



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
 - 130 Powerful Instructions – Most Single Clock Cycle Execution
 - 32 x 8 General Purpose Working Registers
 - Fully Static Operation
 - Up to 16 MIPS Throughput at 16 MHz
 - On-Chip 2-cycle Multiplier
- Non-volatile Program and Data Memories
 - 16K bytes of In-System Self-Programmable Flash
Endurance: 10,000 Write/Erase Cycles
 - Optional Boot Code Section with Independent Lock Bits
In-System Programming by On-chip Boot Program
True Read-While-Write Operation
 - 512 bytes EEPROM
Endurance: 100,000 Write/Erase Cycles
 - 1K byte Internal SRAM
 - Programming Lock for Software Security
- JTAG (IEEE std. 1149.1 compliant) Interface
 - Boundary-scan Capabilities According to the JTAG Standard
 - Extensive On-chip Debug Support
 - Programming of Flash, EEPROM, Fuses, and Lock Bits through the JTAG Interface
- 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
 - Four PWM Channels
 - 8-channel, 10-bit ADC
 - Programmable Serial USART
 - Master/Slave SPI Serial Interface
 - Universal Serial Interface with Start Condition Detector
 - 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
 - Five Sleep Modes: Idle, ADC Noise Reduction, Power-save, Power-down, and Standby
- I/O and Packages
 - 53 Programmable I/O Lines
 - 64-lead TQFP and 64-pad QFN/MLF
- Speed Grade:
 - ATmega165V: 0 - 4 MHz @ 1.8 - 5.5V, 0 - 8 MHz @ 2.7 - 5.5V
 - ATmega165: 0 - 8 MHz @ 2.7 - 5.5V, 0 - 16 MHz @ 4.5 - 5.5V
- Temperature range:
 - -40°C to 85°C Industrial
- Ultra-Low Power Consumption
 - Active Mode:
 - 1 MHz, 1.8V: 350µA
 - 32 kHz, 1.8V: 20µA (including Oscillator)
 - Power-down Mode:
 - 0.1µA at 1.8V



8-bit AVR[®] Microcontroller with 16K Bytes In-System Programmable Flash

ATmega165V
ATmega165

Preliminary

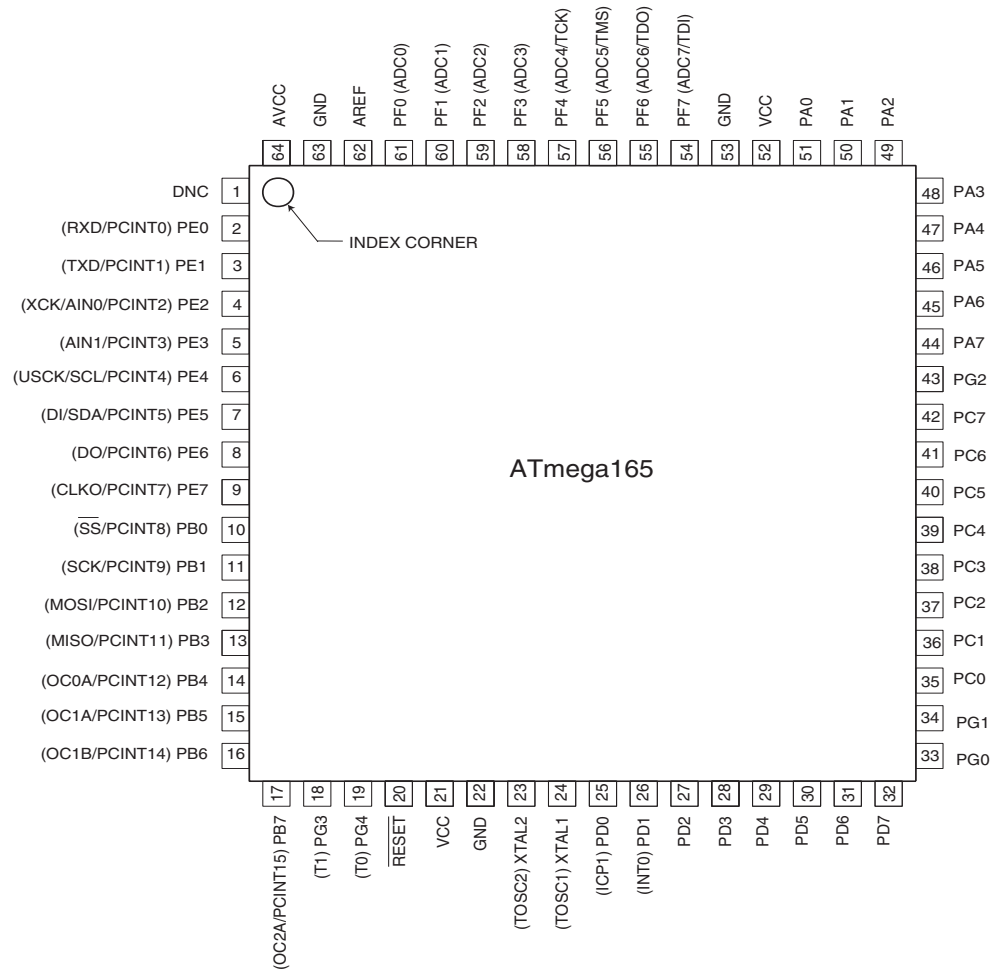
Notice:
Not recommended in new
designs.

