a linear layer

## Syntax

```
lr = maxlinlr(P)
lr = maxlinlr(P,'bias')
```

## Description

`maxlinlr` is used to calculate learning rates for `newlin`.

`maxlinlr(P)` takes one argument,

`P` — R x Q matrix of input vectors

and returns the maximum learning rate for a linear layer without a bias that is to be trained only on the vectors in `P`.

`maxlinlr(P,'bias')` returns the maximum learning rate for a linear layer with a bias.

## Examples

Here we define a batch of four two-element input vectors and find the maximum learning rate for a linear layer with a bias.

```
P = [1 2 -4 7; 0.1 3 10 6];
lr = maxlinlr(P,'bias')
```

## See Also

`linnet`, `newlin`, `newlind`

# midpoint

## Purpose

Midpoint weight initialization function

## Syntax

```
W = midpoint(S,PR)
```

## Description

midpoint is a weight initialization function that sets weight (row) vectors to the center of the input ranges.

midpoint(S,PR) takes two arguments,

S — Number of rows (neurons)

PR — R x 2 matrix of input value ranges = [Pmin Pmax]

and returns an S x R matrix with rows set to (Pmin+Pmax)'/2.

## Examples

Here initial weight values are calculated for a 5 neuron layer with input elements ranging over [0 1] and [-2 2].

```
W = midpoint(5,[0 1; -2 2])
```

## Network Use

You can create a standard network that uses midpoint to initialize weights by calling newc.

To prepare the weights and the bias of layer i of a custom network to initialize with midpoint:

1. Set net.initFcn to 'initlay'. (net.initParam will automatically become initlay's default parameters.)
2. Set net.layers{i}.initFcn to 'initwb'.
3. Set each net.inputWeights{i,j}.initFcn to 'midpoint'. Set each net.layerWeights{i,j}.initFcn to 'midpoint';

To initialize the network call init.

## See Also

initwb, initlay, init

## Purpose

Ranges of matrix rows

## Syntax

```
pr = minmax(p)
```

## Description

`minmax(P)` takes one argument,

`P` — R x Q matrix

and returns the `R` x `2` matrix `PR` of minimum and maximum values for each row of `P`.

## Examples

```
P = [0 1 2; -1 -2 -0.5]
pr = minmax(P)
```

# mse

## Purpose

Mean squared error performance function

## Syntax

```
perf = mse(E,X,PP)
perf = mse(E,net,PP)
info = mse(code)
```

## Description

`mse` is a network performance function. It measures the network's performance according to the mean of squared errors.

`mse(E,X,PP)` takes from one to three arguments,

`E` — Matrix or cell array of error vector(s)

`X` — Vector of all weight and bias values (ignored)

`PP` — Performance parameters (ignored)

and returns the mean squared error.

`mse(E,net,PP)` can take an alternate argument to `X`,

`net` — Neural network from which X can be obtained (ignored)

`mse(code)` returns useful information for each `code` string:

`'deriv'` — Name of derivative function

`'name'` — Full name

`'pnames'` — Names of training parameters

`'pdefaults'` — Default training parameters

## Examples

Here a two-layer feed-forward network is created with a 1-element input ranging from -10 to 10, four hidden `tansig` neurons, and one `purelin` output neuron.

```
net = newff([-10 10],[4 1],{'tansig','purelin'});
```

Here the network is given a batch of inputs `P`. The error is calculated by subtracting the output `A` from target `T`. Then the mean squared error is calculated.

```
p = [-10 -5 0 5 10];
t = [0 0 1 1 1];
y = sim(net,p)
e = t-y
```

```
perf = mse(e)
```

Note that `mse` can be called with only one argument because the other arguments are ignored. `mse` supports those ignored arguments to conform to the standard performance function argument list.

## Network Use

You can create a standard network that uses `mse` with `newff`, `newcf`, or `newelm`.

To prepare a custom network to be trained with `mse`, set `net.performFcn` to `'mse'`. This will automatically set `net.performParam` to the empty matrix `[]`, as `mse` has no performance parameters.

In either case, calling `train` or `adapt` will result in `mse` being used to calculate performance.

See `newff` or `newcf` for examples.

## See Also

`msereg, mae, dmse`

# msereg

**Purpose** Mean squared error with regularization performance function

**Syntax**

```
perf = msereg(E,X,PP)
perf = msereg(E,net,PP)
info = msereg(code)
```

**Description**

`msereg` is a network performance function. It measures network performance as the weight sum of two factors: the mean squared error and the mean squared weight and bias values.

`msereg(E,X,PP)` takes from three arguments,

`E` — Matrix or cell array of error vector(s)

`X` — Vector of all weight and bias values

`PP` — Performance parameter

where `PP` defines one performance parameters,

`PP.ratio` — Relative importance of errors vs. weight and bias values

and returns the sum of mean squared errors (times `PP.ratio`) with the mean squared weight and bias values (times `1 PP.ratio`).

The errors `E` can be given in cell array form,

`E` — `Nt` x `TS` cell array, each element `E{i,ts}` is an `Vi` x `Q` matrix or `[]`

or as a matrix,

`E` — (sum of Vi) x Q matrix

where

`Nt = net.numTargets`

`TS =` Number of time steps

`Q =` Batch size

`Vi = net.targets{i}.size`

`msereg(E,net)` takes an alternate argument to `X` and `PP`,

`net` — Neural network from which `X` and `PP` can be obtained `msereg(code)` returns useful information for each `code` string:

- 'deriv' — Name of derivative function
- 'name' — Full name
- 'pnames' — Names of training parameters
- 'pdefaults' — Default training parameters

## Examples

Here a two-layer feed-forward is created with a one-element input ranging from -2 to 2, four hidden `tansig` neurons, and one `purelin` output neuron.

```
net = newff([-2 2],[4 1]
{'tansig','purelin'},'trainlm','learngdm','mereg');
```

Here the network is given a batch of inputs `P`. The error is calculated by subtracting the output `A` from target `T`. Then the mean squared error is calculated using a ratio of 20/(20+1). (Errors are 20 times as important as weight and bias values).

```
p = [-2 -1 0 1 2];
t = [0 1 1 1 0];
y = sim(net,p)
e = t-y
net.performParam.ratio = 20/(20+1);
perf = mereg(e,net)
```

## Network Use

You can create a standard network that uses `mereg` with `newff`, `newcf`, or `newelm`.

To prepare a custom network to be trained with `mereg`, set `net.performFcn` to 'mereg'. This will automatically set `net.performParam` to `mereg`'s default performance parameters.

In either case, calling `train` or `adapt` will result in `mereg` being used to calculate performance.

See `newff` or `newcf` for examples.

## See Also

`mse, mae, dmsereg`

# negdist

## Purpose

Negative distance weight function

## Syntax

```
Z = negdist(W,P)
df = negdist('deriv')
```

## Description

negdist is a weight function. Weight functions apply weights to an input to get weighted inputs.

negdist(W,P) takes these inputs,

W — S x R weight matrix

P — R x Q matrix of Q input (column) vectors

and returns the S x Q matrix of negative vector distances.

negdist('deriv') returns '' because negdist does not have a derivative function.

## Examples

Here we define a random weight matrix W and input vector P and calculate the corresponding weighted input Z.

```
W = rand(4,3);
P = rand(3,1);
Z = negdist(W,P)
```

## Network Use

You can create a standard network that uses negdist by calling newc or newsom.

To change a network so an input weight uses negdist, set net.inputWeight{i,j}.weightFcn to 'negdist'. For a layer weight set net.inputWeight{i,j}.weightFcn to 'negdist'.

In either case, call sim to simulate the network with negdist. See newc or newsom for simulation examples.

## Algorithm

negdist returns the negative Euclidean distance:

```
z = -sqrt(sum(w-p)^2)
```

## See Also

sim, dotprod, dist

# netprod

## Purpose

Product net input function

## Syntax

```
N = netprod(Z1,Z2,...,Zn)
df = netprod('deriv')
```

## Description

`netprod` is a net input function. Net input functions calculate a layer's net input by combining its weighted inputs and biases.

`netprod(Z1,Z2,...,Zn)` takes,

`Zi` — S x Q matrices

and returns an element-wise sum of `Zi`'s.

`netprod('deriv')` returns `netprod`'s derivative function.

## Examples

Here `netprod` combines two sets of weighted input vectors (which we have defined ourselves).

```
z1 = [1 2 4;3 4 1];
z2 = [-1 2 2; -5 -6 1];
n = netprod(Z1,Z2)
```

Here `netprod` combines the same weighted inputs with a bias vector. Because `Z1` and `Z2` each contain three concurrent vectors, three concurrent copies of `B` must be created with `concur` so that all sizes match up.

```
b = [0; -1];
n = netprod(z1,z2,concur(b,3))
```

## Network Use

You can create a standard network that uses `netprod` by calling `newpnn` or `newgrnn`.

To change a network so that a layer uses `netprod`, set `net.layers{i}.netInputFcn` to `'netprod'`.

In either case, call `sim` to simulate the network with `netprod`. See `newpnn` or `newgrnn` for simulation examples.

## See Also

`sim, dnetprod, netsum, concur`

# netsum

## Purpose

Sum net input function

## Syntax

```
N = netsum(Z1,Z2,...,Zn)
df = netsum('deriv')
```

## Description

netsum is a net input function. Net input functions calculate a layer's net input by combining its weighted inputs and biases.

netsum(Z1,Z2,...,Zn) takes any number of inputs,

Zi — S x Q matrices,

and returns N, the element-wise sum of Zi's.

netsum('deriv') returns netsum's derivative function.

## Examples

Here netsum combines two sets of weighted input vectors (which we have defined ourselves).

```
z1 = [1 2 4;3 4 1];
z2 = [-1 2 2; -5 -6 1];
n = netsum(Z1,Z2)
```

Here netsum combines the same weighted inputs with a bias vector. Because Z1 and Z2 each contain three concurrent vectors, three concurrent copies of B must be created with concur so that all sizes match up.

```
b = [0; -1];
n = netsum(z1,z2,concur(b,3))
```

## Network Use

You can create a standard network that uses netsum by calling newp or newlin.

To change a network so a layer uses netsum, set net.layers{i}.netInputFcn to 'netsum'.

In either case, call sim to simulate the network with netsum. See newp or newlin for simulation examples.

## See Also

sim, dnetprod, netprod, concur

## Purpose

Create a custom neural network

## Syntax

```
net = network
```

```
net = network(numInputs,numLayers,biasConnect,inputConnect,
layerConnect,outputConnect,targetConnect)
```

## To Get Help

Type `help network/network`

## Description

`network` creates new custom networks. It is used to create networks that are then customized by functions such as `newp`, `newlin`, `newff`, etc.

`network` takes these optional arguments (shown with default values):

| | |
|---|---|
| `numInputs` | — Number of inputs, 0 |
| `numLayers` | — Number of layers, 0 |
| `biasConnect` | — numLayers-by-1 Boolean vector, zeros |
| `inputConnect` | — numLayers-by-numInputs Boolean matrix, zeros |
| `layerConnect` | — numLayers-by-numLayers Boolean matrix, zeros |
| `outputConnect` | — 1-by-numLayers Boolean vector, zeros |
| `targetConnect` | — 1-by-numLayers Boolean vector, zeros |

and returns,

`net` — New network with the given property values.

## Properties

Architecture properties:

`net.numInputs`: 0 or a positive integer.

Number of inputs.

`net.numLayers`: 0 or a positive integer.

Number of layers.

`net.biasConnect`: `numLayer`-by-1 Boolean vector.

If `net.biasConnect(i)` is 1, then the layer `i` has a bias and `net.biases{i}` is a structure describing that bias.

`net.inputConnect`: `numLayer`-by-`numInputs` Boolean vector.

If `net.inputConnect(i,j)` is 1, then layer `i` has a weight coming from input `j` and `net.inputWeights{i,j}` is a structure describing that weight.

`net.layerConnect`: numLayer-by-numLayers Boolean vector.

If `net.layerConnect(i,j)` is 1, then layer `i` has a weight coming from layer `j` and `net.layerWeights{i,j}` is a structure describing that weight.

`net.outputConnect`: 1-by-`numLayers` Boolean vector.

If `net.outputConnect(i)` is 1, then the network has an output from layer `i` and `net.outputs{i}` is a structure describing that output.

`net.targetConnect`: 1-by-`numLayers` Boolean vector.

If `net.outputConnect(i)` is 1, then the network has a target from layer `i` and `net.targets{i}` is a structure describing that target.

`net.numOutputs`: 0 or a positive integer. Read only.

Number of network outputs according to `net.outputConnect`.

`net.numTargets`: 0 or a positive integer. Read only.

Number of targets according to `net.targetConnect`.

`net.numInputDelays`: 0 or a positive integer. Read only.

Maximum input delay according to all `net.inputWeight{i,j}`.delays.

`net.numLayerDelays`: 0 or a positive number. Read only.

Maximum layer delay according to all `net.layerWeight{i,j}`.delays.

Subobject structure properties:

`net.inputs`: `numInputs`-by-1 cell array.

`net.inputs{i}` is a structure defining input `i`.

`net.layers`: `numLayers`-by-1 cell array.

`net.layers{i}` is a structure defining layer `i`.

`net.biases`: `numLayers`-by-1 cell array.

If `net.biasConnect(i)` is 1, then `net.biases{i}` is a structure defining the bias for layer `i`.

`net.inputWeights`: `numLayers`-by-`numInputs` cell array.

If `net.inputConnect(i,j)` is 1, then `net.inputWeights{i,j}` is a structure defining the weight to layer `i` from input `j`.

`net.layerWeights`: `numLayers`-by-`numLayers` cell array.

If `net.layerConnect(i,j)` is 1, then `net.layerWeights{i,j}` is a structure defining the weight to layer `i` from layer `j`.

`net.outputs`: 1-by-`numLayers` cell array.

If `net.outputConnect(i)` is 1, then `net.outputs{i}` is a structure defining the network output from layer `i`.

`net.targets`: 1-by-`numLayers` cell array.

If `net.targetConnect(i)` is 1, then `net.targets{i}` is a structure defining the network target to layer `i`.

Function properties:

`net.adaptFcn`: name of a network adaption function or `''`.

`net.initFcn`: name of a network initialization function or `''`.

`net.performFcn`: name of a network performance function or `''`.

`net.trainFcn`: name of a network training function or `''`.

Parameter properties:

`net.adaptParam`: network adaption parameters.

`net.initParam`: network initialization parameters.

`net.performParam`: network performance parameters.

`net.trainParam`: network training parameters.

Weight and bias value properties:

net.IW: numLayers-by-numInputs cell array of input weight values.

net.LW: numLayers-by-numLayers cell array of layer weight values.

net.b: numLayers-by-1 cell array of bias values.

Other properties:

net.userdata: structure you can use to store useful values.

## Examples

Here is the code to create a network without any inputs and layers, and then set its number of inputs and layer to 1 and 2 respectively.

```
net = network
net.numInputs = 1
net.numLayers = 2
```

Here is the code to create the same network with one line of code.

```
net = network(1,2)
```

Here is the code to create a 1 input, 2 layer, feed-forward network. Only the first layer will have a bias. An input weight will connect to layer 1 from input 1. A layer weight will connect to layer 2 from layer 1. Layer 2 will be a network output, and have a target.

```
net = network(1,2,[1;0],[1; 0],[0 0; 1 0],[0 1],[0 1])
```

We can then see the properties of subobjects as follows:

```
net.inputs{1}
net.layers{1}, net.layers{2}
net.biases{1}
net.inputWeights{1,1}, net.layerWeights{2,1}
net.outputs{2}
net.targets{2}
```

We can get the weight matrices and bias vector as follows:

```
net.iw.{1,1}, net.iw{2,1}, net.b{1}
```

We can alter the properties of any of these subobjects. Here we change the transfer functions of both layers:

```
net.layers{1}.transferFcn = 'tansig';
net.layers{2}.transferFcn = 'logsig';
```

Here we change the number of elements in input 1 to 2, by setting each element's range:

```
net.inputs{1}.range = [0 1; -1 1];
```

Next we can simulate the network for a two-element input vector:

```
p = [0.5; -0.1];
y = sim(net,p)
```

## See Also

sim

## Purpose

Create a competitive layer

## Syntax

```
net = newc
net = newc(PR,S,KLR,CLR)
```

## Description

Competitive layers are used to solve classification problems.

`net = newc` creates a new network with a dialog box.

`net = newc(PR,S,KLR,CLR)` takes these inputs,

- `PR` — R x 2 matrix of min and max values for R input elements
- `S` — Number of neurons
- `KLR` — Kohonen learning rate, default = 0.01
- `CLR` — Conscience learning rate, default = 0.001

and returns a new competitive layer.

## Properties

Competitive layers consist of a single layer with the `negdist` weight function, `netsum` net input function, and the `compet` transfer function.

The layer has a weight from the input, and a bias.

Weights and biases are initialized with `midpoint` and `initcon`.

Adaption and training are done with `trains` and `trainr`, which both update weight and bias values with the `learnk` and `learncon` learning functions.

## Examples

Here is a set of four two-element vectors `P`.

```
P = [.1 .8  .1 .9; .2 .9 .1 .8];
```

A competitive layer can be used to divide these inputs into two classes. First a two neuron layer is created with two input elements ranging from 0 to 1, then it is trained.

```
net = newc([0 1; 0 1],2);
net = train(net,P);
```

The resulting network can then be simulated and its output vectors converted to class indices.

```
Y = sim(net,P)
```

```
    Yc = vec2ind(Y)
```

## See Also

sim, init, adapt, train, trains, trainr, newcf

# newcf

## Purpose

Create a trainable cascade-forward backpropagation network

## Syntax

```
net = newcf
net = newcf(PR,[S1 S2...SNl],{TF1 TF2...TFNl},BTF,BLF,PF)
```

## Description

`net = newcf` creates a new network with a dialog box.

`newcf(PR,[S1 S2...SNl],{TF1 TF2...TFNl},BTF,BLF,PF)` takes,

- `PR` — R x 2 matrix of min and max values for R input elements
- `Si` — Size of ith layer, for `Nl` layers
- `TFi` — Transfer function of ith layer, default = '`tansig`'
- `BTF` — Backpropagation network training function, default = '`traingd`'
- `BLF` — Backpropagation weight/bias learning function, default = '`learngdm`'
- `PF` — Performance function, default = '`mse`'

and returns an `N` layer cascade-forward backprop network.

The transfer functions `TFi` can be any differentiable transfer function such as `tansig`, `logsig`, or `purelin`.

The training function `BTF` can be any of the backprop training functions such as `trainlm`, `trainbfg`, `trainrp`, `traingd`, etc.

**Caution:** `trainlm` is the default training function because it is very fast, but it requires a lot of memory to run. If you get an "out-of-memory" error when training try doing one of these:

1. Slow `trainlm` training, but reduce memory requirements by setting `net.trainParam.mem_reduc` to 2 or more. (See `help trainlm`.)
2. Use `trainbfg`, which is slower but more memory-efficient than `trainlm`.
3. Use `trainrp`, which is slower but more memory-efficient than `trainbfg`.

The learning function `BLF` can be either of the backpropagation learning functions such as `learngd` or `learngdm`.

The performance function can be any of the differentiable performance functions such as `mse` or `msereg`.

## Examples

Here is a problem consisting of inputs P and targets T that we would like to solve with a network.

```
P = [0 1 2 3 4 5 6 7 8 9 10];
T = [0 1 2 3 4 3 2 1 2 3 4];
```

Here a two-layer cascade-forward network is created. The network's input ranges from [0 to 10]. The first layer has five tansig neurons, the second layer has one purelin neuron. The trainlm network training function is to be used.

```
net = newcf([0 10],[5 1],{'tansig' 'purelin'});
```

Here the network is simulated and its output plotted against the targets.

```
Y = sim(net,P);
plot(P,T,P,Y,'o')
```

Here the network is trained for 50 epochs. Again the network's output is plotted.

```
net.trainParam.epochs = 50;
net = train(net,P,T);
Y = sim(net,P);
plot(P,T,P,Y,'o')
```

## Algorithm

Cascade-forward networks consist of Nl layers using the dotprod weight function, netsum net input function, and the specified transfer functions.

The first layer has weights coming from the input. Each subsequent layer has weights coming from the input and all previous layers. All layers have biases. The last layer is the network output.

Each layer's weights and biases are initialized with initnw.

Adaption is done with trains, which updates weights with the specified learning function. Training is done with the specified training function. Performance is measured according to the specified performance function.

## See Also

newff, newelm, sim, init, adapt, train, trains

# newelm

## Purpose

Create an Elman backpropagation network

## Syntax

```
net = newelm
net = newelm(PR,[S1 S2...SNl],{TF1 TF2...TFNl},BTF,BLF,PF)
```

## Description

`net = newelm` creates a new network with a dialog box.

`newelm(PR,[S1 S2...SNl],{TF1 TF2...TFNl},BTF,BLF,PF)` takes several arguments,

`PR` — R x 2 matrix of min and max values for R input elements

`Si` — Size of ith layer, for `Nl` layers

`TFi` — Transfer function of ith layer, default = '`tansig`'

`BTF` — Backpropagation network training function, default = '`traingdx`'

`BLF` — Backpropagation weight/bias learning function, default = '`learngdm`'

`PF` — Performance function, default = '`mse`'

and returns an Elman network.

The training function `BTF` can be any of the backprop training functions such as `trainlm`, `trainbfg`, `trainrp`, `traingd`, etc.

**Caution:** `trainlm` is the default training function because it is very fast, but it requires a lot of memory to run. If you get an "out-of-memory" error when training try doing one of these:

1. Slow `trainlm` training, but reduce memory requirements by setting `net.trainParam.mem_reduc` to 2 or more. (See `help trainlm`.)
2. Use `trainbfg`, which is slower but more memory-efficient than `trainlm`.
3. Use `trainrp`, which is slower but more memory-efficient than `trainbfg`.

The learning function `BLF` can be either of the backpropagation learning functions such as `learngd` or `learngdm`.

The performance function can be any of the differentiable performance functions such as `mse` or `msereg`.

## Examples

Here is a series of Boolean inputs P, and another sequence T, which is 1 wherever P has had two 1's in a row.

```
P = round(rand(1,20));
T = [0 (P(1:end-1)+P(2:end) == 2)];
```

We would like the network to recognize whenever two 1's occur in a row. First we arrange these values as sequences.

```
Pseq = con2seq(P);
Tseq = con2seq(T);
```

Next we create an Elman network whose input varies from 0 to 1, and has five hidden neurons and 1 output.

```
net = newelm([0 1],[10 1],{'tansig','logsig'});
```

Then we train the network with a mean squared error goal of 0.1, and simulate it.

```
net = train(net,Pseq,Tseq);
Y = sim(net,Pseq)
```

## Algorithm

Elman networks consist of Nl layers using the dotprod weight function, netsum net input function, and the specified transfer functions.

The first layer has weights coming from the input. Each subsequent layer has a weight coming from the previous layer. All layers except the last have a recurrent weight. All layers have biases. The last layer is the network output.

Each layer's weights and biases are initialized with initnw.

Adaption is done with trains, which updates weights with the specified learning function. Training is done with the specified training function. Performance is measured according to the specified performance function.

## See Also

newff, newcf, sim, init, adapt, train, trains

## Purpose

Create a feed-forward backpropagation network

## Syntax

```
net = newff
net = newff(PR,[S1 S2...SNl],{TF1 TF2...TFNl},BTF,BLF,PF)
```

## Description

`net = newff` creates a new network with a dialog box.

`newff(PR,[S1 S2...SNl],{TF1 TF2...TFNl},BTF,BLF,PF)` takes,

PR — R x 2 matrix of min and max values for R input elements<br>
Si — Size of ith layer, for Nl layers<br>
TFi — Transfer function of ith layer, default = 'tansig'<br>
BTF — Backpropagation network training function, default = 'traingdx'<br>
BLF — Backpropagation weight/bias learning function, default = 'learngdm'<br>
PF — Performance function, default = 'mse'

and returns an N layer feed-forward backprop network.

The transfer functions TFi can be any differentiable transfer function such as tansig, logsig, or purelin.

The training function BTF can be any of the backprop training functions such as trainlm, trainbfg, trainrp, traingd, etc.

**Caution:** trainlm is the default training function because it is very fast, but it requires a lot of memory to run. If you get an "out-of-memory" error when training try doing one of these:

1. Slow trainlm training, but reduce memory requirements by setting net.trainParam.mem_reduc to 2 or more. (See help trainlm.)
2. Use trainbfg, which is slower but more memory-efficient than trainlm.
3. Use trainrp, which is slower but more memory-efficient than trainbfg.

The learning function BLF can be either of the backpropagation learning functions such as learngd or learngdm.

The performance function can be any of the differentiable performance functions such as mse or msereg.

## Examples

Here is a problem consisting of inputs P and targets T that we would like to solve with a network.

```
P = [0 1 2 3 4 5 6 7 8 9 10];
T = [0 1 2 3 4 3 2 1 2 3 4];
```

Here a two-layer feed-forward network is created. The network's input ranges from [0 to 10]. The first layer has five tansig neurons, the second layer has one purelin neuron. The trainlm network training function is to be used.

```
net = newff([0 10],[5 1],{'tansig' 'purelin'});
```

Here the network is simulated and its output plotted against the targets.

```
Y = sim(net,P);
plot(P,T,P,Y,'o')
```

Here the network is trained for 50 epochs. Again the network's output is plotted.

```
net.trainParam.epochs = 50;
net = train(net,P,T);
Y = sim(net,P);
plot(P,T,P,Y,'o')
```

## Algorithm

Feed-forward networks consist of Nl layers using the dotprod weight function, netsum net input function, and the specified transfer functions.

The first layer has weights coming from the input. Each subsequent layer has a weight coming from the previous layer. All layers have biases. The last layer is the network output.

Each layer's weights and biases are initialized with initnw.

Adaption is done with trains, which updates weights with the specified learning function. Training is done with the specified training function. Performance is measured according to the specified performance function.

## See Also

newcf, newelm, sim, init, adapt, train, trains

# newfftd

## Purpose

Create a feed-forward input-delay backpropagation network

## Syntax

```
net = newfftd
net = newfftd(PR,ID,[S1 S2...SNl],{TF1 TF2...TFNl},BTF,BLF,PF)
```

## Description

`net = newfftd` creates a new network with a dialog box.

`newfftd(PR,ID,[S1 S2...SNl],{TF1 TF2...TFNl},BTF,BLF,PF)` takes,

`PR` — `R` x 2 matrix of min and max values for `R` input elements

`ID` — Input delay vector

`Si` — Size of ith layer, for `Nl` layers

`TFi` — Transfer function of ith layer, default = '`tansig`'

`BTF` — Backprop network training function, default = '`traingdx`'

`BLF` — Backprop weight/bias learning function, default = '`learngdm`'

`PF` — Performance function, default = '`mse`'

and returns an `N` layer feed-forward backprop network.

The transfer functions `TFi` can be any differentiable transfer function such as `tansig`, `logsig`, or `purelin`.

The training function `BTF` can be any of the backprop training functions such as `trainlm`, `trainbfg`, `trainrp`, `traingd`, etc.

**Caution:** `trainlm` is the default training function because it is very fast, but it requires a lot of memory to run. If you get an "out-of-memory" error when training try doing one of these:

1. Slow `trainlm` training, but reduce memory requirements by setting `net.trainParam.mem_reduc` to 2 or more. (See `help trainlm`.)
2. Use `trainbfg`, which is slower but more memory-efficient than `trainlm`.
3. Use `trainrp`, which is slower but more memory-efficient than `trainbfg`.

The learning function `BLF` can be either of the backpropagation learning functions such as `learngd` or `learngdm`.

The performance function can be any of the differentiable performance functions such as mse or msereg.

## Examples

Here is a problem consisting of an input sequence P and target sequence T that can be solved by a network with one delay.

```
P = {1  0 0 1 1  0 1  0 0 0 0 1 1  0 0 1};
T = {1 -1 0 1 0 -1 1 -1 0 0 0 1 0 -1 0 1};
```

Here a two-layer feed-forward network is created with input delays of 0 and 1. The network's input ranges from [0 to 1]. The first layer has five tansig neurons, the second layer has one purelin neuron. The trainlm network training function is to be used.

```
net = newfftd([0 1],[0 1],[5 1],{'tansig' 'purelin'});
```

Here the network is simulated.

```
Y = sim(net,P)
```

Here the network is trained for 50 epochs. Again the network's output is calculated.

```
net.trainParam.epochs = 50;
net = train(net,P,T);
Y = sim(net,P)
```

## Algorithm

Feed-forward networks consist of Nl layers using the dotprod weight function, netsum net input function, and the specified transfer functions.

The first layer has weights coming from the input with the specified input delays. Each subsequent layer has a weight coming from the previous layer. All layers have biases. The last layer is the network output.

Each layer's weights and biases are initialized with initnw.

Adaption is done with trains, which updates weights with the specified learning function. Training is done with the specified training function. Performance is measured according to the specified performance function.

## See Also

newcf, newelm, sim, init, adapt, train, trains

# newgrnn

## Purpose

Design a generalized regression neural network (grnn)

## Syntax

```
net = newgrnn
net = newgrnn(P,T,spread)
```

## Description

`net = newgrnn` creates a new network with a dialog box.

Generalized regression neural networks are a kind of radial basis network that is often used for function approximation. `grnn's` can be designed very quickly.

`newgrnn(P,T,spread)` takes three inputs,

`P` — `R` x `Q` matrix of `Q` input vectors

`T` — `S` x `Q` matrix of `Q` target class vectors

`spread` — Spread of radial basis functions, default = 1.0

and returns a new generalized regression neural network.

The larger the `spread,` is the smoother the function approximation will be. To fit data very closely, use a spread smaller than the typical distance between input vectors. To fit the data more smoothly, use a larger `spread`.

## Properties

`newgrnn` creates a two-layer network. The first layer has `radbas` neurons, calculates weighted inputs with `dist` and net input with `netprod`. The second layer has `purelin` neurons, calculates weighted input with `normprod` and net inputs with `netsum`. Only the first layer has biases.

`newgrnn` sets the first layer weights to `P`', and the first layer biases are all set to 0.8326/`spread`, resulting in radial basis functions that cross 0.5 at weighted inputs of +/- `spread`. The second layer weights `W2` are set to `T`.

## Examples

Here we design a radial basis network given inputs `P` and targets `T`.

```
P = [1 2 3];
T = [2.0 4.1 5.9];
net = newgrnn(P,T);
```

Here the network is simulated for a new input.

```
P = 1.5;
Y = sim(net,P)
```

## See Also

`sim`, `newrb`, `newrbe`, `newpnn`

## References

Wasserman, P.D., *Advanced Methods in Neural Computing*, New York: Van Nostrand Reinhold, pp. 155-61, 1993.

# newhop

## Purpose

Create a Hopfield recurrent network

## Syntax

```
net = newhop
net = newhop(T)
```

## Description

Hopfield networks are used for pattern recall.

`net = newhop` creates a new network with a dialog box.

`newhop(T)` takes one input argument,

`T` — `R x Q` matrix of `Q` target vectors (values must be +1 or -1)

and returns a new Hopfield recurrent neural network with stable points at the vectors in `T`.

## Properties

Hopfield networks consist of a single layer with the `dotprod` weight function, `netsum` net input function, and the `satlins` transfer function.

The layer has a recurrent weight from itself and a bias.

## Examples

Here we create a Hopfield network with two three-element stable points `T`.

```
T = [-1 -1 1; 1 -1 1]';
net = newhop(T);
```

Below we check that the network is stable at these points by using them as initial layer delay conditions. If the network is stable we would expect that the outputs `Y` will be the same. (Since Hopfield networks have no inputs, the second argument to `sim` is `Q` = 2 when using matrix notation).

```
Ai = T;
[Y,Pf,Af] = sim(net,2,[],Ai);
Y
```

To see if the network can correct a corrupted vector, run the following code, which simulates the Hopfield network for five time steps. (Since Hopfield networks have no inputs, the second argument to `sim` is `{Q TS} = [1 5]` when using cell array notation.)

```
Ai = {[-0.9; -0.8; 0.7]};
[Y,Pf,Af] = sim(net,{1 5},{},Ai);
Y{1}
```

If you run the above code, `Y{1}` will equal `T(:,1)` if the network has managed to convert the corrupted vector `Ai` to the nearest target vector.

### Algorithm

Hopfield networks are designed to have stable layer outputs as defined by user-supplied targets. The algorithm minimizes the number of unwanted stable points.

### See Also

`sim, satlins`

### References

Li, J., A. N. Michel, and W. Porod, "Analysis and synthesis of a class of neural networks: linear systems operating on a closed hypercube," *IEEE Transactions on Circuits and Systems*, vol. 36, no. 11, pp. 1405-1422, November 1989.

# newlin

## Purpose

Create a linear layer

## Syntax

```
net = newlin
net = newlin(PR,S,ID,LR)
```

## Description

Linear layers are often used as adaptive filters for signal processing and prediction.

`net = newlin` creates a new network with a dialog box.

`newlin(PR,S,ID,LR)` takes these arguments,

- `PR` — `R` x 2 matrix of min and max values for `R` input elements
- `S` — Number of elements in the output vector
- `ID` — Input delay vector, default = [0]
- `LR` — Learning rate, default = 0.01

and returns a new linear layer.

`net = newlin(PR,S,0,P)` takes an alternate argument,

- `P` — Matrix of input vectors

and returns a linear layer with the maximum stable learning rate for learning with inputs `P`.

## Examples

This code creates a single input (range of [-1 1] linear layer with one neuron, input delays of 0 and 1, and a learning rate of 0.01. It is simulated for an input sequence `P1`.

```
net = newlin([-1 1],1,[0 1],0.01);
P1 = {0 -1 1 1 0 -1 1 0 0 1};
Y = sim(net,P1)
```

Here targets `T1` are defined and the layer adapts to them. (Since this is the first call to `adapt`, the default input delay conditions are used.)

```
T1 = {0 -1 0 2 1 -1 0 1 0 1};
[net,Y,E,Pf] = adapt(net,P1,T1); Y
```

Here the linear layer continues to adapt for a new sequence using the previous final conditions `PF` as initial conditions.

```
P2 = {1 0 -1 -1 1 1 1 0 -1};
T2 = {2 1 -1 -2 0 2 2 1 0};
[net,Y,E,Pf] = adapt(net,P2,T2,Pf); Y
```

Here we initialize the layer's weights and biases to new values.

```
net = init(net);
```

Here we train the newly initialized layer on the entire sequence for 200 epochs to an error goal of 0.1.

```
P3 = [P1 P2];
T3 = [T1 T2];
net.trainParam.epochs = 200;
net.trainParam.goal = 0.1;
net = train(net,P3,T3);
Y = sim(net,[P1 P2])
```

## Algorithm

Linear layers consist of a single layer with the dotprod weight function, netsum net input function, and purelin transfer function.

The layer has a weight from the input and a bias.

Weights and biases are initialized with initzero.

Adaption and training are done with trains and trainb, which both update weight and bias values with learnwh. Performance is measured with mse.

## See Also

newlind, sim, init, adapt, train, trains, trainb

# newlind

## Purpose

Design a linear layer

## Syntax

```
net = newlind
net = newlind(P,T,Pi)
```

## Description

net = newlind creates a new network with a dialog box.

newlind(P,T,Pi) takes these input arguments,

P — R x Q matrix of Q input vectors

T — S x Q matrix of Q target class vectors

Pi — 1 x ID cell array of initial input delay states

where each element Pi{i,k} is an RixQ matrix, default = []

and returns a linear layer designed to output T (with minimum sum square error) given input P.

newlind(P,T,Pi) can also solve for linear networks with input delays and multiple inputs and layers by supplying input and target data in cell array form:

P — NixTS cell array, each element P{i,ts} is an Ri x Q input matrix

T — NtxTS cell array, each element P{i,ts} is an Vi x Q matrix

Pi — NixID cell array, each element Pi{i,k} is an Ri x Q matrix, default = []

returns a linear network with ID input delays, Ni network inputs, Nl layers, and designed to output T (with minimum sum square error) given input P.

