Contact us

ATtiny261/ATtiny461/ATtiny861 Automotive

8-bit AVR Microcontroller with 2/4/8K Bytes In-System Programmable Flash

DATASHEET

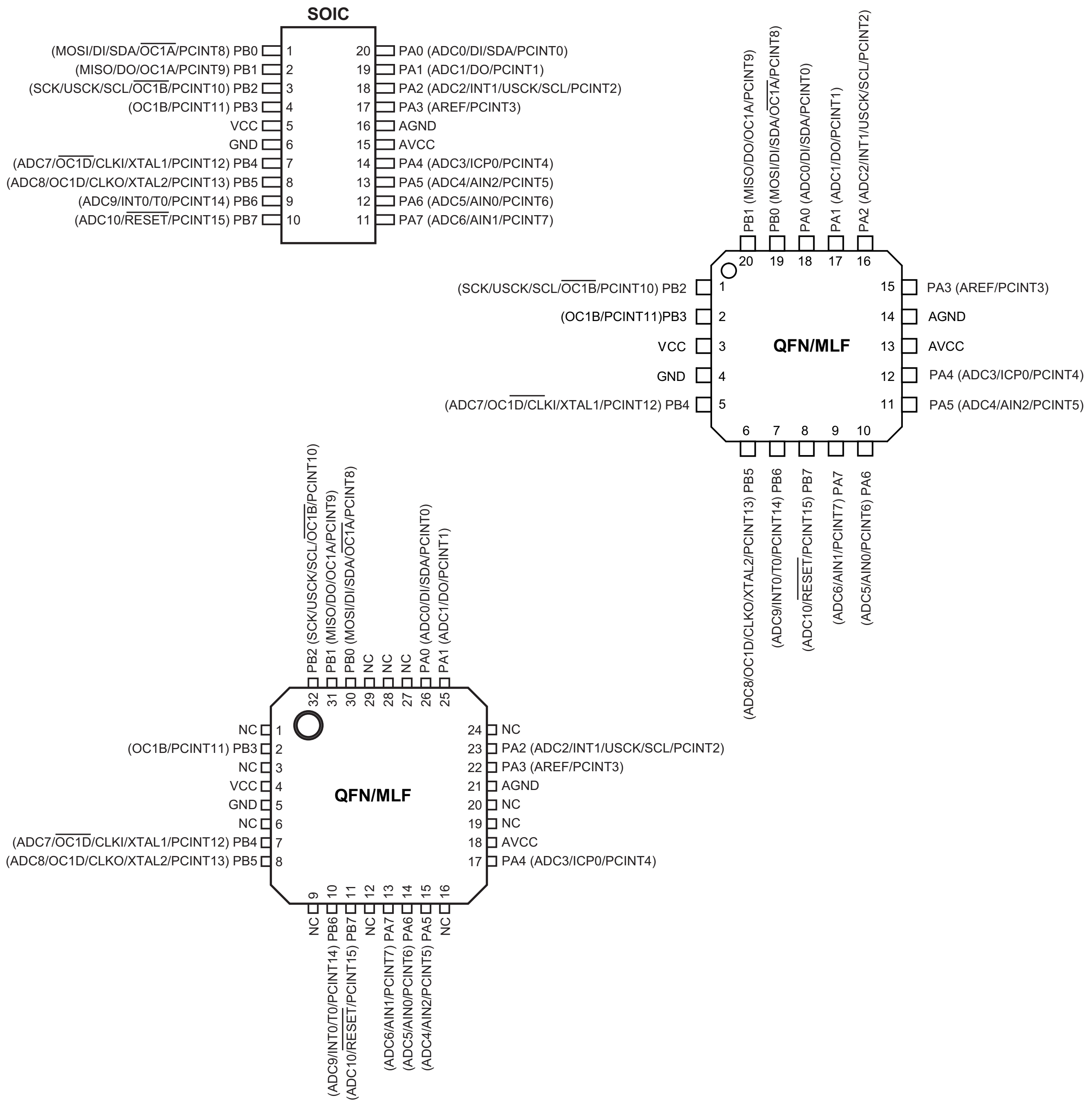
Features

- High performance, low power AVR® 8-Bit microcontroller
- Advanced RISC architecture
  - 123 powerful instructions – most single clock cycle execution
  - 32 x 8 general purpose working registers
  - Fully static operation
- Non-volatile program and data memories
  - 2/4/8K byte of in-system programmable program memory flash (Atmel® ATtiny261/461/861)
    - Endurance: 10,000 write/erase cycles
  - 128/256/512 bytes in-system programmable EEPROM (Atmel ATtiny261/461/861)
    - Endurance: 100,000 write/erase cycles
  - 128/256/512 bytes internal SRAM (ATtiny261/461/861)
  - Programming lock for self-programming flash program and EEPROM data security
- Peripheral features
  - 8/16-bit Timer/Counter with prescaler
  - 8/10-bit high speed Timer/Counter with separate prescaler
    - 3 high frequency PWM outputs with separate output compare registers
    - Programmable dead time generator
  - Universal serial interface with start condition detector
  - 10-bit ADC
    - 11 single ended channels
    - 16 differential ADC channel pairs
    - 15 differential ADC channel pairs with programmable gain (1x, 8x, 20x, 32x)
  - Programmable watchdog timer with separate on-chip oscillator
  - On-chip analog comparator
- Special microcontroller features
  - debugWIRE on-chip debug system
  - In-system programmable via SPI port
  - External and internal interrupt sources
  - Low power idle, ADC noise reduction, and power-down modes
  - Enhanced power-on reset circuit
  - Programmable brown-out detection circuit
  - Internal calibrated oscillator

