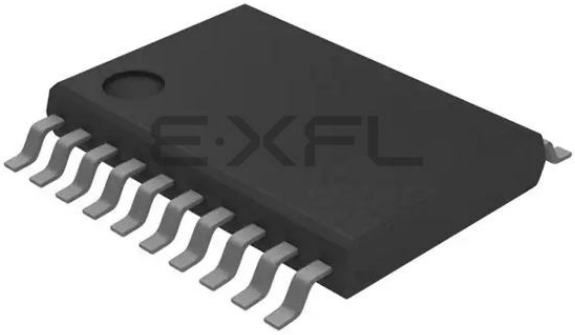


Welcome to E-XFL.COM

What is "Embedded - Microcontrollers"?

"Embedded - Microcontrollers" refer to small, integrated circuits designed to perform specific tasks within larger systems. These microcontrollers are essentially compact computers on a single chip, containing a processor core, memory, and programmable input/output peripherals. They are called "embedded" because they are embedded within electronic devices to control various functions, rather than serving as standalone computers. Microcontrollers are crucial in modern electronics, providing the intelligence and control needed for a wide range of applications.

Applications of "Embedded - Microcontrollers"

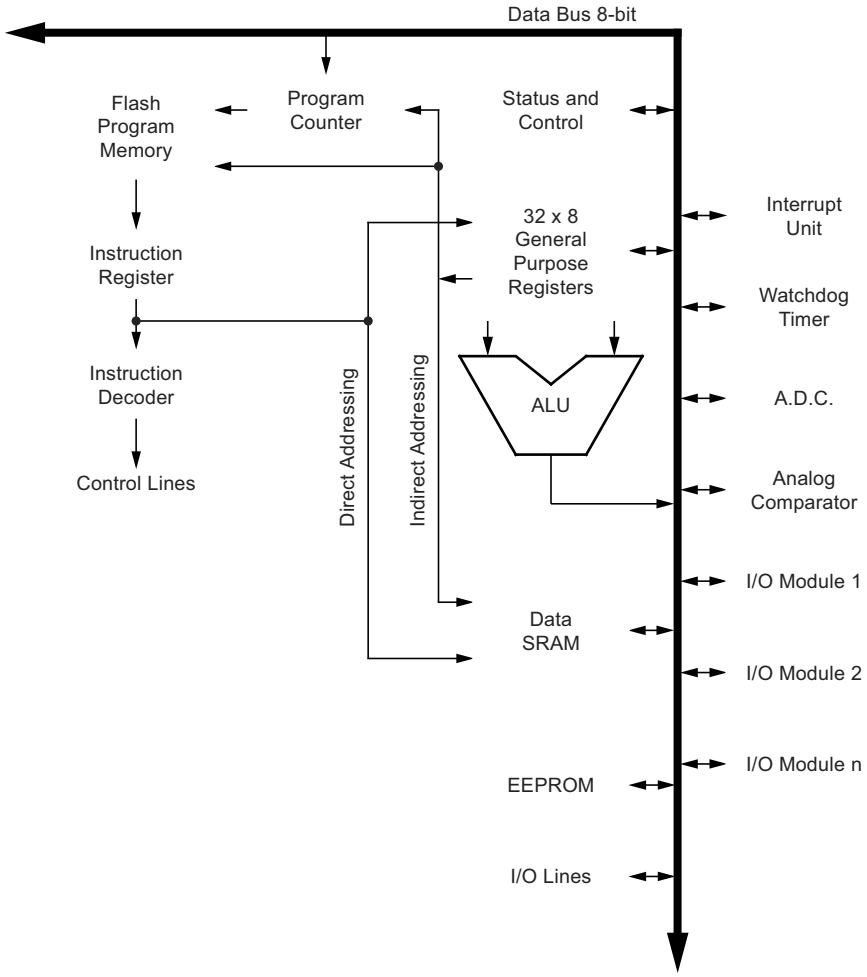
Details	
Product Status	Obsolete
Core Processor	AVR
Core Size	8-Bit
Speed	16MHz
Connectivity	I²C, LINbus, SPI, UART/USART, USI
Peripherals	Brown-out Detect/Reset, POR, PWM, Temp Sensor, WDT
Number of I/O	16
Program Memory Size	16KB (8K x 16)
Program Memory Type	FLASH
EEPROM Size	512 x 8
RAM Size	512 x 8
Voltage - Supply (Vcc/Vdd)	2.7V ~ 5.5V
Data Converters	A/D 11x10b
Oscillator Type	Internal
Operating Temperature	-40°C ~ 125°C (TA)
Mounting Type	Surface Mount
Package / Case	20-TSSOP (0.173", 4.40mm Width)
Supplier Device Package	20-TSSOP
Purchase URL	https://www.e-xfl.com/product-detail/microchip-technology/attiny167-15xz

2. AVR CPU Core

2.1 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 2-1. 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.

