



Chipsmall Limited consists of a professional team with an average of over 10 year of expertise in the distribution of electronic components. Based in Hongkong, we have already established firm and mutual-benefit business relationships with customers from,Europe,America and south Asia,supplying obsolete and hard-to-find components to meet their specific needs.

With the principle of “Quality Parts,Customers Priority,Honest Operation,and Considerate Service”,our business mainly focus on the distribution of electronic components. Line cards we deal with include Microchip,ALPS,ROHM,Xilinx,Pulse,ON,Everlight and Freescale. Main products comprise IC,Modules,Potentiometer,IC Socket,Relay,Connector.Our parts cover such applications as commercial,industrial, and automotives areas.

We are looking forward to setting up business relationship with you and hope to provide you with the best service and solution. Let us make a better world for our industry!



Contact us

Tel: +86-755-8981 8866 Fax: +86-755-8427 6832

Email & Skype: info@chipsmall.com Web: www.chipsmall.com

Address: A1208, Overseas Decoration Building, #122 Zhenhua RD., Futian, Shenzhen, China



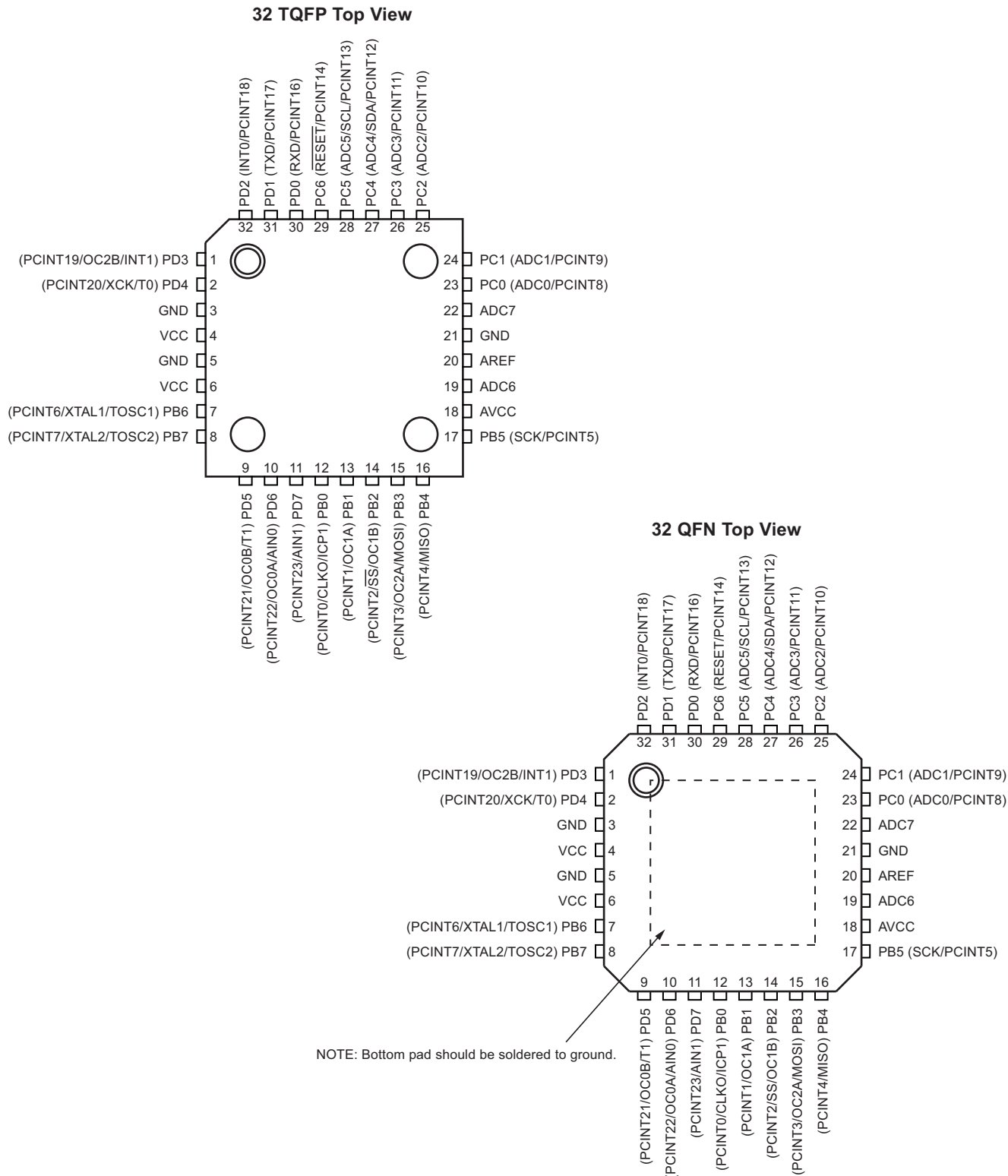
Features

- 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 I²C 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 $+125^{\circ}\text{C}$
- 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: 0.8 μA

1. Pin Configurations

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



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 [Section 14.3.1 “Alternate Functions of Port B” on page 71](#) and [Section 9. “System Clock and Clock Options” on page 24](#).

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 272](#). Shorter pulses are not guaranteed to generate a reset.

The various special features of port C are elaborated in [Section 14.3.2 “Alternate Functions of Port C” on page 74](#).

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 [Section 14.3.3 “Alternate Functions of Port D” on page 76](#).

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)

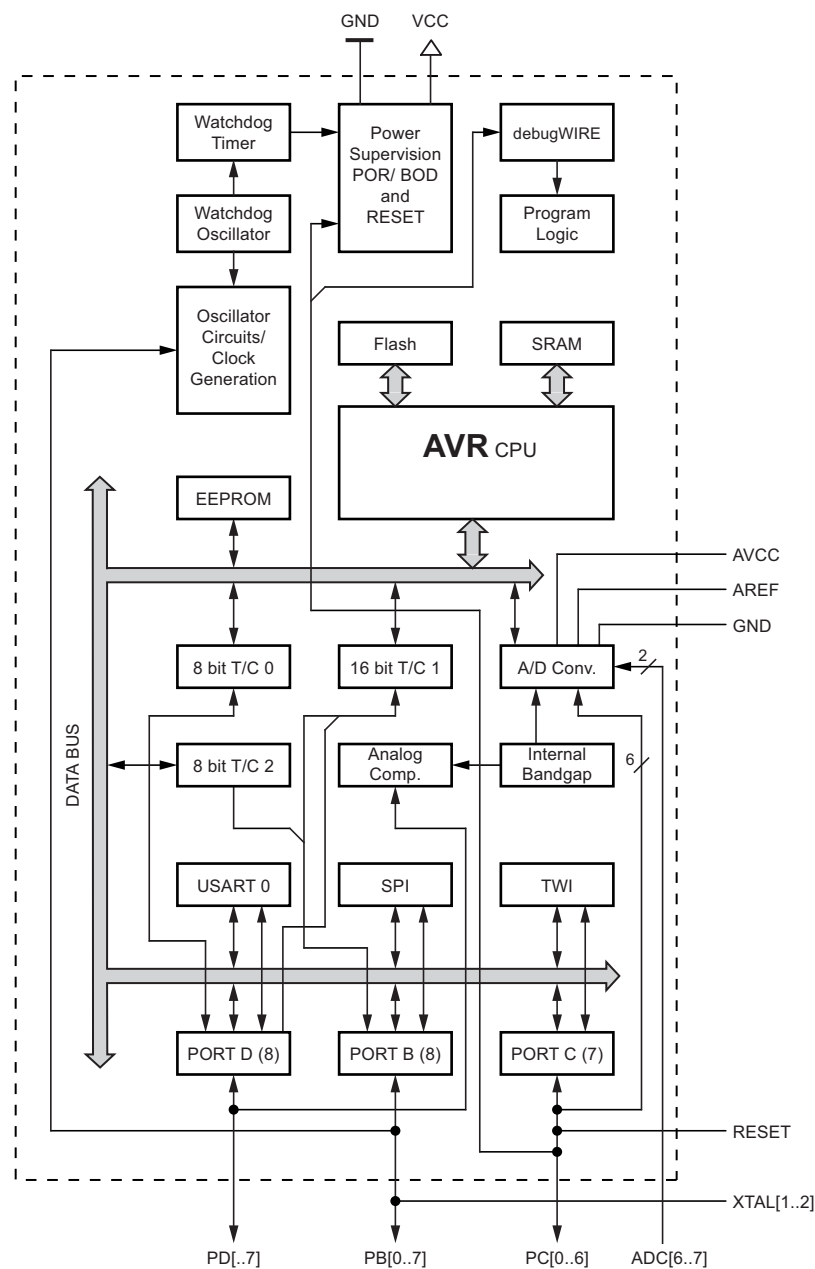
In the TQFP and QFN package, ADC7:6 serve as analog inputs to the A/D converter. These pins are powered from the analog supply and serve as 10-bit ADC channels.