2573G-AVR-07/09



Pin Configurations

Figure 1. Pinout ATmega165



Note: The large center pad underneath the QFN/MLF packages is made of metal and internally connected to GND. It should be soldered or glued to the board to ensure good mechanical stability. If the center pad is left unconnected, the package might loosen from the board.

Disclaimer

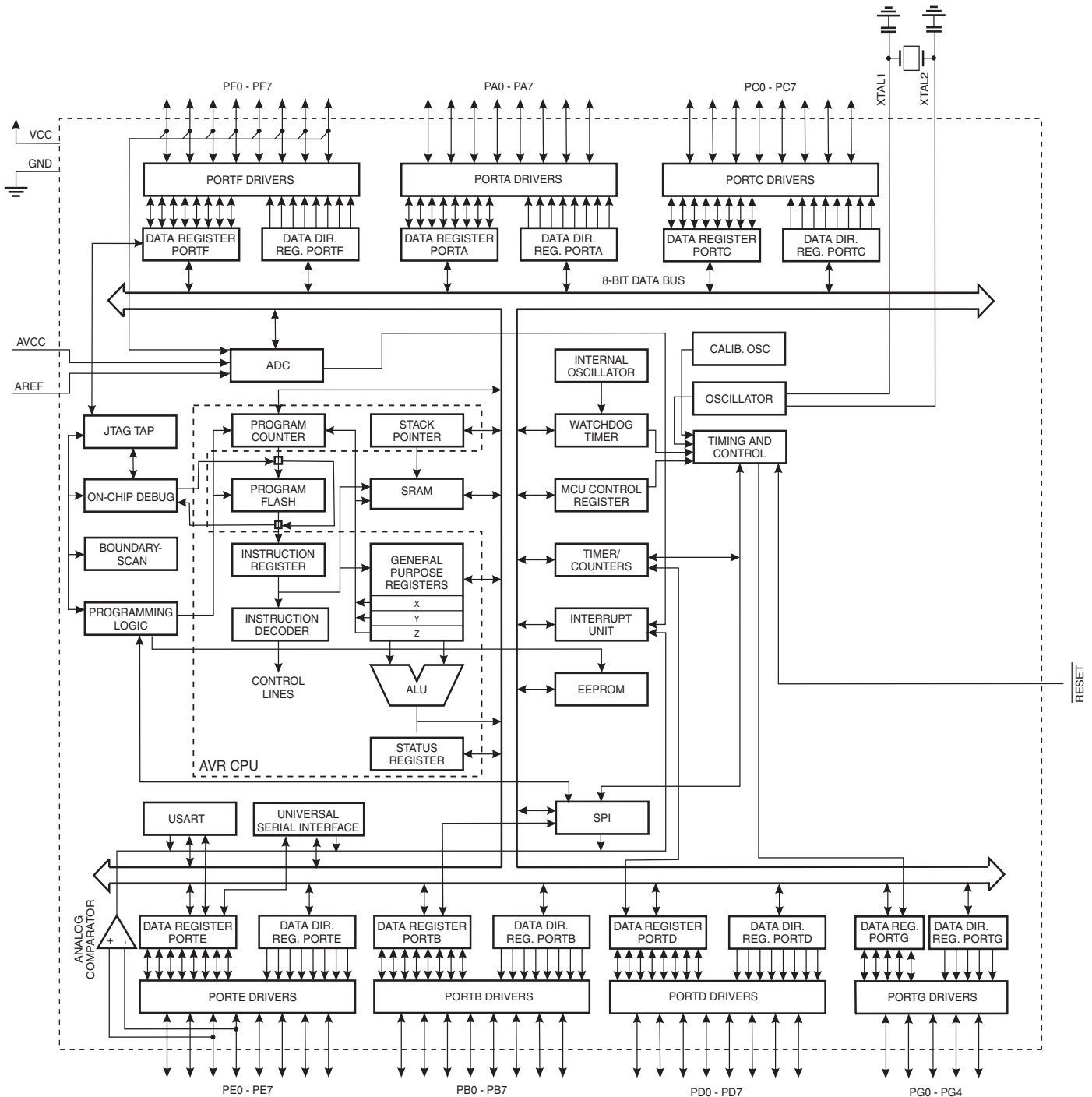
Typical values contained in this datasheet are based on simulations and characterization of other AVR microcontrollers manufactured on the same process technology. Min and Max values will be available after the device is characterized.