2.7.1 Interrupt Behavior

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) */
```

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

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

- **Bits 6:4 – Res: Reserved Bits**

These bits are reserved bits in the Atmel® ATtiny87/167 and will always read as zero.

- **Bits 3:0 – CLKPS3:0: Clock Prescaler Select Bits 3 - 0**

These bits define the division factor between the selected clock source and the internal system clock. These bits can be written run-time to vary the clock frequency to suit the application requirements. As the divider divides the master clock input to the MCU, the speed of all synchronous peripherals is reduced when a division factor is used. The division factors are given in Table 4-10.

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

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

Interrupts must be disabled when changing prescaler setting in order not to disturb the procedure.

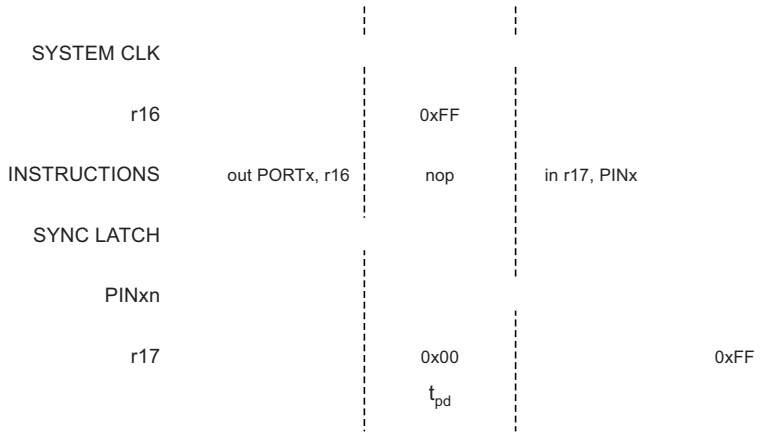
The CKDIV8 fuse determines the initial value of the CLKPS bits. If CKDIV8 is unprogrammed, the CLKPS bits will be reset to “0000”. If CKDIV8 is programmed, CLKPS bits are reset to “0011”, giving a division factor of eight at start up. This feature should be used if the selected clock source has a higher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

Table 4-10. Clock Prescaler Select

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

When reading back a software assigned pin value, a nop instruction must be inserted as indicated in Figure 9-5. The out instruction sets the “SYNC LATCH” signal at the positive edge of the clock. In this case, the delay tpd through the synchronizer is 1 system clock period.

Figure 9-5. Synchronization When Reading a Software Assigned Pin Value



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

Assembly Code Example[1]

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

C Code Example

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

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

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

As for the normal mode of operation, the TOV1 flag is set in the same timer clock cycle that the counter counts from MAX to 0x0000.

12.9.3 Fast PWM Mode

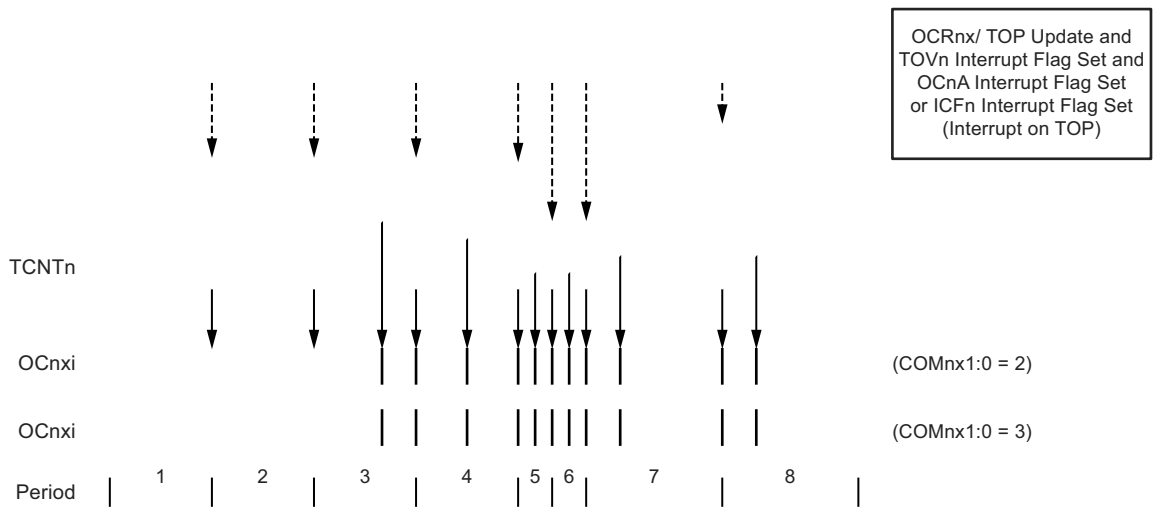
The fast pulse width modulation or fast PWM mode (WGM13:0 = 5, 6, 7, 14, or 15) provides a high frequency PWM waveform generation option. The fast PWM differs from the other PWM options by its single-slope operation. The counter counts from BOTTOM to TOP then restarts from BOTTOM. In non-inverting compare output mode, the output compare (OC1A/B) is set on the compare match between TCNT1 and OCR1A/B, and cleared at TOP. In inverting compare output mode output is cleared on compare match and set at TOP. Due to the single-slope operation, the operating frequency of the fast PWM mode can be twice as high as the phase correct and phase and frequency correct PWM modes that use dual-slope operation. This high frequency makes the fast PWM mode well suited for power regulation, rectification, and DAC applications. High frequency allows physically small sized external components (coils, capacitors), hence reduces total system cost.