- I/O and packages
  - 16 programmable I/O lines
  - 20-pin SOIC, 32-pad MLF and 20-lead TSSOP
- Operating voltage:
  - 2.7 - 5.5V for Atmel ATtiny261/461/861
- Speed grade:
  - Atmel® ATtiny261/461/861: 0 - 8MHz at 2.7 - 5.5V, 0 - 16MHz at 4.5 - 5.5V
  - Operating temperature: Automotive (–40°C to +125°C)
- Low power consumption
  - Active mode ATD On: 1MHz, 2.7V, 25°C: 300µA
  - Power-down mode no watchdog: 2.7V, 25°C: 0.12µA

1. Pin Configurations

Figure 1-1. Pinout ATtiny261/461/861



SOIC
(MOSI/DI/SDA/OC1A/PCINT8) PB0 1
(MISO/DO/OC1A/PCINT9) PB1 2
(SCK/USCK/SCL/OC1B/PCINT10) PB2 3
(OC1B/PCINT11) PB3 4
VCC 5
GND 6
(ADC7/OC1D/CLKI/XTAL1/PCINT12) PB4 7
(ADC8/OC1D/CLKO/XTAL2/PCINT13) PB5 8
(ADC9/INT0/T0/PCINT14) PB6 9
(ADC10/RESET/PCINT15) PB7 10
20 PA0 (ADC0/DI/SDA/PCINT0)
19 PA1 (ADC1/DO/PCINT1)
18 PA2 (ADC2/INT1/USCK/SCL/PCINT2)
17 PA3 (AREF/PCINT3)
16 AGND
15 AVCC
14 PA4 (ADC3/ICP0/PCINT4)
13 PA5 (ADC4/AIN2/PCINT5)
12 PA6 (ADC5/AIN0/PCINT6)
11 PA7 (ADC6/AIN1/PCINT7)
QFN/MLF
PB1 (MISO/DO/OC1A/PCINT9) 20
PB0 (MOSI/DI/SDA/OC1A/PCINT8) 19
PA0 (ADC0/DI/SDA/PCINT0) 18
PA1 (ADC1/DO/PCINT1) 17
PA2 (ADC2/INT1/USCK/SCL/PCINT2) 16
(SCK/USCK/SCL/OC1B/PCINT10) PB2 1
(OC1B/PCINT11)PB3 2
VCC 3
GND 4
(ADC7/OC1D/CLKI/XTAL1/PCINT12) PB4 5
15 PA3 (AREF/PCINT3)
14 AGND
13 AVCC
12 PA4 (ADC3/ICP0/PCINT4)
11 PA5 (ADC4/AIN2/PCINT5)
6 (ADC8/OC1D/CLKO/XTAL2/PCINT13) PB5
7 (ADC9/INT0/T0/PCINT14) PB6
8 (ADC10/RESET/PCINT15) PB7
9 (ADC6/AIN1/PCINT7) PA7
10 (ADC5/AIN0/PCINT6) PA6
QFN/MLF
PB2 (SCK/USCK/SCL/OC1B/PCINT10) 32
PB1 (MISO/DO/OC1A/PCINT9) 31
PB0 (MOSI/DI/SDA/OC1A/PCINT8) 30
NC 29
NC 28
NC 27
PA0 (ADC0/DI/SDA/PCINT0) 26
PA1 (ADC1/DO/PCINT1) 25
NC 1
(OC1B/PCINT11) PB3 2
NC 3
VCC 4
GND 5
NC 6
(ADC7/OC1D/CLKI/XTAL1/PCINT12) PB4 7
(ADC8/OC1D/CLKO/XTAL2/PCINT13) PB5 8
24 NC
23 PA2 (ADC2/INT1/USCK/SCL/PCINT2)
22 PA3 (AREF/PCINT3)
21 AGND
20 NC
19 NC
18 AVCC
17 PA4 (ADC3/ICP0/PCINT4)
9 NC
10 (ADC9/INT0/T0/PCINT14) PB6
11 (ADC10/RESET/PCINT15) PB7
12 NC
13 (ADC6/AIN1/PCINT7) PA7
14 (ADC5/AIN0/PCINT6) PA6
15 (ADC4/AIN2/PCINT5) PA5
16 NC

Note: The large center pad underneath the QFN/MLF package should be soldered to ground on the board to ensure good mechanical stability.

1.1 Disclaimer

Typical values contained in this data sheet 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.

1.2 Automotive Quality Grade

The Atmel® ATtiny261/461/861 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 ATtiny261/461/861 have been verified during regular product qualification as per AEC-Q100 grade 1.

As indicated in the ordering information paragraph, the product is available in only one temperature grade, see Table 1-1.