Overview

The ATmega165 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 ATmega165 achieves throughputs approaching 1 MIPS per MHz allowing the system designer to optimize power consumption versus processing speed.

Block Diagram

Figure 2. 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 ATmega165 provides the following features: 16K bytes of In-System Programmable Flash with Read-While-Write capabilities, 512 bytes EEPROM, 1K byte SRAM, 54 general purpose I/O lines, 32 general purpose working registers, a JTAG interface for Boundary-scan, On-chip Debugging support and programming, three flexible Timer/Counters with compare modes, internal and external interrupts, a serial programmable USART, Universal Serial Interface with Start Condition Detector, an 8-channel, 10-bit ADC, a programmable Watchdog Timer with internal Oscillator, an SPI serial port, and five software selectable power saving modes. The Idle mode stops the CPU while allowing the SRAM, Timer/Counters, 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 ATmega165 is a powerful microcontroller that provides a highly flexible and cost effective solution to many embedded control applications.

The ATmega165 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.

Pin Descriptions

VCC	Digital supply voltage.
GND	Ground.
Port A (PA7..PA0)	<p>Port A is an 8-bit bi-directional I/O port with internal pull-up resistors (selected for each bit). The Port A output buffers have symmetrical drive characteristics with both high sink and source capability. As inputs, Port A pins that are externally pulled low will source current if the pull-up resistors are activated. The Port A pins are tri-stated when a reset condition becomes active, even if the clock is not running.</p>
Port B (PB7..PB0)	<p>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.</p> <p>Port B has better driving capabilities than the other ports.</p> <p>Port B also serves the functions of various special features of the ATmega165 as listed on page 62.</p>
Port C (PC7..PC0)	<p>Port C is an 8-bit bi-directional I/O port with internal pull-up resistors (selected for each bit). The Port C 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.</p>
Port D (PD7..PD0)	<p>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.</p> <p>Port D also serves the functions of various special features of the ATmega165 as listed on page 65.</p>
Port E (PE7..PE0)	<p>Port E is an 8-bit bi-directional I/O port with internal pull-up resistors (selected for each bit). The Port E output buffers have symmetrical drive characteristics with both high sink and source capability. As inputs, Port E pins that are externally pulled low will source current if the pull-up resistors are activated. The Port E pins are tri-stated when a reset condition becomes active, even if the clock is not running.</p> <p>Port E also serves the functions of various special features of the ATmega165 as listed on page 66.</p>
Port F (PF7..PF0)	<p>Port F serves as the analog inputs to the A/D Converter.</p> <p>Port F also serves as an 8-bit bi-directional I/O port, if the A/D Converter is not used. Port pins can provide internal pull-up resistors (selected for each bit). The Port F output buffers have symmetrical drive characteristics with both high sink and source capability. As inputs, Port F pins that are externally pulled low will source current if the pull-up resistors are activated. The Port F pins are tri-stated when a reset condition becomes active, even if the clock is not running. If the JTAG interface is enabled, the pull-up resis-</p>

tors on pins PF7(TDI), PF5(TMS), and PF4(TCK) will be activated even if a reset occurs.

Port F also serves the functions of the JTAG interface.

Port G (PG4..PG0)

Port G is a 5-bit bi-directional I/O port with internal pull-up resistors (selected for each bit). The Port G output buffers have symmetrical drive characteristics with both high sink and source capability. As inputs, Port G pins that are externally pulled low will source current if the pull-up resistors are activated. The Port G pins are tri-stated when a reset condition becomes active, even if the clock is not running.

Port G also serves the functions of various special features of the ATmega165 as listed on page 66.

RESET

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 16 on page 38. Shorter pulses are not guaranteed to generate a reset.

XTAL1

Input to the inverting Oscillator amplifier and input to the internal clock operating circuit.

XTAL2

Output from the inverting Oscillator amplifier.

AVCC

AVCC is the supply voltage pin for Port F and the A/D Converter. 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.

AREF

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

About Code Examples

This documentation contains simple code examples that briefly show how to use various parts of the device. 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.

These code examples assume that the part specific header file is included before compilation. 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".