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 1MIPS per MHz 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/8Kbytes of in-system programmable flash with read-while-write capabilities, 256/512/512 bytes EEPROM, 512/1K/1Kbytes 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 the Atmel 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>.

Note: 1.

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.

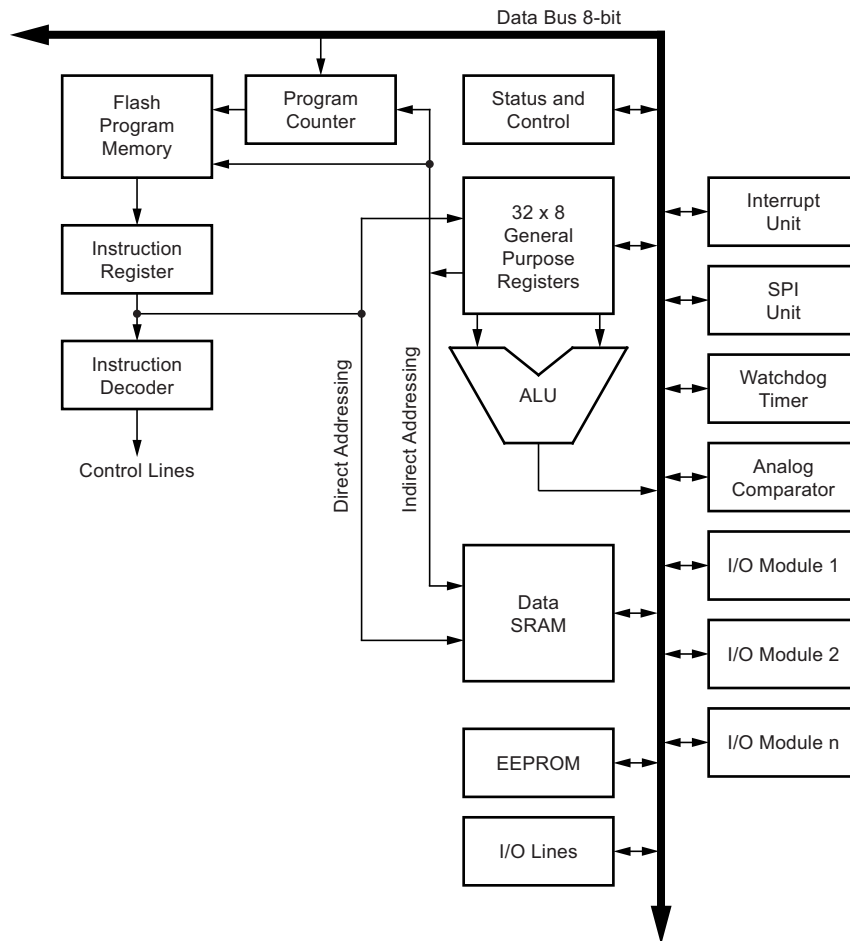
For I/O registers located in extended I/O map, “IN”, “OUT”, “SBIS”, “SBIC”, “CBI”, and “SBI” instructions must be replaced with instructions that allow access to extended I/O. Typically “LDS” and “STS” combined with “SBR”, “SBRC”, “SBR”, and “CBR”.

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 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 [Section 32. “Instruction Set Summary” on page 317](#) 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 [Section 32. “Instruction Set Summary” on page 317](#) for detailed information.