Table 1-1. Temperature Grade Identification for Automotive Products

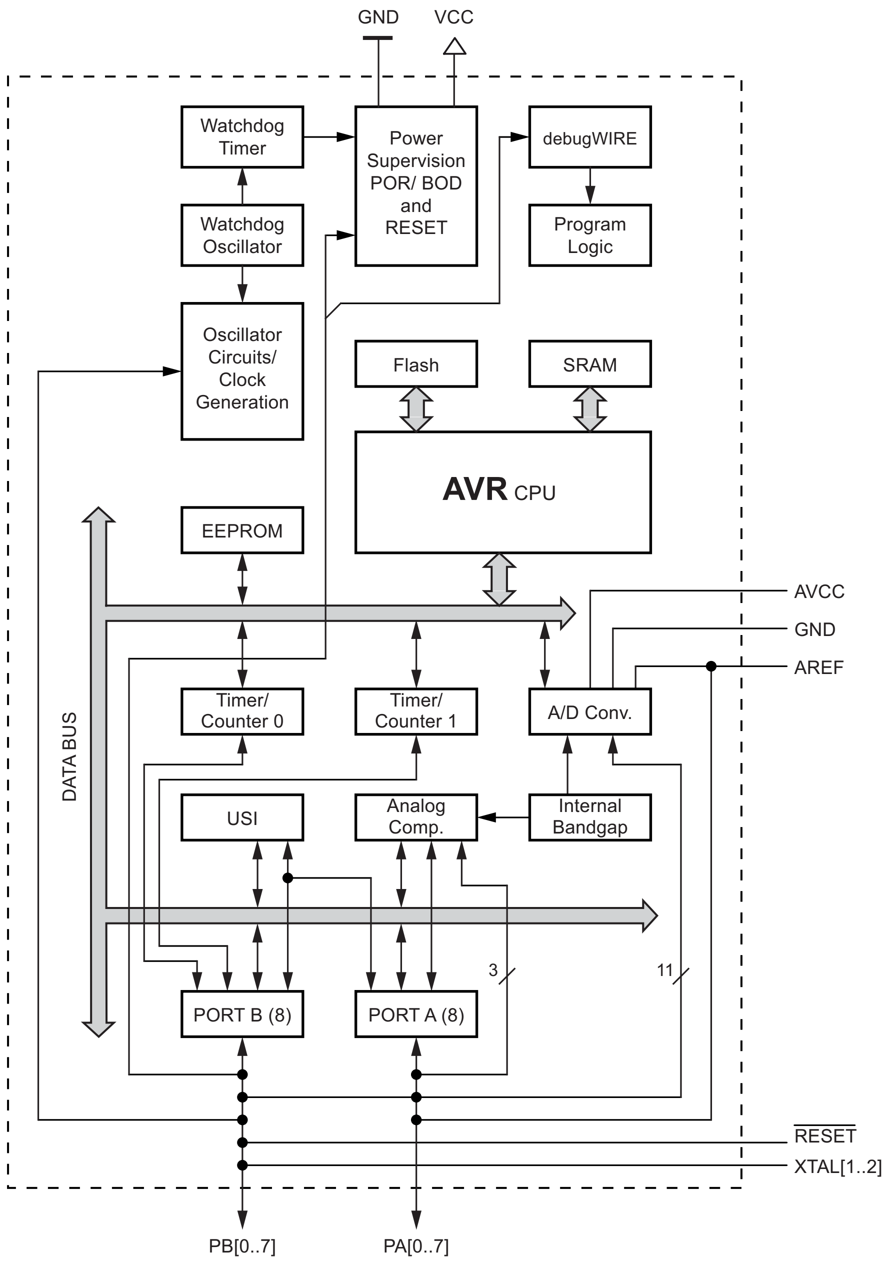
Temperature	Temperature Identifier	Comments
–40; +125	Z	Full automotive temperature range

2. Overview

The Atmel® ATtiny261/461/861 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 ATtiny261/461/861 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® ATtiny261/461/861 provides the following features: 2/4/8Kbyte of in-system programmable flash, 128/256/512 bytes EEPROM, 128/256/512 bytes SRAM, 6 general purpose I/O lines, 32 general purpose working registers, one 8-bit Timer/Counter with compare modes, one 8-bit high speed Timer/Counter, universal serial interface, internal and external interrupts, a 4-channel, 10-bit ADC, a programmable watchdog timer with internal oscillator, and three software selectable power saving modes. The idle mode stops the CPU while allowing the SRAM, Timer/Counter, ADC, analog comparator, and interrupt system to continue functioning. The power-down mode saves the register contents, disabling all chip functions until the next interrupt or hardware reset. The ADC noise reduction mode stops the CPU and all I/O modules except ADC, to minimize switching noise during ADC conversions.

The device is manufactured using Atmel high density non-volatile memory technology. The on-chip ISP flash allows the program memory to be re-programmed in-system through an SPI serial interface, by a conventional non-volatile memory programmer or by an on-chip boot code running on the AVR core.

The Atmel ATtiny261/461/861 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 Pin Descriptions

2.2.1 VCC

Supply voltage.

2.2.2 GND

Ground.

2.2.3 AVCC

Analog supply voltage.

2.2.4 AGND

Analog ground.

2.2.5 Port A (PA7..PA0)

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.

Port A also serves the functions of various special features of the Atmel ATtiny261/461/861 as listed on Section 12.3.2 “Alternate Functions of Port A” on page 62.

2.2.6 Port B (PB7..PB0)

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.

Port B also serves the functions of various special features of the Atmel ATtiny261/461/861 as listed on

Section 12.3.1 “Alternate Functions of Port B” on page 59.

2.2.7 $\overline{\text{RESET}}$

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 23-3 on page 174. Shorter pulses are not guaranteed to generate a reset.