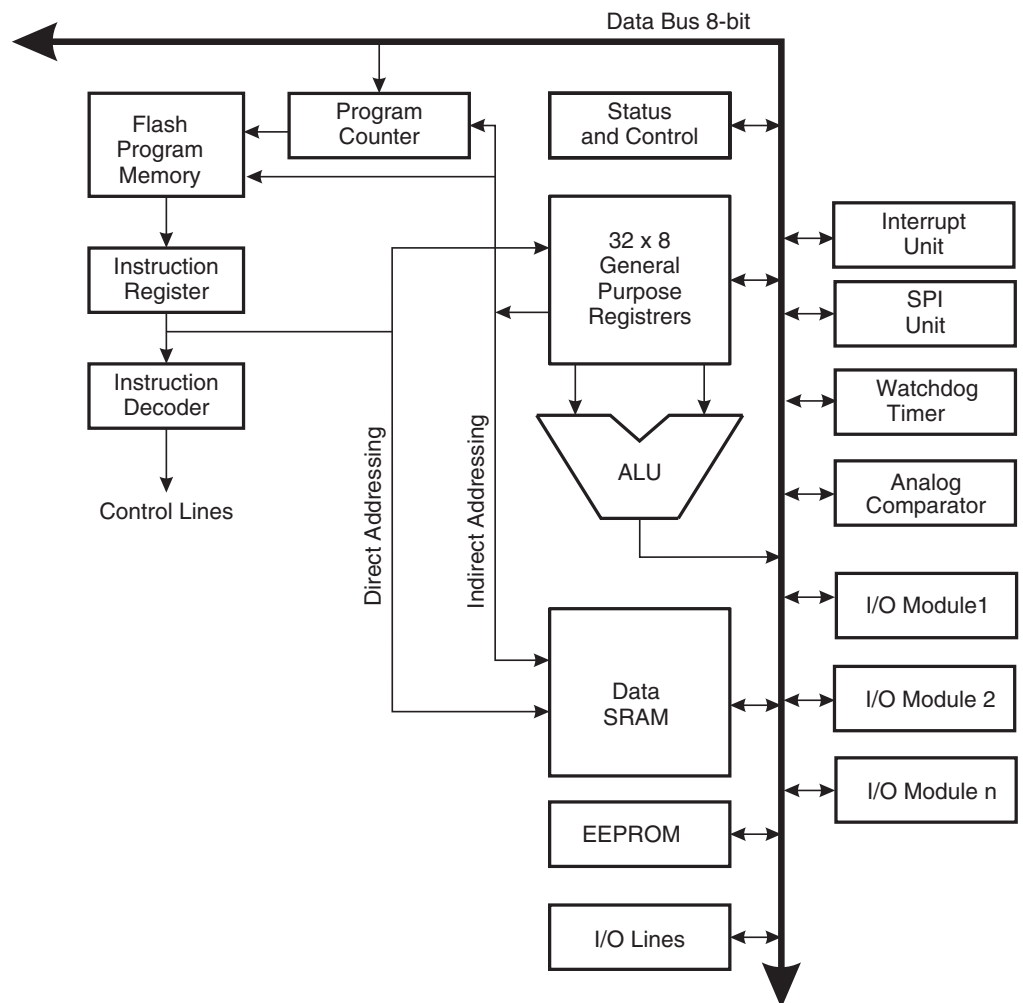
AVR CPU Core

Introduction

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.

Architectural Overview

Figure 3. 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 ATmega165 has Extended I/O space from 0x60 - 0xFF in SRAM where only the ST/STS/STD and LD/LDS/LDD instructions can be used.

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.

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.

The AVR Status Register – SREG – is defined as:

Bit	7	6	5	4	3	2	1	0	
	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, $S = N \oplus V$**

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 arithmetics. 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.

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 4 shows the structure of the 32 general purpose working registers in the CPU.

Figure 4. 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

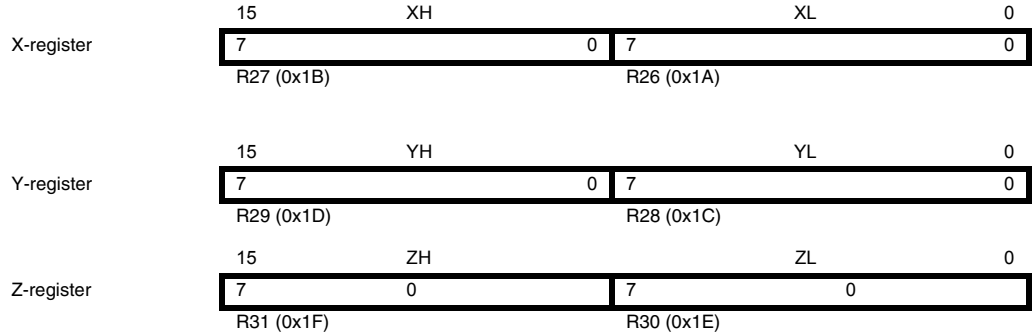
Most of the instructions operating on the Register File have direct access to all registers, and most of them are single cycle instructions.