## Examples

We would like a linear layer that outputs T given P for the following definitions.

```
P = [1 2 3];
T = [2.0 4.1 5.9];
```

Here we use newlind to design such a network and check its response.

```
net = newlind(P,T);
Y = sim(net,P)
```

We would like another linear layer that outputs the sequence T given the sequence P and two initial input delay states Pi.

```
P = {1 2 1 3 3 2};
Pi = {1 3};
```

```
T = {5.0 6.1 4.0 6.0 6.9 8.0};
net = newlind(P,T,Pi);
Y = sim(net,P,Pi)
```

We would like a linear network with two outputs Y1 and Y2 that generate sequences T1 and T2, given the sequences P1 and P2 with three initial input delay states Pi1 for input 1, and three initial delays states Pi2 for input 2.

```
P1 = {1 2 1 3 3 2}; Pi1 = {1 3 0};
P2 = {1 2 1 1 2 1}; Pi2 = {2 1 2};
T1 = {5.0 6.1 4.0 6.0 6.9 8.0};
T2 = {11.0 12.1 10.1 10.9 13.0 13.0};
net = newlind([P1; P2],[T1; T2],[Pi1; Pi2]);
Y = sim(net,[P1; P2],[Pi1; Pi2]);
Y1 = Y(1,:)
Y2 = Y(2,:)
```

## Algorithm

newlind calculates weight W and bias B values for a linear layer from inputs P and targets T by solving this linear equation in the least squares sense:

```
[W b] * [P; ones] = T
```

## See Also

sim, newlin

# newlvq

## Purpose

Create a learning vector quantization network

## Syntax

```
net = newlvq
net = newlvq(PR,S1,PC,LR,LF)
```

## Description

Learning vector quantization (LVQ) networks are used to solve classification problems.

`net = newlvq` creates a new network with a dialog box.

`net = newlvq(PR,S1,PC,LR,LF)` takes these inputs,

| | |
|---|---|
| `PR` | `R` x 2 matrix of min and max values for `R` input elements |
| `S1` | Number of hidden neurons |
| `PC` | `S2` element vector of typical class percentages |
| `LR` | Learning rate, default = 0.01 |
| `LF` | Learning function, default = '`learnlv2`' |

returns a new LVQ network.

The learning function LF can be `learnlv1` or `learnlv2`.

## Properties

`newlvq` creates a two-layer network. The first layer uses the `compet` transfer function, calculates weighted inputs with `negdist`, and net input with `netsum`. The second layer has `purelin` neurons, calculates weighted input with `dotprod` and net inputs with `netsum`. Neither layer has biases.

First layer weights are initialized with `midpoint`. The second layer weights are set so that each output neuron `i` has unit weights coming to it from `PC(i)` percent of the hidden neurons.

Adaption and training are done with `trains` and `trainr`, which both update the first layer weights with the specified learning functions.

## Examples

The input vectors `P` and target classes `Tc` below define a classification problem to be solved by an LVQ network.

```
P = [-3 -2 -2  0  0  0  0 +2 +2 +3; ...
0 +1 -1 +2 +1 -1 -2 +1 -1  0];
Tc = [1 1 1 2 2 2 2 1 1 1];
```

The target classes `Tc` are converted to target vectors `T`. Then, an LVQ network is created (with inputs ranges obtained from `P`, four hidden neurons, and class percentages of 0.6 and 0.4) and is trained.

```
T = ind2vec(Tc);
net = newlvq(minmax(P),4,[.6 .4]);
net = train(net,P,T);
```

The resulting network can be tested.

```
Y = sim(net,P)
Yc = vec2ind(Y)
```

## See Also

`sim, init, adapt, train, trains, trainr, learnlv1, learnlv2`

# newp

## Purpose

Create a perceptron

## Syntax

```
net = newp
net = newp(PR,S,TF,LF)
```

## Description

Perceptrons are used to solve simple (i.e. linearly separable) classification problems.

`net = newp` creates a new network with a dialog box.

`net = newp(PR,S,TF,LF)` takes these inputs,

- `PR` — `R` x 2 matrix of min and max values for `R` input elements
- `S` — Number of neurons
- `TF` — Transfer function, default = '`hardlim`'
- `LF` — Learning function, default = '`learnp`'

and returns a new perceptron.

The transfer function `TF` can be `hardlim` or `hardlims`. The learning function `LF` can be `learnp` or `learnpn`.

## Properties

Perceptrons consist of a single layer with the `dotprod` weight function, the `netsum` net input function, and the specified transfer function.

The layer has a weight from the input and a bias.

Weights and biases are initialized with `initzero`.

Adaption and training are done with `trains` and `trainc`, which both update weight and bias values with the specified learning function. Performance is measured with `mae`.

## Examples

This code creates a perceptron layer with one two-element input (ranges [0 1] and [-2 2]) and one neuron. (Supplying only two arguments to `newp` results in the default perceptron learning function `learnp` being used.)

```
net = newp([0 1; -2 2],1);
```

Here we simulate the network to a sequence of inputs `P`.

```
P1 = {[0; 0] [0; 1] [1; 0] [1; 1]};
Y = sim(net,P1)
```

Here we define a sequence of targets T (together P and T define the operation of an AND gate), and then let the network adapt for 10 passes through the sequence. We then simulate the updated network.

```
T1 = {0 0 0 1};
net.adaptParam.passes = 10;
net = adapt(net,P1,T1);
Y = sim(net,P1)
```

Now we define a new problem, an OR gate, with batch inputs P and targets T.

```
P2 = [0 0 1 1; 0 1 0 1];
T2 = [0 1 1 1];
```

Here we initialize the perceptron (resulting in new random weight and bias values), simulate its output, train for a maximum of 20 epochs, and then simulate it again.

```
net = init(net);
Y = sim(net,P2)
net.trainParam.epochs = 20;
net = train(net,P2,T2);
Y = sim(net,P2)
```

## Notes

Perceptrons can classify linearly separable classes in a finite amount of time. If input vectors have a large variance in their lengths, the learnpn can be faster than learnp.

## See Also

sim, init, adapt, train, hardlim, hardlims, learnp, learnpn, trains, trainc

# newpnn

## Purpose

Design a probabilistic neural network

## Syntax

```
net = newpnn
net = newpnn(P,T,spread)
```

## Description

Probabilistic neural networks (PNN) are a kind of radial basis network suitable for classification problems.

`net = newpnn` creates a new network with a dialog box.

`net = newpnn(P,T,spread)`takes two or three arguments,

| | |
|---|---|
| `P` | — `R` x `Q` matrix of `Q` input vectors |
| `T` | — `S` x `Q` matrix of `Q` target class vectors |
| `spread` | — Spread of radial basis functions, default = 0.1 |

and returns a new probabilistic neural network.

If `spread` is near zero, the network will act as a nearest neighbor classifier. As `spread` becomes larger, the designed network will take into account several nearby design vectors.

## Examples

Here a classification problem is defined with a set of inputs `P` and class indices `Tc`.

```
P = [1 2 3 4 5 6 7];
Tc = [1 2 3 2 2 3 1];
```

Here the class indices are converted to target vectors, and a PNN is designed and tested.

```
T = ind2vec(Tc)
net = newpnn(P,T);
Y = sim(net,P)
Yc = vec2ind(Y)
```

## Algorithm

`newpnn` creates a two-layer network. The first layer has `radbas` neurons, and calculates its weighted inputs with `dist`, and its net input with `netprod`. The second layer has `compet` neurons, and calculates its weighted input with `dotprod` and its net inputs with `netsum`. Only the first layer has biases.

newpnn sets the first layer weights to P', and the first layer biases are all set to 0.8326/spread, resulting in radial basis functions that cross 0.5 at weighted inputs of +/- spread. The second layer weights W2 are set to T.

## See Also

sim, ind2vec, vec2ind, newrb, newrbe, newgrnn

## References

Wasserman, P.D., *Advanced Methods in Neural Computing*, New York: Van Nostrand Reinhold, pp. 35-55, 1993.

# newrb

## Purpose

Design a radial basis network

## Syntax

```
net = newrb
[net,tr] = newrb(P,T,goal,spread,MN,DF)
```

## Description

Radial basis networks can be used to approximate functions. newrb adds neurons to the hidden layer of a radial basis network until it meets the specified mean squared error goal.

net = newrb creates a new network with a dialog box.

newrb(P,T,goal,spread,MN, DF) takes two to these arguments,

P — R x Q matrix of Q input vectors

T — S x Q matrix of Q target class vectors

goal — Mean squared error goal, default = 0.0

spread — Spread of radial basis functions, default = 1.0

MN — Maximum number of neurons, default is Q

DF — Number of neurons to add between displays, default = 25

and returns a new radial basis network.

The larger that spread is, the smoother the function approximation will be. Too large a spread means a lot of neurons will be required to fit a fast changing function. Too small a spread means many neurons will be required to fit a smooth function, and the network may not generalize well. Call newrb with different spreads to find the best value for a given problem.

## Examples

Here we design a radial basis network given inputs P and targets T.

```
P = [1 2 3];
T = [2.0 4.1 5.9];
net = newrb(P,T);
```

Here the network is simulated for a new input.

```
P = 1.5;
Y = sim(net,P)
```

## Algorithm

newrb creates a two-layer network. The first layer has radbas neurons, and calculates its weighted inputs with dist, and its net input with netprod. The second layer has purelin neurons, and calculates its weighted input with dotprod and its net inputs with netsum. Both layers have biases.

Initially the radbas layer has no neurons. The following steps are repeated until the network's mean squared error falls below goal.

1. The network is simulated.
2. The input vector with the greatest error is found.
3. A radbas neuron is added with weights equal to that vector.
4. The purelin layer weights are redesigned to minimize error.

## See Also

sim, newrbe, newgrnn, newpnn

# newrbe

## Purpose

Design an exact radial basis network

## Syntax

```
net = newrbe
net = newrbe(P,T,spread)
```

## Description

Radial basis networks can be used to approximate functions. newrbe very quickly designs a radial basis network with zero error on the design vectors.

net = newrbe creates a new network with a dialog box.

newrbe(P,T,spread) takes two or three arguments,

- P — R x Q matrix of Q input vectors
- T — S x Q matrix of Q target class vectors
- spread — Spread of radial basis functions, default = 1.0

and returns a new exact radial basis network.

The larger the spread is, the smoother the function approximation will be. Too large a spread can cause numerical problems.

## Examples

Here we design a radial basis network given inputs P and targets T.

```
P = [1 2 3];
T = [2.0 4.1 5.9];
net = newrbe(P,T);
```

Here the network is simulated for a new input.

```
P = 1.5;
Y = sim(net,P)
```

## Algorithm

newrbe creates a two-layer network. The first layer has radbas neurons, and calculates its weighted inputs with dist, and its net input with netprod. The second layer has purelin neurons, and calculates its weighted input with dotprod and its net inputs with netsum. Both layers have biases.

newrbe sets the first layer weights to P', and the first layer biases are all set to 0.8326/spread, resulting in radial basis functions that cross 0.5 at weighted inputs of +/- spread.

The second layer weights IW{2,1} and biases b{2} are found by simulating the first layer outputs A{1}, and then solving the following linear expression:

```
[W{2,1} b{2}] * [A{1}; ones] = T
```

### See Also

`sim`, `newrb`, `newgrnn`, `newpnn`

### Purpose

Create a self-organizing map

### Syntax

```
net = newsom
net = newsom(PR,[D1,D2,...],TFCN,DFCN,OLR,OSTEPS,TLR,TND)
```

### Description

Competitive layers are used to solve classification problems.

`net = newsom` creates a new network with a dialog box.

`net = newsom (PR,[D1,D2,...],TFCN,DFCN,OLR,OSTEPS,TLR,TND)` takes,

| | |
|---|---|
| `PR` | — `R` x 2 matrix of min and max values for `R` input elements |
| `Di` | — Size of ith layer dimension, defaults = [5 8] |
| `TFCN` | — Topology function, default ='`hextop`' |
| `DFCN` | — Distance function, default ='`linkdist`' |
| `OLR` | — Ordering phase learning rate, default = 0.9 |
| `OSTEPS` | — Ordering phase steps, default = 1000 |
| `TLR` | — Tuning phase learning rate, default = 0.02 |
| `TND` | — Tuning phase neighborhood distance, default = 1 |

and returns a new self-organizing map.

The topology function `TFCN` can be `hextop`, `gridtop`, or `randtop`. The distance function can be `linkdist`, `dist`, or `mandist`.

### Properties

Self-organizing maps (SOM) consist of a single layer with the `negdist` weight function, `netsum` net input function, and the `compet` transfer function.

The layer has a weight from the input, but no bias. The weight is initialized with `midpoint`.

Adaption and training are done with `trains` and `trainr`, which both update the weight with `learnsom`.

### Examples

The input vectors defined below are distributed over an two-dimension input space varying over [0 2] and [0 1]. This data will be used to train a SOM with dimensions [3 5].

```
P = [rand(1,400)*2; rand(1,400)];
net = newsom([0 2; 0 1],[3 5]);
```

```
plotsom(net.layers{1}.positions)
```

Here the SOM is trained and the input vectors are plotted with the map that the SOM's weights have formed.

```
net = train(net,P);
plot(P(1,:),P(2,:),'.g','markersize',20)
hold on
plotsom(net.iw{1,1},net.layers{1}.distances)
hold off
```

## See Also

sim, init, adapt, train

# nncopy

## Purpose

Copy matrix or cell array

## Syntax

```
nncopy(X,M,N)
```

## Description

`nncopy(X,M,N)` takes two arguments,

`X` — `R` x `C` matrix (or cell array)

`M` — Number of vertical copies

`N` — Number of horizontal copies

and returns a new (`R*M`) x (`C*N`) matrix (or cell array).

## Examples

```
x1 = [1 2 3; 4 5 6];
y1 = nncopy(x1,3,2)
x2 = {[1 2]; [3; 4; 5]}
y2 = nncopy(x2,2,3)
```

## Purpose

Update NNT 2.0 competitive layer

## Syntax

```
net = nnt2c(PR,W,KLR,CLR)
```

## Description

`nnt2c(PR,W,KLR,CLR)` takes these arguments,

`PR` — `R` x 2 matrix of min and max values for `R` input elements

`W` — `S` x `R` weight matrix

`KLR` — Kohonen learning rate, default = 0.01

`CLR` — Conscience learning rate, default = 0.001

and returns a competitive layer.

Once a network has been updated, it can be simulated, initialized, or trained with `sim`, `init`, `adapt`, and `train`.

## See Also

`newc`

# nnt2elm

## Purpose

Update NNT 2.0 Elman backpropagation network

## Syntax

```
net = nnt2elm(PR,W1,B1,W2,B2,BTF,BLF,PF)
```

## Description

`nnt2elm(PR,W1,B1,W2,B2,BTF,BLF,PF)` takes these arguments,

`PR` — `R` x 2 matrix of min and max values for `R` input elements

`W1` — `S1` x `(R+S1)` weight matrix

`B1` — `S1` x 1 bias vector

`W2` — `S2` x `S1` weight matrix

`B2` — `S2` x 1 bias vector

`BTF` — Backpropagation network training function, default = '`traingdx`'

`BLF` — Backpropagation weight/bias learning function, default = '`learngdm`'

`PF` — Performance function, default = '`mse`'

and returns a feed-forward network.

The training function `BTF` can be any of the backpropagation training functions such as `traingd`, `traingdm`, `traingda`, and `traingdx`. Large step-size algorithms, such as `trainlm`, are not recommended for Elman networks.

The learning function `BLF` can be either of the backpropagation learning functions such as `learngd` or `learngdm`.

The performance function can be any of the differentiable performance functions such as `mse` or `msereg`.

Once a network has been updated, it can be simulated, initialized, adapted, or trained with `sim`, `init`, `adapt`, and `train`.

## See Also

`newelm`

## Purpose

Update NNT 2.0 feed-forward network

## Syntax

```
net = nnt2ff(PR,{W1 W2 ...},{B1 B2 ...},{TF1 TF2 ...},BTF,BLR,PF)
```

## Description

`nnt2ff(PR,{W1 W2 ...},{B1 B2 ...},{TF1 TF2 ...},BTF,BLR,PF)` takes these arguments,

`PR` — `R` x 2 matrix of min and max values for `R` input elements

`Wi` — Weight matrix for the ith layer

`Bi` — Bias vector for the ith layer

`TFi` — Transfer function of ith layer, default = '`tansig`'

`BTF` — Backpropagation network training function, default = '`traingdx`'

`BLF` — Backpropagation weight/bias learning function, default = '`learngdm`'

`PF` — Performance function, default = '`mse`'

and returns a feed-forward network.

The training function `BTF` can be any of the backpropagation training functions such as `traingd`, `traingdm`, `traingda`, `traingdx` or `trainlm`.

The learning function `BLF` can be either of the backpropagation learning functions such as `learngd` or `learngdm`.

The performance function can be any of the differentiable performance functions such as `mse` or `msereg`.

Once a network has been updated, it can be simulated, initialized, adapted, or trained with `sim`, `init`, `adapt`, and `train`.

## See Also

`newff, newcf, newfftd, newelm`

# nnt2hop

## Purpose

Update NNT 2.0 Hopfield recurrent network

## Syntax

```
net = nnt2hop(W,B)
```

## Description

`nnt2hop(W,B)` takes these arguments,

`W` — `S` x `S` weight matrix

`B` — `S` x `1` bias vector

and returns a perceptron.

Once a network has been updated, it can be simulated, initialized, adapted, or trained with `sim`, `init`, `adapt`, and `train`.

## See Also

`newhop`

## Purpose

Update NNT 2.0 linear layer

## Syntax

```
net = nnt2lin(PR,W,B,LR)
```

## Description

`nnt2lin(PR,W,B)` takes these arguments,

`PR` — `R` x 2 matrix of min and max values for `R` input elements

`W` — `S` x `R` weight matrix

`B` — `S` x 1 bias vector

`LR` — Learning rate, default = 0.01

and returns a linear layer.

Once a network has been updated, it can be simulated, initialized, adapted, or trained with `sim`, `init`, `adapt`, and `train`.

## See Also

`newlin`

# nnt2lvq

## Purpose

Update NNT 2.0 learning vector quantization network

## Syntax

```
net = nnt2lvq(PR,W1,W2,LR,LF)
```

## Description

`nnt2lvq(PR,W1,W2,LR,LF)` takes these arguments,

`PR` — `R` x 2 matrix of min and max values for `R` input elements

`W1` — `S1` x `R` weight matrix

`W2` — `S2` x `S1` weight matrix

`LR` — Learning rate, default = 0.01

`LF` — Learning function, default = '`learnlv2`'

and returns a radial basis network.

The learning function `LF` can be `learnlv1` or `learnlv2`.

Once a network has been updated, it can be simulated, initialized, adapted, or trained with `sim`, `init`, `adapt`, and `train`.

## See Also

`newlvq`

## Purpose

Update NNT 2.0 perceptron

## Syntax

```
net = nnt2p(PR,W,B,TF,LF)
```

## Description

`nnt2p(PR,W,B,TF,LF)` takes these arguments,

- `PR` — `R` x 2 matrix of min and max values for `R` input elements
- `W` — `S` x `R` weight matrix
- `B` — `S` x 1 bias vector
- `TF` — Transfer function, default = '`hardlim`'
- `LF` — Learning function, default = '`learnp`'

and returns a perceptron.

The transfer function `TF` can be `hardlim` or `hardlims`. The learning function `LF` can be `learnp` or `learnpn`.

Once a network has been updated, it can be simulated, initialized, adapted, or trained with `sim`, `init`, `adapt`, and `train`.

## See Also

`newp`

# nnt2rb

**Purpose** Update NNT 2.0 radial basis network

**Syntax**

```
net = nnt2rb(PR,W1,B1,W2,B2)
```

**Description** `nnt2rb(PR,W1,B1,W2,B2)` takes these arguments,

`PR` — `R` x 2 matrix of min and max values for `R` input elements

`W1` — `S1` x `R` weight matrix

`B1` — `S1` x 1 bias vector

`W2` — `S2` x `S1` weight matrix

`B2` — `S2` x 1 bias vector

and returns a radial basis network.

Once a network has been updated, it can be simulated, initialized, adapted, or trained with `sim`, `init`, `adapt`, and `train`.

**See Also** `newrb, newrbe, newgrnn, newpnn`

## Purpose

Update NNT 2.0 self-organizing map

## Syntax

```
net = nnt2som(PR,[D1,D2,...],W,OLR,OSTEPS,TLR,TND)
```

## Description

`nnt2som(PR,[D1,D2,...],W,OLR,OSTEPS,TLR,TND)` takes these arguments,

| | |
|---|---|
| `PR` | — `R` x 2 matrix of min and max values for `R` input elements |
| `Di` | — Size of ith layer dimension |
| `W` | — `S` x `R` weight matrix |
| `OLR` | — Ordering phase learning rate, default = 0.9 |
| `OSTEPS` | — Ordering phase steps, default = 1000 |
| `TLR` | — Tuning phase learning rate, default = 0.02 |
| `TND` | — Tuning phase neighborhood distance, default = 1 |

and returns a self-organizing map.

`nnt2som` assumes that the self-organizing map has a grid topology (`gridtop`) using link distances (`linkdist`). This corresponds with the neighborhood function in NNT 2.0.

The new network will only output 1 for the neuron with the greatest net input. In NNT 2.0 the network would also output 0.5 for that neuron's neighbors.

Once a network has been updated, it can be simulated, initialized, adapted, or trained with `sim`, `init`, `adapt`, and `train`.

## See Also

`newsom`

# nntool

**Purpose** Neural Network Tool - Graphical User Interface

**Syntax** `nntool`

**Description** `nntool` opens the Network/Data Manager window, which allows you to import, create, use, and export neural networks and data.

## Purpose

Normalize the columns of a matrix

## Syntax

`normc(M)`

## Description

`normc(M)` normalizes the columns of `M` to a length of 1.

## Examples

```
m = [1 2; 3 4];
normc(m)
ans =
     0.3162     0.4472
     0.9487     0.8944
```

## See Also

`normr`

# normprod

## Purpose

Normalized dot product weight function

## Syntax

```
Z = normprod(W,P)
df = normprod('deriv')
```

## Description

normprod is a weight function. Weight functions apply weights to an input to get weighted inputs.

normprod(W,P) takes these inputs,

W — S x R weight matrix

P — R x Q matrix of Q input (column) vectors

and returns the S x Q matrix of normalized dot products.

normprod('deriv') returns '' because normprod does not have a derivative function.

## Examples

Here we define a random weight matrix W and input vector P and calculate the corresponding weighted input Z.

```
W = rand(4,3);
P = rand(3,1);
Z = normprod(W,P)
```

## Network Use

You can create a standard network that uses normprod by calling newgrnn.

To change a network so an input weight uses normprod, set net.inputWeight{i,j}.weightFcn to 'normprod'. For a layer weight, set net.inputWeight{i,j}.weightFcn to 'normprod'.

In either case call sim to simulate the network with normprod. See newgrnn for simulation examples.

## Algorithm

normprod returns the dot product normalized by the sum of the input vector elements.

```
z = w*p/sum(p)
```

## See Also

sim, dotprod, negdist, dist

### Purpose

Normalize the rows of a matrix

### Syntax

`normr(M)`

### Description

`normr(M)` normalizes the columns of `M` to a length of 1.

### Examples

```
m = [1 2; 3 4];
normr(m)
ans =
      0.4472     0.8944
      0.6000     0.8000
```

### See Also

`normc`

# plotbr

**Purpose** Plot network performance for Bayesian regularization training.

**Syntax** `plotbr(TR,name,epoch)`

**Description** `plotbr(tr,name,epoch)` takes these inputs,

`TR` — Training record returned by `train`

`name` — Training function name, default = ''

`epoch` — Number of epochs, default = length of training record

and plots the training sum squared error, the sum squared weight, and the effective number of parameters.

**Examples** Here are input values `P` and associated targets `T`.

```
p = [-1:.05:1];
t = sin(2*pi*p)+0.1*randn(size(p));
```

The code below creates a network and trains it on this problem.

```
net=newff([-1 1],[20,1],{'tansig','purelin'},'trainbr');
[net,tr] = train(net,p,t);
```

During training `plotbr` was called to display the training record. You can also call `plotbr` directly with the final training record `TR`, as shown below.

```
plotbr(tr)
```

## Purpose

Plot a weight-bias position on an error surface

## Syntax

```
h = plotep(W,B,E)
h = plotep(W,B,E,H)
```

## Description

`plotep` is used to show network learning on a plot already created by `plotes`.

`plotep(W,B,E)` takes these arguments,

`W` — Current weight value

`B` — Current bias value

`E` — Current error

and returns a vector `H`, containing information for continuing the plot.

`plotep(W,B,E,H)` continues plotting using the vector `H` returned by the last call to `plotep`.

`H` contains handles to dots plotted on the error surface, so they can be deleted next time, as well as points on the error contour, so they can be connected.

## See Also

`errsurf, plotes`

# plotes

## Purpose

Plot the error surface of a single input neuron

## Syntax

`plotes(WV,BV,ES,V)`

## Description

`plotes(WV,BV,ES,V)` takes these arguments,

`WV` — `1` x `N` row vector of values of `W`

`BV` — `1` x `M` row vector of values of `B`

`ES` — `M` x `N` matrix of error vectors

`V` — View, default = [-37.5, 30]

and plots the error surface with a contour underneath.

Calculate the error surface `ES` with `errsurf`.

## Examples

```
p = [3 2];
t = [0.4 0.8];
wv = -4:0.4:4; bv = wv;
ES = errsurf(p,t,wv,bv,'logsig');
plotes(wv,bv,ES,[60 30])
```

## See Also

`errsurf`

## Purpose

Plot a classification line on a perceptron vector plot

## Syntax

```
plotpc(W,B)
plotpc(W,B,H)
```

## Description

`plotpc(W,B)` takes these inputs,

- `W` — `S` x `R` weight matrix (`R` must be 3 or less)
- `B` — `S` x `1` bias vector

and returns a handle to a plotted classification line.

`plotpc(W,B,H)` takes anadditional input,

- `H` — Handle to last plotted line

and deletes the last line before plotting the new one.

This function does not change the current axis and is intended to be called after `plotpv`.

## Examples

The code below defines and plots the inputs and targets for a perceptron:

```
p = [0 0 1 1; 0 1 0 1];
t = [0 0 0 1];
plotpv(p,t)
```

The following code creates a perceptron with inputs ranging over the values in `P`, assigns values to its weights and biases, and plots the resulting classification line.

```
net = newp(minmax(p),1);
net.iw{1,1} = [-1.2 -0.5];
net.b{1} = 1;
plotpc(net.iw{1,1},net.b{1})
```

## See Also

`plotpv`

# plotperf

## Purpose

Plot network performance

## Syntax

```
plotperf(TR,goal,name,epoch)
```

## Description

`plotperf(TR,goal,name,epoch)` takes these inputs,

| | |
|---|---|
| `TR` | — Training record returned by `train`. |
| `goal` | — Performance goal, default = `NaN`. |
| `name` | — Training function name, default = `''`. |
| `epoch` | — Number of epochs, default = length of training record. |

and plots the training performance, and if available, the performance goal, validation performance, and test performance.

## Examples

Here are eight input values `P` and associated targets `T`, plus a like number of validation inputs `VV.P` and targets `VV.T`.

```
P = 1:8; T = sin(P);
VV.P = P; VV.T = T+rand(1,8)*0.1;
```

The code below creates a network and trains it on this problem.

```
net = newff(minmax(P),[4 1],{'tansig','tansig'});
[net,tr] = train(net,P,T,[],[],VV);
```

During training `plotperf` was called to display the training record. You can also call `plotperf` directly with the final training record `TR`, as shown below.

```
plotperf(tr)
```

## Purpose

Plot perceptron input/target vectors

## Syntax

```
plotpv(P,T)
plotpv(P,T,V)
```

## Description

`plotpv(P,T)` take these inputs,

`P` — `R x Q` matrix of input vectors (`R` must be 3 or less)

`T` — `S` x `Q` matrix of binary target vectors (`S` must be 3 or less)

and plots column vectors in `P` with markers based on `T`

`plotpv(P,T,V)` takes an additional input,

`V` — Graph limits = `[x_min x_max y_min y_max]`

and plots the column vectors with limits set by `V`.

## Examples

The code below defines and plots the inputs and targets for a perceptron:

```
p = [0 0 1 1; 0 1 0 1];
t = [0 0 0 1];
plotpv(p,t)
```

The following code creates a perceptron with inputs ranging over the values in `P`, assigns values to its weights and biases, and plots the resulting classification line.

```
net = newp(minmax(p),1);
net.iw{1,1} = [-1.2 -0.5];
net.b{1} = 1;
plotpc(net.iw{1,1},net.b{1})
```

## See Also

`plotpc`

# plotsom

## Purpose

Plot self-organizing map

## Syntax

```
plotsom(pos)
plotsom(W,D,ND)
```

## Description

plotsom(pos) takes one argument,

POS — NxS matrix of S N-dimension neural positions and plots the neuron positions with red dots, linking the neurons within a Euclidean distance of 1

plotsom(W,d,nd) takes three arguments,

W — SxR weight matrix

D — SxS distance matrix

ND — Neighborhood distance, default = 1

and plots the neuron's weight vectors with connections between weight vectors whose neurons are within a distance of 1.

## Examples

Here are some neat plots of various layer topologies:

```
pos = hextop(5,6); plotsom(pos)
pos = gridtop(4,5); plotsom(pos)
pos = randtop(18,12); plotsom(pos)
pos = gridtop(4,5,2); plotsom(pos)
pos = hextop(4,4,3); plotsom(pos)
```

See newsom for an example of plotting a layer's weight vectors with the input vectors they map.

## See Also

newsom, learnsom, initsom.

## Purpose

Plot vectors as lines from the origin

## Syntax

```
plotv(M,T)
```

## Description

`plotv(M,T)` takes two inputs,

`M` — `R` x `Q` matrix of `Q` column vectors with `R` elements

`T` — (optional) the line plotting type, default = `'-'`

and plots the column vectors of `M`.

`R` must be 2 or greater. If `R` is greater than two, only the first two rows of `M` are used for the plot.

## Examples

```
plotv([-.4 0.7 .2; -0.5 .1 0.5],'-')
```

# plotvec

## Purpose

Plot vectors with different colors

## Syntax

```
plotvec(X,C,M)
```

## Description

`plotvec(X,C,M)` takes these inputs,

- `X` — Matrix of (column) vectors
- `C` — Row vector of color coordinate
- `M` — Marker, default = `'+'`

and plots each ith vector in `X` with a marker `M` and using the ith value in `C` as the color coordinate.

`plotvec(X)` only takes a matrix `X` and plots each ith vector in `X` with marker `'+'` using the index `i` as the color coordinate.

## Examples

```
x = [0 1 0.5 0.7; -1 2 0.5 0.1];
c = [1 2 3 4];
plotvec(x,c)
```

## Purpose

Pseudo-normalize columns of a matrix

## Syntax

`pnormc(X,R)`

## Description

`pnormc(X,R)` takes these arguments,

`X` — `M` x `N` matrix

`R` — (optional) radius to normalize columns to, default = 1

and returns `X` with an additional row of elements, which results in new column vector lengths of `R`.

**Caution:** For this function to work properly, the columns of `X` must originally have vector lengths less than `R`.

## Examples

```
x = [0.1 0.6; 0.3 0.1];
y = pnormc(x)
```

## See Also

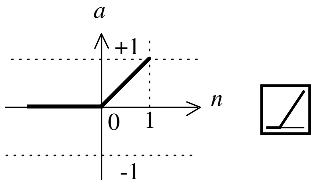
`normc`, `normr`

# poslin

## Purpose

Positive linear transfer function

## Graph and Symbol



*a = poslin(n)*

Positive Linear Transfer Funct.

## Syntax

```
A = poslin(N)
info = poslin(code)
```

## Description

`poslin` is a transfer function. Transfer functions calculate a layer's output from its net input.

`poslin(N)` takes one input,

`N` — `S` x `Q` matrix of net input (column) vectors

and returns the maximum of 0 and each element of `N`.

`poslin(code)` returns useful information for each `code` string:

`'deriv'` — Name of derivative function

`'name'` — Full name

`'output'` — Output range

`'active'` — Active input range

## Examples

Here is the code to create a plot of the `poslin` transfer function.

```
n = -5:0.1:5;
a = poslin(n);
plot(n,a)
```

## Network Use

To change a network so that a layer uses `poslin`, set `net.layers{i}.transferFcn` to 'poslin'.

Call `sim` to simulate the network with `poslin`.

## Algorithm

The transfer function `poslin` returns the output `n` if `n` is greater than or equal to zero and 0 if `n` is less than or equal to zero.

```
poslin(n) = n, if n >= 0; = 0, if n <= 0.
```

## See Also

`sim, purelin, satlin, satlins`

# postmnmx

### Purpose

Postprocess data that has been preprocessed by premnmx

### Syntax

```
[P,T] = postmnmx(PN,minp,maxp,TN,mint,maxt)
[p] = postmnmx(PN,minp,maxp)
```

### Description

postmnmx postprocesses the network training set that was preprocessed by premnmx. It converts the data back into unnormalized units.

postmnmx takes these inputs,

PN — R x Q matrix of normalized input vectors
minp — R x 1 vector containing minimums for each P
maxp — R x 1 vector containing maximums for each P
TN — S x Q matrix of normalized target vectors
mint — S x 1 vector containing minimums for each T
maxt — S x 1 vector containing maximums for each T

and returns,

P — R x Q matrix of input (column) vectors
T — R x Q matrix of target vectors

### Examples

In this example we normalize a set of training data with premnmx, create and train a network using the normalized data, simulate the network, unnormalize the output of the network using postmnmx, and perform a linear regression between the network outputs (unnormalized) and the targets to check the quality of the network training.

```
p = [-0.92 0.73 -0.47 0.74 0.29; -0.08 0.86 -0.67 -0.52 0.93];
t = [-0.08 3.4 -0.82 0.69 3.1];
[pn,minp,maxp,tn,mint,maxt] = premnmx(p,t);
net = newff(minmax(pn),[5 1],{'tansig' 'purelin'},'trainlm');
net = train(net,pn,tn);
an = sim(net,pn);
[a] = postmnmx(an,mint,maxt);
[m,b,r] = postreg(a,t);
```

### Algorithm

```
p = 0.5(pn+1)*(maxp-minp) + minp;
```

**See Also** `premnmx`, `prepca`, `poststd`

# postreg

## Purpose

Postprocess the trained network response with a linear regression

## Syntax

```
[M,B,R] = postreg(A,T)
```

## Description

postreg postprocesses the network training set by performing a linear regression between each element of the network response and the corresponding target.

postreg(A,T) takes these inputs,

- A — 1 x Q array of network outputs. One element of the network output
- T — 1 x Q array of targets. One element of the target vector

and returns,

- M — Slope of the linear regression
- B — Y intercept of the linear regression
- R — Regression R-value. R=1 means perfect correlation

## Examples

In this example we normalize a set of training data with prestd, perform a principal component transformation on the normalized data, create and train a network using the pca data, simulate the network, unnormalize the output of the network using poststd, and perform a linear regression between the network outputs (unnormalized) and the targets to check the quality of the network training.

```
p = [-0.92 0.73 -0.47 0.74 0.29; -0.08 0.86 -0.67 -0.52 0.93];
t = [-0.08 3.4 -0.82 0.69 3.1];
[pn,meanp,stdp,tn,meant,stdt] = prestd(p,t);
[ptrans,transMat] = prepca(pn,0.02);
net = newff(minmax(ptrans),[5 1],{'tansig''purelin'},'trainlm');
net = train(net,ptrans,tn);
an = sim(net,ptrans);
a = poststd(an,meant,stdt);
[m,b,r] = postreg(a,t);
```

## Algorithm

Performs a linear regression between the network response and the target, and then computes the correlation coefficient (R-value) between the network response and the target.