3. Resources

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

4. About Code Examples

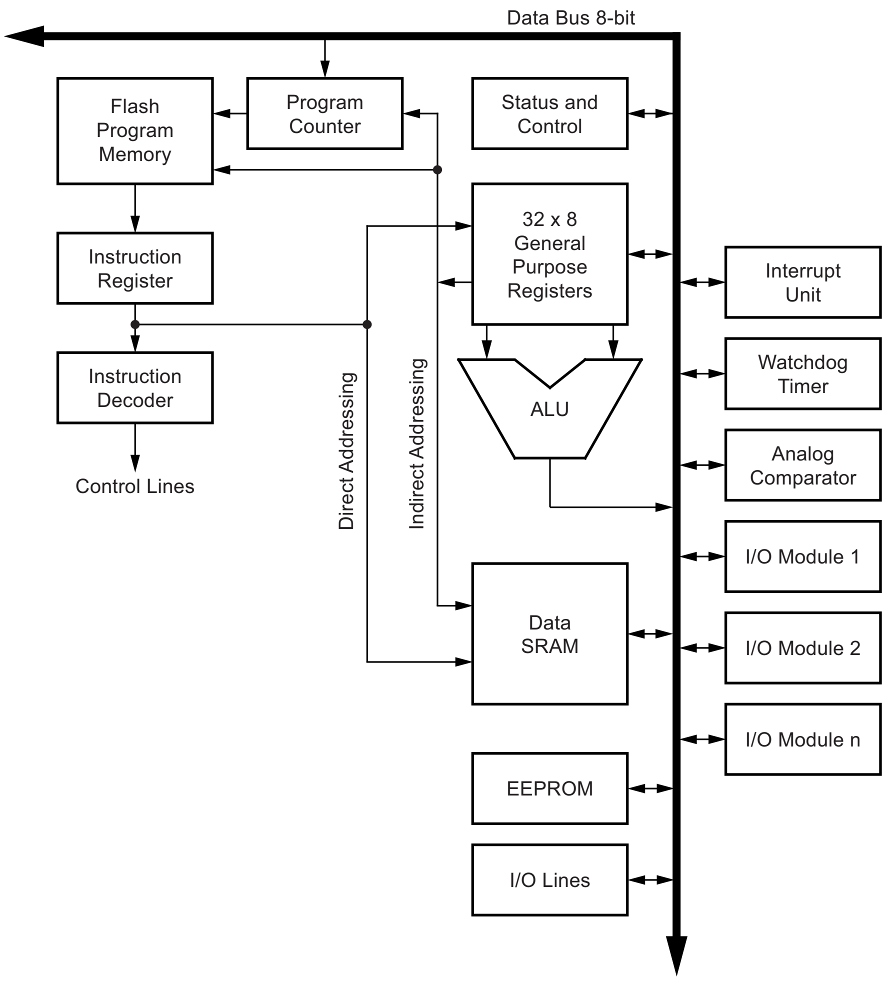
This documentation contains simple code examples that briefly show how to use various parts of the device. These code examples assume that the part specific header file is included before compilation. Be aware that not all C compiler vendors include bit definitions in the header files and interrupt handling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

5. AVR CPU Core

5.1 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 5-1. Block Diagram of the AVR Architecture



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

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

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

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

Program flow is provided by conditional and unconditional jump and call instructions, able to directly address the whole address space. Most AVR® instructions have a single 16-bit word format. Most AVR instructions are 16-bit wide. There are also 32-bit instructions.

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.

5.2 ALU – Arithmetic Logic Unit

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

5.3 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.

5.3.1 SREG – AVR Status Register

The AVR® status register – SREG – is defined as:

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

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

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

- **Bit 6 – T: Bit Copy Storage**

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

- **Bit 5 – H: Half Carry Flag**

The half carry flag H indicates a half carry in some arithmetic operations. Half carry is useful in BCD arithmetic. See the “Instruction Set Description” for detailed information.

- **Bit 4 – S: Sign Bit, S = N $\oplus$ V**

The S-bit is always an exclusive or between the negative flag N and the two 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.

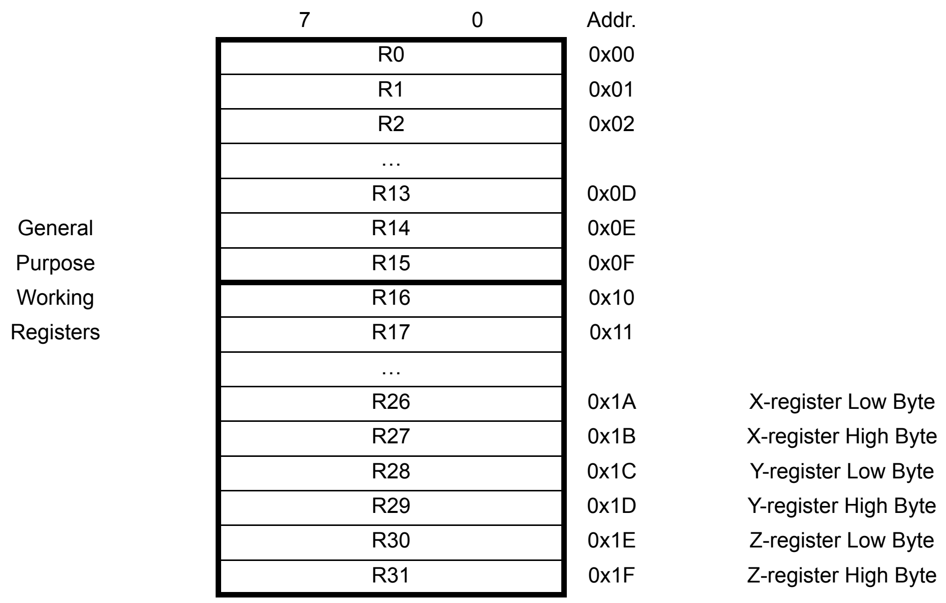
5.4 General Purpose Register File

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 5-2 shows the structure of the 32 general purpose working registers in the CPU.

Figure 5-2. 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

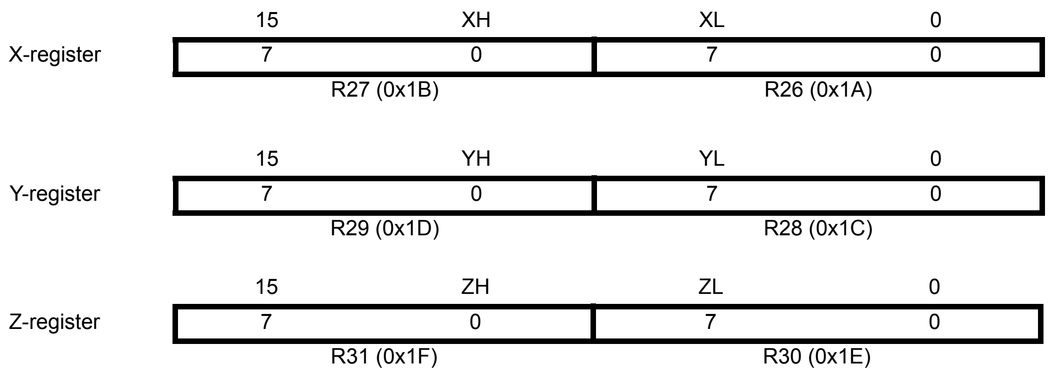
Most of the instructions operating on the register file have direct access to all registers, and most of them are single cycle instructions.