- **Bit 4 – S: Sign Bit, $S = N \hat{A} V$**

The S-bit is always an exclusive or between the negative flag N and the two's complement overflow flag V. See [Section 32. “Instruction Set Summary” on page 317](#) for detailed information.

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

The two's complement overflow flag V supports two's complement arithmetic. See [Section 32. “Instruction Set Summary” on page 317](#) for detailed information.

- **Bit 2 – N: Negative Flag**

The negative flag N indicates a negative result in an arithmetic or logic operation. See [Section 32. “Instruction Set Summary” on page 317](#) for detailed information.

- **Bit 1 – Z: Zero Flag**

The zero flag Z indicates a zero result in an arithmetic or logic operation. See [Section 32. “Instruction Set Summary” on page 317](#) for detailed information.

- **Bit 0 – C: Carry Flag**

The carry flag C indicates a carry in an arithmetic or logic operation. See [Section 32. “Instruction Set Summary” on page 317](#) 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.	
General Purpose Working Registers	R0		0x00	
	R1		0x01	
	R2		0x02	
	...			
	R13		0x0D	
	R14		0x0E	
	R15		0x0F	
	R16		0x10	
	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 [Section 32. "Instruction Set Summary" on page 317](#)).

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 17](#).

See [Table 7-1](#) for stack pointer details.

Table 7-1. Stack Pointer Instructions

Instruction	Stack pointer	Description
PUSH	Decrement by 1	Data is pushed onto the stack
CALL ICALL RCALL	Decrement by 2	Return address is pushed onto the stack with a subroutine call or interrupt
POP	Increment by 1	Data is popped from the stack
RET RETI	Increment 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 clk_{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 1MIPS per MHz 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

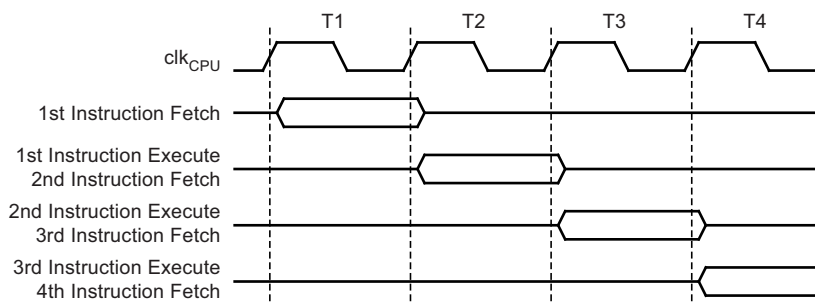
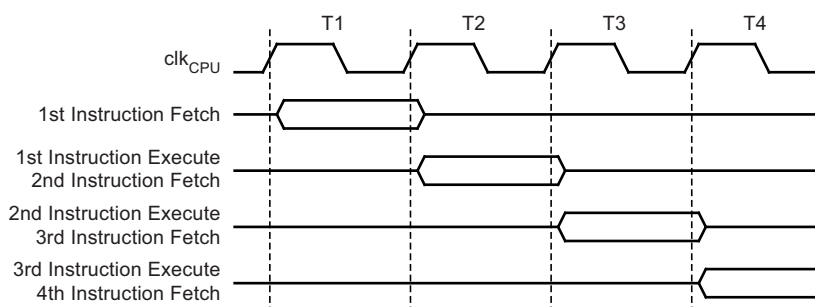


Figure 7-5 shows the internal timing concept for the register file. In a single clock cycle an ALU operation using two register operands is executed, and the result is stored back to the destination register.

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 [Section 28. “Memory Programming” on page 251](#) 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 [Section 12. “Interrupts” on page 50](#). 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 [Section 12. “Interrupts” on page 50](#) for more information. The reset vector can also be moved to the start of the boot flash section by programming the BOOTRST fuse, see [Section 27. “Boot Loader Support – Read-While-Write Self-Programming” on page 237](#).