## See Also

`premnmx`, `prepca`

# poststd

## Purpose

Postprocess data which has been preprocessed by prestd

## Syntax

```
[P,T] = poststd(PN,meanp,stdp,TN,meant,stdt)
[p] = poststd(PN,meanp,stdp)
```

## Description

poststd postprocesses the network training set that was preprocessed by prestd. It converts the data back into unnormalized units.

poststd takes these inputs,

PN — R x Q matrix of normalized input vectors

meanp — R x 1 vector containing standard deviations for each P

stdp — R x 1 vector containing standard deviations for each P

TN — S x Q matrix of normalized target vectors

meant — S x 1 vector containing standard deviations for each T

stdt — S x 1 vector containing standard deviations for each T

and returns,

P — R x Q matrix of input (column) vectors

T — S x Q matrix of target vectors

## Examples

In this example we normalize a set of training data with prestd, create and train a network using the normalized data, simulate the network, unnormalize the output of the network using poststd, and perform a linear regression between the network outputs (unnormalized) and the targets to check the quality of the network training.

```
p = [-0.92 0.73 -0.47 0.74 0.29; -0.08 0.86 -0.67 -0.52 0.93];
t = [-0.08 3.4 -0.82 0.69 3.1];
[pn,meanp,stdp,tn,meant,stdt] = prestd(p,t);
net = newff(minmax(pn),[5 1],{'tansig' 'purelin'},'trainlm');
net = train(net,pn,tn);
an = sim(net,pn);
a = poststd(an,meant,stdt);
[m,b,r] = postreg(a,t);
```

## Algorithm

```
p = stdp*pn + meanp;
```

**See Also** premnmx, prepca, postmnmx, prestd

# premnmx

## Purpose

Preprocess data so that minimum is -1 and maximum is 1

## Syntax

```
[PN,minp,maxp,TN,mint,maxt] = premnmx(P,T)
[PN,minp,maxp] = premnmx(P)
```

## Description

premnmx preprocesses the network training set by normalizing the inputs and targets so that they fall in the interval [-1,1].

premnmx(P,T) takes these inputs,

P — R x Q matrix of input (column) vectors

T — S x Q matrix of target vectors

and returns,

| | |
|---|---|
| PN | — R x Q matrix of normalized input vectors |
| minp | — R x 1 vector containing minimums for each P |
| maxp | — R x 1 vector containing maximums for each P |
| TN | — S x Q matrix of normalized target vectors |
| mint | — S x 1 vector containing minimums for each T |
| maxt | — S x 1 vector containing maximums for each T |

## Examples

Here is the code to normalize a given data set so that the inputs and targets will fall in the range [-1,1].

```
p = [-10 -7.5 -5 -2.5 0 2.5 5 7.5 10];
t = [0 7.07 -10 -7.07 0 7.07 10 7.07 0];
[pn,minp,maxp,tn,mint,maxt] = premnmx(p,t);
```

If you just want to normalize the input,

```
[pn,minp,maxp] = premnmx(p);
```

## Algorithm

```
pn = 2*(p-minp)/(maxp-minp) - 1;
```

## See Also

prestd, prepca, postmnmx

## Purpose

Principal component analysis

## Syntax

```
[ptrans,transMat] = prepca(P,min_frac)
```

## Description

prepca preprocesses the network input training set by applying a principal component analysis. This analysis transforms the input data so that the elements of the input vector set will be uncorrelated. In addition, the size of the input vectors may be reduced by retaining only those components which contribute more than a specified fraction (min_frac) of the total variation in the data set.

prepca(P,min_frac) takes these inputs

P — R x Q matrix of centered input (column) vectors

min_frac — Minimum fraction variance component to keep

and returns

ptrans — Transformed data set

transMat — Transformation matrix

## Examples

Here is the code to perform a principal component analysis and retain only those components that contribute more than two percent to the variance in the data set. prestd is called first to create zero mean data, which is needed for prepca.

```
p=[-1.5 -0.58 0.21 -0.96 -0.79; -2.2 -0.87 0.31 -1.4  -1.2];
[pn,meanp,stdp] = prestd(p);
[ptrans,transMat] = prepca(pn,0.02);
```

Since the second row of p is almost a multiple of the first row, this example will produce a transformed data set that contains only one row.

## Algorithm

This routine uses singular value decomposition to compute the principal components. The input vectors are multiplied by a matrix whose rows consist of the eigenvectors of the input covariance matrix. This produces transformed input vectors whose components are uncorrelated and ordered according to the magnitude of their variance.

Those components that contribute only a small amount to the total variance in the data set are eliminated. It is assumed that the input data set has already been normalized so that it has a zero mean. The function `prestd` can be used to normalize the data.

## See Also

`prestd, premnmx`

## References

Jolliffe, I.T., *Principal Component Analysis*, New York: Springer-Verlag, 1986.

## Purpose

Preprocess data so that its mean is 0 and the standard deviation is 1

## Syntax

```
[pn,meanp,stdp,tn,meant,stdt] = prestd(p,t)
[pn,meanp,stdp] = prestd(p)
```

## Description

`prestd` preprocesses the network training set by normalizing the inputs and targets so that they have means of zero and standard deviations of 1.

`prestd(p,t)` takes these inputs,

- `p` — `R` x `Q` matrix of input (column) vectors
- `t` — `S` x `Q` matrix of target vectors

and returns,

- `pn` — `R` x `Q` matrix of normalized input vectors
- `meanp` — `R` x `1` vector containing mean for each `P`
- `stdp` — `R` x `1` vector containing standard deviations for each `P`
- `tn` — `S` x `Q` matrix of normalized target vectors
- `meant` — `S` x `1` vector containing mean for each `T`
- `stdt` — `S` x `1` vector containing standard deviations for each `T`

## Examples

Here is the code to normalize a given data set so that the inputs and targets will have means of zero and standard deviations of 1.

```
p = [-0.92 0.73 -0.47 0.74 0.29; -0.08 0.86 -0.67 -0.52 0.93];
t = [-0.08 3.4 -0.82 0.69 3.1];
[pn,meanp,stdp,tn,meant,stdt] = prestd(p,t);
```

If you just want to normalize the input,

```
[pn,meanp,stdp] = prestd(p);
```

## Algorithm

```
pn = (p-meanp)/stdp;
```

## See Also

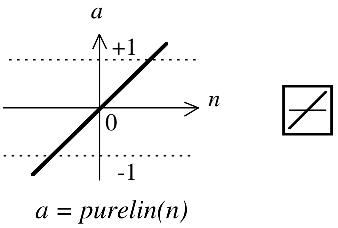
`premnmx`, `prepca`

# purelin

## Purpose

Linear transfer function

## Graph and Symbol



Linear Transfer Function

## Syntax

```
A = purelin(N)
info = purelin(code)
```

## Description

`purelin` is a transfer function. Transfer functions calculate a layer's output from its net input.

`purelin(N)` takes one input,

`N` — `S` x `Q` matrix of net input (column) vectors

and returns `N`.

`purelin(code)` returns useful information for each `code` string:

'`deriv`' — Name of derivative function

'`name`' — Full name

'`output`' — Output range

'`active`' — Active input range

## Examples

Here is the code to create a plot of the `purelin` transfer function.

```
n = -5:0.1:5;
a = purelin(n);
plot(n,a)
```

## Network Use

You can create a standard network that uses `purelin` by calling `newlin` or `newlind`.

To change a network so a layer uses `purelin`, set `net.layers{i}.transferFcn` to '`purelin`'.

In either case, call `sim` to simulate the network with `purelin`. See `newlin` or `newlind` for simulation examples.

## Algorithm

```
purelin(n) = n
```

## See Also

`sim`, `dpurelin`, `satlin`, `satlins`

# quant

## Purpose

Discretize values as multiples of a quantity

## Syntax

`quant(X,Q)`

## Description

`quant(X,Q)` takes two inputs,

`X` — Matrix, vector or scalar

`Q` — Minimum value

and returns values in `X` rounded to nearest multiple of `Q`.

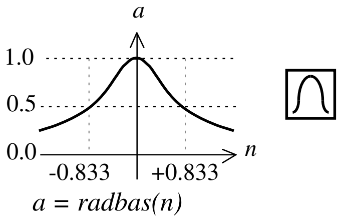
## Examples

```
x = [1.333 4.756 -3.897];
y = quant(x,0.1)
```

## Purpose

Radial basis transfer function

## Graph and Symbol



Radial Basis Function

## Syntax

```
A = radbas(N)
info = radbas(code)
```

## Description

`radbas` is a transfer function. Transfer functions calculate a layer's output from its net input.

`radbas(N)` takes one input,

`N` — `S` x `Q` matrix of net input (column) vectors

and returns each element of `N` passed through a radial basis function.

`radbas(code)` returns useful information for each `code` string:

`'deriv'` — Name of derivative function

`'name'` — Full name

`'output'` — Output range

`'active'` — Active input range

## Examples

Here we create a plot of the `radbas` transfer function.

```
n = -5:0.1:5;
a = radbas(n);
plot(n,a)
```

## Network Use

You can create a standard network that uses `radbas` by calling `newpnn` or `newgrnn`.

To change a network so that a layer uses `radbas`, set `net.layers{i}.transferFcn` to '`radbas`'.

In either case, call `sim` to simulate the network with `radbas`. See `newpnn` or `newgrnn` for simulation examples.

## Algorithm

`radbas(N)` calculates its output as:

```
  a = exp(-n2)
```

## See Also

`sim, tribas, dradbas`

## Purpose

Normalized column weight initialization function

## Syntax

```
W = randnc(S,PR)
W = randnc(S,R)
```

## Description

`randnc` is a weight initialization function.

randnc`(S,P)` takes two inputs,

`S` — Number of rows (neurons)

`PR` — `R` x 2 matrix of input value ranges = `[Pmin Pmax]`

and returns an `S` x `R` random matrix with normalized columns.

Can also be called as `randnc(S,R)`.

## Examples

A random matrix of four normalized three-element columns is generated:

```
M = randnc(3,4)
M =
     0.6007    0.4715    0.2724    0.5596
     0.7628    0.6967    0.9172    0.7819
     0.2395    0.5406    0.2907    0.2747
```

## See Also

`randnr`

# randnr

## Purpose

Normalized row weight initialization function

## Syntax

```
W = randnr(S,PR)
W = randnr(S,R)
```

## Description

randnr is a weight initialization function.

randnr(S,PR) takes two inputs,

S — Number of rows (neurons)

PR — R x 2 matrix of input value ranges = [Pmin Pmax]

and returns an S x R random matrix with normalized rows.

Can also be called as randnr(S,R).

## Examples

A matrix of three normalized four-element rows is generated:

```
M = randnr(3,4)
M =
    0.9713    0.0800    0.1838    0.1282
    0.8228    0.0338    0.1797    0.5381
    0.3042    0.5725    0.5436    0.5331
```

## See Also

randnc

## Purpose

Symmetric random weight/bias initialization function

## Syntax

```
W = rands(S,PR)
M = rands(S,R)
v = rands(S);
```

## Description

rands is a weight/bias initialization function.

rands(S,PR) takes,

S — Number of neurons

PR — R x 2 matrix of R input ranges

and returns an S-by-R weight matrix of random values between -1 and 1.

rands(S,R) returns an S-by-R matrix of random values. rands(S) returns an S-by-1 vector of random values.

## Examples

Here three sets of random values are generated with rands.

```
rands(4,[0 1; -2 2])
rands(4)
rands(2,3)
```

## Network Use

To prepare the weights and the bias of layer i of a custom network to be initialized with rands

1. Set net.initFcn to 'initlay'. (net.initParam will automatically become initlay's default parameters.)
2. Set net.layers{i}.initFcn to 'initwb'.
3. Set each net.inputWeights{i,j}.initFcn to 'rands'. Set each net.layerWeights{i,j}.initFcn to 'rands'. Set each net.biases{i}.initFcn to 'rands'.

To initialize the network call init.

## See Also

randnr, randnc, initwb, initlay, init

# randtop

**Purpose** Random layer topology function

**Syntax**

```
pos = randtop(dim1,dim2,...,dimN)
```

**Description**

`randtop` calculates the neuron positions for layers whose neurons are arranged in an `N` dimensional random pattern.

`randtop(dim1,dim2,...,dimN))` takes `N` arguments,

`dimi` — Length of layer in dimension `i`

and returns an `N` x `S` matrix of `N` coordinate vectors, where `S` is the product of `dim1*dim2*...*dimN`.

**Examples**

This code creates and displays a two-dimensional layer with 192 neurons arranged in a 16-by-12 random pattern.

```
pos = randtop(16,12); plotsom(pos)
```

This code plots the connections between the same neurons, but shows each neuron at the location of its weight vector. The weights are generated randomly so that the layer is very unorganized, as is evident in the plot.

```
W = rands(192,2); plotsom(W,dist(pos))
```

**See Also** `gridtop`, `hextop`

## Purpose

Change network weights and biases to previous initialization values

## Syntax

```
net = revert(net)
```

## Description

`revert (net)` returns neural network `net` with weight and bias values restored to the values generated the last time the network was initialized.

If the network has been altered so that it has different weight and bias connections or different input or layer sizes, then `revert` cannot set the weights and biases to their previous values and they will be set to zeros instead.

## Examples

Here a perceptron is created with a two-element input (with ranges of 0 to 1, and -2 to 2) and one neuron. Once it is created we can display the neuron's weights and bias.

```
net = newp([0 1;-2 2],1);
```

The initial network has weights and biases with zero values.

```
net.iw{1,1}, net.b{1}
```

We can change these values as follows.

```
net.iw{1,1} = [1 2];
net.b{1} = 5;
net.iw{1,1}, net.b{1}
```

We can recover the network's initial values as follows.

```
net = revert(net);
net.iw{1,1}, net.b{1}
```

## See Also

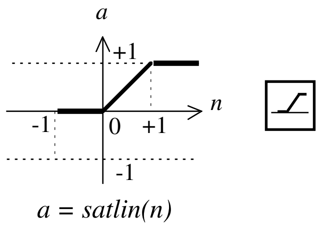
`init`, `sim`, `adapt`, `train`.

# satlin

## Purpose

Saturating linear transfer function

## Graph and Symbol



Satlin Transfer Function

## Syntax

```
A = satlin(N)
info = satlin(code)
```

## Description

`satlin` is a transfer function. Transfer functions calculate a layer's output from its net input.

`satlin(N)` takes one input,

`N` — `S` x `Q` matrix of net input (column) vectors

and returns values of `N` truncated into the interval `[-1, 1]`.

`satlin(code)` returns useful information for each `code` string:

'`deriv`' — Name of derivative function.

'`name`' — Full name.

'`output`' — Output range.

'`active`' — Active input range.

## Examples

Here is the code to create a plot of the `satlin` transfer function.

```
n = -5:0.1:5;
a = satlin(n);
plot(n,a)
```

## Network Use

To change a network so that a layer uses `satlin`, set `net.layers{i}.transferFcn` to '`satlin`'.

Call sim to simulate the network with satlin. See newhop for simulation examples.

## Algorithm

```
satlin(n) = 0, if n <= 0; n, if 0 <= n <= 1; 1, if 1 <= n.
```

## See Also

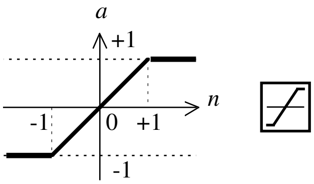
sim, poslin, satlins, purelin

# satlins

## Purpose

Symmetric saturating linear transfer function

## Graph and Symbol



*a = satlins(n)*

Satlins Transfer Function

## Syntax

```
A = satlins(N)
info = satlins(code)
```

## Description

`satlins` is a transfer function. Transfer functions calculate a layer's output from its net input.

`satlins(N)` takes one input,

`N` — `S` x `Q` matrix of net input (column) vectors

and returns values of `N` truncated into the interval `[-1, 1]`.

`satlins(code)` returns useful information for each `code` string:

`'deriv'` — Name of derivative function

`'name'` — Full name

`'output'` — Output range

`'active'` — Active input range

## Examples

Here is the code to create a plot of the `satlins` transfer function.

```
n = -5:0.1:5;
a = satlins(n);
plot(n,a)
```

## Network Use

You can create a standard network that uses `satlins` by calling `newhop`.

To change a network so that a layer uses `satlins`, set `net.layers{i}.transferFcn` to '`satlins`'.

In either case, call `sim` to simulate the network with `satlins`. See `newhop` for simulation examples.

## Algorithm

```
satlins(n) = -1, if n <= -1; n, if -1 <= n <= 1; 1, if 1 <= n.
```

## See Also

```
sim, satlin, poslin, purelin
```

# seq2con

## Purpose

Convert sequential vectors to concurrent vectors

## Syntax

```
b = seq2con(s)
```

## Description

The Neural Network Toolbox represents batches of vectors with a matrix, and sequences of vectors with multiple columns of a cell array.

seq2con and con2seq allow concurrent vectors to be converted to sequential vectors, and back again.

seq2con(S) takes one input,

S — N x TS cell array of matrices with M columns

and returns,

B — N x 1 cell array of matrices with M*TS columns.

## Examples

Here three sequential values are converted to concurrent values.

```
p1 = {1 4 2}
p2 = seq2con(p1)
```

Here two sequences of vectors over three time steps are converted to concurrent vectors.

```
p1 = {[1; 1] [5; 4] [1; 2]; [3; 9] [4; 1] [9; 8]}
p2 = seq2con(p1)
```

## See Also

con2seq, concur

## Purpose

Set all network weight and bias values with a single vector

## Syntax

```
net = setx(net,X)
```

## Description

This function sets a networks weight and biases to a vector of values.

```
net = setx(net,X)
```

`net` — Neural network

`X` — Vector of weight and bias values

## Examples

Here we create a network with a two-element input, and one layer of three neurons.

```
net = newff([0 1; -1 1],[3]);
```

The network has six weights (3 neurons * 2 input elements) and three biases (3 neurons) for a total of nine weight and bias values. We can set them to random values as follows:

```
net = setx(net,rand(9,1));
```

We can then view the weight and bias values as follows:

```
net.iw{1,1}
net.b{1}
```

## See Also

`getx`, `formx`

# sim

### Purpose

Simulate a neural network

### Syntax

```
[Y,Pf,Af,E,perf] = sim(net,P,Pi,Ai,T)
[Y,Pf,Af,E,perf] = sim(net,{Q TS},Pi,Ai,T)
[Y,Pf,Af,E,perf] = sim(net,Q,Pi,Ai,T)
```

### To Get Help

Type `help network/sim`

### Description

`sim` simulates neural networks.

`[Y,Pf,Af,E,perf] = sim(net,P,PiAi,T)` takes,

`net` — Network<br>
`P` — Network inputs<br>
`Pi` — Initial input delay conditions, default = zeros<br>
`Ai` — Initial layer delay conditions, default = zeros<br>
`T` — Network targets, default = zeros

and returns,

`Y` — Network outputs<br>
`Pf` — Final input delay conditions<br>
`Af` — Final layer delay conditions<br>
`E` — Network errors<br>
`perf` — Network performance

Note that arguments `Pi`, `Ai`, `Pf`, and `Af` are optional and need only be used for networks that have input or layer delays.

`sim`'s signal arguments can have two formats: cell array or matrix.

The cell array format is easiest to describe. It is most convenient for networks with multiple inputs and outputs, and allows sequences of inputs to be presented:

P — Ni x TS cell array, each element P{i,ts} is an Ri x Q matrix

Pi — Ni x ID cell array, each element Pi{i,k} is an Ri x Q matrix

Ai — Nl x LD cell array, each element Ai{i,k} is an Si x Q matrix

T — Nt x TS cell array, each element P{i,ts} is an Vi x Q matrix

Y — NO x TS cell array, each element Y{i,ts} is a Ui x Q matrix

Pf — Ni x ID cell array, each element Pf{i,k} is an Ri x Q matrix

Af — Nl x LD cell array, each element Af{i,k} is an Si x Q matrix

E — Nt x TS cell array, each element P{i,ts} is an Vi x Q matrix

where

Ni = net.numInputs

Nl = net.numLayers

No = net.numOutputs

D = net.numInputDelays

LD = net.numLayerDelays

TS = Number of time steps

Q = Batch size

Ri = net.inputs{i}.size

Si = net.layers{i}.size

Ui = net.outputs{i}.size

The columns of Pi, Ai, Pf, and Af are ordered from oldest delay condition to most recent:

Pi{i,k} = input i at time ts=k ID

Pf{i,k} = input i at time ts=TS+k ID

Ai{i,k} = layer output i at time ts=k LD

Af{i,k} = layer output i at time ts=TS+k LD

The matrix format can be used if only one time step is to be simulated (TS = 1). It is convenient for networks with only one input and output, but can also be used with networks that have more.

Each matrix argument is found by storing the elements of the corresponding cell array argument into a single matrix:

- P — (sum of Ri) x Q matrix
- Pi — (sum of Ri) x (ID*Q) matrix
- Ai — (sum of Si) x (LD*Q) matrix
- T — (sum of Vi)xQ matrix
- Y — (sum of Ui) x Q matrix
- Pf — (sum of Ri) x (ID*Q) matrix
- Af — (sum of Si) x (LD*Q) matrix
- E — (sum of Vi)xQ matrix

[Y,Pf,Af] = sim(net,{Q TS},Pi,Ai) is used for networks which do not have an input, such as Hopfield networks, when cell array notation is used.

## Examples

Here newp is used to create a perceptron layer with a two-element input (with ranges of [0 1]), and a single neuron.

```
net = newp([0 1;0 1],1);
```

Here the perceptron is simulated for an individual vector, a batch of three vectors, and a sequence of three vectors.

```
p1 = [.2; .9]; a1 = sim(net,p1)
p2 = [.2 .5 .1; .9 .3 .7]; a2 = sim(net,p2)
p3 = {[.2; .9] [.5; .3] [.1; .7]}; a3 = sim(net,p3)
```

Here newlind is used to create a linear layer with a three-element input, two neurons.

```
net = newlin([0 2;0 2;0 2],2,[0 1]);
```

Here the linear layer is simulated with a sequence of two input vectors using the default initial input delay conditions (all zeros).

```
p1 = {[2; 0.5; 1] [1; 1.2; 0.1]};
[y1,pf] = sim(net,p1)
```

Here the layer is simulated for three more vectors using the previous final input delay conditions as the new initial delay conditions.

```
p2 = {[0.5; 0.6; 1.8] [1.3; 1.6; 1.1] [0.2; 0.1; 0]};
```

```
[y2,pf] = sim(net,p2,pf)
```

Here newelm is used to create an Elman network with a one-element input, and a layer 1 with three tansig neurons followed by a layer 2 with two purelin neurons. Because it is an Elman network it has a tap delay line with a delay of 1 going from layer 1 to layer 1.

```
net = newelm([0 1],[3 2],{'tansig','purelin'});
```

Here the Elman network is simulated for a sequence of three values using default initial delay conditions.

```
p1 = {0.2 0.7 0.1};
[y1,pf,af] = sim(net,p1)
```

Here the network is simulated for four more values, using the previous final delay conditions as the new initial delay conditions.

```
p2 = {0.1 0.9 0.8 0.4};
[y2,pf,af] = sim(net,p2,pf,af)
```

## Algorithm

sim uses these properties to simulate a network net.

```
net.numInputs, net.numLayers
net.outputConnect, net.biasConnect
net.inputConnect, net.layerConnect
```

These properties determine the network's weight and bias values, and the number of delays associated with each weight:

```
net.IW{i,j}
net.LW{i,j}
net.b{i}
net.inputWeights{i,j}.delays
net.layerWeights{i,j}.delays
```

These function properties indicate how sim applies weight and bias values to inputs to get each layer's output:

```
net.inputWeights{i,j}.weightFcn
net.layerWeights{i,j}.weightFcn
net.layers{i}.netInputFcn
net.layers{i}.transferFcn
```

See Chapter 2, “Neuron Model and Network Architectures” for more information on network simulation.

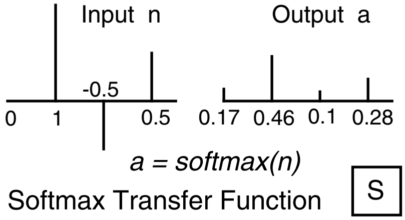
## See Also

`init`, `adapt`, `train`, `revert`

## Purpose

Soft max transfer function

## Graph and Symbol



## Syntax

```
A = softmax(N)
info = softmax(code)
```

## Description

softmax is a transfer function. Transfer functions calculate a layer's output from its net input.

softmax(N) takes one input argument,

N — S x Q matrix of net input (column) vectors

and returns output vectors with elements between 0 and 1, but with their size relations intact.

softmax('code') returns information about this function.

These codes are defined:

'deriv' — Name of derivative function.

'name' — Full name.

'output' — Output range.

'active' — Active input range.

compet does not have a derivative function.

## Examples

Here we define a net input vector N, calculate the output, and plot both with bar graphs.

```
n = [0; 1; -0.5; 0.5];
a = softmax(n);
subplot(2,1,1), bar(n), ylabel('n')
subplot(2,1,2), bar(a), ylabel('a')
```

### Network Use

To change a network so that a layer uses `softmax`, set `net.layers{i,j}.transferFcn` to '`softmax`'.

Call `sim` to simulate the network with `softmax`. See `newc` or `newpnn` for simulation examples.

### See Also

`sim`, `compet`

## Purpose

One-dimensional minimization using backtracking

## Syntax

```
[a,gX,perf,retcode,delta,tol] =
srchbac(net,X,Pd,Tl,Ai,Q,TS,dX,gX,perf,dperf,delta,TOL,ch_perf)
```

## Description

srchbac is a linear search routine. It searches in a given direction to locate the minimum of the performance function in that direction. It uses a technique called backtracking.

srchbac(net,X,Pd,Tl,Ai,Q,TS,dX,gX,perf,dperf,delta,TOL,ch_perf) takes these inputs,

net — Neural network

X — Vector containing current values of weights and biases

Pd — Delayed input vectors

Tl — Layer target vectors

Ai — Initial input delay conditions

Q — Batch size

TS — Time steps

dX — Search direction vector

gX — Gradient vector

perf — Performance value at current X

dperf — Slope of performance value at current X in direction of dX

delta — Initial step size

tol — Tolerance on search

ch_perf — Change in performance on previous step

and returns,

a — Step size, which minimizes performance

gX — Gradient at new minimum point

perf — Performance value at new minimum point

retcode — Return code which has three elements. The first two elements correspond to the number of function evaluations in the two stages of the search. The third element is a return code. These will have different

meanings for different search algorithms. Some may not be used in this function.

0 - normal; 1 - minimum step taken;

2 - maximum step taken; 3 - beta condition not met.

delta — New initial step size. Based on the current step size

tol — New tolerance on search

Parameters used for the backstepping algorithm are:

alpha — Scale factor that determines sufficient reduction in perf

beta — Scale factor that determines sufficiently large step size

low_lim — Lower limit on change in step size

up_lim — Upper limit on change in step size

maxstep — Maximum step length

minstep — Minimum step length

scale_tol — Parameter which relates the tolerance tol to the initial step size delta. Usually set to 20

The defaults for these parameters are set in the training function that calls it. See traincgf, traincgb, traincgp, trainbfg, trainoss.

Dimensions for these variables are:

Pd — No x Ni x TS cell array, each element P{i,j,ts} is a Dij x Q matrix

Tl — Nl x TS cell array, each element P{i,ts} is an Vi x Q matrix

Ai — Nl x LD cell array, each element Ai{i,k} is an Si x Q matrix

where

```
Ni = net.numInputs
Nl = net.numLayers
LD = net.numLayerDelays
Ri = net.inputs{i}.size
Si = net.layers{i}.size
Vi = net.targets{i}.size
Dij = Ri * length(net.inputWeights{i,j}.delays)
```

## Examples

Here is a problem consisting of inputs p and targets t that we would like to solve with a network.

```
p = [0 1 2 3 4 5];
t = [0 0 0 1 1 1];
```

Here a two-layer feed-forward network is created. The network's input ranges from `[0 to 10]`. The first layer has two `tansig` neurons, and the second layer has one `logsig` neuron. The `traincgf` network training function and the `srchbac` search function are to be used.

### Create and Test a Network

```
net = newff([0 5],[2 1],{'tansig','logsig'},'traincgf');
a = sim(net,p)
```

### Train and Retest the Network

```
net.trainParam.searchFcn = 'srchbac';
net.trainParam.epochs = 50;
net.trainParam.show = 10;
net.trainParam.goal = 0.1;
net = train(net,p,t);
a = sim(net,p)
```

## Network Use

You can create a standard network that uses `srchbac` with `newff`, `newcf`, or `newelm`.

To prepare a custom network to be trained with `traincgf`, using the line search function `srchbac`

1. Set `net.trainFcn` to '`traincgf`'. This will set `net.trainParam` to `traincgf`'s default parameters.
2. Set `net.trainParam.searchFcn` to '`srchbac`'.

The `srchbac` function can be used with any of the following training functions: `traincgf`, `traincgb`, `traincgp`, `trainbfg`, `trainoss`.

## Algorithm

`srchbac` locates the minimum of the performance function in the search direction `dX`, using the backtracking algorithm described on page 126 and 328 of Dennis and Schnabel's book noted below.

## See Also

`srchcha, srchgol, srchhyb`

## References

Dennis, J. E., and R. B. Schnabel, *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*, Englewood Cliffs, NJ: Prentice-Hall, 1983.

## Purpose

One-dimensional interval location using Brent's method

## Syntax

```
[a,gX,perf,retcode,delta,tol] =
srchbre(net,X,Pd,Tl,Ai,Q,TS,dX,gX,perf,dperf,delta,tol,ch_perf)
```

## Description

`srchbre` is a linear search routine. It searches in a given direction to locate the minimum of the performance function in that direction. It uses a technique called Brent's technique.

`srchbre(net,X,Pd,Tl,Ai,Q,TS,dX,gX,perf,dperf,delta,tol,ch_perf)` takes these inputs,

| | |
|---|---|
| `net` | — Neural network |
| `X` | — Vector containing current values of weights and biases |
| `Pd` | — Delayed input vectors |
| `Tl` | — Layer target vectors |
| `Ai` | — Initial input delay conditions |
| `Q` | — Batch size |
| `TS` | — Time steps |
| `dX` | — Search direction vector |
| `gX` | — Gradient vector |
| `perf` | — Performance value at current `X` |
| `dperf` | — Slope of performance value at current `X` in direction of `dX` |
| `delta` | — Initial step size |
| `tol` | — Tolerance on search |
| `ch_perf` | — Change in performance on previous step |

and returns,

| | |
|---|---|
| `a` | — Step size, which minimizes performance |
| `gX` | — Gradient at new minimum point |
| `perf` | — Performance value at new minimum point |

`retcode` — Return code, which has three elements. The first two elements correspond to the number of function evaluations in the two stages of the search. The third element is a return code. These will have different

meanings for different search algorithms. Some may not be used in this function.

0 - normal; 1 - minimum step taken;

2 - maximum step taken; 3 - beta condition not met.

`delta` — New initial step size. Based on the current step size

`tol` — New tolerance on search

Parameters used for the brent algorithm are:

`alpha` — Scale factor, which determines sufficient reduction in `perf`

`beta` — Scale factor, which determines sufficiently large step size

`bmax` — Largest step size

`scale_tol` — Parameter which relates the tolerance `tol` to the initial step size `delta`. Usually set to 20

The defaults for these parameters are set in the training function that calls it. See `traincgf`, `traincgb`, `traincgp`, `trainbfg`, `trainoss`.

Dimensions for these variables are:

`Pd` — `No` x `Ni` x `TS` cell array, each element `P{i,j,ts}` is a `Dij` x `Q` matrix

`Tl` — `Nl` x `TS` cell array, each element `P{i,ts}` is an `Vi` x `Q` matrix

`Ai` — `Nl` x `LD` cell array, each element `Ai{i,k}` is an `Si` x `Q` matrix

where

```
Ni = net.numInputs
Nl = net.numLayers
LD = net.numLayerDelays
Ri = net.inputs{i}.size
Si = net.layers{i}.size
Vi = net.targets{i}.size
Dij = Ri * length(net.inputWeights{i,j}.delays)
```

## Examples

Here is a problem consisting of inputs `p` and targets `t` that we would like to solve with a network.

```
p = [0 1 2 3 4 5];
t = [0 0 0 1 1 1];
```

Here a two-layer feed-forward network is created. The network's input ranges from `[0 to 10]`. The first layer has two `tansig` neurons, and the second layer has one `logsig` neuron. The `traincgf` network training function and the `srchbac` search function are to be used.

### Create and Test a Network

```
net = newff([0 5],[2 1],{'tansig','logsig'},'traincgf');
a = sim(net,p)
```

### Train and Retest the Network

```
net.trainParam.searchFcn = 'srchbre';
net.trainParam.epochs = 50;
net.trainParam.show = 10;
net.trainParam.goal = 0.1;
net = train(net,p,t);
a = sim(net,p)
```

## Network Use

You can create a standard network that uses `srchbre` with `newff`, `newcf`, or `newelm`.

To prepare a custom network to be trained with `traincgf`, using the line search function `srchbre`

1 Set `net.trainFcn` to '`traincgf`'. This will set `net.trainParam` to `traincgf`'s default parameters.

2 Set `net.trainParam.searchFcn` to '`srchbre`'.

The `srchbre` function can be used with any of the following training functions: `traincgf`, `traincgb`, `traincgp`, `trainbfg`, `trainoss.`

## Algorithm

`srchbre` brackets the minimum of the performance function in the search direction `dX`, using Brent's algorithm described on page 46 of Scales (see reference below). It is a hybrid algorithm based on the golden section search and the quadratic approximation.

## See Also

`srchbac, srchcha, srchgol, srchhyb`

## References

Scales, L. E., *Introduction to Non-Linear Optimization*, New York: Springer-Verlag, 1985.

# srchcha

## Purpose

One-dimensional minimization using Charalambous' method

## Syntax

```
[a,gX,perf,retcode,delta,tol] =
srchcha(net,X,Pd,Tl,Ai,Q,TS,dX,gX,perf,dperf,delta,tol,ch_perf)
```

## Description

`srchcha` is a linear search routine. It searches in a given direction to locate the minimum of the performance function in that direction. It uses a technique based on Charalambous' method.

`srchcha(net,X,Pd,Tl,Ai,Q,TS,dX,gX,perf,dperf,delta,tol,ch_perf)` takes these inputs,

`net` — Neural network

`X` — Vector containing current values of weights and biases

`Pd` — Delayed input vectors

`Tl` — Layer target vectors

`Ai` — Initial input delay conditions

`Q` — Batch size

`TS` — Time steps

`dX` — Search direction vector

`gX` — Gradient vector

`perf` — Performance value at current `X`

`dperf` — Slope of performance value at current `X` in direction of `dX`

`delta` — Initial step size

`tol` — Tolerance on search

`ch_perf` — Change in performance on previous step

and returns,

`a` — Step size, which minimizes performance

`gX` — Gradient at new minimum point

`perf` — Performance value at new minimum point

`retcode` — Return code, which has three elements. The first two elements correspond to the number of function evaluations in the two stages of the search. The third element is a return code. These will have different

meanings for different search algorithms. Some may not be used in this function.

0 - normal; 1 - minimum step taken;

2 - maximum step taken; 3 - beta condition not met.

`delta` — New initial step size. Based on the current step size

`tol` — New tolerance on search

Parameters used for the Charalambous algorithm are:

`alpha` — Scale factor, which determines sufficient reduction in `perf`

`beta` — Scale factor, which determines sufficiently large step size

`gama` — Parameter to avoid small reductions in performance. Usually set to 0.1

`scale_tol` — Parameter, which relates the tolerance `tol` to the initial step size delta. Usually set to 20

The defaults for these parameters are set in the training function that calls it. See `traincgf`, `traincgb`, `traincgp`, `trainbfg`, `trainoss.`

Dimensions for these variables are

| | |
|---|---|
| `Pd` | `No` x `Ni` x `TS` cell array, each element `P{i,j,ts}` is a `Dij` x `Q` matrix |
| `Tl` | `Nl` x `TS` cell array, each element `P{i,ts}` is an `Vi` x `Q` matrix |
| `Ai` | `Nl` x `LD` cell array, each element `Ai{i,k}` is an `Si` x `Q` matrix |

where

```
Ni = net.numInputs
Nl = net.numLayers
LD = net.numLayerDelays
Ri = net.inputs{i}.size
Si = net.layers{i}.size
Vi = net.targets{i}.size
Dij = Ri * length(net.inputWeights{i,j}.delays)
```

## Examples

Here is a problem consisting of inputs `p` and targets `t` that we would like to solve with a network.

```
p = [0 1 2 3 4 5];
t = [0 0 0 1 1 1];
```

Here a two-layer feed-forward network is created. The network's input ranges from [0 to 10]. The first layer has two tansig neurons, and the second layer has one logsig neuron. The traincgf network training function and the srchcha search function are to be used.