As shown in Figure 5-2, 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.

5.4.1 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 5-3.

Figure 5-3. The X-, Y-, and Z-registers



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

5.5 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. 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 0x60. 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.

5.5.1 SPH and SPL – Stack Pointer 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	

5.6 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 5-4 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 5-4. The Parallel Instruction Fetches and Instruction Executions

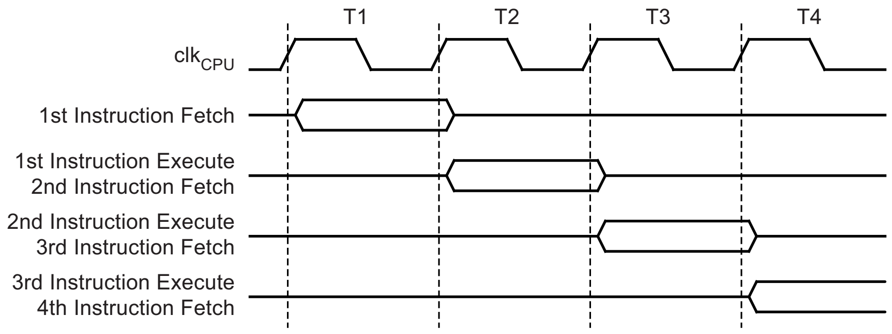
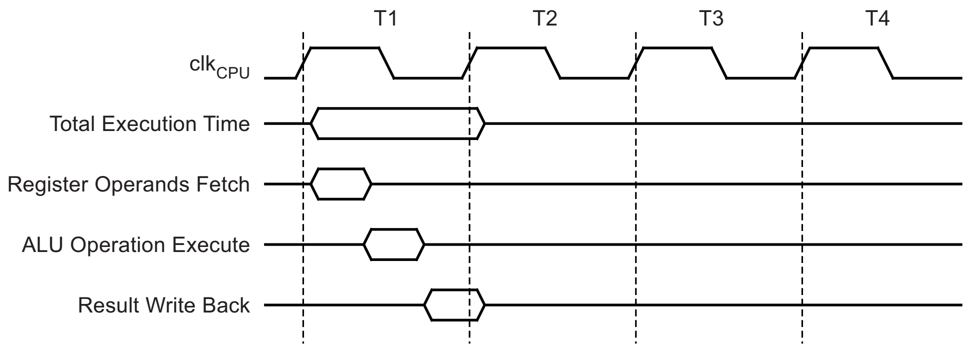


Figure 5-5 shows the internal timing concept for the register file. In a single clock cycle an ALU operation using two register operands is executed, and the result is stored back to the destination register.

Figure 5-5. Single Cycle ALU Operation



5.7 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.

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 10. "Interrupts" on page 47. 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.

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

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

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

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

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

When using the CLI instruction to disable interrupts, the interrupts will be immediately disabled. No interrupt will be executed after the CLI instruction, even if it occurs simultaneously with the CLI instruction.

The following example shows how this can be used to avoid interrupts during the timed EEPROM write sequence.

Assembly Code Example

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

C Code Example

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

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

Assembly Code Example

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

C Code Example

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

5.7.1 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.

6. AVR Memories

This section describes the different memories in the Atmel® ATtiny261/461/861. The AVR architecture has two main memory spaces, the Data memory and the Program memory space. In addition, the Atmel ATtiny261/461/861 features an EEPROM memory for data storage. All three memory spaces are linear and regular.

6.1 In-System Re-programmable Flash Program Memory

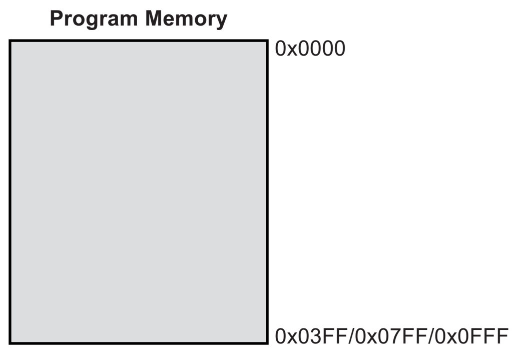
The Atmel ATtiny261/461/861 contains 2/4/8K byte on-chip in-system reprogrammable flash memory for program storage. Since all AVR® instructions are 16 or 32 bits wide, the flash is organized as 1024/2048/4096 × 16.

The flash memory has an endurance of at least 10,000 write/erase cycles. The Atmel ATtiny261/461/861 program counter (PC) is 10/11/12 bits wide, thus addressing the 1024/2048/4096 program memory locations. Section 22. "Memory Programming" on page 156 contains a detailed description on flash data serial downloading using the SPI pins.

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 5.6 "Instruction Execution Timing" on page 13.

Figure 6-1. Program Memory Map



6.2 SRAM Data Memory

Figure 6-2 shows how the Atmel ATtiny261/461/861 SRAM memory is organized.

The lower 224/352/608 data memory locations address both the register file, the I/O memory and the internal data SRAM. The first 32 locations address the register file, the next 64 locations the standard I/O memory, and the last 128/256/512 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.