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	
<pre> 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) */ </pre>	

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

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 27.9.1 “SPMCSR – Store Program Memory Control and Status Register” on page 249](#) 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 [Section 26. “Self-Programming the Flash, Atmel ATmega48PA” on page 231](#) and [Section 27. “Boot Loader Support – Read-While-Write Self-Programming” on page 237](#). [Section 28. “Memory Programming” on page 251](#) 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 [Section 7.6 “Instruction Execution Timing” on page 13](#).

Figure 8-1. Program Memory Map Atmel ATmega48PA

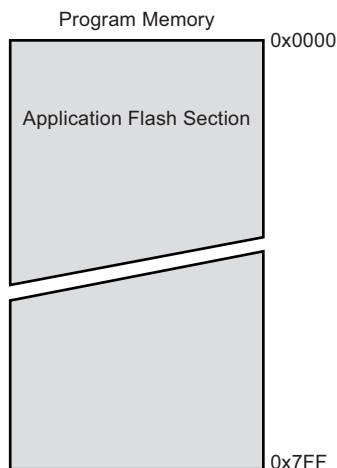
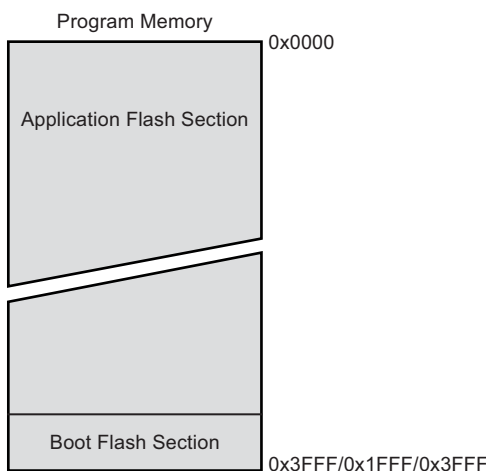


Figure 8-2. Program Memory Map Atmel ATmega88PA, Atmel ATmega168PA



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.

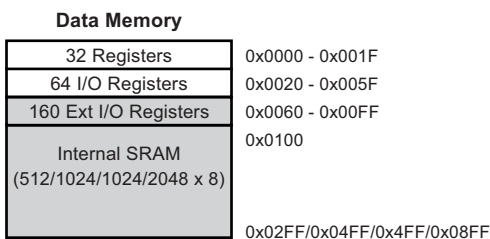
The direct addressing reaches the entire data space.

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 [Section 7.4 "General Purpose Register File" on page 11](#).

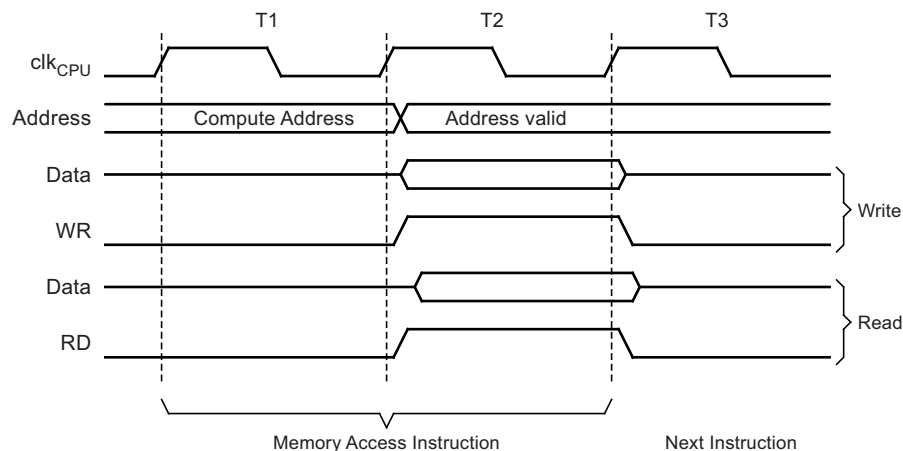
Figure 8-3. Data Memory Map



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.

[Section 28. "Memory Programming" on page 251](#) 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 on page 22](#). 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 [Section 8.4.2 "Preventing EEPROM Corruption" on page 19](#) 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 [Section 31. “Register Summary” on page 310](#).

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 AVR, 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 EEPF 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 EEPF to one causes the EEPROM to be written. When EEMPE is set, setting EEPF within four clock cycles will write data to the EEPROM at the selected address. If EEMPE is zero, setting EEPF 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 EEPF bit for an EEPROM write procedure.