### Create and Test a Network

```
net = newff([0 5],[2 1],{'tansig','logsig'},'traincgf');
a = sim(net,p)
```

### Train and Retest the Network

```
net.trainParam.searchFcn = 'srchcha';
net.trainParam.epochs = 50;
net.trainParam.show = 10;
net.trainParam.goal = 0.1;
net = train(net,p,t);
a = sim(net,p)
```

## Network Use

You can create a standard network that uses srchcha with newff, newcf, or newelm.

To prepare a custom network to be trained with traincgf, using the line search function srchcha

1. Set net.trainFcn to 'traincgf'. This will set net.trainParam to traincgf's default parameters.
2. Set net.trainParam.searchFcn to 'srchcha'.

The srchcha function can be used with any of the following training functions: traincgf, traincgb, traincgp, trainbfg, trainoss.

## Algorithm

srchcha locates the minimum of the performance function in the search direction dX, using an algorithm based on the method described in Charalambous (see reference below).

## See Also

srchbac, srchbre, srchgol, srchhyb

## References

Charalambous, C.,"Conjugate gradient algorithm for efficient training of artificial neural networks," *IEEE Proceedings*, vol. 139, no. 3, pp. 301–310, June 1992.

## Purpose

One-dimensional minimization using golden section search

## Syntax

```
[a,gX,perf,retcode,delta,tol] =
srchgol(net,X,Pd,Tl,Ai,Q,TS,dX,gX,perf,dperf,delta,tol,ch_perf)
```

## Description

`srchgol` is a linear search routine. It searches in a given direction to locate the minimum of the performance function in that direction. It uses a technique called the golden section search.

`srchgol(net,X,Pd,Tl,Ai,Q,TS,dX,gX,perf,dperf,delta,tol,ch_perf)` takes these inputs,

`net` — Neural network

`X` — Vector containing current values of weights and biases

`Pd` — Delayed input vectors

`Tl` — Layer target vectors

`Ai` — Initial input delay conditions

`Q` — Batch size

`TS` — Time steps

`dX` — Search direction vector

`gX` — Gradient vector

`perf` — Performance value at current `X`

`dperf` — Slope of performance value at current `X` in direction of `dX`

`delta` — Initial step size

`tol` — Tolerance on search

`ch_perf` — Change in performance on previous step

and returns,

`a` — Step size, which minimizes performance

`gX` — Gradient at new minimum point

`perf` — Performance value at new minimum point

`retcode` — Return code, which has three elements. The first two elements correspond to the number of function evaluations in the two stages of the search. The third element is a return code. These will have different

meanings for different search algorithms. Some may not be used in this function.

0 - normal; 1 - minimum step taken;

2 - maximum step taken; 3 - beta condition not met.

`delta` — New initial step size. Based on the current step size.

`tol` — New tolerance on search

Parameters used for the golden section algorithm are:

`alpha` — Scale factor, which determines sufficient reduction in `perf`

`bmax` — Largest step size

`scale_tol` — Parameter, which relates the tolerance `tol` to the initial step size `delta`. Usually set to 20

The defaults for these parameters are set in the training function that calls it. See `traincgf, traincgb, traincgp, trainbfg, trainoss.`

Dimensions for these variables are:

`Pd` — `No x Ni x TS` cell array, each element `P{i,j,ts}` is a `Dij x Q` matrix

`Tl` — `Nl x TS` cell array, each element `P{i,ts}` is an `Vi x Q` matrix

`Ai` — `Nl x LD` cell array, each element `Ai{i,k}` is an `Si x Q` matrix

where

```
Ni = net.numInputs
Nl = net.numLayers
LD = net.numLayerDelays
Ri = net.inputs{i}.size
Si = net.layers{i}.size
Vi = net.targets{i}.size
Dij = Ri * length(net.inputWeights{i,j}.delays)
```

## Examples

Here is a problem consisting of inputs `p` and targets `t` that we would like to solve with a network.

```
p = [0 1 2 3 4 5];
t = [0 0 0 1 1 1];
```

Here a two-layer feed-forward network is created. The network's input ranges from `[0 to 10]`. The first layer has two `tansig` neurons, and the second layer has one logsig neuron. The traincgf network training function and the srchgol search function are to be used.

### Create and Test a Network

```
net = newff([0 5],[2 1],{'tansig','logsig'},'traincgf');
a = sim(net,p)
```

### Train and Retest the Network

```
net.trainParam.searchFcn = 'srchgol';
net.trainParam.epochs = 50;
net.trainParam.show = 10;
net.trainParam.goal = 0.1;
net = train(net,p,t);
a = sim(net,p)
```

## Network Use

You can create a standard network that uses srchgol with newff, newcf, or newelm.

To prepare a custom network to be trained with traincgf, using the line search function srchgol

1. Set net.trainFcn to 'traincgf'. This will set net.trainParam to traincgf's default parameters.
2. Set net.trainParam.searchFcn to 'srchgol'.

The srchgol function can be used with any of the following training functions: traincgf, traincgb, traincgp, trainbfg, trainoss.

## Algorithm

srchgol locates the minimum of the performance function in the search direction dX, using the golden section search. It is based on the algorithm as described on page 33 of Scales (see reference below).

## See Also

srchbac, srchbre, srchcha, srchhyb

## References

Scales, L. E., *Introduction to Non-Linear Optimization*, New York: Springer-Verlag, 1985.

# srchhyb

**Purpose** One-dimensional minimization using a hybrid bisection-cubic search

**Syntax**

```
[a,gX,perf,retcode,delta,tol] =
srchhyb(net,X,P,T,Q,TS,dX,gX,perf,dperf,delta,tol,ch_perf)
```

**Description** `srchhyb` is a linear search routine. It searches in a given direction to locate the minimum of the performance function in that direction. It uses a technique that is a combination of a bisection and a cubic interpolation.

`srchhyb(net,X,Pd,Tl,Ai,Q,TS,dX,gX,perf,dperf,delta,tol,ch_perf)` takes these inputs,

`net` — Neural network

`X` — Vector containing current values of weights and biases

`Pd` — Delayed input vectors

`Tl` — Layer target vectors

`Ai` — Initial input delay conditions

`Q` — Batch size

`TS` — Time steps

`dX` — Search direction vector

`gX` — Gradient vector

`perf` — Performance value at current `X`

`dperf` — Slope of performance value at current `X` in direction of `dX`

`delta` — Initial step size

`tol` — Tolerance on search

`ch_perf` — Change in performance on previous step

and returns,

`a` — Step size, which minimizes performance

`gX` — Gradient at new minimum point

`perf` — Performance value at new minimum point

`retcode` — Return code, which has three elements. The first two elements correspond to the number of function evaluations in the two stages of the search. The third element is a return code. These will have different

meanings for different search algorithms. Some may not be used in this function.

0 - normal; 1 - minimum step taken;

2 - maximum step taken; 3 - beta condition not met.

`delta` — New initial step size. Based on the current step size.

`tol` — New tolerance on search

Parameters used for the hybrid bisection-cubic algorithm are:

`alpha` — Scale factor, which determines sufficient reduction in `perf`

`beta` — Scale factor, which determines sufficiently large step size

`bmax` — Largest step size

`scale_tol` — Parameter, which relates the tolerance `tol` to the initial step size `delta`. Usually set to 20.

The defaults for these parameters are set in the training function that calls it. See `traincgf`, `traincgb`, `traincgp`, `trainbfg`, `trainoss.`

Dimensions for these variables are:

`Pd` — `No x Ni x TS` cell array, each element `P{i,j,ts}` is a `Dij x Q` matrix

`Tl` — `Nl x TS` cell array, each element `P{i,ts}` is an `Vi x Q` matrix

`Ai` — `Nl x LD` cell array, each element `Ai{i,k}` is an `Si x Q` matrix

where

```
Ni = net.numInputs
Nl = net.numLayers
LD = net.numLayerDelays
Ri = net.inputs{i}.size
Si = net.layers{i}.size
Vi = net.targets{i}.size
Dij = Ri * length(net.inputWeights{i,j}.delays)
```

## Examples

Here is a problem consisting of inputs `p` and targets `t` that we would like to solve with a network.

```
p = [0 1 2 3 4 5];
t = [0 0 0 1 1 1];
```

Here a two-layer feed-forward network is created. The network's input ranges from [0 to 10]. The first layer has two tansig neurons, and the second layer has one logsig neuron. The traincgf network training function and the srchhyb search function are to be used.

### Create and Test a Network

```
net = newff([0 5],[2 1],{'tansig','logsig'},'traincgf');
a = sim(net,p)
```

### Train and Retest the Network

```
net.trainParam.searchFcn = 'srchhyb';
net.trainParam.epochs = 50;
net.trainParam.show = 10;
net.trainParam.goal = 0.1;
net = train(net,p,t);
a = sim(net,p)
```

## Network Use

You can create a standard network that uses srchhyb with newff, newcf, or newelm.

To prepare a custom network to be trained with traincgf, using the line search function srchhyb

1. Set net.trainFcn to 'traincgf'. This will set net.trainParam to traincgf's default parameters.
2. Set net.trainParam.searchFcn to 'srchhyb'.

The srchhyb function can be used with any of the following training functions: traincgf, traincgb, traincgp, trainbfg, trainoss.

## Algorithm

srchhyb locates the minimum of the performance function in the search direction dX, using the hybrid bisection-cubic interpolation algorithm described on page 50 of Scales (see reference below).

## See Also

srchbac, srchbre, srchcha, srchgol

## References

Scales, L. E., *Introduction to Non-Linear Optimization*, New York: Springer-Verlag, 1985.

## Purpose

Sum squared error performance function

## Syntax

```
perf = sse(E,X,PP)
perf = sse(E,net,PP)
info = sse(code)
```

## Description

`sse` is a network performance function. It measures performance according to the sum of squared errors.

`sse(E,X,PP)` takes from one to three arguments,

`E` — Matrix or cell array of error vector(s)

`X` — Vector of all weight and bias values (ignored)

`PP` — Performance parameters (ignored)

and returns the sum squared error.

`sse(E,net,PP)` can take an alternate argument to `X`,

`net` — Neural network from which `X` can be obtained (ignored)

`sse(code)` returns useful information for each `code` string:

`'deriv'` — Name of derivative function

`'name'` — Full name

`'pnames'` — Names of training parameters

`'pdefaults'` — Default training parameters

## Examples

Here a two-layer feed-forward is created with a 1-element input ranging from -10 to 10, four hidden `tansig` neurons, and one `purelin` output neuron.

```
net = newff([-10 10],[4 1],{'tansig','purelin'});
```

Here the network is given a batch of inputs `P`. The error is calculated by subtracting the output `A` from target `T`. Then the sum squared error is calculated.

```
p = [-10 -5 0 5 10];
t = [0 0 1 1 1];
y = sim(net,p)
e = t-y
perf = sse(e)
```

Note that sse can be called with only one argument because the other arguments are ignored. sse supports those arguments to conform to the standard performance function argument list.

## Network Use

To prepare a custom network to be trained with sse, set net.performFcn to 'sse'. This will automatically set net.performParam to the empty matrix [], as sse has no performance parameters.

Calling train or adapt will result in sse being used to calculate performance.

## See Also

dsse

**Purpose** Sum squared elements of a matrix

**Syntax** `sumsqr(m)`

**Description** `sumsqr(M)` returns the sum of the squared elements in `M`.

**Examples**

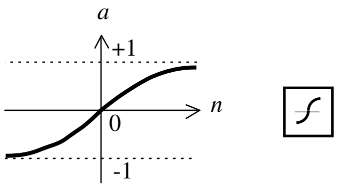
```
s = sumsqr([1 2;3 4])
```

# tansig

## Purpose

Hyperbolic tangent sigmoid transfer function

## Graph and Symbol



*a = tansig(n)*

Tan-Sigmoid Transfer Function

## Syntax

```
A = tansig(N)
info = tansig(code)
```

## Description

`tansig` is a transfer function. Transfer functions calculate a layer's output from its net input.

`tansig(N)` takes one input,

`N` — `S` x `Q` matrix of net input (column) vectors

and returns each element of `N` squashed between -1 and 1.

`tansig(code)` return useful information for each `code` string:

`'deriv'` — Name of derivative function

`'name'` — Full name

`'output'` — Output range

`'active'` — Active input range

`tansig` is named after the hyperbolic tangent, which has the same shape. However, `tanh` may be more accurate and is recommended for applications that require the hyperbolic tangent.

## Examples

Here is the code to create a plot of the `tansig` transfer function.

```
n = -5:0.1:5;
a = tansig(n);
plot(n,a)
```

## Network Use

You can create a standard network that uses `tansig` by calling `newff` or `newcf`.

To change a network so a layer uses `tansig`, set `net.layers{i,j}.transferFcn` to 'tansig'.

In either case, call `sim` to simulate the network with `tansig`. See `newff` or `newcf` for simulation examples.

## Algorithm

`tansig(N)` calculates its output according to:

```
n = 2/(1+exp(-2*n))-1
```

This is mathematically equivalent to `tanh(N)`. It differs in that it runs faster than the MATLAB® implementation of `tanh`, but the results can have very small numerical differences. This function is a good trade off for neural networks, where speed is important and the exact shape of the transfer function is not.

## See Also

`sim, dtansig, logsig`

## References

Vogl, T. P., J.K. Mangis, A.K. Rigler, W.T. Zink, and D.L. Alkon, "Accelerating the convergence of the backpropagation method," *Biological Cybernetics*, vol. 59, pp. 257-263, 1988.

## Purpose

Train a neural network

## Syntax

`[net,tr,Y,E,Pf,Af] = train(net,P,T,Pi,Ai,VV,TV)`

## To Get Help

Type `help network/train`

## Description

`train` trains a network `net` according to `net.trainFcn` and `net.trainParam`.

`train(NET,P,T,Pi,Ai,VV,TV)` takes,

- `net` — Neural Network
- `P` — Network inputs
- `T` — Network targets, default = zeros
- `Pi` — Initial input delay conditions, default = zeros
- `Ai` — Initial layer delay conditions, default = zeros
- `VV` — Structure of validation vectors, default = []
- `TV` — Structure of test vectors, default = []

and returns,

- `net` — New network
- `TR` — Training record (`epoch` and `perf`)
- `Y` — Network outputs
- `E` — Network errors.
- `Pf` — Final input delay conditions
- `Af` — Final layer delay conditions

Note that `T` is optional and need only be used for networks that require targets. `Pi` and `Pf` are also optional and need only be used for networks that have input or layer delays.

Optional arguments `VV` and `TV` are described below.

`train`'s signal arguments can have two formats: cell array or matrix.

The cell array format is easiest to describe. It is most convenient for networks with multiple inputs and outputs, and allows sequences of inputs to be presented:

P — Ni x TS cell array, each element P{i,ts} is an Ri x Q matrix
T — Nt x TS cell array, each element P{i,ts} is an Vi x Q matrix
Pi — Ni x ID cell array, each element Pi{i,k} is an Ri x Q matrix
Ai — Nl x LD cell array, each element Ai{i,k} is an Si x Q matrix
Y — NO x TS cell array, each element Y{i,ts} is an Ui x Q matrix
E — Nt x TS cell array, each element P{i,ts} is an Vi x Q matrix
Pf — Ni x ID cell array, each element Pf{i,k} is an Ri x Q matrix
Af — Nl x LD cell array, each element Af{i,k} is an Si x Q matrix

where

Ni = net.numInputs
Nl = net.numLayers
Nt = net.numTargets
ID = net.numInputDelays
LD = net.numLayerDelays
TS = Number of time steps
Q = Batch size
Ri = net.inputs{i}.size
Si = net.layers{i}.size
Vi = net.targets{i}.size

The columns of Pi, Pf, Ai, and Af are ordered from the oldest delay condition to the most recent:

Pi{i,k} = input i at time ts=k ID.
Pf{i,k} = input i at time ts=TS+k ID.
Ai{i,k} = layer output i at time ts=k LD.
Af{i,k} = layer output i at time ts=TS+k LD.

The matrix format can be used if only one time step is to be simulated (TS = 1). It is convenient for networks with only one input and output, but can be used with networks that have more.

Each matrix argument is found by storing the elements of the corresponding cell array argument into a single matrix:

P — (sum of Ri) x Q matrix

T — (sum of Vi) x Q matrix

Pi — (sum of Ri) x (ID*Q) matrix

Ai — (sum of Si) x (LD*Q) matrix

Y — (sum of Ui) x Q matrix

E — (sum of Vi) x Q matrix

Pf — (sum of Ri) x (ID*Q) matrix

Af — (sum of Si) x (LD*Q) matrix

If VV and TV are supplied they should be an empty matrix [] or a structure with the following fields:

VV.P, TV.P — Validation/test inputs

VV.T, TV.T — Validation/test targets, default = zeros

VV.Pi, TV.Pi — Validation/test initial input delay conditions, default = zeros

VV.Ai, TV.Ai — Validation/test layer delay conditions, default = zeros

The validation vectors are used to stop training early if further training on the primary vectors will hurt generalization to the validation vectors. Test vector performance can be used to measure how well the network generalizes beyond primary and validation vectors. If VV.T, VV.Pi, or VV.Ai are set to an empty matrix or cell array, default values will be used. The same is true for TV.T, TV.Pi, TV.Ai.

## Examples

Here input P and targets T define a simple function which we can plot:

```
p = [0 1 2 3 4 5 6 7 8];
t = [0 0.84 0.91 0.14 -0.77 -0.96 -0.28 0.66 0.99];
plot(p,t,'o')
```

Here newff is used to create a two-layer feed-forward network. The network will have an input (ranging from 0 to 8), followed by a layer of 10 tansig neurons, followed by a layer with 1 purelin neuron. trainlm backpropagation is used. The network is also simulated.

```
net = newff([0 8],[10 1],{'tansig' 'purelin'},'trainlm');
```

```
y1 = sim(net,p)
plot(p,t,'o',p,y1,'x')
```

Here the network is trained for up to 50 epochs to a error goal of 0.01, and then resimulated.

```
net.trainParam.epochs = 50;
net.trainParam.goal = 0.01;
net = train(net,p,t);
y2 = sim(net,p)
plot(p,t,'o',p,y1,'x',p,y2,'*')
```

## Algorithm

train calls the function indicated by net.trainFcn, using the training parameter values indicated by net.trainParam.

Typically one epoch of training is defined as a single presentation of all input vectors to the network. The network is then updated according to the results of all those presentations.

Training occurs until a maximum number of epochs occurs, the performance goal is met, or any other stopping condition of the function net.trainFcn occurs.

Some training functions depart from this norm by presenting only one input vector (or sequence) each epoch. An input vector (or sequence) is chosen randomly each epoch from concurrent input vectors (or sequences). newc and newsom return networks that use trainr, a training function that presents each input vector once in random order.

## See Also

sim, init, adapt, revert

## Purpose

Batch training with weight and bias learning rules.

## Syntax

```
[net,TR,Ac,El] = trainb(net,Pd,Tl,Ai,Q,TS,VV,TV)
info = trainb(code)
```

## Description

`trainb` is not called directly. Instead it is called by `train` for networks whose `net.trainFcn` property is set to '`trainb`'.

`trainb` trains a network with weight and bias learning rules with batch updates. The weights and biases are updated at the end of an entire pass through the input data.

`trainb(net,Pd,Tl,Ai,Q,TS,VV,TV)` takes these inputs,

`net` — Neural network

`Pd` — Delayed inputs

`Tl` — Layer targets

`Ai` — Initial input conditions

`Q` — Batch size

`TS` — Time steps

`VV` — Empty matrix [] or structure of validation vectors

`TV` — Empty matrix [] or structure of test vectors

and returns,

`net` — Trained network

`TR` — Training record of various values over each epoch:

- `TR.epoch` — Epoch number
- `TR.perf` — Training performance
- `TR.vperf` — Validation performance
- `TR.tperf` — Test performance

`Ac` — Collective layer outputs for last epoch.

`El` — Layer errors for last epoch

Training occurs according to the `trainb`'s training parameters, shown here with their default values:

| | | |
|---|---|---|
| `net.trainParam.epochs` | `100` | Maximum number of epochs to train |
| `net.trainParam.goal` | `0` | Performance goal |
| `net.trainParam.max_fail` | `5` | Maximum validation failures |
| `net.trainParam.show` | `25` | Epochs between displays (`NaN` for no displays) |
| `net.trainParam.time` | `inf` | Maximum time to train in seconds |

Dimensions for these variables are:

`Pd` — `No x Ni x TS` cell array, each element `Pd{i,j,ts}` is a `Dij x Q` matrix

`Tl` — `Nl x TS` cell array, each element `P{i,ts}` is a `Vi x Q` matrix or []

`Ai` — `Nl x LD` cell array, each element `Ai{i,k}` is an `Si x Q` matrix

where

```
Ni = net.numInputs
Nl = net.numLayers
LD = net.numLayerDelays
Ri = net.inputs{i}.size
Si = net.layers{i}.size
Vi = net.targets{i}.size
Dij = Ri * length(net.inputWeights{i,j}.delays)
```

If `VV` or `TV` is not [], it must be a structure of vectors:

`VV.PD, TV.PD` — Validation/test delayed inputs

`VV.Tl, TV.Tl` — Validation/test layer targets

`VV.Ai, TV.Ai` — Validation/test initial input conditions

`VV.Q,  TV.Q` — Validation/test batch size

`VV.TS, TV.TS` — Validation/test time steps

Validation vectors are used to stop training early if the network performance on the validation vectors fails to improve or remains the same for `max_fail` epochs in a row. Test vectors are used as a further check that the network is generalizing well, but do not have any effect on training.

`trainb(CODE)` returns useful information for each `CODE` string:

`'pnames'` — Names of training parameters

`'pdefaults'` — Default training parameters

## Network Use

You can create a standard network that uses `trainb` by calling `newlin`.

To prepare a custom network to be trained with `trainb`

1. Set `net.trainFcn` to '`trainb`'.

   (This will set `NET.trainParam` to `trainb`'s default parameters.)
2. Set each `NET.inputWeights{i,j}.learnFcn` to a learning function.
3. Set each `NET.layerWeights{i,j}.learnFcn` to a learning function.
4. Set each `NET.biases{i}.learnFcn` to a learning function. (Weight and bias learning parameters will automatically be set to default values for the given learning function.)

To train the network

1. Set `NET.trainParam` properties to desired values.
2. Set weight and bias learning parameters to desired values.
3. Call `train`.

See `newlin` for training examples

## Algorithm

Each weight and bias updates according to its learning function after each epoch (one pass through the entire set of input vectors).

Training stops when any of these conditions are met:

- The maximum number of `epochs` (repetitions) is reached.
- Performance has been minimized to the `goal`.
- The maximum amount of `time` has been exceeded.
- Validation performance has increase more than `max_fail` times since the last time it decreased (when using validation).

## See Also

`newp, newlin, train`

## Purpose

BFGS quasi-Newton backpropagation

## Syntax

```
[net,TR,Ac,El] = trainbfg(net,Pd,Tl,Ai,Q,TS,VV,TV)
info = trainbfg(code)
```

## Description

`trainbfg` is a network training function that updates weight and bias values according to the BFGS quasi-Newton method.

`trainbfg(net,Pd,Tl,Ai,Q,TS,VV,TV)` takes these inputs,

- `net` — Neural network
- `Pd` — Delayed input vectors
- `Tl` — Layer target vectors
- `Ai` — Initial input delay conditions
- `Q` — Batch size
- `TS` — Time steps
- `VV` — Either empty matrix `[]` or structure of validation vectors
- `TV` — Either empty matrix `[]` or structure of test vectors

and returns,

- `net` — Trained network
- `TR` — Training record of various values over each epoch:
  - `TR.epoch` — Epoch number
  - `TR.perf` — Training performance
  - `TR.vperf` — Validation performance
  - `TR.tperf` — Test performance
- `Ac` — Collective layer outputs for last epoch
- `El` — Layer errors for last epoch

Training occurs according to `trainbfg`'s training parameters, shown here with their default values:

| | | |
|---|---|---|
| `net.trainParam.epochs` | `100` | Maximum number of epochs to train |
| `net.trainParam.show` | `25` | Epochs between showing progress |
| `net.trainParam.goal` | `0` | Performance goal |
| `net.trainParam.time` | `inf` | Maximum time to train in seconds |
| `net.trainParam.min_grad` | `1e-6` | Minimum performance gradient |
| `net.trainParam.max_fail` | `5` | Maximum validation failures |
| `net.trainParam.searchFcn`<br>`'srchcha'` | | Name of line search routine to use. |

Parameters related to line search methods (not all used for all methods):

`net.trainParam.scal_tol` `20`

Divide into `delta` to determine tolerance for linear search.

`net.trainParam.alpha` `0.001`

Scale factor, which determines sufficient reduction in `perf`.

`net.trainParam.beta` `0.1`

Scale factor, which determines sufficiently large step size.

`net.trainParam.delta` `0.01`

Initial step size in interval location step.

`net.trainParam.gama` `0.1`

Parameter to avoid small reductions in performance. Usually set to 0.1. (See use in `srch_cha`.)

| | | |
|---|---|---|
| `net.trainParam.low_lim` | `0.1` | Lower limit on change in step size. |
| `net.trainParam.up_lim` | `0.5` | Upper limit on change in step size. |
| `net.trainParam.maxstep` | `100` | Maximum step length. |
| `net.trainParam.minstep` | `1.0e-6` | Minimum step length. |
| `net.trainParam.bmax` | `26` | Maximum step size. |

Dimensions for these variables are:

`Pd` — `No` x `Ni` x `TS` cell array, each element `P{i,j,ts}` is a `Dij` x `Q` matrix

`Tl` — `Nl` x `TS` cell array, each element `P{i,ts}` is a `Vi` x `Q` matrix

`Ai` — `Nl` x `LD` cell array, each element `Ai{i,k}` is an `Si` x `Q` matrix

where

```
Ni = net.numInputs
Nl = net.numLayers
LD = net.numLayerDelays
Ri = net.inputs{i}.size
Si = net.layers{i}.size
Vi = net.targets{i}.size
Dij = Ri * length(net.inputWeights{i,j}.delays)
```

If `VV` is not `[]`, it must be a structure of validation vectors,

`VV.PD` — Validation delayed inputs

`VV.Tl` — Validation layer targets

`VV.Ai` — Validation initial input conditions

`VV.Q`  — Validation batch size

`VV.TS` — Validation time steps

which is used to stop training early if the network performance on the validation vectors fails to improve or remains the same for `max_fail` epochs in a row.

If TV is not [], it must be a structure of validation vectors,

TV.PD — Validation delayed inputs

TV.Tl — Validation layer targets

TV.Ai — Validation initial input conditions

TV.Q — Validation batch size

TV.TS — Validation time steps

which is used to test the generalization capability of the trained network.

trainbfg(code) returns useful information for each code string:

'pnames' — Names of training parameters

'pdefaults' — Default training parameters

## Examples

Here is a problem consisting of inputs P and targets T that we would like to solve with a network.

```
P = [0 1 2 3 4 5];
T = [0 0 0 1 1 1];
```

Here a two-layer feed-forward network is created. The network's input ranges from [0 to 10]. The first layer has two tansig neurons, and the second layer has one logsig neuron. The trainbfg network training function is to be used.

### Create and Test a Network

```
net = newff([0 5],[2 1],{'tansig','logsig'},'trainbfg');
a = sim(net,p)
```

### Train and Retest the Network

```
net.trainParam.epochs = 50;
net.trainParam.show = 10;
net.trainParam.goal = 0.1;
net = train(net,p,t);
a = sim(net,p)
```

See newff, newcf, and newelm for other examples

## Network Use

You can create a standard network that uses trainbfg with newff, newcf, or newelm.

To prepare a custom network to be trained with `trainbfg`:

1. Set `net.trainFcn` to '`trainbfg`'. This will set `net.trainParam` to `trainbfg`'s default parameters.
2. Set `net.trainParam` properties to desired values.

In either case, calling `train` with the resulting network will train the network with `trainbfg`.

## Algorithm

`trainbfg` can train any network as long as its weight, net input, and transfer functions have derivative functions.

Backpropagation is used to calculate derivatives of performance `perf` with respect to the weight and bias variables `X`. Each variable is adjusted according to the following:

```
X = X + a*dX;
```

where `dX` is the search direction. The parameter `a` is selected to minimize the performance along the search direction. The line search function `searchFcn` is used to locate the minimum point. The first search direction is the negative of the gradient of performance. In succeeding iterations the search direction is computed according to the following formula:

```
dX = -H\gX;
```

where `gX` is the gradient and `H` is an approximate Hessian matrix. See page 119 of Gill, Murray, and Wright (see reference below) for a more detailed discussion of the BFGS quasi-Newton method.

Training stops when any of these conditions occur:

- The maximum number of `epochs` (repetitions) is reached.
- The maximum amount of `time` has been exceeded.
- Performance has been minimized to the `goal`.
- The performance gradient falls below `mingrad`.
- Validation performance has increased more than `max_fail` times since the last time it decreased (when using validation).

## See Also

`newff, newcf, traingdm, traingda, traingdx, trainlm, trainrp, traincgf, traincgb, trainscg, traincgp, trainoss.`

## References

Gill, P. E.,W. Murray, and M. H. Wright, *Practical Optimization*, New York: Academic Press, 1981.

## Purpose

Bayesian regularization backpropagation

## Syntax

```
[net,TR,Ac,El] = trainbr(net,Pd,Tl,Ai,Q,TS,VV,TV)
info = trainbr(code)
```

## Description

`trainbr` is a network training function that updates the weight and bias values according to Levenberg-Marquardt optimization. It minimizes a combination of squared errors and weights, and then determines the correct combination so as to produce a network that generalizes well. The process is called Bayesian regularization.

`trainbr(net,Pd,Tl,Ai,Q,TS,VV,TV)` takes these inputs,

`net` — Neural network

`Pd` — Delayed input vectors

`Tl` — Layer target vectors

`Ai` — Initial input delay conditions

`Q` — Batch size

`TS` — Time steps

`VV` — Either empty matrix `[]` or structure of validation vectors

`TV` — Either empty matrix `[]` or structure of test vectors

and returns,

`net` — Trained network

`TR` — Training record of various values over each epoch:

`TR.epoch` — Epoch number

`TR.perf` — Training performance

`TR.vperf` — Validation performance

`TR.tperf` — Test performance

`TR.mu` — Adaptive `mu` value

`Ac` — Collective layer outputs for last epoch.

`El` — Layer errors for last epoch

Training occurs according to the trainlm's training parameters, shown here with their default values:

| | | |
|---|---|---|
| net.trainParam.epochs | 100 | Maximum number of epochs to train |
| net.trainParam.goal | 0 | Performance goal |
| net.trainParam.mu | 0.005 | Marquardt adjustment parameter |
| net.trainParam.mu_dec | 0.1 | Decrease factor for mu |
| net.trainParam.mu_inc | 10 | Increase factor for mu |
| net.trainParam.mu_max | 1e-10 | Maximum value for mu |
| net.trainParam.max_fail | 5 | Maximum validation failures |
| net.trainParam.mem_reduc | 1 | Factor to use for memory/speed trade-off |
| net.trainParam.min_grad | 1e-10 | Minimum performance gradient |
| net.trainParam.show | 25 | Epochs between showing progress |
| net.trainParam.time | inf | Maximum time to train in seconds |

Dimensions for these variables are:

Pd — No x Ni x TS cell array, each element P{i,j,ts} is a Dij x Q matrix

Tl — Nl x TS cell array, each element P{i,ts} is a Vi x Q matrix

Ai — Nl x LD cell array, each element Ai{i,k} is an Si x Q matrix

where

```
Ni = net.numInputs
Nl = net.numLayers
LD = net.numLayerDelays
Ri = net.inputs{i}.size
Si = net.layers{i}.size
Vi = net.targets{i}.size
Dij = Ri * length(net.inputWeights{i,j}.delays)
```

If VV is not [], it must be a structure of validation vectors,

VV.PD — Validation delayed inputs

VV.Tl — Validation layer targets

VV.Ai — Validation initial input conditions

VV.Q — Validation batch size

VV.TS — Validation time steps

which is normally used to stop training early if the network performance on the validation vectors fails to improve or remains the same for max_fail epochs in a row.

If TV is not [], it must be a structure of validation vectors,

TV.PD — Validation delayed inputs

TV.Tl — Validation layer targets

TV.Ai — Validation initial input conditions

TV.Q — Validation batch size

TV.TS — Validation time steps

which is used to test the generalization capability of the trained network.

trainbr(code) returns useful information for each code string:

'pnames' — Names of training parameters

'pdefaults' — Default training parameters

## Examples

Here is a problem consisting of inputs p and targets t that we would like to solve with a network. It involves fitting a noisy sine wave.

```
p = [-1:.05:1];
t = sin(2*pi*p)+0.1*randn(size(p));
```

Here a two-layer feed-forward network is created. The network's input ranges from [-1 to 1]. The first layer has 20 tansig neurons, the second layer has one purelin neuron. The trainbr network training function is to be used. The plot of the resulting network output should show a smooth response, without overfitting.

### Create a Network

```
net=newff([-1 1],[20,1],{'tansig','purelin'},'trainbr');
```

### Train and Test the Network

```
net.trainParam.epochs = 50;
net.trainParam.show = 10;
net = train(net,p,t);
a = sim(net,p)
plot(p,a,p,t,'+')
```

## Network Use

You can create a standard network that uses `trainbr` with `newff`, `newcf`, or `newelm`.

To prepare a custom network to be trained with `trainbr`

1. Set `net.trainFcn` to '`trainlm`'. This will set `net.trainParam` to `trainbr`'s default parameters.
2. Set `net.trainParam` properties to desired values.

In either case, calling `train` with the resulting network will train the network with `trainbr`.

See `newff`, `newcf`, and `newelm` for examples.

## Algorithm

`trainbr` can train any network as long as its weight, net input, and transfer functions have derivative functions.

Bayesian regularization minimizes a linear combination of squared errors and weights. It also modifies the linear combination so that at the end of training the resulting network has good generalization qualities. See MacKay (*Neural Computation*) and Foresee and Hagan (*Proceedings of the International Joint Conference on Neural Networks*) for more detailed discussions of Bayesian regularization.

This Bayesian regularization takes place within the Levenberg-Marquardt algorithm. Backpropagation is used to calculate the Jacobian `jX` of performance `perf` with respect to the weight and bias variables `X`. Each variable is adjusted according to Levenberg-Marquardt,

```
jj = jX * jX
je = jX * E
dX = -(jj+I*mu) \ je
```

where `E` is all errors and `I` is the identity matrix.

The adaptive value mu is increased by mu_inc until the change shown above results in a reduced performance value. The change is then made to the network and mu is decreased by mu_dec.

The parameter mem_reduc indicates how to use memory and speed to calculate the Jacobian jX. If mem_reduc is 1, then trainlm runs the fastest, but can require a lot of memory. Increasing mem_reduc to 2 cuts some of the memory required by a factor of two, but slows trainlm somewhat. Higher values continue to decrease the amount of memory needed and increase the training times.

Training stops when any one of these conditions occurs:

- The maximum number of epochs (repetitions) is reached.
- The maximum amount of time has been exceeded.
- Performance has been minimized to the goal.
- The performance gradient falls below mingrad.
- mu exceeds mu_max.
- Validation performance has increased more than max_fail times since the last time it decreased (when using validation).

## See Also

newff, newcf, traingdm, traingda, traingdx, trainlm, trainrp, traincgf, traincgb, trainscg, traincgp, trainoss

## References

Foresee, F. D., and M. T. Hagan, “Gauss-Newton approximation to Bayesian regularization,” *Proceedings of the 1997 International Joint Conference on Neural Networks*, 1997.

MacKay, D. J. C., “Bayesian interpolation,” *Neural Computation*, vol. 4, no. 3, pp. 415-447, 1992.