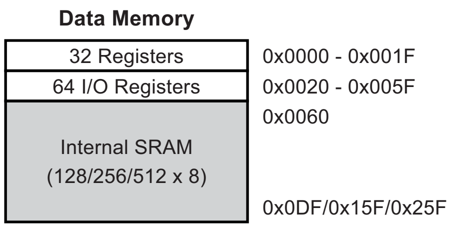
The direct addressing reaches the entire data space.

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, and the 128/256/512 bytes of internal data SRAM in the Atmel ATtiny261/461/861 are all accessible through all these addressing modes. The register file is described in Section 5.4 "General Purpose Register File" on page 11.

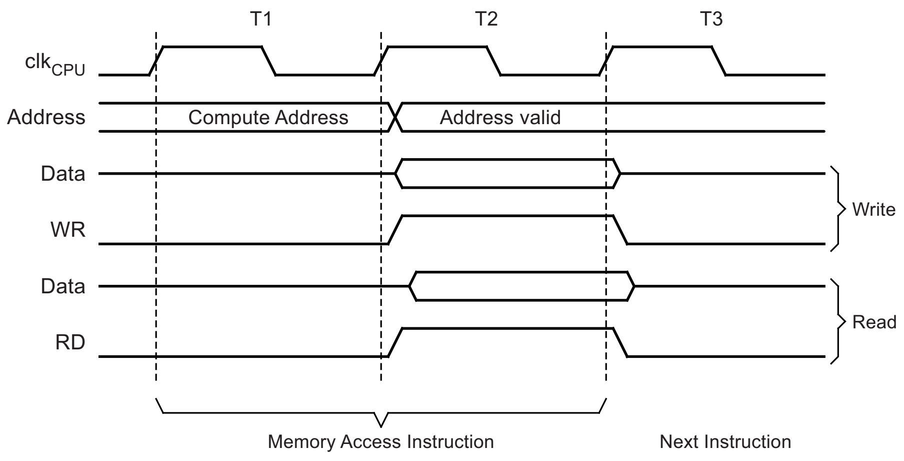
Figure 6-2. Data Memory Map



6.2.1 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 6-3.

Figure 6-3. On-chip Data SRAM Access Cycles



6.3 EEPROM Data Memory

The Atmel® ATtiny261/461/861 contains 128/256/512 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 serial data downloading to the EEPROM, see Section 22.9 “Serial Downloading” on page 167.

6.3.1 EEPROM Read/Write Access

The EEPROM access registers are accessible in the I/O space.

The write access times for the EEPROM are given in Table 6-1 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 6.3.6 “Preventing EEPROM Corruption” on page 20 for details on how to avoid problems in these situations.

In order to prevent unintentional EEPROM writes, a specific write procedure must be followed. Refer to Section 6.3.2 “Atomic Byte Programming” on page 18 and Section 6.3.3 “Split Byte Programming” on page 18 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.

6.3.2 Atomic Byte Programming

Using atomic byte programming is the simplest mode. When writing a byte to the EEPROM, the user must write the address into the EEARL register and data into EEDR register. If the EEPMn bits are zero, writing EEPE (within four cycles after EEMPE is written) will trigger the erase/write operation. Both the erase and write cycle are done in one operation and the total programming time is given in Table 1. The EEPE bit remains set until the erase and write operations are completed. While the device is busy with programming, it is not possible to do any other EEPROM operations.

6.3.3 Split Byte Programming

It is possible to split the erase and write cycle in two different operations. This may be useful if the system requires short access time for some limited period of time (typically if the power supply voltage falls). In order to take advantage of this method, it is required that the locations to be written have been erased before the write operation. But since the erase and write operations are split, it is possible to do the erase operations when the system allows doing time-critical operations (typically after power-up).

6.3.4 Erase

To erase a byte, the address must be written to EEAR. If the EEPMn bits are 0b01, writing the EEPE (within four cycles after EEMPE is written) will trigger the erase operation only (programming time is given in Table 1). The EEPE bit remains set until the erase operation completes. While the device is busy programming, it is not possible to do any other EEPROM operations.

6.3.5 Write

To write a location, the user must write the address into EEAR and the data into EEDR. If the EEPMn bits are 0b10, writing the EEPE (within four cycles after EEMPE is written) will trigger the write operation only (programming time is given in Table 1-1 on page 4). The EEPE bit remains set until the write operation completes. If the location to be written has not been erased before write, the data that is stored must be considered as lost. While the device is busy with programming, it is not possible to do any other EEPROM operations.

The calibrated oscillator is used to time the EEPROM accesses. Make sure the oscillator frequency is within the requirements described in Section 7.12.1 “OSCCAL – Oscillator Calibration Register” on page 31.

The following code examples show one assembly and one C function for erase, write, or atomic write of 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.

Assembly Code Example

```
EEPROM_write:
        ; Wait for completion of previous write
        sbic    EECR,EEPE
        rjmp    EEPROM_write
        ; Set Programming mode
        ldi    r16, (0<<EEPM1)|(0<<EEPM0)
        out    EECR, r16
        ; Set up address (r18:r17) in address register
        out     EEARH, r18
        out     EEARL, r17
        ; Write data (r16) to data register
        out     EEDR, r16
        ; Write logical one to EEMPE
        sbi     EECR,EEMPE
        ; Start eeprom write by setting EEPE
        sbi     EECR,EEPE
        ret
```

C Code Example

```c
void EEPROM_write(unsigned char ucAddress, unsigned char ucData)
{
        /* Wait for completion of previous write */
        while(EECR & (1<<EEPE))
        ;
        /* Set Programming mode */
        EECR = (0<<EEPM1)|(0<<EEPM0);
        /* Set up address and data registers */
        EEAR = ucAddress;
        EEDR = ucData;
        /* Write logical one to EEMPE */
        EECR |= (1<<EEMPE);
        /* Start eeprom write by setting EEPE */
        EECR |= (1<<EEPE);
}
```

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

Assembly Code Example

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

C Code Example

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

6.3.6 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.