As shown in Figure 4, 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.

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 5.

Figure 5. 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).

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. The Stack Pointer Register always points to the top of the Stack. Note that the Stack is implemented as growing from higher memory locations to lower memory locations. This implies that a Stack PUSH command decreases the Stack Pointer.

The Stack Pointer points to the data SRAM Stack area where the Subroutine and Interrupt Stacks are located. This Stack space in the data SRAM must be defined by the program before any subroutine calls are executed or interrupts are enabled. The Stack Pointer must be set to point above 0xFF. The Stack Pointer is decremented by one when data is pushed onto the Stack with the PUSH instruction, and it is decremented by two when the return address is pushed onto the Stack with subroutine call or interrupt. The Stack Pointer is incremented by one when data is popped from the Stack with the POP instruction, and it is incremented by two when data is popped from the Stack with return from subroutine RET or return from interrupt RETI.

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.

Bit	15	14	13	12	11	10	9	8	
	–	–	–	–	–	SP10	SP9	SP8	SPH
	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	0	0	0	0	0	0	0	0	
	0	0	0	0	0	0	0	0	

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 6 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 per MHz with the corresponding unique results for functions per cost, functions per clocks, and functions per power-unit.

Figure 6. The Parallel Instruction Fetches and Instruction Executions

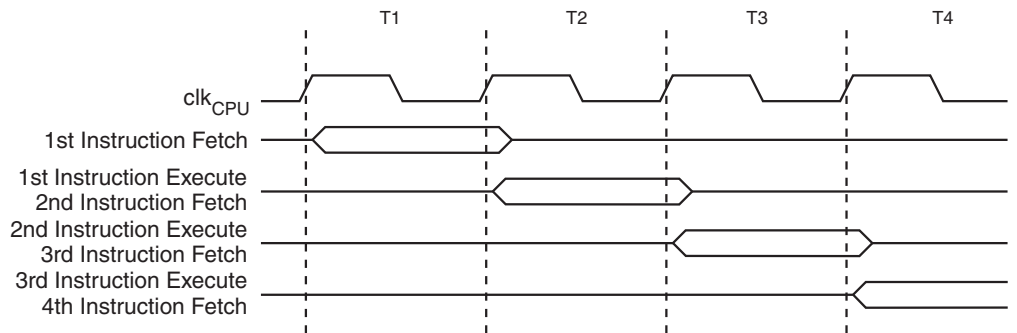
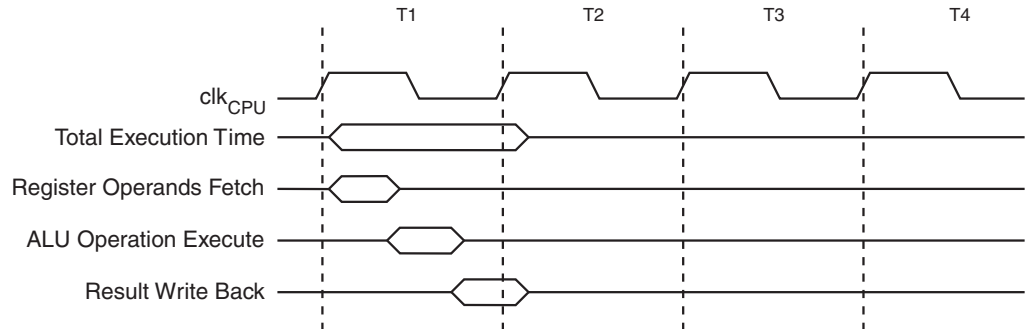


Figure 7 shows the internal timing concept for the Register File. In a single clock cycle an ALU operation using two register operands is executed, and the result is stored back to the destination register.

Figure 7. Single Cycle ALU Operation



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 246 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 46. 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 INTO – 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 46 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 232.

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, EEMWE  ; start EEPROM write
sbi EECR, EEWE
out SREG, r16    ; restore SREG value (I-bit)
    
```

C Code Example

```

char cSREG;
cSREG = SREG; /* store SREG value */
/* disable interrupts during timed sequence */
__disable_interrupt();
EECR |= (1<<EEMWE); /* start EEPROM write */
EECR |= (1<<EEWE);
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
<pre> sei ; set Global Interrupt Enable sleep ; enter sleep, waiting for interrupt ; note: will enter sleep before any pending ; interrupt(s) </pre>
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>

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.

AVR ATmega165 Memories

This section describes the different memories in the ATmega165. The AVR architecture has two main memory spaces, the Data Memory and the Program Memory space. In addition, the ATmega165 features an EEPROM Memory for data storage. All three memory spaces are linear and regular.