# trainc

## Purpose

Cyclical order incremental training with learning functions

## Syntax

```
[net,TR,Ac,El] = trainc(net,Pd,Tl,Ai,Q,TS,VV,TV)
info = trainc(code)
```

## Description

`trainc` is not called directly. Instead it is called by `train` for networks whose `net.trainFcn` property is set to `'trainc'`.

`trainc` trains a network with weight and bias learning rules with incremental updates after each presentation of an input. Inputs are presented in cyclic order.

`trainc(net,Pd,Tl,Ai,Q,TS,VV,TV)` takes these inputs,

- `net` — Neural network
- `Pd` — Delayed inputs
- `Tl` — Layer targets
- `Ai` — Initial input conditions
- `Q` — Batch size
- `TS` — Time steps
- `VV` — Ignored
- `TV` — Ignored

and returns,

- `net` — Trained network
- `TR` — Training record of various values over each epoch:
  - `TR.epoch` — Epoch number
  - `TR.perf` — Training performance
- `Ac` — Collective layer outputs
- `El` — Layer errors

Training occurs according to the trainc's training parameters shown here with their default values:

| | | |
|---|---|---|
| net.trainParam.epochs | 100 | Maximum number of epochs to train |
| net.trainParam.goal | 0 | Performance goal |
| net.trainParam.show | 25 | Epochs between displays (NaN for no displays) |
| net.trainParam.time | inf | Maximum time to train in seconds |

Dimensions for these variables are:

Pd — No x Ni x TS cell array, each element Pd{i,j,ts} is a Dij x Q matrix

Tl — Nl x TS cell array, each element P{i,ts} is a Vi x Q matrix or []

Ai — Nl x LD cell array, each element Ai{i,k} is an Si x Q matrix

where

```
Ni = net.numInputs
Nl = net.numLayers
LD = net.numLayerDelays
Ri = net.inputs{i}.size
Si = net.layers{i}.size
Vi = net.targets{i}.size
Dij = Ri * length(net.inputWeights{i,j}.delays)
```

trainc does not implement validation or test vectors, so arguments VV and TV are ignored.

trainc(code) returns useful information for each code string:

'pnames' — Names of training parameters

'pdefaults' — Default training parameters

## Network Use

You can create a standard network that uses trainc by calling newp.

To prepare a custom network to be trained with trainc

1 Set net.trainFcn to 'trainc'.

(This will set net.trainParam to trainc default parameters.)

2 Set each net.inputWeights{i,j}.learnFcn to a learning function.

3 Set each net.layerWeights{i,j}.learnFcn to a learning function.

4. Set each `net.biases{i}.learnFcn` to a learning function. (Weight and bias learning parameters will automatically be set to default values for the given learning function.)

To train the network

1. Set `net.trainParam` properties to desired values.
2. Set weight and bias learning parameters to desired values.
3. Call `train`.

See `newp` for training examples.

## Algorithm

For each epoch, each vector (or sequence) is presented in order to the network with the weight and bias values updated accordingly after each individual presentation.

Training stops when any of these conditions are met:

- The maximum number of `epochs` (repetitions) is reached.
- Performance has been minimized to the `goal`.
- The maximum amount of `time` has been exceeded.

## See Also

`newp, newlin, train`

## Purpose

Conjugate gradient backpropagation with Powell-Beale restarts

## Syntax

```
[net,TR,Ac,El] = traincgb(net,Pd,Tl,Ai,Q,TS,VV,TV)
info = traincgb(code)
```

## Description

`traincgb` is a network training function that updates weight and bias values according to the conjugate gradient backpropagation with Powell-Beale restarts.

`traincgb(net,Pd,Tl,Ai,Q,TS,VV,TV)` takes these inputs,

- `net` — Neural network
- `Pd` — Delayed input vectors
- `Tl` — Layer target vectors
- `Ai` — Initial input delay conditions
- `Q` — Batch size
- `TS` — Time steps
- `VV` — Either empty matrix `[]` or structure of validation vectors
- `TV` — Either empty matrix `[]` or structure of test vectors

and returns,

- `net` — Trained network
- `TR` — Training record of various values over each epoch:
  - `TR.epoch` — Epoch number
  - `TR.perf` — Training performance
  - `TR.vperf` — Validation performance
  - `TR.tperf` — Test performance
- `Ac` — Collective layer outputs for last epoch
- `El` — Layer errors for last epoch

Training occurs according to the traincgb's training parameters, shown here with their default values:

| | | |
|---|---|---|
| net.trainParam.epochs | 100 | Maximum number of epochs to train |
| net.trainParam.show | 25 | Epochs between showing progress |
| net.trainParam.goal | 0 | Performance goal |
| net.trainParam.time | inf | Maximum time to train in seconds |
| net.trainParam.min_grad | 1e-6 | Minimum performance gradient |
| net.trainParam.max_fail | 5 | Maximum validation failures |
| net.trainParam.searchFcn<br>'srchcha' | | Name of line search routine to use. |

Parameters related to line search methods (not all used for all methods):

| | | |
|---|---|---|
| net.trainParam.scal_tol | 20 | Divide into delta to determine tolerance for linear search. |
| net.trainParam.alpha | 0.001 | Scale factor, which determines sufficient reduction in perf. |
| net.trainParam.beta | 0.1 | Scale factor, which determines sufficiently large step size. |
| net.trainParam.delta | 0.01 | Initial step size in interval location step. |
| net.trainParam.gama | 0.1 | Parameter to avoid small reductions in performance. Usually set to 0.1. (See use in srch_cha.) |
| net.trainParam.low_lim | 0.1 | Lower limit on change in step size. |
| net.trainParam.up_lim | 0.5 | Upper limit on change in step size. |
| net.trainParam.maxstep | 100 | Maximum step length. |
| net.trainParam.minstep | 1.0e-6 | Minimum step length. |
| net.trainParam.bmax | 26 | Maximum step size. |

Dimensions for these variables are

Pd — No x Ni x TS cell array, each element P{i,j,ts} is a Dij x Q matrix.

Tl — Nl x TS cell array, each element P{i,ts} is a Vi x Q matrix.

Ai — Nl x LD cell array, each element Ai{i,k} is an Si x Q matrix.

where

```
Ni = net.numInputs
Nl = net.numLayers
LD = net.numLayerDelays
Ri = net.inputs{i}.size
Si = net.layers{i}.size
Vi = net.targets{i}.size
Dij = Ri * length(net.inputWeights{i,j}.delays)
```

If VV is not [], it must be a structure of validation vectors,

VV.PD — Validation delayed inputs.

VV.Tl — Validation layer targets.

VV.Ai — Validation initial input conditions.

VV.Q — Validation batch size.

VV.TS — Validation time steps.

which is used to stop training early if the network performance on the validation vectors fails to improve or remains the same for max_fail epochs in a row.

If TV is not [], it must be a structure of validation vectors,

TV.PD — Validation delayed inputs.

TV.Tl — Validation layer targets.

TV.Ai — Validation initial input conditions.

TV.Q — Validation batch size.

TV.TS — Validation time steps.

which is used to test the generalization capability of the trained network.

traincgb(code) returns useful information for each code string:

'pnames' — Names of training parameters.

'pdefaults' — Default training parameters.

## Examples

Here is a problem consisting of inputs p and targets t that we would like to solve with a network.

```
p = [0 1 2 3 4 5];
t = [0 0 0 1 1 1];
```

Here a two-layer feed-forward network is created. The network's input ranges from [0 to 10]. The first layer has two tansig neurons, and the second layer has one logsig neuron. The traincgb network training function is to be used.

### Create and Test a Network

```
net = newff([0 5],[2 1],{'tansig','logsig'},'traincgb');
a = sim(net,p)
```

### Train and Retest the Network

```
net.trainParam.epochs = 50;
net.trainParam.show = 10;
net.trainParam.goal = 0.1;
net = train(net,p,t);
a = sim(net,p)
```

See newff, newcf, and newelm for other examples.

## Network Use

You can create a standard network that uses traincgb with newff, newcf, or newelm.

To prepare a custom network to be trained with traincgb

1. Set net.trainFcn to 'traincgb'. This will set net.trainParam to traincgb's default parameters.
2. Set net.trainParam properties to desired values.

In either case, calling train with the resulting network will train the network with traincgb.

## Algorithm

traincgb can train any network as long as its weight, net input, and transfer functions have derivative functions.

Backpropagation is used to calculate derivatives of performance perf with respect to the weight and bias variables X. Each variable is adjusted according to the following:

```
X = X + a*dX;
```

where dX is the search direction. The parameter a is selected to minimize the performance along the search direction. The line search function searchFcn is used to locate the minimum point. The first search direction is the negative of the gradient of performance. In succeeding iterations the search direction is computed from the new gradient and the previous search direction according to the formula:

```
dX = -gX + dX_old*Z;
```

where gX is the gradient. The parameter Z can be computed in several different ways. The Powell-Beale variation of conjugate gradient is distinguished by two features. First, the algorithm uses a test to determine when to reset the search direction to the negative of the gradient. Second, the search direction is computed from the negative gradient, the previous search direction, and the last search direction before the previous reset. See Powell, *Mathematical Programming,* for a more detailed discussion of the algorithm.

Training stops when any of these conditions occur:

- The maximum number of epochs (repetitions) is reached.
- The maximum amount of time has been exceeded.
- Performance has been minimized to the goal.
- The performance gradient falls below mingrad.
- Validation performance has increased more than max_fail times since the last time it decreased (when using validation).

## See Also

newff, newcf, traingdm, traingda, traingdx, trainlm, traincgp, traincgf, traincgb, trainscg, trainoss, trainbfg

## References

Powell, M. J. D.,“Restart procedures for the conjugate gradient method,” *Mathematical Programming*, vol. 12, pp. 241-254, 1977.

## Purpose

Conjugate gradient backpropagation with Fletcher-Reeves updates

## Syntax

```
[net,TR,Ac,El] = traincgf(net,Pd,Tl,Ai,Q,TS,VV,TV)
info = traincgf(code)
```

## Description

`traincgf` is a network training function that updates weight and bias values according to the conjugate gradient backpropagation with Fletcher-Reeves updates.

`traincgf(NET,Pd,Tl,Ai,Q,TS,VV,TV)` takes these inputs,

`net` — Neural network.

`Pd` — Delayed input vectors.

`Tl` — Layer target vectors.

`Ai` — Initial input delay conditions.

`Q` — Batch size.

`TS` — Time steps.

`VV` — Either empty matrix `[]` or structure of validation vectors.

`TV` — Either empty matrix `[]` or structure of test vectors.

and returns,

`net` — Trained network.

`TR` — Training record of various values over each epoch:

`TR.epoch` — Epoch number.

`TR.perf` — Training performance.

`TR.vperf` — Validation performance.

`TR.tperf` — Test performance.

`Ac` — Collective layer outputs for last epoch.

`El` — Layer errors for last epoch.

Training occurs according to the `traincgf`'s training parameters, shown here with their default values:

| | | |
|---|---|---|
| `net.trainParam.epochs` | `100` | Maximum number of epochs to train |
| `net.trainParam.show` | `25` | Epochs between showing progress |
| `net.trainParam.goal` | `0` | Performance goal |
| `net.trainParam.time` | `inf` | Maximum time to train in seconds |
| `net.trainParam.min_grad` | `1e-6` | Minimum performance gradient |
| `net.trainParam.max_fail` | `5` | Maximum validation failures |
| `net.trainParam.searchFcn` `'srchcha'` | | Name of line search routine to use |

Parameters related to line search methods (not all used for all methods):

`net.trainParam.scal_tol` `20`

Divide into `delta` to determine tolerance for linear search.

`net.trainParam.alpha` `0.001`

Scale factor, which determines sufficient reduction in `perf`.

`net.trainParam.beta` `0.1`

Scale factor, which determines sufficiently large step size.

`net.trainParam.delta` `0.01`

Initial step size in interval location step.

`net.trainParam.gama` `0.1`

Parameter to avoid small reductions in performance. Usually set to 0.1. (See use in `srch_cha`.)

| | | |
|---|---|---|
| `net.trainParam.low_lim` | `0.1` | Lower limit on change in step size. |
| `net.trainParam.up_lim` | `0.5` | Upper limit on change in step size. |
| `net.trainParam.maxstep` | `100` | Maximum step length. |
| `net.trainParam.minstep` | `1.0e-6` | Minimum step length. |
| `net.trainParam.bmax` | `26` | Maximum step size. |

Dimensions for these variables are:

Pd — No x Ni x TS cell array, each element P{i,j,ts} is a Dij x Q matrix.

Tl — Nl x TS cell array, each element P{i,ts} is a Vi x Q matrix.

Ai — Nl x LD cell array, each element Ai{i,k} is an Si x Q matrix.

where

```
Ni = net.numInputs
Nl = net.numLayers
LD = net.numLayerDelays
Ri = net.inputs{i}.size
Si = net.layers{i}.size
Vi = net.targets{i}.size
Dij = Ri * length(net.inputWeights{i,j}.delays)
```

If VV is not [], it must be a structure of validation vectors,

VV.PD — Validation delayed inputs.

VV.Tl — Validation layer targets.

VV.Ai — Validation initial input conditions.

VV.Q — Validation batch size.

VV.TS — Validation time steps.

which is used to stop training early if the network performance on the validation vectors fails to improve or remains the same for max_fail epochs in a row.

If TV is not [], it must be a structure of validation vectors,

TV.PD — Validation delayed inputs.

TV.Tl — Validation layer targets.

TV.Ai — Validation initial input conditions.

TV.Q — Validation batch size.

TV.TS — Validation time steps.

which is used to test the generalization capability of the trained network.

traincgf(code) returns useful information for each code string:

'pnames' — Names of training parameters

'pdefaults' — Default training parameters

## Examples

Here is a problem consisting of inputs p and targets t that we would like to solve with a network.

```
p = [0 1 2 3 4 5];
t = [0 0 0 1 1 1];
```

Here a two-layer feed-forward network is created. The network's input ranges from [0 to 10]. The first layer has two tansig neurons, and the second layer has one logsig neuron. The traincgf network training function is to be used.

### Create and Test a Network

```
net = newff([0 5],[2 1],{'tansig','logsig'},'traincgf');
a = sim(net,p)
```

### Train and Retest the Network

```
net.trainParam.epochs = 50;
net.trainParam.show = 10;
net.trainParam.goal = 0.1;
net = train(net,p,t);
a = sim(net,p)
```

See newff, newcf, and newelm for other examples.

## Network Use

You can create a standard network that uses traincgf with newff, newcf, or newelm.

To prepare a custom network to be trained with traincgf

1. Set net.trainFcn to 'traincgf'. This will set net.trainParam to traincgf's default parameters.
2. Set net.trainParam properties to desired values.

In either case, calling train with the resulting network will train the network with traincgf.

## Algorithm

traincgf can train any network as long as its weight, net input, and transfer functions have derivative functions.

Backpropagation is used to calculate derivatives of performance perf with respect to the weight and bias variables X. Each variable is adjusted according to the following:

```
X = X + a*dX;
```

where dX is the search direction. The parameter a is selected to minimize the performance along the search direction. The line search function searchFcn is used to locate the minimum point. The first search direction is the negative of the gradient of performance. In succeeding iterations the search direction is computed from the new gradient and the previous search direction, according to the formula:

```
dX = -gX + dX_old*Z;
```

where gX is the gradient. The parameter Z can be computed in several different ways. For the Fletcher-Reeves variation of conjugate gradient it is computed according to

```
Z=normnew_sqr/norm_sqr;
```

where norm_sqr is the norm square of the previous gradient and normnew_sqr is the norm square of the current gradient. See page 78 of Scales (*Introduction to Non-Linear Optimization*) for a more detailed discussion of the algorithm.

Training stops when any of these conditions occur:

- The maximum number of epochs (repetitions) is reached.
- The maximum amount of time has been exceeded.
- Performance has been minimized to the goal.
- The performance gradient falls below mingrad.
- Validation performance has increased more than max_fail times since the last time it decreased (when using validation).

## See Also

newff, newcf, traingdm, traingda, traingdx, trainlm, traincgp, traincgb, trainscg, traincgp, trainoss, trainbfg

## References

Scales, L. E., *Introduction to Non-Linear Optimization*, New York: Springer-Verlag, 1985.

# traincgp

## Purpose

Conjugate gradient backpropagation with Polak-Ribiere updates

## Syntax

```
[net,TR,Ac,El] = traincgp(net,Pd,Tl,Ai,Q,TS,VV,TV)
info = traincgp(code)
```

## Description

`traincgp` is a network training function that updates weight and bias values according to the conjugate gradient backpropagation with Polak-Ribiere updates.

`traincgp(net,Pd,Tl,Ai,Q,TS,VV,TV)` takes these inputs,

- `net` — Neural network.
- `Pd` — Delayed input vectors.
- `Tl` — Layer target vectors.
- `Ai` — Initial input delay conditions.
- `Q` — Batch size.
- `TS` — Time steps.
- `VV` — Either empty matrix `[]` or structure of validation vectors.
- `TV` — Either empty matrix `[]` or structure of test vectors.

and returns,

- `net` — Trained network.
- `TR` — Training record of various values over each epoch:
  - `TR.epoch` — Epoch number.
  - `TR.perf` — Training performance.
  - `TR.vperf` — Validation performance.
  - `TR.tperf` — Test performance.
- `Ac` — Collective layer outputs for last epoch.
- `El` — Layer errors for last epoch.

Training occurs according to the traincgp's training parameters shown here with their default values:

| | | |
|---|---|---|
| net.trainParam.epochs | 100 | Maximum number of epochs to train |
| net.trainParam.show | 25 | Epochs between showing progress |
| net.trainParam.goal | 0 | Performance goal |
| net.trainParam.time | inf | Maximum time to train in seconds |
| net.trainParam.min_grad | 1e-6 | Minimum performance gradient |
| net.trainParam.max_fail | 5 | Maximum validation failures |
| net.trainParam.searchFcn<br>'srchcha' | | Name of line search routine to use |

Parameters related to line search methods (not all used for all methods):

net.trainParam.scal_tol 20

Divide into delta to determine tolerance for linear search.

net.trainParam.alpha 0.001

Scale factor which determines sufficient reduction in perf.

net.trainParam.beta 0.1

Scale factor which determines sufficiently large step size.

net.trainParam.delta 0.01

Initial step size in interval location step.

net.trainParam.gama 0.1

Parameter to avoid small reductions in performance. Usually set to 0.1. (See use in srch_cha.)

| | | |
|---|---|---|
| net.trainParam.low_lim | 0.1 | Lower limit on change in step size. |
| net.trainParam.up_lim | 0.5 | Upper limit on change in step size. |
| net.trainParam.maxstep | 100 | Maximum step length. |
| net.trainParam.minstep | 1.0e-6 | Minimum step length. |
| net.trainParam.bmax | 26 | Maximum step size. |

Dimensions for these variables are:

Pd — No x Ni x TS cell array, each element P{i,j,ts} is a Dij x Q matrix.

Tl — Nl x TS cell array, each element P{i,ts} is a Vi x Q matrix.

Ai — Nl x LD cell array, each element Ai{i,k} is an Si x Q matrix.

where

```
Ni = net.numInputs
Nl = net.numLayers
LD = net.numLayerDelays
Ri = net.inputs{i}.size
Si = net.layers{i}.size
Vi = net.targets{i}.size
Dij = Ri * length(net.inputWeights{i,j}.delays)
```

If VV is not [], it must be a structure of validation vectors,

VV.PD — Validation delayed inputs.

VV.Tl — Validation layer targets.

VV.Ai — Validation initial input conditions.

VV.Q — Validation batch size.

VV.TS — Validation time steps.

which is used to stop training early if the network performance on the validation vectors fails to improve or remains the same for max_fail epochs in a row.

If TV is not [], it must be a structure of validation vectors,

TV.PD — Validation delayed inputs.

TV.Tl — Validation layer targets.

TV.Ai — Validation initial input conditions.

TV.Q — Validation batch size.

TV.TS — Validation time steps.

which is used to test the generalization capability of the trained network.

traincgp(code) returns useful information for each code string:

'pnames' — Names of training parameters

'pdefaults' — Default training parameters

## Examples

Here is a problem consisting of inputs p and targets t that we would like to solve with a network.

```
p = [0 1 2 3 4 5];
t = [0 0 0 1 1 1];
```

Here a two-layer feed-forward network is created. The network's input ranges from [0 to 10]. The first layer has two tansig neurons, and the second layer has one logsig neuron. The traincgp network training function is to be used.

### Create and Test a Network

```
net = newff([0 5],[2 1],{'tansig','logsig'},'traincgp');
a = sim(net,p)
```

### Train and Retest the Network

```
net.trainParam.epochs = 50;
net.trainParam.show = 10;
net.trainParam.goal = 0.1;
net = train(net,p,t);
a = sim(net,p)
```

See newff, newcf, and newelm for other examples.

## Network Use

You can create a standard network that uses traincgp with newff, newcf, or newelm.

To prepare a custom network to be trained with traincgp

1. Set net.trainFcn to 'traincgp'. This will set net.trainParam to traincgp's default parameters.
2. Set net.trainParam properties to desired values.

In either case, calling train with the resulting network will train the network with traincgp.

## Algorithm

traincgp can train any network as long as its weight, net input, and transfer functions have derivative functions.

Backpropagation is used to calculate derivatives of performance perf with respect to the weight and bias variables X. Each variable is adjusted according to the following:

```
X = X + a*dX;
```

where dX is the search direction. The parameter a is selected to minimize the performance along the search direction. The line search function searchFcn is used to locate the minimum point. The first search direction is the negative of the gradient of performance. In succeeding iterations the search direction is computed from the new gradient and the previous search direction according to the formula:

```
dX = -gX + dX_old*Z;
```

where gX is the gradient. The parameter Z can be computed in several different ways. For the Polak-Ribiere variation of conjugate gradient it is computed according to

```
Z = ((gX - gX_old)'*gX)/norm_sqr;
```

where norm_sqr is the norm square of the previous gradient and gX_old is the gradient on the previous iteration. See page 78 of Scales (*Introduction to Non-Linear Optimization*) for a more detailed discussion of the algorithm.

Training stops when any of these conditions occur:

- The maximum number of epochs (repetitions) is reached.
- The maximum amount of time has been exceeded.
- Performance has been minimized to the goal.
- The performance gradient falls below mingrad.
- Validation performance has increased more than max_fail times since the last time it decreased (when using validation).

## See Also

newff, newcf, traingdm, traingda, traingdx, trainlm, trainrp, traincgf, traincgb, trainscg, trainoss, trainbfg

## References

Scales, L. E., *Introduction to Non-Linear Optimization*, New York: Springer-Verlag, 1985.

# traingd

## Purpose

Gradient descent backpropagation

## Syntax

```
[net,TR,Ac,El] = traingd(net,Pd,Tl,Ai,Q,TS,VV,TV)
info = traingd(code)
```

## Description

`traingd` is a network training function that updates weight and bias values according to gradient descent.

`traingd(net,Pd,Tl,Ai,Q,TS,VV)` takes these inputs,

`net` — Neural network.

`Pd` — Delayed input vectors.

`Tl` — Layer target vectors.

`Ai` — Initial input delay conditions.

`Q` — Batch size.

`TS` — Time steps.

`VV` — Either an empty matrix `[]` or a structure of validation vectors.

`TV` — Empty matrix [] or structure of test vectors.

and returns,

`net` — Trained network.

`TR` — Training record of various values over each epoch:

`TR.epoch` — Epoch number.

`TR.perf` — Training performance.

`TR.vperf` — Validation performance.

`TR.tperf` — Test performance.

`Ac` — Collective layer outputs for last epoch.

`El` — Layer errors for last epoch.

Training occurs according to the traingd's training parameters shown here with their default values:

| | | |
|---|---|---|
| net.trainParam.epochs | 10 | Maximum number of epochs to train |
| net.trainParam.goal | 0 | Performance goal |
| net.trainParam.lr | 0.01 | Learning rate |
| net.trainParam.max_fail | 5 | Maximum validation failures |
| net.trainParam.min_grad | 1e-10 | Minimum performance gradient |
| net.trainParam.show | 25 | Epochs between showing progress |
| net.trainParam.time | inf | Maximum time to train in seconds |

Dimensions for these variables are

Pd — No x Ni x TS cell array, each element P{i,j,ts} is a Dij x Q matrix.

Tl — Nl x TS cell array, each element P{i,ts} is an Vi x Q matrix.

Ai — Nl x LD cell array, each element Ai{i,k} is an Si x Q matrix.

where

```
Ni = net.numInputs
Nl = net.numLayers
LD = net.numLayerDelays
Ri = net.inputs{i}.size
Si = net.layers{i}.size
Vi = net.targets{i}.size
Dij = Ri * length(net.inputWeights{i,j}.delays)
```

If VV or TV is not [], it must be a structure of validation vectors,

VV.PD, TV.PD — Validation/test delayed inputs.

VV.Tl, TV.Tl — Validation/test layer targets.

VV.Ai, TV.Ai — Validation/test initial input conditions.

VV.Q, TV.Q — Validation/test batch size.

VV.TS, TV.TS — Validation/test time steps.

Validation vectors are used to stop training early if the network performance on the validation vectors fails to improve or remains the same for max_fail epochs in a row. Test vectors are used as a further check that the network is generalizing well, but do not have any effect on training.

traingd(code) returns useful information for each code string:

'pnames'    Names of training parameters.

'pdefaults'    Default training parameters.

## Network Use

You can create a standard network that uses traingd with newff, newcf, or newelm.

To prepare a custom network to be trained with traingd:

1. Set net.trainFcn to 'traingd'. This will set net.trainParam to traingd's default parameters.
2. Set net.trainParam properties to desired values.

In either case, calling train with the resulting network will train the network with traingd.

See newff, newcf, and newelm for examples.

## Algorithm

traingd can train any network as long as its weight, net input, and transfer functions have derivative functions.

Backpropagation is used to calculate derivatives of performance perf with respect to the weight and bias variables X. Each variable is adjusted according to gradient descent:

```
dX = lr * dperf/dX
```

Training stops when any of these conditions occur:

- The maximum number of epochs (repetitions) is reached.
- The maximum amount of time has been exceeded.
- Performance has been minimized to the goal.
- The performance gradient falls below mingrad.
- Validation performance has increased more than max_fail times since the last time it decreased (when using validation).

## See Also

newff, newcf, traingdm, traingda, traingdx, trainlm

## Purpose

Gradient descent with adaptive learning rate backpropagation

## Syntax

```
[net,TR,Ac,El] = traingda(net,Pd,Tl,Ai,Q,TS,VV,TV)
info = traingda(code)
```

## Description

`traingda` is a network training function that updates weight and bias values according to gradient descent with adaptive learning rate.

`traingda(net,Pd,Tl,Ai,Q,TS,VV)` takes these inputs,

`net` — Neural network.<br>
`Pd` — Delayed input vectors.<br>
`Tl` — Layer target vectors.<br>
`Ai` — Initial input delay conditions.<br>
`Q` — Batch size.<br>
`TS` — Time steps.<br>
`VV` — Either empty matrix `[]` or structure of validation vectors.<br>
`TV` — Empty matrix [] or structure of test vectors.

and returns,

`net` — Trained network.

`TR` — Training record of various values over each epoch:

- `TR.epoch` — Epoch number.
- `TR.perf` — Training performance.
- `TR.vperf` — Validation performance.
- `TR.tperf` — Test performance.
- `TR.lr` — Adaptive learning rate.

`Ac` — Collective layer outputs for last epoch.

`El` — Layer errors for last epoch.

Training occurs according to the `traingda`'s training parameters, shown here with their default values:

| | | |
|---|---|---|
| `net.trainParam.epochs` | `10` | Maximum number of epochs to train |
| `net.trainParam.goal` | `0` | Performance goal |
| `net.trainParam.lr` | `0.01` | Learning rate |
| `net.trainParam.lr_inc` | `1.05` | Ratio to increase learning rate |
| `net.trainParam.lr_dec` | `0.7` | Ratio to decrease learning rate |
| `net.trainParam.max_fail` | `5` | Maximum validation failures |
| `net.trainParam.max_perf_inc` | `1.04` | Maximum performance increase |
| `net.trainParam.min_grad` | `1e-10` | Minimum performance gradient |
| `net.trainParam.show` | `25` | Epochs between showing progress |
| `net.trainParam.time` | `inf` | Maximum time to train in seconds |

Dimensions for these variables are

`Pd` — `No` x `Ni` x `TS` cell array, each element `P{i,j,ts}` is a `Dij` x `Q` matrix.

`Tl` — `Nl` x `TS` cell array, each element `P{i,ts}` is a `Vi` x `Q` matrix.

`Ai` — `Nl` x `LD` cell array, each element `Ai{i,k}` is an `Si` x `Q` matrix.

where

```
Ni = net.numInputs
Nl = net.numLayers
LD = net.numLayerDelays
Ri = net.inputs{i}.size
Si = net.layers{i}.size
Vi = net.targets{i}.size
Dij = Ri * length(net.inputWeights{i,j}.delays)
```

If `VV` or `TV` is not `[]`, it must be a structure of validation vectors,

`VV.PD, TV.PD` — Validation/test delayed inputs

`VV.Tl, TV.Tl` — Validation/test layer targets

`VV.Ai, TV.Ai` — Validation/test initial input conditions

`VV.Q,  TV.Q` — Validation/test batch size

`VV.TS, TV.TS` — Validation/test time steps

Validation vectors are used to stop training early if the network performance on the validation vectors fails to improve or remains the same for max_fail epochs in a row. Test vectors are used as a further check that the network is generalizing well, but do not have any effect on training.

traingda(code) returns useful information for each code string:

'pnames' — Names of training parameters

'pdefaults' — Default training parameters

## Network Use

You can create a standard network that uses traingda with newff, newcf, or newelm.

To prepare a custom network to be trained with traingda

1. Set net.trainFcn to 'traingda'. This will set net.trainParam to traingda's default parameters.
2. Set net.trainParam properties to desired values.

In either case, calling train with the resulting network will train the network with traingda.

See newff, newcf, and newelm for examples.

## Algorithm

traingda can train any network as long as its weight, net input, and transfer functions have derivative functions.

Backpropagation is used to calculate derivatives of performance dperf with respect to the weight and bias variables X. Each variable is adjusted according to gradient descent:

```
dX = lr*dperf/dX
```

At each epoch, if performance decreases toward the goal, then the learning rate is increased by the factor lr_inc. If performance increases by more than the factor max_perf_inc, the learning rate is adjusted by the factor lr_dec and the change, which increased the performance, is not made.

Training stops when any of these conditions occur:

- The maximum number of epochs (repetitions) is reached.
- The maximum amount of time has been exceeded.

- Performance has been minimized to the `goal`.
- The performance gradient falls below `mingrad`.
- Validation performance has increased more than `max_fail` times since the last time it decreased (when using validation).

### See Also

`newff`, `newcf`, `traingd`, `traingdm`, `traingdx`, `trainlm`

## Purpose

Gradient descent with momentum backpropagation

## Syntax

```
[net,TR,Ac,El] = traingdm(net,Pd,Tl,Ai,Q,TS,VV,TV)
info = traingdm(code)
```

## Description

`traingdm` is a network training function that updates weight and bias values according to gradient descent with momentum.

`traingdm(net,Pd,Tl,Ai,Q,TS,VV)` takes these inputs,

`net` — Neural network
`Pd` — Delayed input vectors
`Tl` — Layer target vectors
`Ai` — Initial input delay conditions
`Q` — Batch size
`TS` — Time steps
`VV` — Either empty matrix `[]` or structure of validation vectors
`TV` — Empty matrix `[]` or structure of test vectors

and returns,

`net` — Trained network
`TR` — Training record of various values over each epoch:
  `TR.epoch` — Epoch number
  `TR.perf` — Training performance
  `TR.vperf` — Validation performance
  `TR.tperf` — Test performance
`Ac` — Collective layer outputs for last epoch
`El` — Layer errors for last epoch

Training occurs according to the traingdm's training parameters shown here with their default values:

| | | |
|---|---|---|
| net.trainParam.epochs | 10 | Maximum number of epochs to train |
| net.trainParam.goal | 0 | Performance goal |
| net.trainParam.lr | 0.01 | Learning rate |
| net.trainParam.max_fail | 5 | Maximum validation failures |
| net.trainParam.mc | 0.9 | Momentum constant. |
| net.trainParam.min_grad | 1e-10 | Minimum performance gradient |
| net.trainParam.show | 25 | Epochs between showing progress |
| net.trainParam.time | inf | Maximum time to train in seconds |

Dimensions for these variables are

Pd — No x Ni x TS cell array, each element P{i,j,ts} is a Dij x Q matrix.

Tl — Nl x TS cell array, each element P{i,ts} is a Vi x Q matrix.

Ai — Nl x LD cell array, each element Ai{i,k} is an Si x Q matrix.

where

```
Ni = net.numInputs
Nl = net.numLayers
LD = net.numLayerDelays
Ri = net.inputs{i}.size
Si = net.layers{i}.size
Vi = net.targets{i}.size
Dij = Ri * length(net.inputWeights{i,j}.delays)
```

If VV or TV is not [], it must be a structure of validation vectors,

VV.PD, TV.PD — Validation/test delayed inputs

VV.Tl, TV.Tl — Validation/test layer targets

VV.Ai, TV.Ai — Validation/test initial input conditions

VV.Q, TV.Q — Validation/test batch size

VV.TS, TV.TS — Validation/test time steps

Validation vectors are used to stop training early if the network performance on the validation vectors fails to improve or remains the same for max_fail epochs in a row. Test vectors are used as a further check that the network is generalizing well, but do not have any effect on training.

`traingdm(code)` returns useful information for each `code` string:

`'pnames'` — Names of training parameters

`'pdefaults'` — Default training parameters

## Network Use

You can create a standard network that uses `traingdm` with `newff`, `newcf`, or `newelm`.

To prepare a custom network to be trained with `traingdm`

1. Set `net.trainFcn` to `'traingdm'`. This will set `net.trainParam` to `traingdm`'s default parameters.
2. Set `net.trainParam` properties to desired values.

In either case, calling `train` with the resulting network will train the network with `traingdm`.

See `newff`, `newcf`, and `newelm` for examples.

## Algorithm

`traingdm` can train any network as long as its weight, net input, and transfer functions have derivative functions.

Backpropagation is used to calculate derivatives of performance `perf` with respect to the weight and bias variables `X`. Each variable is adjusted according to gradient descent with momentum,

```
dX = mc*dXprev + lr*(1-mc)*dperf/dX
```

where `dXprev` is the previous change to the weight or bias.

Training stops when any of these conditions occur:

- The maximum number of `epochs` (repetitions) is reached.
- The maximum amount of `time` has been exceeded.
- Performance has been minimized to the `goal`.
- The performance gradient falls below `mingrad`.
- Validation performance has increase more than `max_fail` times since the last time it decreased (when using validation).

## See Also

`newff, newcf, traingd, traingda, traingdx, trainlm`

# traingdx

## Purpose

Gradient descent with momentum and adaptive learning rate backpropagation

## Syntax

```
[net,TR,Ac,El] = traingdx(net,Pd,Tl,Ai,Q,TS,VV,TV)
info = traingdx(code)
```

## Description

`traingdx` is a network training function that updates weight and bias values according to gradient descent momentum and an adaptive learning rate.

`traingdx(net,Pd,Tl,Ai,Q,TS,VV)` takes these inputs,

`net` — Neural network.

`Pd` — Delayed input vectors.

`Tl` — Layer target vectors.

`Ai` — Initial input delay conditions.

`Q` — Batch size.

`TS` —Time steps.

`VV` — Either empty matrix `[]` or structure of validation vectors.

`TV` — Empty matrix `[]` or structure of test vectors.

and returns,

`net` — Trained network.

`TR` — Training record of various values over each epoch:

`TR.epoch` — Epoch number.

`TR.perf` — Training performance.

`TR.vperf` — Validation performance.

`TR.tperf` — Test performance.

`TR.lr` — Adaptive learning rate.

`Ac` — Collective layer outputs for last epoch.

`El` — Layer errors for last epoch.