6.4 I/O Memory

The I/O space definition of the Atmel® ATtiny261/461/861 is shown in Section 25. "Register Summary" on page 189.

All Atmel ATtiny261/461/861 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.

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, 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.

6.4.1 General Purpose I/O Registers

The Atmel ATtiny261/461/861 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.

6.5 Register Description

6.5.1 EEARH and EEARL – EEPROM Address Register

Bit	7	6	5	4	3	2	1	0	
0x1F (0x3F)	-	-	-	-	-	-	-	**EEAR8**	**EEARH**
0x1E (0x3E)	**EEAR7**	**EEAR6**	**EEAR5**	**EEAR4**	**EEAR3**	**EEAR2**	**EEAR1**	**EEAR0**	**EEARL**
Bit	7	6	5	4	3	2	1	0	
Read/Write	R	R	R	R	R	R	R	R/W	
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	X	
Initial Value	X	X	X	X	X	X	X	X	

- **Bit 7:1 – Res6:0: Reserved Bits**

These bits are reserved for future use and will always read as 0 in Atmel ATtiny261/461/861.

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

The EEPROM address registers – EEARH and EEARL – specifies the high EEPROM address in the 128/256/512 bytes EEPROM space. The EEPROM data bytes are addressed linearly between 0 and 127/255/511. The initial value of EEAR is undefined. A proper value must be written before the EEPROM may be accessed.

6.5.2 EEDR – EEPROM Data Register

Bit	7	6	5	4	3	2	1	0	
0x1D (0x3D)	**EEDR7**	**EEDR6**	**EEDR5**	**EEDR4**	**EEDR3**	**EEDR2**	**EEDR1**	**EEDR0**	**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.

6.5.3 EECR – EEPROM Control Register

Bit	7	6	5	4	3	2	1	0	
0x1C (0x3C)	–	–	**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	

- **Bit 7 – Res: Reserved Bit**

This bit is reserved for future use and will always read as 0 in Atmel® ATtiny261/461/861. For compatibility with future AVR® devices, always write this bit to zero. After reading, mask out this bit.

- **Bit 6 – Res: Reserved Bit**

This bit is reserved in the Atmel ATtiny261/461/861 and will always read as zero.

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

The EEPROM programming mode bits 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 6-1. 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 6-1. 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 non-volatile memory is ready for programming.

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

The EEMPE bit determines whether writing EEPE to one will have effect or not.

When EEMPE is set, setting EEPE within four clock cycles will program the EEPROM at the selected address. If EEMPE is zero, setting EEPE will have no effect. When EEMPE has been written to one by software, hardware clears the bit to zero after four clock cycles.

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

The EEPROM program enable signal EEPE is the programming enable signal to the EEPROM. When EEPE is written, the EEPROM will be programmed according to the EEPMn bits setting. The EEMPE bit must be written to one before a logical one is written to EEPE, otherwise no EEPROM write takes place. When the write access time has elapsed, the EEPE bit is cleared by hardware. When EEPE has been set, the CPU is halted for two cycles before the next instruction is executed.

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

The EEPROM read enable signal – EERE – is the read strobe to the EEPROM. When the correct address is set up in the EEAR register, the EERE bit must be written to one to trigger the EEPROM read. The EEPROM read access takes one instruction, and the requested data is available immediately. When the EEPROM is read, the CPU is halted for four cycles before the next instruction is executed. The user should poll the EEPE bit before starting the read operation. If a write operation is in progress, it is neither possible to read the EEPROM, nor to change the EEAR register.

6.5.4 GPIOR2 – General Purpose I/O Register 2

Bit	7	6	5	4	3	2	1	0	
0x0C (0x2C)	**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	

6.5.5 GPIOR1 – General Purpose I/O Register 1

Bit	7	6	5	4	3	2	1	0	
0x0B (0x2B)	**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	