In-System Reprogrammable Flash Program Memory

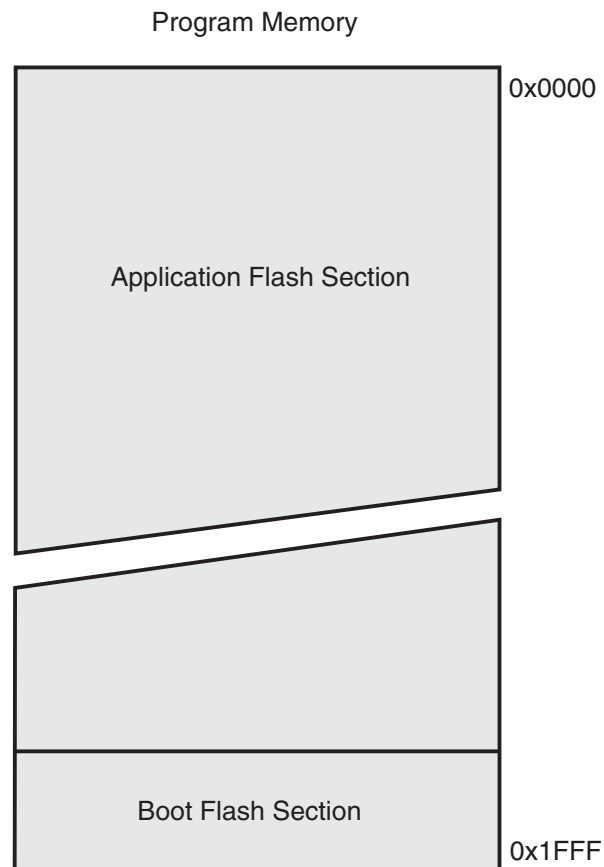
The ATmega165 contains 16K bytes On-chip In-System Reprogrammable Flash memory for program storage. Since all AVR instructions are 16 or 32 bits wide, the Flash is organized as 8K x 16. For software security, the Flash Program memory space is divided into two sections, Boot Program section and Application Program section.

The Flash memory has an endurance of at least 10,000 write/erase cycles. The ATmega165 Program Counter (PC) is 13 bits wide, thus addressing the 8K program memory locations. The operation of Boot Program section and associated Boot Lock bits for software protection are described in detail in “Boot Loader Support – Read-While-Write Self-Programming” on page 232. “Memory Programming” on page 246 contains a detailed description on Flash data serial downloading using the SPI pins or the JTAG interface.

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 12.

Figure 8. Program Memory Map



SRAM Data Memory

Figure 9 shows how the ATmega165 SRAM Memory is organized.

The ATmega165 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 1,280 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 1024 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.

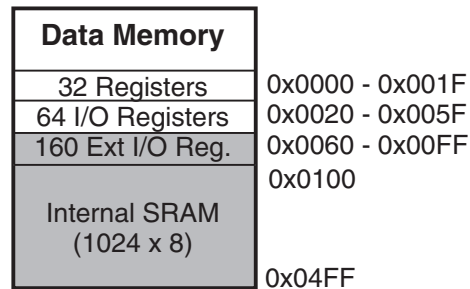
The direct addressing reaches the entire data space.

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 1,024 bytes of internal data SRAM in the ATmega165 are all accessible through all these addressing modes. The Register File is described in "General Purpose Register File" on page 10.

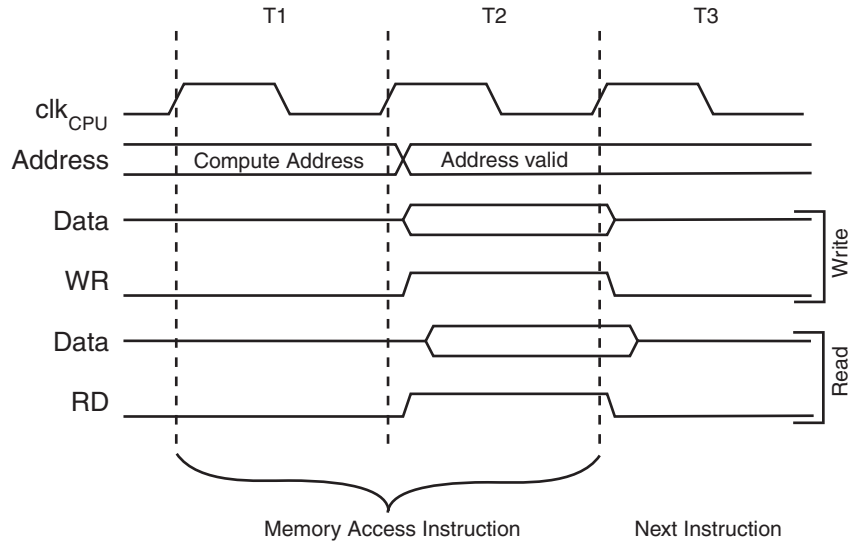
Figure 9. Data Memory Map



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 10.