Training occurs according to the traingdx's training parameters shown here with their default values:

| | | |
|---|---|---|
| net.trainParam.epochs | 10 | Maximum number of epochs to train |
| net.trainParam.goal | 0 | Performance goal |
| net.trainParam.lr | 0.01 | Learning rate |
| net.trainParam.lr_inc | 1.05 | Ratio to increase learning rate |
| net.trainParam.lr_dec | 0.7 | Ratio to decrease learning rate |
| net.trainParam.max_fail | 5 | Maximum validation failures |
| net.trainParam.max_perf_inc | 1.04 | Maximum performance increase |
| net.trainParam.mc | 0.9 | Momentum constant. |
| net.trainParam.min_grad | 1e-10 | Minimum performance gradient |
| net.trainParam.show | 25 | Epochs between showing progress |
| net.trainParam.time | inf | Maximum time to train in seconds |

Dimensions for these variables are

Pd — No x Ni x TS cell array, each element P{i,j,ts} is a Dij x Q matrix

Tl — Nl x TS cell array, each element P{i,ts} is a Vi x Q matrix

Ai — Nl x LD cell array, each element Ai{i,k} is an Si x Q matrix

where

```
Ni = net.numInputs
Nl = net.numLayers
LD = net.numLayerDelays
Ri = net.inputs{i}.size
Si = net.layers{i}.size
Vi = net.targets{i}.size
Dij = Ri * length(net.inputWeights{i,j}.delays)
```

If VV or TV is not [], it must be a structure of validation vectors,

VV.PD, TV.PD — Validation/test delayed inputs

VV.Tl, TV.Tl — Validation/test layer targets

VV.Ai, TV.Ai — Validation/test initial input conditions

VV.Q, TV.Q — Validation/test batch size

VV.TS, TV.TS — Validation/test time steps

Validation vectors are used to stop training early if the network performance on the validation vectors fails to improve or remains the same for max_fail epochs in a row. Test vectors are used as a further check that the network is generalizing well, but do not have any effect on training.

traingdx(code) return useful information for each code string:

'pnames' — Names of training parameters

'pdefaults' — Default training parameters

## Network Use

You can create a standard network that uses traingdx with newff, newcf, or newelm.

To prepare a custom network to be trained with traingdx

1. Set net.trainFcn to 'traingdx'. This will set net.trainParam to traingdx's default parameters.
2. Set net.trainParam properties to desired values.

In either case, calling train with the resulting network will train the network with traingdx.

See newff, newcf, and newelm for examples.

## Algorithm

traingdx can train any network as long as its weight, net input, and transfer functions have derivative functions.

Backpropagation is used to calculate derivatives of performance perf with respect to the weight and bias variables X. Each variable is adjusted according to gradient descent with momentum,

```
dX = mc*dXprev + lr*mc*dperf/dX
```

where dXprev is the previous change to the weight or bias.

For each epoch, if performance decreases toward the goal, then the learning rate is increased by the factor lr_inc. If performance increases by more than the factor max_perf_inc, the learning rate is adjusted by the factor lr_dec and the change, which increased the performance, is not made.

Training stops when any of these conditions occur:

- The maximum number of epochs (repetitions) is reached.

- The maximum amount of `time` has been exceeded.
- Performance has been minimized to the `goal`.
- The performance gradient falls below `mingrad`.
- Validation performance has increase more than `max_fail` times since the last time it decreased (when using validation).

### See Also

`newff, newcf, traingd, traingdm, traingda, trainlm`

### Purpose

Levenberg-Marquardt backpropagation

### Syntax

```
[net,TR] = trainlm(net,Pd,Tl,Ai,Q,TS,VV,TV)
info = trainlm(code)
```

### Description

`trainlm` is a network training function that updates weight and bias values according to Levenberg-Marquardt optimization.

`trainlm(net,Pd,Tl,Ai,Q,TS,VV,TV)` takes these inputs,

`net` — Neural network.

`Pd` — Delayed input vectors.

`Tl` — Layer target vectors.

`Ai` — Initial input delay conditions.

`Q` — Batch size.

`TS` — Time steps.

`VV` — Either empty matrix `[]` or structure of validation vectors.

`TV` — Either empty matrix [] or structure of validation vectors.

and returns,

`net` — Trained network.

`TR` — Training record of various values over each epoch:

`TR.epoch` — Epoch number.

`TR.perf` — Training performance.

`TR.vperf` — Validation performance.

`TR.tperf` — Test performance.

`TR.mu` — Adaptive `mu` value.

Training occurs according to the `trainlm`'s training parameters shown here with their default values:

| | | |
|---|---|---|
| `net.trainParam.epochs` | `100` | Maximum number of epochs to train |
| `net.trainParam.goal` | `0` | Performance goal |
| `net.trainParam.max_fail` | `5` | Maximum validation failures |
| `net.trainParam.mem_reduc` | `1` | Factor to use for memory/speed tradeoff |
| `net.trainParam.min_grad` | `1e-10` | Minimum performance gradient |
| `net.trainParam.mu` | `0.001` | Initial Mu |
| `net.trainParam.mu_dec` | `0.1` | Mu decrease factor |
| `net.trainParam.mu_inc` | `10` | Mu increase factor |
| `net.trainParam.mu_max` | `1e10` | Maximum Mu |
| `net.trainParam.show` | `25` | Epochs between showing progress |
| `net.trainParam.time` | `inf` | Maximum time to train in seconds |

Dimensions for these variables are

`Pd` — `No` x `Ni` x `TS` cell array, each element `P{i,j,ts}` is a `Dij` x `Q` matrix.

`Tl` — `Nl` x `TS` cell array, each element `P{i,ts}` is a `Vi` x `Q` matrix.

`Ai` — `Nl` x `LD` cell array, each element `Ai{i,k}` is an `Si` x `Q` matrix.

where

```
Ni = net.numInputs
Nl = net.numLayers
LD = net.numLayerDelays
Ri = net.inputs{i}.size
Si = net.layers{i}.size
Vi = net.targets{i}.size
Dij = Ri * length(net.inputWeights{i,j}.delays)
```

If VV or TV is not [], it must be a structure of vectors,

VV.PD, TV.PD — Validation/test delayed inputs
VV.Tl, TV.Tl — Validation/test layer targets
VV.Ai, TV.Ai — Validation/test initial input conditions
VV.Q, TV.Q — Validation/test batch size
VV.TS, TV.TS — Validation/test time steps

Validation vectors are used to stop training early if the network performance on the validation vectors fails to improve or remains the same for max_fail epochs in a row. Test vectors are used as a further check that the network is generalizing well, but do not have any effect on training.

trainlm(code) returns useful information for each code string:

'pnames' — Names of training parameters
'pdefaults' — Default training parameters

## Network Use

You can create a standard network that uses trainlm with newff, newcf, or newelm.

To prepare a custom network to be trained with trainlm

1. Set net.trainFcn to 'trainlm'. This will set net.trainParam to trainlm's default parameters.
2. Set net.trainParam properties to desired values.

In either case, calling train with the resulting network will train the network with trainlm.

See newff, newcf, and newelm for examples.

## Algorithm

trainlm can train any network as long as its weight, net input, and transfer functions have derivative functions.

Backpropagation is used to calculate the Jacobian jX of performance perf with respect to the weight and bias variables X. Each variable is adjusted according to Levenberg-Marquardt,

```
jj = jX * jX
je = jX * E
dX = -(jj+I*mu) \ je
```

where E is all errors and I is the identity matrix.

The adaptive value mu is increased by mu_inc until the change above results in a reduced performance value. The change is then made to the network and mu is decreased by mu_dec.

The parameter mem_reduc indicates how to use memory and speed to calculate the Jacobian jX. If mem_reduc is 1, then trainlm runs the fastest, but can require a lot of memory. Increasing mem_reduc to 2 cuts some of the memory required by a factor of two, but slows trainlm somewhat. Higher values continue to decrease the amount of memory needed and increase training times.

Training stops when any of these conditions occur:

- The maximum number of epochs (repetitions) is reached.
- The maximum amount of time has been exceeded.
- Performance has been minimized to the goal.
- The performance gradient falls below mingrad.
- mu exceeds mu_max.
- Validation performance has increased more than max_fail times since the last time it decreased (when using validation).

## See Also

newff, newcf, traingd, traingdm, traingda, traingdx

# trainoss

## Purpose

One step secant backpropagation

## Syntax

```
[net,TR,Ac,El] = trainoss(net,Pd,Tl,Ai,Q,TS,VV,TV)
info = trainoss(code)
```

## Description

`trainoss` is a network training function that updates weight and bias values according to the one step secant method.

`trainoss(net,Pd,Tl,Ai,Q,TS,VV,TV)` takes these inputs,

`net` — Neural network.

`Pd` — Delayed input vectors.

`Tl` — Layer target vectors.

`Ai` — Initial input delay conditions.

`Q` — Batch size.

`TS` — Time steps.

`VV` — Either empty matrix `[]` or structure of validation vectors.

`TV` — Either empty matrix `[]` or structure of test vectors.

`TV` — Either empty matrix `[]` or structure of test vectors.

and returns,

`net` — Trained network.

`TR` — Training record of various values over each epoch:

`TR.epoch` — Epoch number.

`TR.perf` — Training performance.

`TR.vperf` — Validation performance.

`TR.tperf` — Test performance.

`Ac` — Collective layer outputs for last epoch.

`El` — Layer errors for last epoch.

Training occurs according to the trainoss's training parameters, shown here with their default values:

| | | |
|---|---|---|
| net.trainParam.epochs | 100 | Maximum number of epochs to train |
| net.trainParam.show | 25 | Epochs between showing progress |
| net.trainParam.goal | 0 | Performance goal |
| net.trainParam.time | inf | Maximum time to train in seconds |
| net.trainParam.min_grad | 1e-6 | Minimum performance gradient |
| net.trainParam.max_fail | 5 | Maximum validation failures |
| net.trainParam.searchFcn | 'srchcha' | Name of line search routine to use |

Parameters related to line search methods (not all used for all methods):

net.trainParam.scal_tol 20

Divide into delta to determine tolerance for linear search.

net.trainParam.alpha 0.001

Scale factor, which determines sufficient reduction in perf.

net.trainParam.beta 0.1

Scale factor, which determines sufficiently large step size.

net.trainParam.delta 0.01

Initial step size in interval location step.

net.trainParam.gama 0.1

Parameter to avoid small reductions in performance. Usually set to 0.1. (See use in srch_cha.)

| | | |
|---|---|---|
| net.trainParam.low_lim | 0.1 | Lower limit on change in step size. |
| net.trainParam.up_lim | 0.5 | Upper limit on change in step size. |
| net.trainParam.maxstep | 100 | Maximum step length. |
| net.trainParam.minstep | 1.0e-6 | Minimum step length. |
| net.trainParam.bmax | 26 | Maximum step size. |

Dimensions for these variables are

Pd — No x Ni x TS cell array, each element P{i,j,ts} is a Dij x Q matrix.

Tl — Nl x TS cell array, each element P{i,ts} is a Vi x Q matrix.

Ai — Nl x LD cell array, each element Ai{i,k} is an Si x Q matrix.

where

```
Ni = net.numInputs
Nl = net.numLayers
LD = net.numLayerDelays
Ri = net.inputs{i}.size
Si = net.layers{i}.size
Vi = net.targets{i}.size
Dij = Ri * length(net.inputWeights{i,j}.delays)
```

If VV is not [], it must be a structure of validation vectors,

VV.PD — Validation delayed inputs.

VV.Tl — Validation layer targets.

VV.Ai — Validation initial input conditions.

VV.Q — Validation batch size.

VV.TS — Validation time steps.

which is used to stop training early if the network performance on the validation vectors fails to improve or remains the same for max_fail epochs in a row.

If TV is not [], it must be a structure of validation vectors,

TV.PD — Validation delayed inputs.

TV.Tl — Validation layer targets.

TV.Ai — Validation initial input conditions.

TV.Q — Validation batch size.

TV.TS — Validation time steps.

which is used to test the generalization capability of the trained network.

trainoss(code) returns useful information for each code string:

'pnames' — Names of training parameters

'pdefaults' — Default training parameters

## Examples

Here is a problem consisting of inputs p and targets t that we would like to solve with a network.

```
p = [0 1 2 3 4 5];
t = [0 0 0 1 1 1];
```

Here a two-layer feed-forward network is created. The network's input ranges from [0 to 10]. The first layer has two tansig neurons, and the second layer has one logsig neuron. The trainoss network training function is to be used.

### Create and Test a Network

```
net = newff([0 5],[2 1],{'tansig','logsig'},'trainoss');
a = sim(net,p)
```

### Train and Retest the Network

```
net.trainParam.epochs = 50;
net.trainParam.show = 10;
net.trainParam.goal = 0.1;
net = train(net,p,t);
a = sim(net,p)
```

See newff, newcf, and newelm for other examples.

## Network Use

You can create a standard network that uses trainoss with newff, newcf, or newelm.

To prepare a custom network to be trained with trainoss

1 Set net.trainFcn to 'trainoss'. This will set net.trainParam to trainoss's default parameters.

2 Set net.trainParam properties to desired values.

In either case, calling train with the resulting network will train the network with trainoss.

## Algorithm

trainoss can train any network as long as its weight, net input, and transfer functions have derivative functions.

Backpropagation is used to calculate derivatives of performance perf with respect to the weight and bias variables X. Each variable is adjusted according to the following:

```
X = X + a*dX;
```

where dX is the search direction. The parameter a is selected to minimize the performance along the search direction. The line search function searchFcn is used to locate the minimum point. The first search direction is the negative of the gradient of performance. In succeeding iterations the search direction is computed from the new gradient and the previous steps and gradients according to the following formula:

```
dX = -gX + Ac*X_step + Bc*dgX;
```

where gX is the gradient, X_step is the change in the weights on the previous iteration, and dgX is the change in the gradient from the last iteration. See Battiti (*Neural Computation*) for a more detailed discussion of the one step secant algorithm.

Training stops when any of these conditions occur:

- The maximum number of epochs (repetitions) is reached.
- The maximum amount of time has been exceeded.
- Performance has been minimized to the goal.
- The performance gradient falls below mingrad.
- Validation performance has increased more than max_fail times since the last time it decreased (when using validation).

## See Also

newff, newcf, traingdm, traingda, traingdx, trainlm, trainrp, traincgf, traincgb, trainscg, traincgp, trainbfg

## References

Battiti, R. “First and second order methods for learning: Between steepest descent and Newton’s method,” *Neural Computation*, vol. 4, no. 2, pp. 141–166, 1992.

## Purpose

Random order incremental training with learning functions.

## Syntax

```
[net,TR,Ac,El] = trainr(net,Pd,Tl,Ai,Q,TS,VV,TV)
info = trainr(code)
```

## Description

`trainr` is not called directly. Instead it is called by `train` for networks whose `net.trainFcn` property is set to `'trainr'`.

`trainr` trains a network with weight and bias learning rules with incremental updates after each presentation of an input. Inputs are presented in random order.

`trainr(net,Pd,Tl,Ai,Q,TS,VV)` takes these inputs,

`net` — Neural network.

`Pd` — Delayed inputs.

`Tl` — Layer targets.

`Ai` — Initial input conditions.

`Q` — Batch size.

`TS` — Time steps.

`VV` — Ignored.

`TV` — Ignored.

and returns,

`net` — Trained network.

`TR` — Training record of various values over each epoch:

`TR.epoch` — Epoch number.

`TR.perf` — Training performance.

`Ac` — Collective layer outputs.

`El` — Layer errors.

Training occurs according to trainr's training parameters shown here with their default values:

| | | |
|---|---|---|
| net.trainParam.epochs | 100 | Maximum number of epochs to train |
| net.trainParam.goal | 0 | Performance goal |
| net.trainParam.show | 25 | Epochs between displays (NaN for no displays) |
| net.trainParam.time | inf | Maximum time to train in seconds |

Dimensions for these variables are:

Pd — No x Ni x TS cell array, each element Pd{i,j,ts} is a Dij x Q matrix.

Tl — Nl x TS cell array, each element P{i,ts} is a Vi x Q matrix or [].

Ai — Nl x LD cell array, each element Ai{i,k} is an Si x Q matrix.

where

```
Ni = net.numInputs
Nl = net.numLayers
LD = net.numLayerDelays
Ri = net.inputs{i}.size
Si = net.layers{i}.size
Vi = net.targets{i}.size
Dij = Ri * length(net.inputWeights{i,j}.delays)
```

trainr does not implement validation or test vectors, so arguments VV and TV are ignored.

trainr(code) returns useful information for each code string:

'pnames' — Names of training parameters

'pdefaults' — Default training parameters

## Network Use

You can create a standard network that uses trainr by calling newc or newsom.

To prepare a custom network to be trained with trainr

1 Set net.trainFcn to 'trainr'.

(This will set net.trainParam to trainr's default parameters.)

2 Set each net.inputWeights{i,j}.learnFcn to a learning function.

3 Set each `net.layerWeights{i,j}.learnFcn` to a learning function.

4 Set each `net.biases{i}.learnFcn` to a learning function. (Weight and bias learning parameters will automatically be set to default values for the given learning function.)

To train the network

1 Set `net.trainParam` properties to desired values.

2 Set weight and bias learning parameters to desired values.

3 Call `train`.

See `newc` and `newsom` for training examples.

## Algorithm

For each epoch, all training vectors (or sequences) are each presented once in a different random order with the network and weight and bias values updated accordingly after each individual presentation.

Training stops when any of these conditions are met:

- The maximum number of `epochs` (repetitions) is reached.
- Performance has been minimized to the `goal`.
- The maximum amount of `time` has been exceeded.

## See Also

`newp, newlin, train`

# trainrp

## Purpose

Resilient backpropagation

## Syntax

```
[net,TR,Ac,El] = trainrp(net,Pd,Tl,Ai,Q,TS,VV,TV)
info = trainrp(code)
```

## Description

`trainrp` is a network training function that updates weight and bias values according to the resilient backpropagation algorithm (RPROP).

`trainrp(net,Pd,Tl,Ai,Q,TS,VV,TV)` takes these inputs,

`net` — Neural network.

`Pd` — Delayed input vectors.

`Tl` — Layer target vectors.

`Ai` — Initial input delay conditions.

`Q` — Batch size.

`TS` — Time steps.

`VV` — Either empty matrix `[]` or structure of validation vectors.

`TV` — Either empty matrix `[]` or structure of test vectors.

and returns,

`net` — Trained network.

`TR` — Training record of various values over each epoch:

`TR.epoch` — Epoch number.

`TR.perf` — Training performance.

`TR.vperf` — Validation performance.

`TR.tperf` — Test performance.

`Ac` — Collective layer outputs for last epoch.

`El` — Layer errors for last epoch.

Training occurs according to the trainrp's training parameters shown here with their default values:

| | | |
|---|---|---|
| net.trainParam.epochs | 100 | Maximum number of epochs to train |
| net.trainParam.show | 25 | Epochs between showing progress |
| net.trainParam.goal | 0 | Performance goal |
| net.trainParam.time | inf | Maximum time to train in seconds |
| net.trainParam.min_grad | 1e-6 | Minimum performance gradient |
| net.trainParam.max_fail | 5 | Maximum validation failures |
| net.trainParam.lr | 0.01 | Learning rate |
| net.trainParam.delt_inc | 1.2 | Increment to weight change |
| net.trainParam.delt_dec | 0.5 | Decrement to weight change |
| net.trainParam.delta0 | 0.07 | Initial weight change |
| net.trainParam.deltamax | 50.0 | Maximum weight change |

Dimensions for these variables are

Pd — No x Ni x TS cell array, each element P{i,j,ts} is a Dij x Q matrix.

Tl — Nl x TS cell array, each element P{i,ts} is a Vi x Q matrix.

Ai — Nl x LD cell array, each element Ai{i,k} is an Si x Q matrix.

where

```
Ni = net.numInputs
Nl = net.numLayers
LD = net.numLayerDelays
Ri = net.inputs{i}.size
Si = net.layers{i}.size
Vi = net.targets{i}.size
Dij = Ri * length(net.inputWeights{i,j}.delays)
```

If VV is not [], it must be a structure of validation vectors,

VV.PD — Validation delayed inputs.

VV.Tl — Validation layer targets.

VV.Ai — Validation initial input conditions.

VV.Q — Validation batch size.

VV.TS — Validation time steps.

which is used to stop training early if the network performance on the validation vectors fails to improve or remains the same for `max_fail` epochs in a row.

If `TV` is not `[]`, it must be a structure of validation vectors,

`TV.PD` — Validation delayed inputs

`TV.Tl` — Validation layer targets

`TV.Ai` — Validation initial input conditions

`TV.Q` — Validation batch size

`TV.TS` — Validation time steps

which is used to test the generalization capability of the trained network.

`trainrp(code)` returns useful information for each `code` string:

`'pnames'` — Names of training parameters

`'pdefaults'` — Default training parameters

## Examples

Here is a problem consisting of inputs `p` and targets `t` that we would like to solve with a network.

```
p = [0 1 2 3 4 5];
t = [0 0 0 1 1 1];
```

Here a two-layer feed-forward network is created. The network's input ranges from `[0 to 10]`. The first layer has two `tansig` neurons, and the second layer has one `logsig` neuron. The `trainrp` network training function is to be used.

### Create and Test a Network

```
net = newff([0 5],[2 1],{'tansig','logsig'},'trainrp');
a = sim(net,p)
```

### Train and Retest the Network

```
net.trainParam.epochs = 50;
net.trainParam.show = 10;
net.trainParam.goal = 0.1;
net = train(net,p,t);
a = sim(net,p)
```

See `newff`, `newcf`, and `newelm` for other examples.

## Network Use

You can create a standard network that uses `trainrp` with `newff`, `newcf`, or `newelm`.

To prepare a custom network to be trained with `trainrp`

1. Set `net.trainFcn` to `'trainrp'`. This will set `net.trainParam` to `trainrp`'s default parameters.
2. Set `net.trainParam` properties to desired values.

In either case, calling `train` with the resulting network will train the network with `trainrp`.

## Algorithm

`trainrp` can train any network as long as its weight, net input, and transfer functions have derivative functions.

Backpropagation is used to calculate derivatives of performance `perf` with respect to the weight and bias variables `X`. Each variable is adjusted according to the following:

```
dX = deltaX.*sign(gX);
```

where the elements of `deltaX` are all initialized to `delta0` and `gX` is the gradient. At each iteration the elements of `deltaX` are modified. If an element of `gX` changes sign from one iteration to the next, then the corresponding element of `deltaX` is decreased by `delta_dec`. If an element of `gX` maintains the same sign from one iteration to the next, then the corresponding element of `deltaX` is increased by `delta_inc`. See Reidmiller and Braun, *Proceedings of the IEEE International Conference on Neural Networks*.

Training stops when any of these conditions occur:

- The maximum number of `epochs` (repetitions) is reached.
- The maximum amount of `time` has been exceeded.
- Performance has been minimized to the `goal`.
- The performance gradient falls below `mingrad`.
- Validation performance has increased more than `max_fail` times since the last time it decreased (when using validation).

## See Also

`newff`, `newcf`, `traingdm`, `traingda`, `traingdx`, `trainlm`, `traincgp`, `traincgf`, `traincgb`, `trainscg`, `trainoss`, `trainbfg`

## References

Riedmiller, M., and H. Braun, “A direct adaptive method for faster backpropagation learning: The RPROP algorithm,” *Proceedings of the IEEE International Conference on Neural Networks*, San Francisco,1993.

## Purpose

Sequential order incremental training w/learning functions

## Syntax

```
[net,TR,Ac,El] = trains(net,Pd,Tl,Ai,Q,TS,VV,TV)
info = trains(code)
```

## Description

`trains` is not called directly. Instead it is called by `train` for networks whose `net.trainFcn` property is set to `'trains'`.

`trains` trains a network with weight and bias learning rules with sequential updates. The sequence of inputs is presented to the network with updates occurring after each time step.

This incremental training algorithm is commonly used for adaptive applications.

`trains` takes these inputs:

`net` — Neural network
`Pd` — Delayed inputs
`Tl` — Layer targets
`Ai` — Initial input conditions
`Q` — Batch size
`TS` — Time steps
`VV` — Ignored
`TV` — Ignored

and after training the network with its weight and bias learning functions returns:

`net` — Updated network
`TR` — Training record
  `TR.time steps` — Number of time steps
  `TR.perf` — Performance for each time step
`Ac` — Collective layer outputs
`El` — Layer errors

Training occurs according to `trains`'s training parameter shown here with its default value:

`net.trainParam.passes` 1 Number of times to present sequence

Dimensions for these variables are

`Pd` — `No x NixTS` cell array, each element `P{i,j,ts}` is a `Zij x Q` matrix

`Tl` — `Nl x TS` cell array, each element `P{i,ts}` is an `Vi x Q` matrix or []

`Ai` — `Nl x LD` cell array, each element `Ai{i,k}` is an `Si x Q` matrix

`Ac` — `Nl x (LD+TS)` cell array, each element `Ac{i,k}` is an `Si x Q` matrix

`El` — `Nl x TS` cell array, each element `El{i,k}` is an `Si x Q` matrix or []

where

```
Ni = net.numInputs
Nl = net.numLayers
LD = net.numLayerDelays
Ri = net.inputs{i}.size
Si = net.layers{i}.size
Vi = net.targets{i}.size
Zij = Ri * length(net.inputWeights{i,j}.delays)
```

`trains`(code) returns useful information for each `code` string:

`'pnames'` — Names of training parameters

`'pdefaults'` — Default training parameters

## Network Use

You can create a standard network that uses `trains` for adapting by calling `newp` or `newlin`.

To prepare a custom network to adapt with `trains`

1. Set `net.adaptFcn` to '`trains`'.

   (This will set `net.adaptParam` to `trains`'s default parameters.)

2. Set each `net.inputWeights{i,j}.learnFcn` to a learning function.
3. Set each `net.layerWeights{i,j}`.learnFcn to a learning function.
4. Set each `net.biases{i}.learnFcn` to a learning function. (Weight and bias learning parameters will automatically be set to default values for the given learning function.)

To allow the network to adapt

1. Set weight and bias learning parameters to desired values.
2. Call `adapt`.

See `newp` and `newlin` for adaption examples.

## Algorithm

Each weight and bias is updated according to its learning function after each time step in the input sequence.

## See Also

`newp, newlin, train, trainb, trainc, trainr`

# trainscg

**Purpose** Scaled conjugate gradient backpropagation

**Syntax**

```
[net,TR,Ac,El] = trainscg(net,Pd,Tl,Ai,Q,TS,VV,TV)
info = trainscg(code)
```

**Description** `trainscg` is a network training function that updates weight and bias values according to the scaled conjugate gradient method.

`trainscg(net,Pd,Tl,Ai,Q,TS,VV,TV)` takes these inputs,

`net` — Neural network.

`Pd` — Delayed input vectors.

`Tl` — Layer target vectors.

`Ai` — Initial input delay conditions.

`Q` — Batch size.

`TS` — Time steps.

`VV` — Either empty matrix `[]` or structure of validation vectors.

`TV` — Either empty matrix `[]` or structure of test vectors.

and returns,

`net` — Trained network.

`TR` — Training record of various values over each epoch:

- `TR.epoch` — Epoch number.
- `TR.perf` — Training performance.
- `TR.vperf` — Validation performance.
- `TR.tperf` — Test performance.

`Ac` — Collective layer outputs for last epoch.

`El` — Layer errors for last epoch.

Training occurs according to the `trainscg`'s training parameters shown here with their default values:

| | | |
|---|---|---|
| `net.trainParam.epochs` | `100` | Maximum number of epochs to train |
| `net.trainParam.show` | `25` | Epochs between showing progress |
| `net.trainParam.goal` | `0` | Performance goal |
| `net.trainParam.time` | `inf` | Maximum time to train in seconds |
| `net.trainParam.min_grad` | `1e-6` | Minimum performance gradient |
| `net.trainParam.max_fail` | `5` | Maximum validation failures |
| `net.trainParam.sigma` | `5.0e-5` | Determines change in weight for second derivative approximation. |
| `net.trainParam.lambda` | `5.0e-7` | Parameter for regulating the indefiniteness of the Hessian. |

Dimensions for these variables are

`Pd` — `No` x `Ni` x `TS` cell array, each element `P{i,j,ts}` is a `Dij` x `Q` matrix.

`Tl` — `Nl` x `TS` cell array, each element `P{i,ts}` is a `Vi` x `Q` matrix.

`Ai` — `Nl` x `LD` cell array, each element `Ai{i,k}` is an `Si` x `Q` matrix.

where

```
Ni = net.numInputs
Nl = net.numLayers
LD = net.numLayerDelays
Ri = net.inputs{i}.size
Si = net.layers{i}.size
Vi = net.targets{i}.size
Dij = Ri * length(net.inputWeights{i,j}.delays)
```

If `VV` is not `[]`, it must be a structure of validation vectors,

`VV.PD` — Validation delayed inputs.

`VV.Tl` — Validation layer targets.

`VV.Ai` — Validation initial input conditions.

`VV.Q` — Validation batch size.

`VV.TS` — Validation time steps.

which is used to stop training early if the network performance on the validation vectors fails to improve or remains the same for max_fail epochs in a row.

If TV is not [], it must be a structure of validation vectors,

TV.PD — Validation delayed inputs
TV.Tl — Validation layer targets
TV.Ai — Validation initial input conditions
TV.Q — Validation batch size
TV.TS — Validation time steps

which is used to test the generalization capability of the trained network.

trainscg(code) returns useful information for each code string:

'pnames' — Names of training parameters
'pdefaults' — Default training parameters

## Examples

Here is a problem consisting of inputs p and targets t that we would like to solve with a network.

```
p = [0 1 2 3 4 5];
t = [0 0 0 1 1 1];
```

Here a two-layer feed-forward network is created. The network's input ranges from [0 to 10]. The first layer has two tansig neurons, and the second layer has one logsig neuron. The trainscg network training function is used.

### Create and Test a Network

```
net = newff([0 5],[2 1],{'tansig','logsig'},'trainscg');
a = sim(net,p)
```

### Train and Retest the Network

```
net.trainParam.epochs = 50;
net.trainParam.show = 10;
net.trainParam.goal = 0.1;
net = train(net,p,t);
a = sim(net,p)
```

See newff, newcf, and newelm for other examples.

## Network Use

You can create a standard network that uses `trainscg` with `newff`, `newcf`, or `newelm`.

To prepare a custom network to be trained with `trainscg`

1. Set `net.trainFcn` to `'trainscg'`. This will set `net.trainParam` to `trainscg`'s default parameters.
2. Set `net.trainParam` properties to desired values.

In either case, calling `train` with the resulting network will train the network with `trainscg`.

## Algorithm

`trainscg` can train any network as long as its weight, net input, and transfer functions have derivative functions. Backpropagation is used to calculate derivatives of performance `perf` with respect to the weight and bias variables `X`.

The scaled conjugate gradient algorithm is based on conjugate directions, as in `traincgp`, `traincgf` and `traincgb`, but this algorithm does not perform a line search at each iteration. See Moller (*Neural Networks*) for a more detailed discussion of the scaled conjugate gradient algorithm.

Training stops when any of these conditions occur:

- The maximum number of `epochs` (repetitions) is reached.
- The maximum amount of `time` has been exceeded.
- Performance has been minimized to the `goal`.
- The performance gradient falls below `mingrad`.
- Validation performance has increased more than `max_fail` times since the last time it decreased (when using validation).

## See Also

`newff, newcf, traingdm, traingda, traingdx, trainlm, trainrp, traincgf, traincgb, trainbfg, traincgp, trainoss`

## References

Moller, M. F., “A scaled conjugate gradient algorithm for fast supervised learning,” *Neural Networks*, vol. 6, pp. 525-533, 1993.

# tramnmx

## Purpose

Transform data using a precalculated minimum and maximum value

## Syntax

```
[PN] = tramnmx(P,minp,maxp)
```

## Description

tramnmx transforms the network input set using minimum and maximum values that were previously computed by premnmx. This function needs to be used when a network has been trained using data normalized by premnmx. All subsequent inputs to the network need to be transformed using the same normalization.

tramnmx(P,minp, maxp)takes these inputs

P — R x Q matrix of input (column) vectors.

minp — R x 1 vector containing original minimums for each input.

maxp — R x 1 vector containing original maximums for each input.

and returns,

PN — R x Q matrix of normalized input vectors

## Examples

Here is the code to normalize a given data set, so that the inputs and targets will fall in the range [-1,1], using premnmx, and also code to train a network with the normalized data.

```
p = [-10 -7.5 -5 -2.5 0 2.5 5 7.5 10];
t = [0 7.07 -10 -7.07 0 7.07 10 7.07 0];
[pn,minp,maxp,tn,mint,maxt] = premnmx(p,t);
net = newff(minmax(pn),[5 1],{'tansig' 'purelin'},'trainlm');
net = train(net,pn,tn);
```

If we then receive new inputs to apply to the trained network, we will use tramnmx to transform them first. Then the transformed inputs can be used to simulate the previously trained network. The network output must also be unnormalized using postmnmx.

```
p2 = [4 -7];
[p2n] = tramnmx(p2,minp,maxp);
an = sim(net,pn);
[a] = postmnmx(an,mint,maxt);
```

## Algorithm

```
pn = 2*(p-minp)/(maxp-minp) - 1;
```

## See Also

premnmx, prestd, prepca, trastd, trapca

# trapca

## Purpose

Principal component transformation

## Syntax

```
[Ptrans] = trapca(P,transMat)
```

## Description

trapca preprocesses the network input training set by applying the principal component transformation that was previously computed by prepca. This function needs to be used when a network has been trained using data normalized by prepca. All subsequent inputs to the network need to be transformed using the same normalization.

trapca(P,transMat) takes these inputs,

P — R x Q matrix of centered input (column) vectors.

transMat — Transformation matrix.

and returns,

Ptrans — Transformed data set.

## Examples

Here is the code to perform a principal component analysis and retain only those components that contribute more than two percent to the variance in the data set. prestd is called first to create zero mean data, which is needed for prepca.

```
p = [-1.5 -0.58 0.21 -0.96 -0.79; -2.2 -0.87 0.31 -1.4  -1.2];
t = [-0.08 3.4 -0.82 0.69 3.1];
[pn,meanp,stdp,tn,meant,stdt] = prestd(p,t);
[ptrans,transMat] = prepca(pn,0.02);
net = newff(minmax(ptrans),[5 1],{'tansig''purelin'},'trainlm');
net = train(net,ptrans,tn);
```

If we then receive new inputs to apply to the trained network, we will use trastd and trapca to transform them first. Then the transformed inputs can be used to simulate the previously trained network. The network output must also be unnormalized using poststd.

```
p2 = [1.5 -0.8;0.05 -0.3];
[p2n] = trastd(p2,meanp,stdp);
[p2trans] = trapca(p2n,transMat)
an = sim(net,p2trans);
[a] = poststd(an,meant,stdt);
```

## Algorithm

```
Ptrans = transMat*P;
```

## See Also

`prestd, premnmx, prepca, trastd, tramnmx`

## Purpose

Preprocess data using a precalculated mean and standard deviation

## Syntax

```
[PN] = trastd(P,meanp,stdp)
```

## Description

`trastd` preprocesses the network training set using the mean and standard deviation that were previously computed by `prestd`. This function needs to be used when a network has been trained using data normalized by `prestd`. All subsequent inputs to the network need to be transformed using the same normalization.

`trastd(P,T)` takes these inputs,

`P` — `R` x `Q` matrix of input (column) vectors.

`meanp` — `R` x `1` vector containing the original means for each input.

`stdp` — `R` x `1` vector containing the original standard deviations for each input.

and returns,

`PN` — `R` x `Q` matrix of normalized input vectors.