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

The EEPROM write enable signal EEPF is the write strobe to the EEPROM. When address and data are correctly set up, the EEPF 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 EEPF, 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 EEPF 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 EEPF in EECR.
6. Within four clock cycles after setting EEMPE, write a logical one to EEPF.

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 [Section 27: “Boot Loader Support – Read-While-Write Self-Programming” on page 237](#) 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 EEPF bit is cleared by hardware. The user software can poll this bit and wait for a zero before writing the next byte. When EEPF 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 EEPF 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 (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
<pre> 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 </pre>
C Code Example
<pre> 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); } </pre>

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	
<pre> 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 </pre>	
C Code Example	
<pre> 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; } </pre>	

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

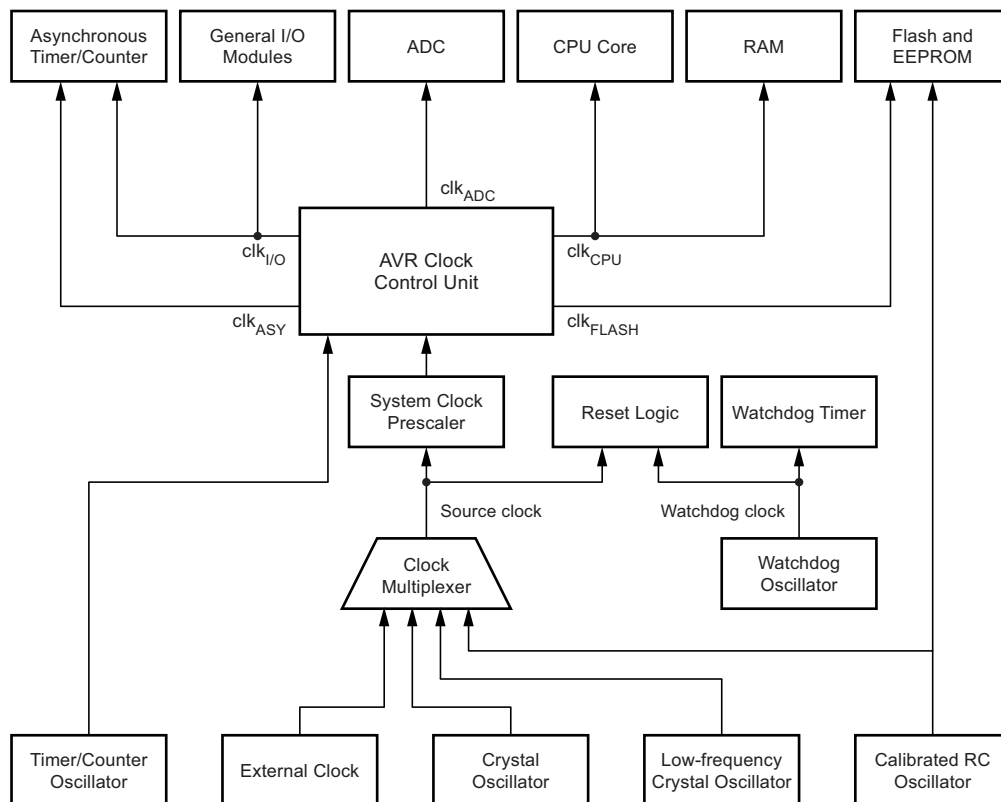
Bit	7	6	5	4	3	2	1	0	
0x1E (0x3E)	MSB							LSB	GPIOR0
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

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 [Section 10. “Power Management and Sleep Modes” on page 35](#). 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 [Section 9. “System Clock and Clock Options” on page 24](#).

9.1.3 Flash Clock – $\text{clk}_{\text{FLASH}}$

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_{TOUIT}) after the device reset is released by all other reset sources. [Section 11. “System Control and Reset” on page 41](#) describes the start conditions for the internal reset. The delay (t_{TOUIT}) 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 [Section 30. “Typical Characteristics” on page 279](#).

Table 9-2. Number of Watchdog Oscillator Cycles

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