Figure 10. On-chip Data SRAM Access Cycles



EEPROM Data Memory

The ATmega165 contains 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.

For a detailed description of SPI, JTAG and Parallel data downloading to the EEPROM, see page 261, page 266, and page 249 respectively.

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 1. 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 21. 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.

The EEPROM Address Register – EEARH and EEARL

Bit	15	14	13	12	11	10	9	8	
	-	-	-	-	-	-	-	EEAR8	EEARH
	EEAR7	EEAR6	EEAR5	EEAR4	EEAR3	EEAR2	EEAR1	EEAR0	EEARL
Read/Write	R	R	R	R	R	R	R	R/W	
Initial Value	0	0	0	0	0	0	0	X	
	X	X	X	X	X	X	X	X	

- **Bits 15..9 – Res: Reserved Bits**

These bits are reserved bits in the ATmega165 and will always read as zero.

- **Bits 8..0 – EEAR8..0: EEPROM Address**

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

The EEPROM Data Register – EEDR

Bit	7	6	5	4	3	2	1	0	
	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 – EEDR7..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.

The EEPROM Control Register – EECR

Bit	7	6	5	4	3	2	1	0	
	-	-	-	-	EERIE	EEMWE	EEWE	EERE	EECR
Read/Write	R	R	R	R	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	X	0	

- **Bits 7..4 – Res: Reserved Bits**

These bits are reserved bits in the ATmega165 and will always read as zero.

- **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 EEWE is cleared.

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

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

• **Bit 1 – EEW: EEPROM Write Enable**

The EEPROM Write Enable Signal EEW is the write strobe to the EEPROM. When address and data are correctly set up, the EEW bit must be written to one to write the value into the EEPROM. The EEMWE bit must be written to one before a logical one is written to EEW, 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 EEW becomes zero.
2. Wait until SPMEN 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 EEMWE bit while writing a zero to EEW in EECR.
6. Within four clock cycles after setting EEMWE, write a logical one to EEW.

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 232 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 EEW bit is cleared by hardware. The user software can poll this bit and wait for a zero before writing the next byte. When EEW 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 EEW 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 1 lists the typical programming time for EEPROM access from the CPU.

Table 1. EEPROM Programming Time

Symbol	Number of Calibrated RC Oscillator Cycles	Typ Programming Time
EEPROM write (from CPU)	67 584	8.5 ms

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,EEWE
    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 EEMWE
    sbi  EECR,EEMWE
    ; Start eeprom write by setting EEWE
    sbi  EECR,EEWE
    ret
```

C Code Example

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

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,EWE
    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<<EWE))
        ;
    /* Set up address register */
    EEAR = uiAddress;
    /* Start eeprom read by writing EERE */
    EECR |= (1<<EERE);
    /* Return data from Data Register */
    return EEDR;
}
```

EEPROM Write During Power-down Sleep Mode

When entering Power-down sleep mode while an EEPROM write operation is active, the EEPROM write operation will continue, and will complete before the Write Access time has passed. However, when the write operation is completed, the clock continues running, and as a consequence, the device does not enter Power-down entirely. It is therefore recommended to verify that the EEPROM write operation is completed before entering Power-down.

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.

I/O Memory

The I/O space definition of the ATmega165 is shown in “Register Summary” on page 319.

All ATmega165 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 ATmega165 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.

General Purpose I/O Registers

The ATmega165 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.