## Examples

Here is the code to normalize a given data set so that the inputs and targets will have means of zero and standard deviations of 1.

```
p = [-0.92 0.73 -0.47 0.74 0.29; -0.08 0.86 -0.67 -0.52 0.93];
t = [-0.08 3.4 -0.82 0.69 3.1];
[pn,meanp,stdp,tn,meant,stdt] = prestd(p,t);
net = newff(minmax(pn),[5 1],{'tansig' 'purelin'},'trainlm');
net = train(net,pn,tn);
```

If we then receive new inputs to apply to the trained network, we will use `trastd` to transform them first. Then the transformed inputs can be used to simulate the previously trained network. The network output must also be unnormalized using `poststd`.

```
p2 = [1.5 -0.8;0.05 -0.3];
[p2n] = trastd(p2,meanp,stdp);
an = sim(net,pn);
[a] = poststd(an,meant,stdt);
```

## Algorithm

```
pn = (p-meanp)/stdp;
```

### See Also

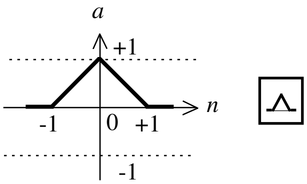
`premnmx`, `prepca`, `prestd`, `trapca`, `tramnmx`

# tribas

## Purpose

Triangular basis transfer function

## Graph and Symbol



*a = tribas(n)*

Triangular Basis Function

## Syntax

```
A = tribas(N)
info = tribas(code)
```

## Description

`tribas` is a transfer function. Transfer functions calculate a layer's output from its net input.

`tribas(N)` takes one input,

`N` — `S` x `Q` matrix of net input (column) vectors.

and returns each element of `N` passed through a radial basis function.

`tribas(code)` returns useful information for each `code` string:

'`deriv`' — Name of derivative function.

'`name`' — Full name.

'`output`' — Output range.

'`active`' — Active input range.

## Examples

Here we create a plot of the `tribas` transfer function.

```
n = -5:0.1:5;
a = tribas(n);
plot(n,a)
```

## Network Use

To change a network so that a layer uses `tribas`, set `net.layers{i}.transferFcn` to '`tribas`'.

Call sim to simulate the network with tribas.

### Algorithm

tribas(N) calculates its output with according to:

tribas(n) = 1-abs(n), if -1 <= n <= 1; = 0, otherwise.

### See Also

sim, radbas

# vec2ind

## Purpose

Convert vectors to indices

## Syntax

```
ind = vec2ind(vec)
```

## Description

`ind2vec` and `vec2ind` allow indices to be represented either by themselves or as vectors containing a 1 in the row of the index they represent.

`vec2ind(vec)` takes one argument,

`vec` — Matrix of vectors, each containing a single 1.

and returns the indices of the 1's.

## Examples

Here four vectors (each containing only one "1" element) are defined and the indices of the 1's are found.

```
vec = [1 0 0 0; 0 0 1 0; 0 1 0 1]
ind = vec2ind(vec)
```

## See Also

`ind2vec`

# A

# Glossary

**ADALINE** - An acronym for a linear neuron: ADAptive LINear Element.

**adaption** - A training method that proceeds through the specified sequence of inputs, calculating the output, error and network adjustment for each input vector in the sequence as the inputs are presented.

**adaptive learning rate** - A learning rate that is adjusted according to an algorithm during training to minimize training time.

**adaptive filter** - A network that contains delays and whose weights are adjusted after each new input vector is presented. The network "adapts" to changes in the input signal properties if such occur. This kind of filter is used in long distance telephone lines to cancel echoes.

**architecture** - A description of the number of the layers in a neural network, each layer's transfer function, the number of neurons per layer, and the connections between layers.

**backpropagation learning rule** - A learning rule in which weights and biases are adjusted by error-derivative (delta) vectors backpropagated through the network. Backpropagation is commonly applied to feedforward multilayer networks. Sometimes this rule is called the generalized delta rule.

**backtracking search** - Linear search routine that begins with a step multiplier of 1 and then backtracks until an acceptable reduction in the performance is obtained.

**batch** - A matrix of input (or target) vectors applied to the network "simultaneously." Changes to the network weights and biases are made just once for the entire set of vectors in the input matrix. (This term is being replaced by the more descriptive expression "concurrent vectors.")

**batching** - The process of presenting a set of input vectors for simultaneous calculation of a matrix of output vectors and/or new weights and biases.

**Bayesian framework** - Assumes that the weights and biases of the network are random variables with specified distributions.

**BFGS quasi-Newton algorithm** - A variation of Newton's optimization algorithm, in which an approximation of the Hessian matrix is obtained from gradients computed at each iteration of the algorithm.

**bias** - A neuron parameter that is summed with the neuron's weighted inputs and passed through the neuron's transfer function to generate the neuron's output.

**bias vector** - A column vector of bias values for a layer of neurons.

**Brent's search** - A linear search that is a hybrid combination of the golden section search and a quadratic interpolation.

**Charalambous' search** - A hybrid line search that uses a cubic interpolation, together with a type of sectioning.

**cascade forward network** - A layered network in which each layer only receives inputs from previous layers.

**classification** - An association of an input vector with a particular target vector.

**competitive layer** - A layer of neurons in which only the neuron with maximum net input has an output of 1 and all other neurons have an output of 0. Neurons compete with each other for the right to respond to a given input vector.

**competitive learning** - The unsupervised training of a competitive layer with the instar rule or Kohonen rule. Individual neurons learn to become feature detectors. After training, the layer categorizes input vectors among its neurons.

**competitive transfer function** - Accepts a net input vector for a layer and returns neuron outputs of 0 for all neurons except for the "winner," the neuron associated with the most positive element of the net input **n**.

**concurrent input vectors** - Name given to a matrix of input vectors that are to be presented to a network "simultaneously." All the vectors in the matrix will be used in making just one set of changes in the weights and biases.

**conjugate gradient algorithm** - In the conjugate gradient algorithms a search is performed along conjugate directions, which produces generally faster convergence than a search along the steepest descent directions.

**connection** - A one-way link between neurons in a network.

**connection strength** - The strength of a link between two neurons in a network. The strength, often called weight, determines the effect that one neuron has on another.

**cycle** - A single presentation of an input vector, calculation of output, and new weights and biases.

**dead neurons** - A competitive layer neuron that never won any competition during training and so has not become a useful feature detector. Dead neurons do not respond to any of the training vectors.

**decision boundary** - A line, determined by the weight and bias vectors, for which the net input $n$ is zero.

**delta rule** - See the Widrow-Hoff learning rule.

**delta vector** - The delta vector for a layer is the derivative of a network's output error with respect to that layer's net input vector.

**distance** - The distance between neurons, calculated from their positions with a distance function.

**distance function** - A particular way of calculating distance, such as the Euclidean distance between two vectors.

**early stopping** - A technique based on dividing the data into three subsets. The first subset is the training set used for computing the gradient and updating the network weights and biases. The second subset is the validation set. When the validation error increases for a specified number of iterations, the training is stopped, and the weights and biases at the minimum of the validation error are returned. The third subset is the test set. It is used to verify the network design.

**epoch** - The presentation of the set of training (input and/or target) vectors to a network and the calculation of new weights and biases. Note that training vectors can be presented one at a time or all together in a batch.

**error jumping** - A sudden increase in a network's sum-squared error during training. This is often due to too large a learning rate.

**error ratio** - A training parameter used with adaptive learning rate and momentum training of backpropagation networks.

**error vector** - The difference between a network's output vector in response to an input vector and an associated target output vector.

**feedback network** - A network with connections from a layer's output to that layer's input. The feedback connection can be direct or pass through several layers.

**feedforward network** - A layered network in which each layer only receives inputs from previous layers.

**Fletcher-Reeves update** - A method developed by Fletcher and Reeves for computing a set of conjugate directions. These directions are used as search directions as part of a conjugate gradient optimization procedure.

**function approximation** - The task performed by a network trained to respond to inputs with an approximation of a desired function.

**generalization** - An attribute of a network whose output for a new input vector tends to be close to outputs for similar input vectors in its training set.

**generalized regression network** - Approximates a continuous function to an arbitrary accuracy, given a sufficient number of hidden neurons.

**global minimum** - The lowest value of a function over the entire range of its input parameters. Gradient descent methods adjust weights and biases in order to find the global minimum of error for a network.

**golden section search** - A linear search that does not require the calculation of the slope. The interval containing the minimum of the performance is subdivided at each iteration of the search, and one subdivision is eliminated at each iteration.

**gradient descent** - The process of making changes to weights and biases, where the changes are proportional to the derivatives of network error with respect to those weights and biases. This is done to minimize network error.

**hard-limit transfer function** - A transfer that maps inputs greater-than or equal-to 0 to 1, and all other values to 0.

**Hebb learning rule** - Historically the first proposed learning rule for neurons. Weights are adjusted proportional to the product of the outputs of pre- and post-weight neurons.

**hidden layer** - A layer of a network that is not connected to the network output. (For instance, the first layer of a two-layer feedforward network.)

**home neuron** - A neuron at the center of a neighborhood.

**hybrid bisection-cubicsearch** - A line search that combines bisection and cubic interpolation.

**input layer** - A layer of neurons receiving inputs directly from outside the network.

**initialization** - The process of setting the network weights and biases to their original values.

**input space** - The range of all possible input vectors.

**input vector** - A vector presented to the network.

**input weights** - The weights connecting network inputs to layers.

**input weight vector** - The row vector of weights going to a neuron.

**Jacobian matrix -** Contains the first derivatives of the network errors with respect to the weights and biases.

**Kohonen learning rule** - A learning rule that trains selected neuron's weight vectors to take on the values of the current input vector.

**layer** - A group of neurons having connections to the same inputs and sending outputs to the same destinations.

**layer diagram** - A network architecture figure showing the layers and the weight matrices connecting them. Each layer's transfer function is indicated with a symbol. Sizes of input, output, bias and weight matrices are shown. Individual neurons and connections are not shown. (See Chapter 2.)

**layer weights** - The weights connecting layers to other layers. Such weights need to have non-zero delays if they form a recurrent connection (i.e., a loop).

**learning** - The process by which weights and biases are adjusted to achieve some desired network behavior.

**learning rate** - A training parameter that controls the size of weight and bias changes during learning.

**learning rules** - Methods of deriving the next changes that might be made in a network OR a procedure for modifying the weights and biases of a network.

**Levenberg-Marquardt** - An algorithm that trains a neural network 10 to 100 faster than the usual gradient descent backpropagation method. It will always compute the approximate Hessian matrix, which has dimensions *n*-by-*n*.

**line search function** - Procedure for searching along a given search direction (line) to locate the minimum of the network performance.

**linear transfer function** - A transfer function that produces its input as its output.

**link distance** - The number of links, or steps, that must be taken to get to the neuron under consideration.

**local minimum** - The minimum of a function over a limited range of input values. A local minimum may not be the global minimum.

**log-sigmoid transfer function** - A squashing function of the form shown below that maps the input to the interval (0,1). (The toolbox function is `logsig`.)

$$f(n) = \frac{1}{1+e^{-n}}$$

**Manhattan distance** - The Manhattan distance between two vectors **x** and **y** is calculated as:

```
D = sum(abs(x-y))
```

**maximum performance increase** - The maximum amount by which the performance is allowed to increase in one iteration of the variable learning rate training algorithm.

**maximum step size** - The maximum step size allowed during a linear search. The magnitude of the weight vector is not allowed to increase by more than this maximum step size in one iteration of a training algorithm.

**mean square error function** - The performance function that calculates the average squared error between the network outputs **a** and the target outputs **t**.

**momentum** - A technique often used to make it less likely for a backpropagation networks to get caught in a shallow minima.

**momentum constant** - A training parameter that controls how much "momentum" is used.

**mu parameter** - The initial value for the scalar μ.

**neighborhood** - A group of neurons within a specified distance of a particular neuron. The neighborhood is specified by the indices for all of the neurons that lie within a radius $d$ of the winning neuron $i^*$ :

$$N_i(d) = \{j, d_{ij} \le d\}$$

**net input vector** - The combination, in a layer, of all the layer's weighted input vectors with its bias.

**neuron** - The basic processing element of a neural network. Includes weights and bias, a summing junction and an output transfer function. Artificial neurons, such as those simulated and trained with this toolbox, are abstractions of biological neurons.

**neuron diagram** - A network architecture figure showing the neurons and the weights connecting them. Each neuron's transfer function is indicated with a symbol.

**ordering phase** - Period of training during which neuron weights are expected to order themselves in the input space consistent with the associated neuron positions.

**output layer** - A layer whose output is passed to the world outside the network.

**output vector** - The output of a neural network. Each element of the output vector is the output of a neuron.

**output weight vector** - The column vector of weights coming from a neuron or input. (See outstar learning rule.)

**outstar learning rule** - A learning rule that trains a neuron's (or input's) output weight vector to take on the values of the current output vector of the post-weight layer. Changes in the weights are proportional to the neuron's output.

**overfitting** - A case in which the error on the training set is driven to a very small value, but when new data is presented to the network, the error is large.

**pass** - Each traverse through all of the training input and target vectors.

**pattern** - A vector.

**pattern association** - The task performed by a network trained to respond with the correct output vector for each presented input vector.

**pattern recognition** - The task performed by a network trained to respond when an input vector close to a learned vector is presented. The network "recognizes" the input as one of the original target vectors.

**performance function** - Commonly the mean squared error of the network outputs. However, the toolbox also considers other performance functions. Type `nnets` and look under performance functions.

**perceptron** - A single-layer network with a hard-limit transfer function. This network is often trained with the perceptron learning rule.

**perceptron learning rule** - A learning rule for training single-layer hard-limit networks. It is guaranteed to result in a perfectly functioning network in finite time, given that the network is capable of doing so.

**performance** - The behavior of a network.

**Polak-Ribiére update** - A method developed by Polak and Ribiére for computing a set of conjugate directions. These directions are used as search directions as part of a conjugate gradient optimization procedure.

**positive linear transfer function** - A transfer function that produces an output of zero for negative inputs and an output equal to the input for positive inputs.

**postprocessing** - Converts normalized outputs back into the same units that were used for the original targets.

**Powell-Beale restarts** - A method developed by Powell and Beale for computing a set of conjugate directions. These directions are used as search directions as part of a conjugate gradient optimization procedure. This procedure also periodically resets the search direction to the negative of the gradient.

**preprocessing** - Perform some transformation of the input or target data before it is presented to the neural network.

**principal component analysis** - Orthogonalize the components of network input vectors. This procedure can also reduce the dimension of the input vectors by eliminating redundant components.

**quasi-Newton algorithm** - Class of optimization algorithm based on Newton's method. An approximate Hessian matrix is computed at each iteration of the algorithm based on the gradients.

**radial basis networks** - A neural network that can be designed directly by fitting special response elements where they will do the most good.

**radial basis transfer function** - The transfer function for a radial basis neuron is:

$$radbas(n) = e^{-n^2}$$

**regularization** - Involves modifying the performance function, which is normally chosen to be the sum of squares of the network errors on the training set, by adding some fraction of the squares of the network weights.

**resilient backpropagation** - A training algorithm that eliminates the harmful effect of having a small slope at the extreme ends of the sigmoid "squashing" transfer functions.

**saturating linear transfer function** - A function that is linear in the interval (-1,+1) and saturates outside this interval to -1 or +1. (The toolbox function is `satlin`.)

**scaled conjugate gradient algorithm** - Avoids the time consuming line search of the standard conjugate gradient algorithm.

**sequential input vectors** - A set of vectors that are to be presented to a network "one after the other." The network weights and biases are adjusted on the presentation of each input vector.

**sigma parameter** - Determines the change in weight for the calculation of the approximate Hessian matrix in the scaled conjugate gradient algorithm.

**sigmoid** - Monotonic S-shaped function mapping numbers in the interval (-∞,∞) to a finite interval such as (-1,+1) or (0,1).

**simulation -** Takes the network input **p**, and the network object `net`, and returns the network outputs **a.**

**spread constant -** The distance an input vector must be from a neuron's weight vector to produce an output of 0.5.

**squashing function** - A monotonic increasing function that takes input values between -∞ and +∞ and returns values in a finite interval.

**star learning rule -** A learning rule that trains a neuron's weight vector to take on the values of the current input vector. Changes in the weights are proportional to the neuron's output.

**sum-squared error** - The sum of squared differences between the network targets and actual outputs for a given input vector or set of vectors.

**supervised learning** - A learning process in which changes in a network's weights and biases are due to the intervention of any external teacher. The teacher typically provides output targets.

**symmetric hard-limit transfer function** - A transfer that maps inputs greater-than or equal-to 0 to +1, and all other values to -1.

**symmetric saturating linear transfer function** - Produces the input as its output as long as the input i in the range -1 to 1. Outside that range the output is -1 and +1 respectively.

**tan-sigmoid transfer function** - A squashing function of the form shown below that maps the input to the interval (-1,1). (The toolbox function is `tansig`.)

$$f(n) = \frac{1}{1 + e^{-n}}$$

**tapped delay line** - A sequential set of delays with outputs available at each delay output.

**target vector** - The desired output vector for a given input vector.

**test vectors** - A set of input vectors (not used directly in training) that is used to test the trained network.

**topology functions** - Ways to arrange the neurons in a grid, box, hexagonal, or random topology.

**training** - A procedure whereby a network is adjusted to do a particular job. Commonly viewed as an "offline" job, as opposed to an adjustment made during each time interval as is done in adaptive training.

**training vector** - An input and/or target vector used to train a network.

**transfer function** - The function that maps a neuron's (or layer's) net output **n** to its actual output.

**tuning phase** - Period of SOFM training during which weights are expected to spread out relatively evenly over the input space while retaining their topological order found during the ordering phase.

**underdetermined system** - A system that has more variables than constraints.

**unsupervised learning** - A learning process in which changes in a network's weights and biases are not due to the intervention of any external teacher. Commonly changes are a function of the current network input vectors, output vectors, and previous weights and biases.

**update -** Make a change in weights and biases. The update can occur after presentation of a single input vector or after accumulating changes over several input vectors.

**validation vectors -** A set of input vectors (not used directly in training) that is used to monitor training progress so as to keep the network from overfitting.

**weighted input vector -** The result of applying a weight to a layer's input, whether it is a network input or the output of another layer.

**weight function -** Weight functions apply weights to an input to get weighted inputs as specified by a particular function.

**weight matrix -** A matrix containing connection strengths from a layer's inputs to its neurons. The element $w_{i,j}$ of a weight matrix W refers to the connection strength from input j to neuron i.

**Widrow-Hoff learning rule -** A learning rule used to trained single-layer linear networks. This rule is the predecessor of the backpropagation rule and is sometimes referred to as the delta rule.

# Bibliography

**[Batt92]** Battiti, R., “First and second order methods for learning: Between steepest descent and Newton’s method,” *Neural Computation*, vol. 4, no. 2, pp. 141–166, 1992.

**[Beal72]** Beale, E. M. L., “A derivation of conjugate gradients,” in F. A. Lootsma, ed., *Numerical methods for nonlinear optimization*, London: Academic Press, 1972.

**[Bren73]** Brent, R. P., *Algorithms for Minimization Without Derivatives*, Englewood Cliffs, NJ: Prentice-Hall, 1973.

**[Caud89]** Caudill, M., *Neural Networks Primer*, San Francisco, CA: Miller Freeman Publications, 1989.

This collection of papers from the *AI Expert Magazine* gives an excellent introduction to the field of neural networks. The papers use a minimum of mathematics to explain the main results clearly. Several good suggestions for further reading are included.

**[CaBu92]** Caudill, M., and C. Butler, *Understanding Neural Networks: Computer Explorations, Vols. 1 and 2,* Cambridge, MA: the MIT Press, 1992.

This is a two volume workbook designed to give students “hands on” experience with neural networks. It is written for a laboratory course at the senior or first-year graduate level. Software for IBM PC and Apple Macintosh computers is included. The material is well written, clear and helpful in understanding a field that traditionally has been buried in mathematics.

**[Char92]** Charalambous, C.,“Conjugate gradient algorithm for efficient training of artificial neural networks,” *IEEE Proceedings*, vol. 139, no. 3, pp. 301–310, 1992.

**[ChCo91]** Chen, S., C. F. N. Cowan, and P. M. Grant, “Orthogonal least squares learning algorithm for radial basis function networks,” *IEEE Transactions on Neural Networks*, vol. 2, no. 2, pp. 302-309, 1991.

This paper gives an excellent introduction to the field of radial basis functions. The papers use a minimum of mathematics to explain the main results clearly. Several good suggestions for further reading are included.

**[DARP88]** *DARPA Neural Network Study*, Lexington, MA: M.I.T. Lincoln Laboratory, 1988.

This book is a compendium of knowledge of neural networks as they were known to 1988. It presents the theoretical foundations of neural networks and discusses their current applications. It contains sections on associative memories, recurrent networks, vision, speech recognition, and robotics. Finally, it discusses simulation tools and implementation technology.

**[DeSc83]** Dennis, J. E., and R. B. Schnabel, *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*, Englewood Cliffs, NJ: Prentice-Hall, 1983.

**[Elma90]** Elman, J. L.,"Finding structure in time," *Cognitive Science*, vol. 14, pp. 179-211, 1990.

This paper is a superb introduction to the Elman networks described in Chapter 10, "Recurrent Networks."

**[FlRe64]** Fletcher, R., and C. M. Reeves, "Function minimization by conjugate gradients," *Computer Journal*, vol. 7, pp. 149-154, 1964.

**[FoHa97]** Foresee, F. D., and M. T. Hagan, "Gauss-Newton approximation to Bayesian regularization," *Proceedings of the 1997 International Joint Conference on Neural Networks*, pages 1930-1935, 1997.

**[GiMu81]** Gill, P. E., W. Murray, and M. H. Wright, *Practical Optimization*, New York: Academic Press, 1981.

**[Gros82]** Grossberg,  S., *Studies of the Mind and Brain*, Drodrecht, Holland: Reidel Press, 1982.

This book contains articles summarizing Grossberg's theoretical psychophysiology work up to 1980. Each article contains a preface explaining the main points.

**[HaDe99]** Hagan,M.T. and H.B. Demuth, "Neural Networks for Control," *Proceedings of the 1999 American Control Conference*, San Diego, CA, 1999, pp. 1642-1656.

**[HaJe99]** Hagan, M.T., O. De Jesus, and R. Schultz, "Training Recurrent Networks for Filtering and Control," Chapter 12 in *Recurrent Neural Networks: Design and Applications*, L. Medsker and L.C. Jain, Eds., CRC Press, 1999, pp. 311-340.

**[HaMe94]** Hagan, M. T., and M. Menhaj, "Training feedforward networks with the Marquardt algorithm," *IEEE Transactions on Neural Networks*, vol. 5, no. 6, pp. 989–993, 1994.

This paper reports the first development of the Levenberg-Marquardt algorithm for neural networks. It describes the theory and application of the algorithm, which trains neural networks at a rate 10 to 100 times faster than the usual gradient descent backpropagation method.

**[HDB96]** Hagan, M. T., H. B. Demuth, and M. H. Beale, *Neural Network Design*, Boston, MA: PWS Publishing, 1996.

This book provides a clear and detailed survey of basic neural network architectures and learning rules. It emphasizes mathematical analysis of networks, methods of training networks, and application of networks to practical engineering problems. It has demonstration programs, an instructor's guide and transparency overheads for teaching.

**[Hebb49]** Hebb, D. O., *The Organization of Behavior,* New York: Wiley, 1949.

This book proposed neural network architectures and the first learning rule. The learning rule is used to form a theory of how collections of cells might form a concept.

**[Himm72]** Himmelblau, D. M., *Applied Nonlinear Programming*, New York: McGraw-Hill, 1972.

[Joll86] Jolliffe, I. T., *Principal Component Analysis*, New York: Springer-Verlag, 1986.

**[HuSb92]** Hunt, K.J., D. Sbarbaro, R. Zbikowski, and P.J. Gawthrop, Neural Networks for Control System - A Survey," *Automatica*, Vol. 28, 1992, pp. 1083-1112.

**[Koho87]** Kohonen, T., *Self-Organization and Associative Memory, 2nd Edition*, Berlin: Springer-Verlag, 1987.

This book analyzes several learning rules. The Kohonen learning rule is then introduced and embedded in self-organizing feature maps. Associative networks are also studied.

**[Koho97]** Kohonen, T., *Self-Organizing Maps*, Second Edition, Berlin: Springer-Verlag, 1997.

This book discusses the history, fundamentals, theory, applications and hardware of self-organizing maps. It also includes a comprehensive literature survey.

**[LiMi89]** Li, J., A. N. Michel, and W. Porod, “Analysis and synthesis of a class of neural networks: linear systems operating on a closed hypercube,” *IEEE Transactions on Circuits and Systems*, vol. 36, no. 11, pp. 1405-1422, 1989.

This paper discusses a class of neural networks described by first order linear differential equations that are defined on a closed hypercube. The systems considered retain the basic structure of the Hopfield model but are easier to analyze and implement. The paper presents an efficient method for determining the set of asymptotically stable equilibrium points and the set of unstable equilibrium points. Examples are presented. The method of Li et. al. is implemented in Chapter 9 of this *User's Guide*.

**[Lipp87]** Lippman, R. P., “An introduction to computing with neural nets,” *IEEE ASSP Magazine*, pp. 4-22, 1987.

This paper gives an introduction to the field of neural nets by reviewing six neural net models that can be used for pattern classification. The paper shows how existing classification and clustering algorithms can be performed using simple components that are like neurons. This is a highly readable paper.

**[MacK92]** MacKay, D. J. C., “Bayesian interpolation,” *Neural Computation*, vol. 4, no. 3, pp. 415-447, 1992.

**[McPi43]** McCulloch, W. S., and W. H. Pitts, “A logical calculus of ideas immanent in nervous activity,” *Bulletin of Mathematical Biophysics*, vol. 5, pp. 115-133, 1943.

A classic paper that describes a model of a neuron that is binary and has a fixed threshold. A network of such neurons can perform logical operations.

**[Moll93]** Moller, M. F., “A scaled conjugate gradient algorithm for fast supervised learning,” *Neural Networks*, vol. 6, pp. 525-533, 1993.

**[MuNe92]**Murray, R., D. Neumerkel, and D. Sbarbaro, “Neural Networks for Modeling and Control of a Non-linear Dynamic System,” *Proceedings of the 1992 IEEE International Symposium on Intelligent Control*, 1992, pp. 404-409.

**[NaMu97]**Narendra, K.S. and S. Mukhopadhyay, “Adaptive Control Using Neural Networks and Approximate Models,” *IEEE Transactions on Neural Networks* Vol. 8, 1997, pp. 475-485.

**[NgWi89]** Nguyen, D., and B. Widrow, “The truck backer-upper: An example of self-learning in neural networks,” *Proceedings of the International Joint Conference on Neural Networks*, vol 2, pp. 357-363, 1989.

This paper describes a two-layer network that first learned the truck dynamics and then learned how to back the truck to a specified position at a loading dock. To do this, the neural network had to solve a highly nonlinear control systems problem.

**[NgWi90]** Nguyen, D., and B. Widrow, “Improving the learning speed of 2-layer neural networks by choosing initial values of the adaptive weights,” *Proceedings of the International Joint Conference on Neural Networks*, vol 3, pp. 21-26, 1990.

Nguyen and Widrow demonstrate that a two-layer sigmoid/linear network can be viewed as performing a piecewise linear approximation of any learned function. It is shown that weights and biases generated with certain constraints result in an initial network better able to form a function approximation of an arbitrary function. Use of the Nguyen-Widrow (instead of purely random) initial conditions often shortens training time by more than an order of magnitude.

**[Powe77]** Powell, M. J. D., “Restart procedures for the conjugate gradient method,” *Mathematical Programming*, vol. 12, pp. 241-254, 1977.

**[Pulu92]** N. Purdie, E.A. Lucas and M.B. Talley, “Direct measure of total cholesterol and its distribution among major serum lipoproteins,” Clinical Chemistry, vol. 38, no. 9, pp. 1645-1647, 1992.

**[RiBr93]** Riedmiller, M., and H. Braun, “A direct adaptive method for faster backpropagation learning: The RPROP algorithm,” *Proceedings of the IEEE International Conference on Neural Networks*, 1993.

**[Rose61]** Rosenblatt, F., *Principles of Neurodynamics*, Washington D.C.: Spartan Press, 1961.

This book presents all of Rosenblatt’s results on perceptrons. In particular, it presents his most important result, the *perceptron learning theorem*.

**[RuHi86a]** Rumelhart, D. E., G. E. Hinton, and R. J. Williams, “Learning internal representations by error propagation,”, in D. E. Rumelhart and J. L. McClelland, eds. *Parallel Data Processing, vol.1*, Cambridge, MA: The M.I.T. Press, pp. 318-362, 1986.

This is a basic reference on backpropagation.

**[RuHi86b]** Rumelhart, D. E., G. E. Hinton, and R. J. Williams, “Learning representations by back-propagating errors,” *Nature*, vol. 323, pp. 533–536, 1986.

**[RuMc86]** Rumelhart, D. E., J. L. McClelland, and the PDP Research Group, eds., *Parallel Distributed Processing, Vols. 1 and 2*, Cambridge, MA: The M.I.T. Press, 1986.

These two volumes contain a set of monographs that present a technical introduction to the field of neural networks. Each section is written by different authors. These works present a summary of most of the research in neural networks to the date of publication.

**[Scal85]** Scales, L. E., *Introduction to Non-Linear Optimization*, New York: Springer-Verlag, 1985.

**[SoHa96]** Soloway, D. and P.J. Haley, “Neural Generalized Predictive Control,” *Proceedings of the 1996 IEEE International Symposium on Intelligent Control*, 1996, pp. 277-281.

**[VoMa88]** Vogl, T. P., J. K. Mangis, A. K. Rigler, W. T. Zink, and D. L. Alkon, “Accelerating the convergence of the backpropagation method,” *Biological Cybernetics*, vol. 59, pp. 256-264, 1988.

Backpropagation learning can be speeded up and made less sensitive to small features in the error surface such as shallow local minima by combining techniques such as batching, adaptive learning rate, and momentum.

**[Wass93]** Wasserman, P. D., *Advanced Methods in Neural Computing*, New York: Van Nostrand Reinhold, 1993.

**[WiHo60]** Widrow, B., and M. E. Hoff, “Adaptive switching circuits,” *1960 IRE WESCON Convention Record, New York IRE*, pp. 96-104, 1960.

**[WiSt85]** Widrow, B., and S. D. Sterns, *Adaptive Signal Processing*, New York: Prentice-Hall, 1985.

This is a basic paper on adaptive signal processing.

C

# Demonstrations and Applications

# Tables of Demonstrations and Applications

## Chapter 2: Neuron Model and Network Architectures

| | Filename | Page |
|---|---|---|
| Simple neuron and transfer functions | `nnd2n1` | page 2-5 |
| Neuron with vector input | `nnd2n2` | page 2-7 |

## Chapter 3: Perceptrons

| | Filename | Page |
|---|---|---|
| Decision boundaries | `nnd4db` | page 3-5 |
| Perceptron learning rule. Pick boundaries | `nnd4pr` | page 3-15 |
| Classification with a two-input perceptron | `demop1` | page 3-20 |
| Outlier input vectors | `demop4` | page 3-21 |
| Normalized perceptron rule | `demop5` | page 3-22 |
| Linearly nonseparable vectors | `demop6` | page 3-21 |

## Chapter 4: Linear Filters

| | Filename | Page |
|---|---|---|
| Pattern association showing error surface | `demolin1` | page 4-9 |
| Training a linear neuron | `demolin2` | page 4-17 |
| Linear classification system | `nnd10lc` | page 4-17 |
| Linear fit of nonlinear problem | `demolin4` | page 4-18 |
| Underdetermined problem | `demolin5` | page 4-18 |
| Linearly dependent problem | `demolin6` | page 4-19 |
| Too large a learning rate | `demolin7` | page 4-19 |

## Chapter 5: Backpropagation

| | Filename | Page |
|---|---|---|
| Generalization | `nnd11gn` | page 5-51 |
| Steepest descent backpropagation | `nnd12sd1` | page 5-11 |
| Momentum backpropagation | `nnd12mo` | page 5-13 |
| Variable learning rate backpropagation | `nnd12vl` | page 5-15 |
| Conjugate gradient backpropagation | `nnd12cg` | page 5-19 |
| Marquardt backpropagation | `nnd12m` | page 5-30 |
| Sample training session | `demobp1` | page 5-30 |

## Chapter 7: Radial Basis Networks

| | Filename | Page |
|---|---|---|
| Radial basis approximation | demorb1 | page 7-8 |
| Radial basis underlapping neurons | demorb3 | page 7-8 |
| Radial basis overlapping neurons | demorb4 | page 7-8 |
| GRNN function approximation | demogrn1 | page 7-11 |
| PNN classification | demopnn1 | page 7-14 |

## Chapter 8: Self-Organizing and Learn. Vector Quant. Nets

| | Filename | Page |
|---|---|---|
| Competitive learning | democ1 | page 8-8 |
| One-dimensional self-organizing map | demosm1 | page 8-23 |
| Two-dimensional self-organizing map | demosm2 | page 8-23 |
| Learning vector quantization | demolvq1 | page 8-38 |

## Chapter 9: Recurrent Networks

| | Filename | Page |
|---|---|---|
| Hopfield two neuron design | demohop1 | page 9-1 |
| Hopfield unstable equilibria | demohop2 | page 9-14 |
| Hopfield three neuron design | demohop3 | page 9-14 |
| Hopfield spurious stable points | demohop4 | page 9-14 |

## Chapter 10: Adaptive Networks

| | Filename | Page |
|---|---|---|
| Adaptive noise cancellation, toolbox example | demolin8 | page 10-16 |
| Adaptive noise cancellation in airplane cockpit | nnd10nc | page 10-14 |

## Chapter 11: Applications

| | Filename | Page |
|---|---|---|
| Linear design | applin1 | page 11-3 |
| Adaptive linear prediction | applin2 | page 11-7 |
| Elman amplitude detection | appelm1 | page 11-11 |
| Character recognition | appcr1 | page 11-16 |

D

# Simulink

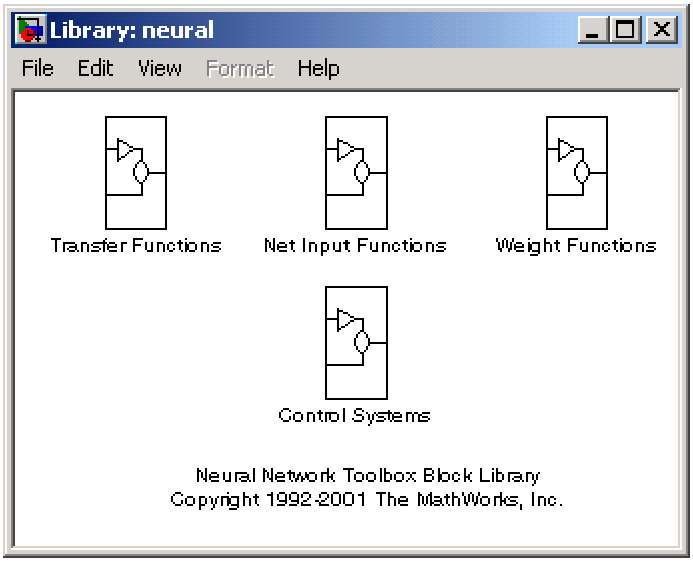
# Block Set

The Neural Network Toolbox provides a set of blocks you can use to build neural networks in Simulink or which can be used by the function `gensim` to generate the Simulink version of any network you have created in MATLAB®.

Bring up the Neural Network Toolbox blockset with this command.

```
neural
```

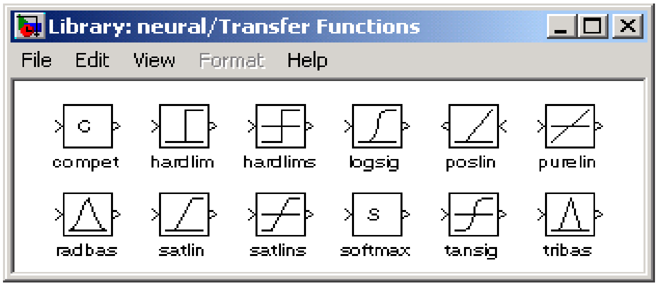
The result is a window that contains four blocks. Each of these blocks contains additional blocks.



Neural Network Toolbox Block Library
Copyright 1992-2001 The MathWorks, Inc.

## Transfer Function Blocks

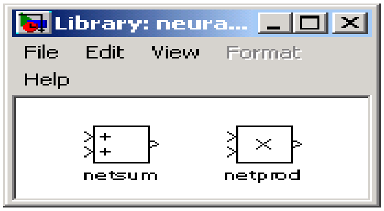
Double-click on the Transfer Functions block in the **Neural** window to bring up a window containing several transfer function blocks.