6.5.6 GPIOR0 – General Purpose I/O Register 0

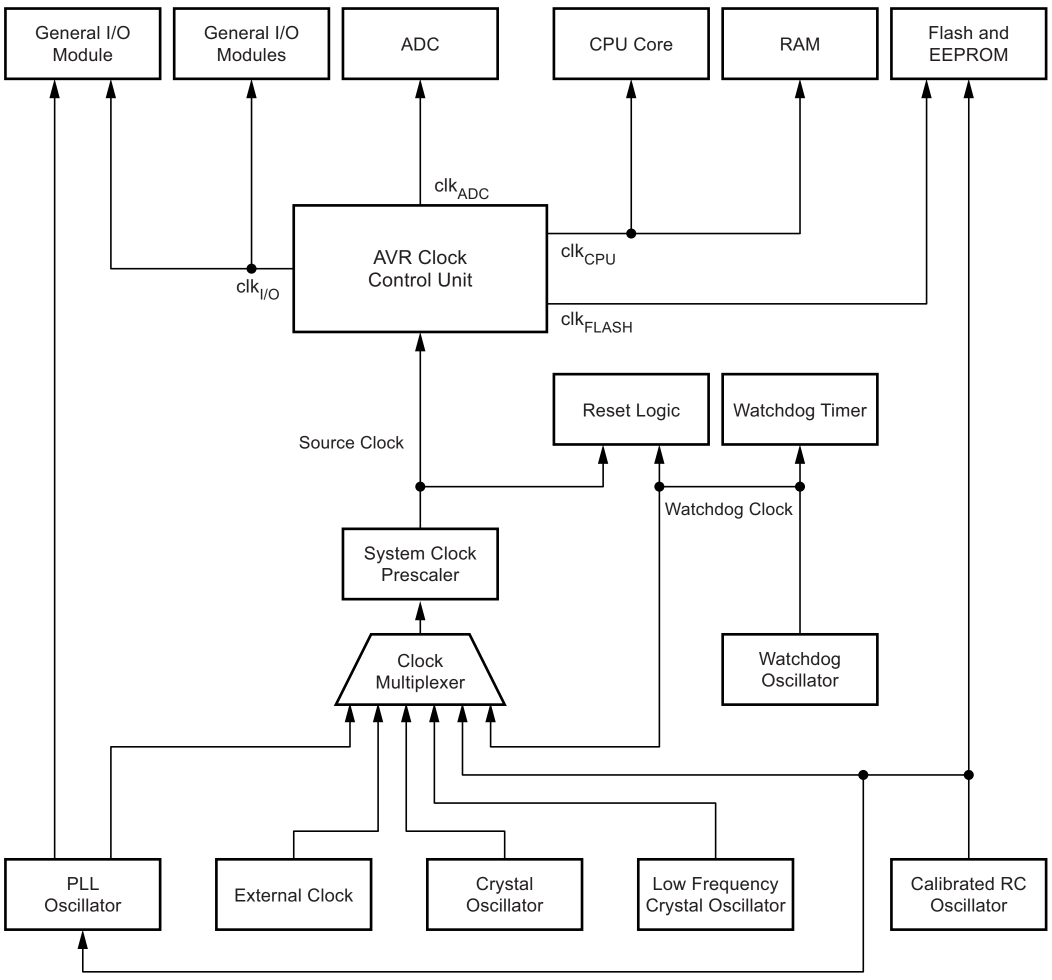
Bit	7	6	5	4	3	2	1	0	
0x0A (0x2A)	**MSB**							**LSB**	**GPIOR0**
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

7. System Clock and Clock Options

7.1 Clock Systems and their Distribution

Figure 7-1 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 8. “Power Management and Sleep Modes” on page 34. The clock systems are detailed below.

Figure 7-1. Clock Distribution



7.1.1 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.

7.1.2 I/O Clock – clk$_{I/O}$

The I/O clock is used by the majority of the I/O modules, like Timer/Counter. 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.

7.1.3 Flash Clock – clk_{FLASH}

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

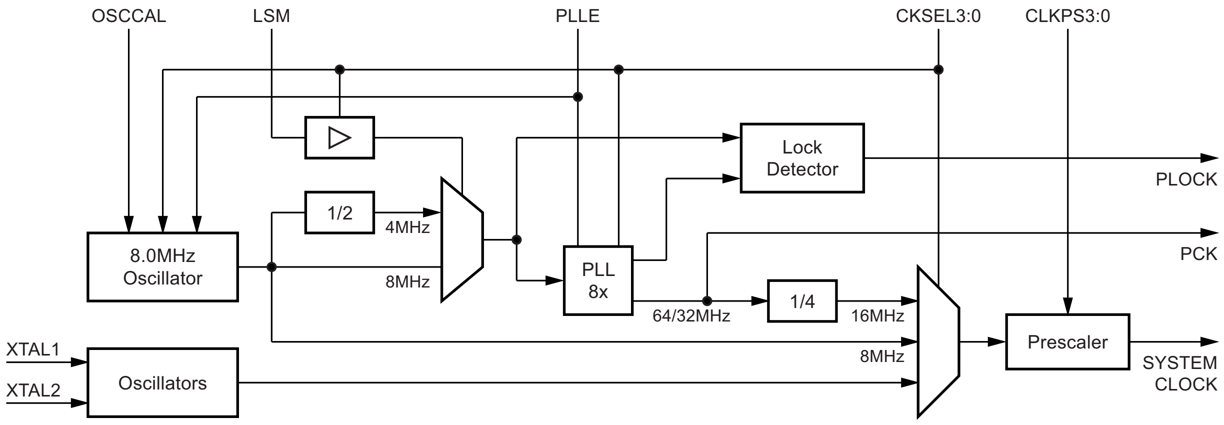
7.1.4 ADC Clock – clk_{ADC}

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.

7.1.5 Internal PLL for Fast Peripheral Clock Generation - clk_{PCK}

The internal PLL in Atmel® ATtiny261/461/861 generates a clock frequency that is 8x multiplied from a source input. By default, the PLL uses the output of the internal 8.0MHz RC oscillator as source. Alternatively, if the LSM bit of the PLLCSR is set the PLL will use the output of the RC oscillator divided by two. Thus the output of the PLL, the fast peripheral clock is 64MHz. The fast peripheral clock, or a clock prescaled from that, can be selected as the clock source for Timer/Counter1or as a system clock. See Figure 7-2. The frequency of the fast peripheral clock is divided by two when LSM of PLLCSR is set, resulting in a clock frequency of 32MHz. Note, that LSM can not be set if PLL_{CLK} is used as a system clock.

Figure 7-2. PCK Clocking System



The PLL is locked on the RC oscillator and adjusting the RC oscillator via OSCCAL register will adjust the fast peripheral clock at the same time. However, even if the RC oscillator is taken to a higher frequency than 8MHz, the fast peripheral clock frequency saturates at 85MHz (worst case) and remains oscillating at the maximum frequency. It should be noted that the PLL in this case is not locked any longer with the RC oscillator clock. Therefore, it is recommended not to take the OSCCAL adjustments to a higher frequency than 8MHz in order to keep the PLL in the correct operating range.

The internal PLL is enabled when:

- The PLLE bit of the PLLCSR register is set.
- The CKSEL fuse are programmed to ‘0001’.

The PLLCSR bit PLOCK is set when PLL is locked.

Both internal RC oscillator and PLL are switched off in power down and stand-by sleep modes.