General Purpose I/O Register 2 – GPIOR2

Bit	7	6	5	4	3	2	1	0	
	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	

General Purpose I/O Register 1 – GPIOR1

Bit	7	6	5	4	3	2	1	0	
	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	

General Purpose I/O Register 0 – GPIOR0

Bit	7	6	5	4	3	2	1	0	
	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	

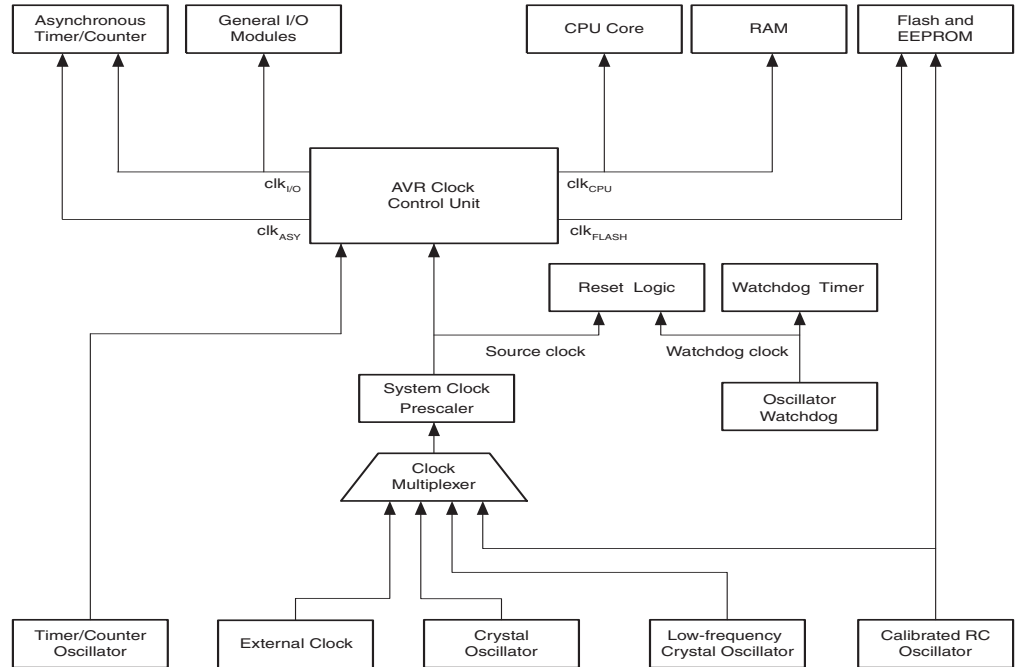


System Clock and Clock Options

Clock Systems and their Distribution

Figure 11 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 32. The clock systems are detailed below.

Figure 11. Clock Distribution



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.

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 some external interrupts are detected by asynchronous logic, allowing such interrupts to be detected even if the I/O clock is halted. Also note that start condition detection in the USI module is carried out asynchronously when $clk_{I/O}$ is halted, enabling USI start condition detection in all sleep modes.

Flash Clock – clk_{FLASH}

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

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 32 kHz clock crystal. The dedicated clock domain allows using this Timer/Counter as a real-time counter even when the device is in sleep mode.

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.

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 2. Device Clocking Options Select⁽¹⁾

Device Clocking Option	CKSEL3..0
External Crystal/Ceramic Resonator	1111 - 1000
External Low-frequency Crystal	0111 - 0110
Calibrated Internal RC Oscillator	0010
External Clock	0000
Reserved	0011, 0001, 0101, 0100

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

The various choices for each clocking option is given in the following sections. When the CPU wakes up from Power-down or Power-save, the selected clock source is used to time the start-up, ensuring stable Oscillator operation before instruction execution starts. When the CPU starts from reset, there is an additional delay allowing the power to reach a stable level before commencing normal operation. The Watchdog Oscillator is used for timing this real-time part of the start-up time. The number of WDT Oscillator cycles used for each time-out is shown in Table 3. The frequency of the Watchdog Oscillator is voltage dependent as shown in “ATmega165 Typical Characteristics” on page 285.

Table 3. Number of Watchdog Oscillator Cycles

Typ Time-out ($V_{CC} = 5.0V$)	Typ Time-out ($V_{CC} = 3.0V$)	Number of Cycles
4.1 ms	4.3 ms	4K (4,096)
65 ms	69 ms	64K (65,536)

Default Clock Source

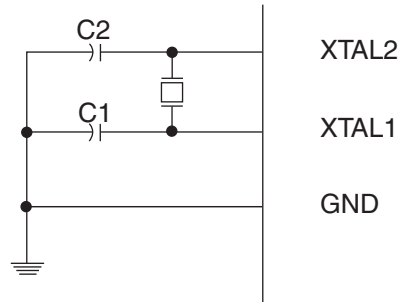
The device is shipped with CKSEL = “0010”, SUT = “10”, and CKDIV8 programmed. The default clock source setting is the Internal RC Oscillator with longest start-up time and an initial system clock prescaling of 8. This default setting ensures that all users can make their desired clock source setting using an In-System or Parallel programmer.

Crystal Oscillator

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 12. Either a quartz crystal or a ceramic resonator may be used.

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 4. For ceramic resonators, the capacitor values given by the manufacturer should be used.

Figure 12. Crystal Oscillator Connections



The 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 4.

Table 4. Crystal Oscillator Operating Modes

CKSEL3..1	Frequency Range (MHz)	Recommended Range for Capacitors C1 and C2 for Use with Crystals (pF)
100 ⁽¹⁾	0.4 - 0.9	–
101	0.9 - 3.0	12 - 22
110	3.0 - 8.0	12 - 22
111	8.0 -	12 - 22

Notes: 1. This option should not be used with crystals, only with ceramic resonators.

The CKSEL0 Fuse together with the SUT1..0 Fuses select the start-up times as shown in Table 5.