Each of these blocks takes a net input vector and generates a corresponding output vector whose dimensions are the same as the input vector.

## Net Input Blocks

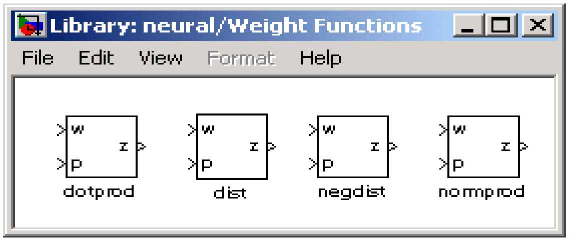
Double-click on the Net Input Functions block in the **Neural** window to bring up a window containing two net-input function blocks.



Each of these blocks takes any number of weighted input vectors, weight layer output vectors, and bias vectors, and returns a net-input vector.

## Weight Blocks

Double-click on the Weight Functions block in the **Neural** window to bring up a window containing three weight function blocks.



Each of these blocks takes a neuron's weight vector and applies it to an input vector (or a layer output vector) to get a weighted input value for a neuron.

It is important to note that the blocks above expect the neuron's weight vector to be defined as a column vector. This is because Simulink signals can be column vectors, but cannot be matrices or row vectors.

It is also important to note that because of this limitation you have to create $S$ weight function blocks (one for each row), to implement a weight matrix going to a layer with $S$ neurons.

This contrasts with the other two kinds of blocks. Only one net input function and one transfer function block are required for each layer.

# Block Generation

The function gensim generates block descriptions of networks so you can simulate them in Simulink.

```
gensim(net,st)
```

The second argument to gensim determines the sample time, which is normally chosen to be some positive real value.

If a network has no delays associated with its input weights or layer weights, this value can be set to -1. A value of -1 tells gensim to generate a network with continuous sampling.

## Example

Here is a simple problem defining a set of inputs p and corresponding targets t.

```
p = [1 2 3 4 5];
t = [1 3 5 7 9];
```

The code below designs a linear layer to solve this problem.

```
net = newlind(p,t)
```

We can test the network on our original inputs with sim.

```
y = sim(net,p)
```

The results returned show the network has solved the problem.

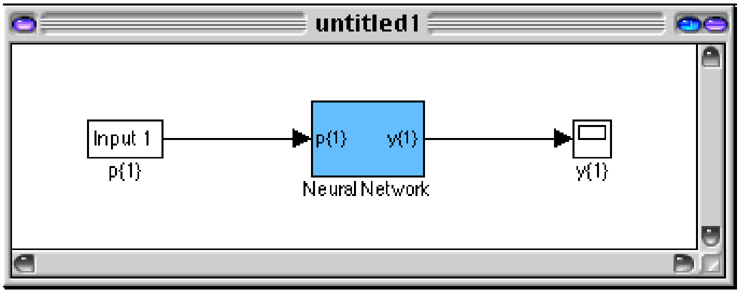
```
y =
       1.0000    3.0000    5.0000    7.0000    9.0000
```

Call gensim as follows to generate a Simulink version of the network.

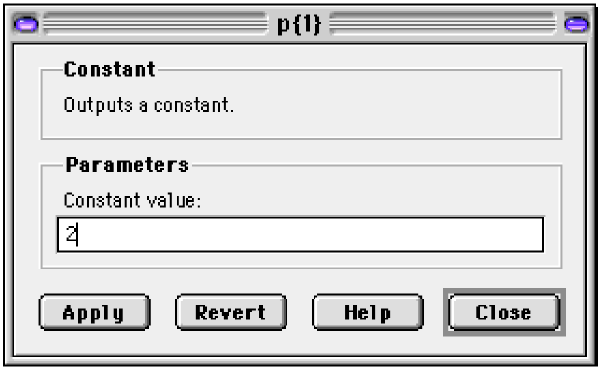
```
gensim(net,-1)
```

The second argument is -1 so the resulting network block samples continuously.

The call to gensim results in the following screen. It contains a Simulink system consisting of the linear network connected to a sample input and a scope.



To test the network, double-click on the `Input 1` block at left.



p{1}

**Constant**

Outputs a constant.

**Parameters**

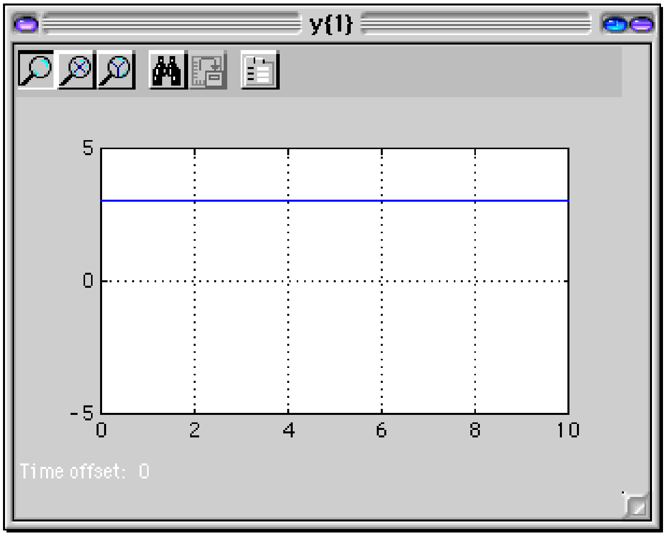
Constant value:

2

The input block is actually a standard Constant block. Change the constant value from the initial randomly generated value to `2`, and then select **Close**.

Select **Start** from the **Simulation** menu. Simulink momentarily pauses as it simulates the system.

When the simulation is over, double-click the scope at the right to see the following display of the network's response.



Note that the output is 3, which is the correct output for an input of 2.

## Exercises

Here are a couple of exercises you can try.

### Changing Input Signal

Replace the constant input block with a signal generator from the standard Simulink block set Sources. Simulate the system and view the network's response.

### Discrete Sample Time

Recreate the network, but with a discrete sample time of 0.5, instead of continuous sampling.

```
gensim(net,0.5)
```

Again replace the constant input with a signal generator. Simulate the system and view the network's response.

E

# Code Notes

## Dimensions

The following code dimensions are used in describing both the network signals that users commonly see, and those used by the utility functions:

Ni = number of network inputs = net.numInputs

Ri = number of elements in input i = net.inputs{i}.size

Nl = number of layers = net.numLayers

Si = number of neurons in layer i = net.layers{i}.size

Nt = number of targets = net.numTargets

Vi = number of elements in target i, equal to Sj, where j is the ith layer with a target. (A layer n has a target if net.targets(n) == 1.)

No = number of network outputs = net.numOutputs

Ui = number of elements in output i, equal to Sj, where j is the ith layer with an output (A layer n has an output if net.outputs(n) == 1.)

ID = number of input delays = net.numInputDelays

LD = number of layer delays = net.numLayerDelays

TS = number of time steps

Q  = number of concurrent vectors or sequences.

# Variables

The variables a user commonly uses when defining a simulation or training session are

P

Network inputs.

Ni-by-TS cell array, each element P{i,ts} is an Ri-by-Q matrix.

Pi

Initial input delay conditions.

Ni-by-ID cell array, each element Pi{i,k} is an Ri-by-Q matrix.

Ai

Initial layer delay conditions.

Nl-by-LD cell array, each element Ai{i,k} is an Si-by-Q matrix.

T

Network targets.

Nt-by-TS cell array, each element P{i,ts} is an Vi-by-Q matrix.

These variables are returned by simulation and training

calls:

Y

Network outputs.

No-by-TS cell array, each element Y{i,ts} is a Ui-by-Q matrix.

E

Network errors.

Nt-by-TS cell array, each element P{i,ts} is an Vi-by-Q matrix.

perf

network performance

## Utility Function Variables

These variables are used only by the utility functions.

Pc

Combined inputs.

Ni-by-(ID+TS) cell array, each element P{i,ts} is an Ri-by-Q matrix.

Pc = [Pi P] = Initial input delay conditions and network inputs.

Pd

Delayed inputs.

Ni-by-Nj-by-TS cell array, each element Pd{i,j,ts} is an (Ri*IWD(i,j))-by-Q matrix, where IWD(i,j) is the number of delay taps associated with input weight to layer i from input j.

Equivalently, IWD(i,j) = length(net.inputWeights{i,j}.delays).

Pd is the result of passing the elements of P through each input weights tap delay lines. Since inputs are always transformed by input delays in the same way it saves time to only do that operation once, instead of for every training step.

BZ

Concurrent bias vectors.

Nl-by-1 cell array, each element BZ{i} is a Si-by-Q matrix.

Each matrix is simply Q copies of the net.b{i} bias vector.

IWZ

Weighted inputs.

Ni-by-Nl-by-TS cell array, each element IWZ{i,j,ts} is a Si-by--by-Q matrix.

LWZ

Weighed layer outputs.

Ni-by-Nl-by-TS cell array, each element LWZ{i,j,ts} is a Si-by-Q matrix.

N

Net inputs.

Ni-by-TS cell array, each element N{i,ts} is a Si-by-Q matrix.

A

Layer outputs.

Nl-by-TS cell array, each element A{i,ts} is a Si-by-Q matrix.

Ac

Combined layer outputs.

Nl-by-(LD+TS) cell array, each element A{i,ts} is a Si-by-Q matrix.

Ac = [Ai A] = Initial layer delay conditions and layer outputs.

Tl

Layer targets.

Nl-by-TS cell array, each element Tl{i,ts} is a Si-by-Q matrix.

Tl contains empty matrices [] in rows of layers i not associated with targets, indicated by net.targets(i) == 0.

El

Layer errors.

Nl-by-TS cell array, each element El{i,ts} is a Si-by-Q matrix.

El contains empty matrices [] in rows of layers i not associated with targets, indicated by net.targets(i) == 0.

X

Column vector of all weight and bias values.

# Functions

The following functions are the utility functions that you can call to perform a lot of the work of simulating or training a network. You can read about them in their respective help comments.

These functions calculate signals.

`calcpd`, `calca`, `calca1`, `calce`, `calce1`, `calcperf`

These functions calculate derivatives, Jacobians, and values associated with Jacobians.

`calcgx`, `calcjx`, `calcjejj`

`calcgx` is used for gradient algorithms; `calcjx` and `calcjejj` can be used for calculating approximations of the Hessian for algorithms like Levenberg-Marquardt.

These functions allow network weight and bias values to be accessed and altered in terms of a single vector X.

`setx`, `getx`, `formx`

## Code Efficiency

The functions sim, train, and adapt all convert a network object to a structure,

```
net = struct(net);
```

before simulation and training, and then recast the structure back to a network.

```
net = class(net,'network')
```

This is done for speed efficiency since structure fields are accessed directly, while object fields are accessed using the MATLAB® object method handling system. If users write any code that uses utility functions outside of sim, train, or adapt, they should use the same technique.

# Argument Checking

These functions are only recommended for advanced users.

None of the utility functions do any argument checking, which means that the only feedback you get from calling them with incorrectly sized arguments is an error.

The lack of argument checking allows these functions to run as fast as possible.

For “safer” simulation and training, use `sim`, `train` and `adapt`.

# Index

RA# 32118-1

Megger.

METROLOGY DEPARTMENT

# CERTIFICATE OF CALIBRATION

Megger Metrology Department certifies that the instrument identified below has been calibrated using reference standards whose accuracies are traceable to the National Institute of Standards and Technology (NIST) within the limitations of the Institute's calibration services, or have been derived from accepted values of natural physical constants and that quality assurance is maintained compatible with 10 CFR 50, Appendix "B" and ANSI/NCSL Z540-1. Megger shall not be held responsible for malfunctions or accuracy of instrument if misused, tampered with or subjected to conditions adverse to functional integrity.

The results presented in this report are expressed in terms of the NIST realization of the new 1990 representation of the volt and ohm, based on the Josephson and Quantum Hall effects, respectively.

This certificate shall not be reproduced except in full, without the written approval of Megger.

P0# P0-02982-MEG

INSTRUMENT Relay Test Set

MANUFACTURER Megger MODEL NO. SMRT36-30P0F0A1S1

SERIAL NO. 201110070014 TOLERANCE MFG. SPECS

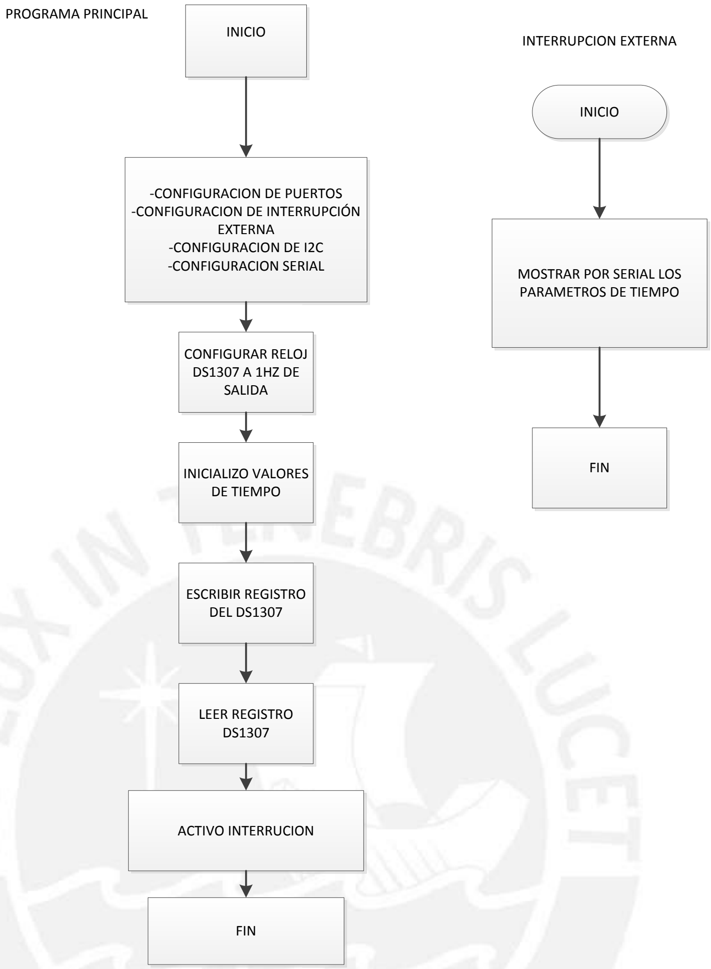
LOCATION OEM Electric SAC

DATE CALIBRATED 10/24/2013 DUE DATE 10/24/2014

CALIBRATED BY Kelly Combs LEVEL III

APPROVED BY

## ANEXO 14- Diagrama de flujo de prueba del reloj DS1307


PROGRAMA PRINCIPAL
INICIO
-CONFIGURACION DE PUERTOS
-CONFIGURACION DE INTERRUPCIÓN EXTERNA
-CONFIGURACION DE I2C
-CONFIGURACION SERIAL
CONFIGURAR RELOJ DS1307 A 1HZ DE SALIDA
INICIALIZO VALORES DE TIEMPO
ESCRIBIR REGISTRO DEL DS1307
LEER REGISTRO DS1307
ACTIVO INTERRUCION
FIN
INTERRUPCION EXTERNA
INICIO
MOSTRAR POR SERIAL LOS PARAMETROS DE TIEMPO
FIN

# ANEXO 15 –Prueba con el módulo Bluetooth

**Zigbee vs Bluetooth.**

Ahora, como se cuenta con un módulo Bluetooth, se va realizar una prueba comparativa con los módulos Xbee, en el aspecto del ruido.

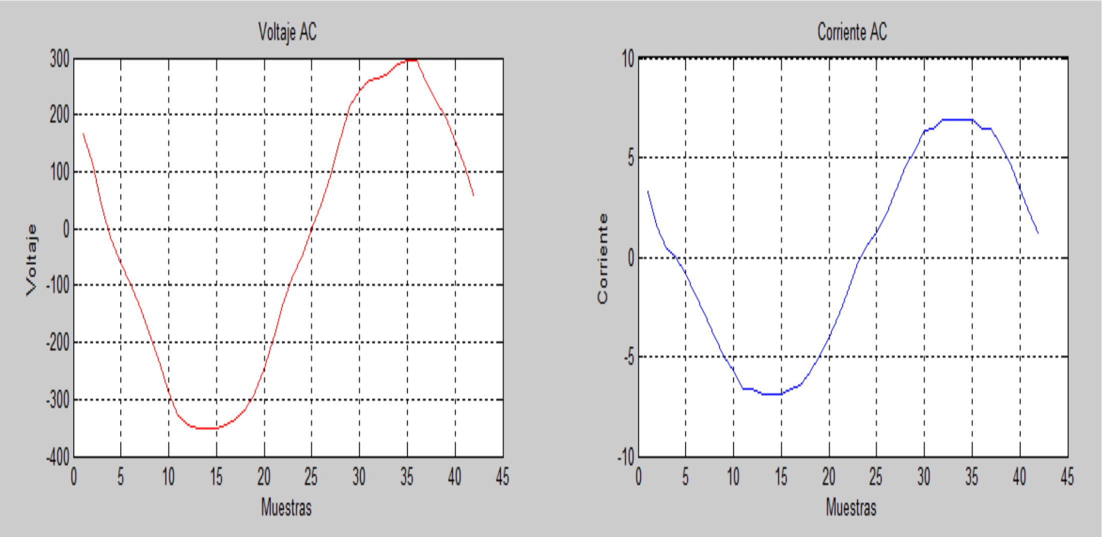


**Fig4.24: Señal obtenida por medio del módulo Bluetooth**

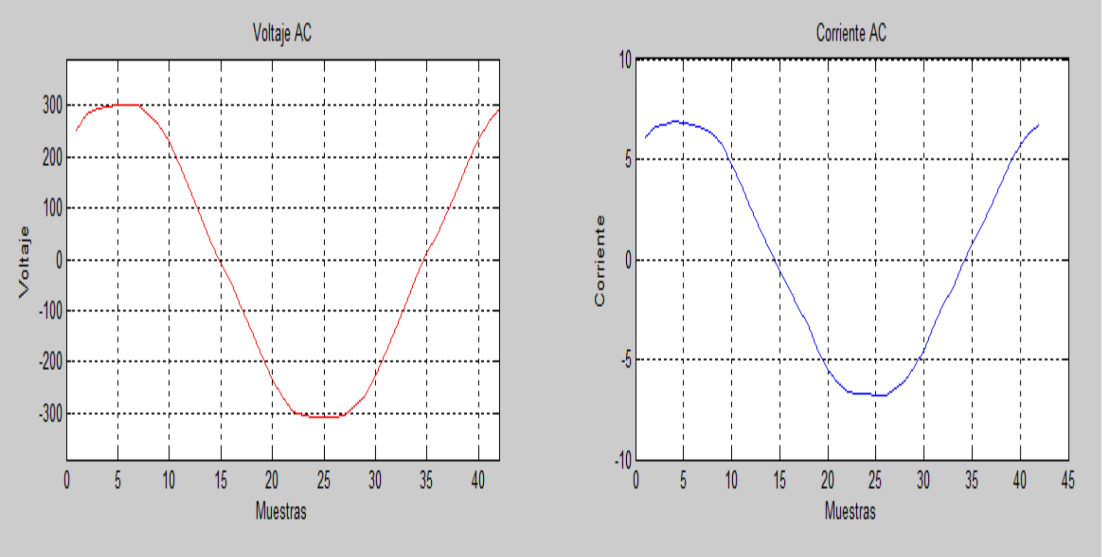


**Fig4.25: Señal obtenida por medio de los módulos Xbee**

Como se puede observar en las gráficas, hay mayor inmunidad al ruido mediante el protocolo Zigbee, ya que por el tipo de modulación el Zigbee se presenta como más robusto que el Bluetooth.

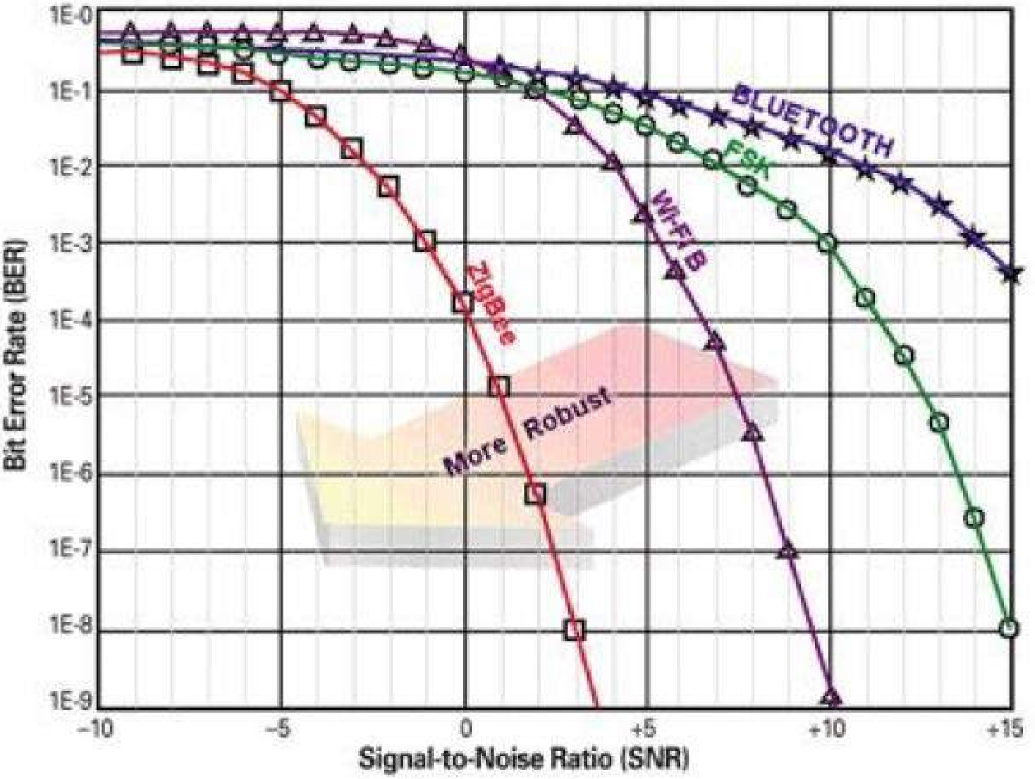


**Fig4.10: Gráfico comparativo de la relación señal a ruido en función del BER**

De la figura anterior se puede corroborar con las pruebas realizadas que el protocolo Zigbee presenta una mayor inmunidad al ruido frente al protocolo Bluetooth.

Resolución vs Número de muestras

A continuación, se muestran dos gráficas obtenidas en la captura de datos por serial y graficadas en el programa MATLAB, la diferencia radica en la resolución y el número de muestras que se obtiene para un periodo.

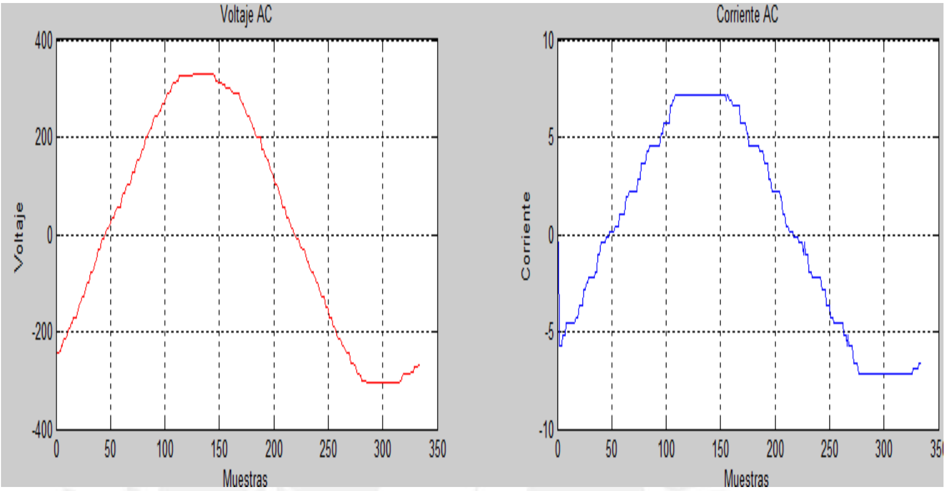


**Fig4.10: Señal de voltaje y corriente AC graficada en el programa Matlab**

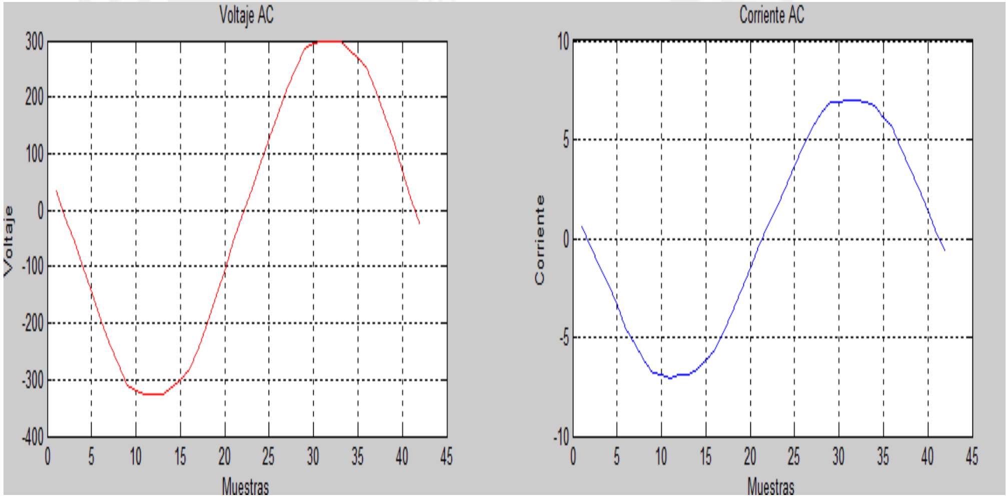


**Fig4.11: Señal de voltaje y corriente AC graficada en el programa Matlab**

Se realizó la implementación de ambos sensores con la finalidad de obtener mejores resultados, tal como se observan en la figura 24. La señal de voltaje es obtenida a través de la línea 220VRMS y la señal de corriente corresponde a la corriente consumida en este caso por una plancha. En ambas señales se usó el conversor ADC interno del microcontrolador. En la figura 23 se usó una resolución de 8 bits, con la finalidad de poder obtener más muestras Mientras que en la figura 24 se empleó una resolución de 10 bits, pero con menor número de muestras Se puede observar que se obtiene una mejor forma de onda en la figura 24 sobre todo en la onda de corriente.

# HC-SR501 PIR MOTION DETECTOR

## Product Discription

HC-SR501 is based on infrared technology, automatic control module, using Germany imported LHI778 probe design, high sensitivity, high reliability, ultra-low-voltage operating mode, widely used in various auto-sensing electrical equipment, especially for battery-powered automatic controlled products.

### Specification:

- Voltage: 5V – 20V
- Power Consumption: 65mA
- TTL output: 3.3V, 0V
- Delay time: Adjustable (.3->5min)
- Lock time: 0.2 sec
- Trigger methods: L – disable repeat trigger, H enable repeat trigger
- Sensing range: less than 120 degree, within 7 meters
- Temperature: – 15 ~ +70
- Dimension: 32*24 mm, distance between screw 28mm, M2, Lens dimension in diameter: 23mm

### Application:

Automatically sensing light for Floor, bathroom, basement, porch, warehouse, Garage, etc, ventilator, alarm, etc.

### Features:

- Automatic induction: to enter the sensing range of the output is high, the person leaves the sensing range of the automatic delay off high, output low.
- Photosensitive control (optional, not factory-set) can be set photosensitive control, day or light intensity without induction.
- Temperature compensation (optional, factory reset): In the summer when the ambient temperature rises to 30 ° C to 32 ° C, the detection distance is slightly shorter, temperature compensation can be used for performance compensation.
- Triggered in two ways: (jumper selectable)
  - non-repeatable trigger: the sensor output high, the delay time is over, the output is automatically changed from high level to low level;
  - repeatable trigger: the sensor output high, the delay period, if there is human activity in its sensing range, the output will always remain high until the people left after the delay will be high level goes low (sensor module detects a time delay period will be automatically extended every human activity, and the starting point for the delay time to the last event of the time).
- With induction blocking time (the default setting: 2.5s blocked time): sensor module after each sensor output (high into low), followed by a blockade set period of time, during this time period sensor does not accept any sensor signal. This feature can be achieved sensor output time “and” blocking time “interval between the work can be applied to interval detection products; This function can inhibit a variety of interference in the process of load switching. (This time can be set at zero seconds – a few tens of seconds).
- Wide operating voltage range: default voltage DC4.5V-20V.
- Micropower consumption: static current <50 microamps, particularly suitable for battery-powered automatic control products.
- Output high signal: easy to achieve docking with the various types of circuit.

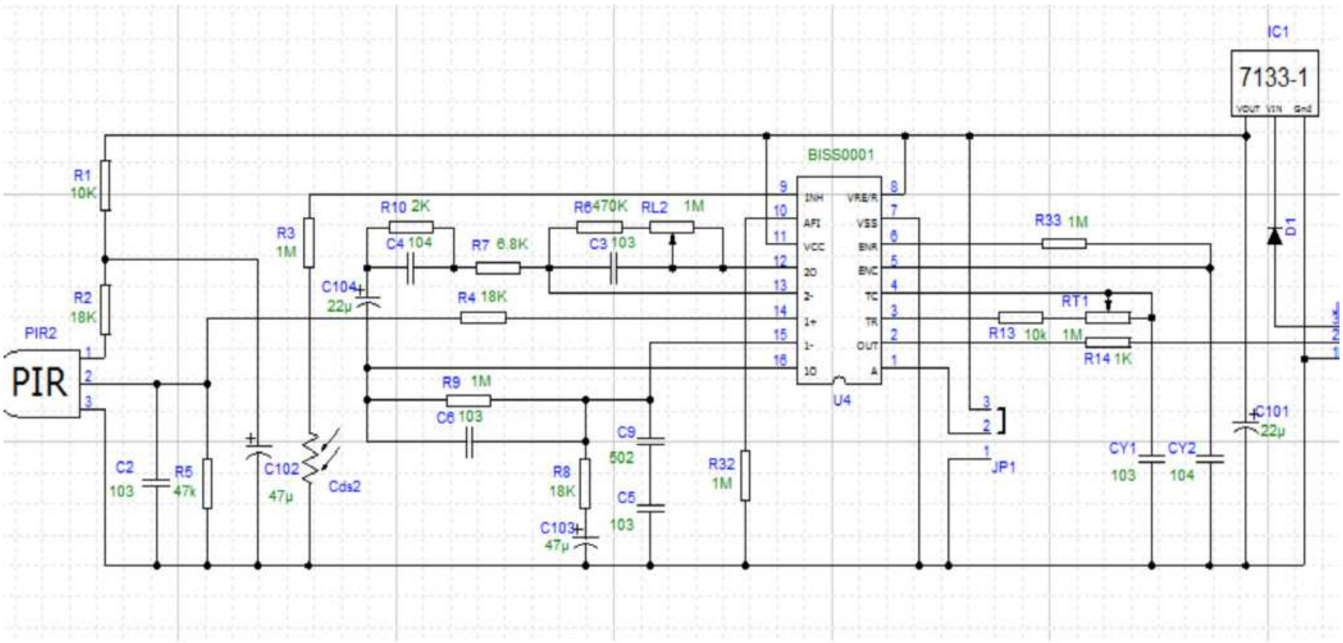
### Adjustment:

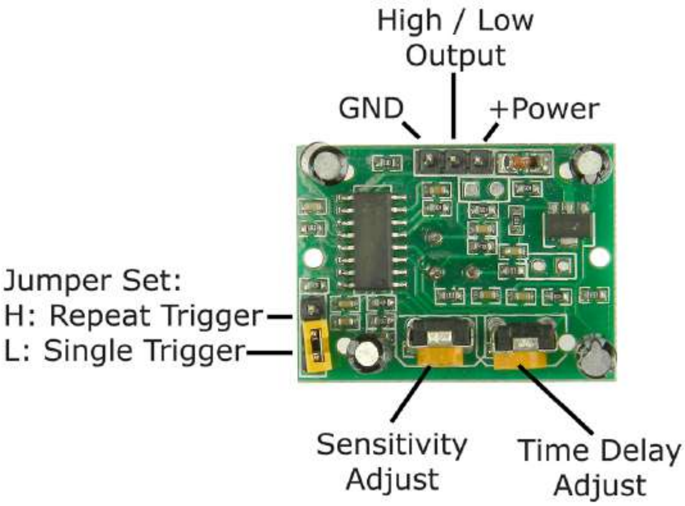
- Adjust the distance potentiometer clockwise rotation, increased sensing distance (about 7 meters), on the contrary, the sensing distance decreases (about 3 meters).
- Adjust the delay potentiometer clockwise rotation sensor the delay lengthened (300S), on the contrary, shorten the induction delay (5S).

### Instructions for use:

- Sensor module is powered up after a minute, in this initialization time intervals during this module will output 0-3 times, a minute later enters the standby state.
- Should try to avoid the lights and other sources of interference close direct module surface of the lens, in order to avoid the introduction of interference signal malfunction; environment should avoid the wind flow, the wind will cause interference on the sensor.
- Sensor module with dual probe, the probe window is rectangular, dual (A B) in both ends of the longitudinal direction
  - so when the human body from left to right or right to left through the infrared spectrum to reach dual time, distance difference, the greater the difference, the more sensitive the sensor,
  - when the human body from the front to the probe or from top to bottom or from bottom to top on the direction traveled, double detects changes in the distance of less than infrared spectroscopy, no difference value the sensor insensitive or does not work;
- The dual direction of sensor should be installed parallel as far as possible in inline with human movement. In order to increase the sensor angle range, the module using a circular lens also makes the probe surrounded induction, but the left and right sides still up and down in both directions sensing range, sensitivity, still need to try to install the above requirements.

# HC-SR501 PIR MOTION DETECTOR





1 working voltage range :DC 4.5-20V

2 Quiescent Current :50uA

3 high output level 3.3 V / Low 0V

4. Trigger L trigger can not be repeated / H repeated trigger

5. circuit board dimensions :32 * 24 mm

6. maximum 110 ° angle sensor

7. 7 m maximum sensing distance

| Product Type | HC--SR501 Body Sensor Module |
|---|---|
| Operating Voltage Range | 5-20VDC |
| Quiescent Current | <50uA |
| Level output | High 3.3 V /Low 0V |
| Trigger | L can not be repeated trigger/H can be repeated trigger(Default repeated trigger) |
| Delay time | 5-300S( adjustable) Range (approximately .3Sec -5Min) |
| Block time | 2.5S(default)Can be made a range(0.xx to tens of seconds |
| Board Dimensions | 32mm*24mm |
| Angle Sensor | <110 ° cone angle |
| Operation Temp. | -15-+70 degrees |
| Lens size sensor | Diameter:23mm(Default) |

Application scope

•Security products

•Body induction toys

•Body induction lamps

•Industrial automation control etc

Pyroelectric infrared switch is a passive infrared switch which consists of BISS0001 ,pyroelectric infrared sensors and a few external components. It can au
open all kinds of equipments, inculding incandescent lamp, fluorescent lamp, intercom, automatic, electric fan, dryer and automatic washing machine, etc.
It is widely used in enterprises, hotels, stores, and corridor and other sensitive area for automatical lamplight, lighting and alarm system.

Instructions

Induction module needs a minute or so to initialize. During initializing time, it will output 0-3 times. One minute later it comes into standby.
Keep the surface of the lens from close lighting source and wind, which will introduce interference.
Induction module has double -probe whose window is rectangle. The two sub-probe (A and B) is located at the two ends of rectangle. When human body r
to right, or from right to left, Time for IR to reach to reach the two sub-probes differs.The lager the time diffference is, the more sensitive this module is. Wh
body moves face-to probe, or up to down, or down to up, there is no time difference. So it does not work. So instal the module in the direction in which mos
activities behaves, to guarantee the induction of human by dual sub-probes. In order to increase the induction range, this module uses round lens which ca
from all direction. However, induction from right or left is more sensitivity than from up or down.