The PWM resolution for fast PWM can be fixed to 8-, 9-, or 10-bit, or defined by either ICR1 or OCR1A. The minimum resolution allowed is 2-bit (ICR1 or OCR1A set to 0x0003), and the maximum resolution is 16-bit (ICR1 or OCR1A set to MAX). The PWM resolution in bits can be calculated by using the following equation:

$$R_{FPWM} = \frac{\log(TOP+1)}{\log(2)}$$

In fast PWM mode the counter is incremented until the counter value matches either one of the fixed values 0x00FF, 0x01FF, or 0x03FF (WGM13:0 = 5, 6, or 7), the value in ICR1 (WGM13:0 = 14), or the value in OCR1A (WGM13:0 = 15). The counter is then cleared at the following timer clock cycle. The timing diagram for the fast PWM mode is shown in Figure 12-8. The figure shows fast PWM mode when OCR1A or ICR1 is used to define TOP. The TCNT1 value is in the timing diagram shown as a histogram for illustrating the single-slope operation. The diagram includes non-inverted and inverted PWM outputs. The small horizontal line marks on the TCNT1 slopes represent compare matches between OCR1A/B and TCNT1. The OC1A/B interrupt flag will be set when a compare match occurs.

Figure 12-8. Fast PWM Mode, Timing Diagram



The timer/counter overflow flag (TOV1) is set each time the counter reaches TOP. In addition the OC1A or ICF1 flag is set at the same timer clock cycle as TOV1 is set when either OCR1A or ICR1 is used for defining the TOP value. If one of the interrupts are enabled, the interrupt handler routine can be used for updating the TOP and compare values.

When changing the TOP value the program must ensure that the new TOP value is higher or equal to the value of all of the compare registers. If the TOP value is lower than any of the compare registers, a compare match will never occur between the TCNT1 and the OCR1A/B. Note that when using fixed TOP values the unused bits are masked to zero when any of the OCR1A/B registers are written.

- **Bit 6 – WCOL: Write COLlision Flag**

The WCOL bit is set if the SPI data register (SPDR) is written during a data transfer. The WCOL bit (and the SPIF bit) are cleared by first reading the SPI status register with WCOL set, and then accessing the SPI data register.

- **Bit 5..1 – Res: Reserved Bits**

These bits are reserved bits in the Atmel® ATtiny87/167 and will always read as zero.

- **Bit 0 – SPI2X: Double SPI Speed Bit**

When this bit is written logic one the SPI speed (SCK Frequency) will be doubled when the SPI is in master mode (see Table 13-4 on page 135). This means that the minimum SCK period will be two CPU clock periods. When the SPI is configured as slave, the SPI is only guaranteed to work at $f_{clkio}/4$ or lower.

The SPI interface on the Atmel ATtiny87/167 is also used for program memory and EEPROM downloading or uploading. See Section 21.8 “Serial Downloading” on page 218 for serial programming and verification.

13.2.5 SPI Data Register – SPDR

Bit	7	6	5	4	3	2	1	0	
	SPD7	SPD6	SPD5	SPD4	SPD3	SPD2	SPD1	SPD0	SPDR
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	X	X	X	X	X	X	X	X	Undefined

- **Bits 7:0 - SPD7:0: SPI Data**

The SPI data register is a read/write register used for data transfer between the register file and the SPI shift register. Writing to the register initiates data transmission. Reading the register causes the shift register receive buffer to be read.

13.3 Data Modes

There are four combinations of SCK phase and polarity with respect to serial data, which are determined by control bits CPHA and CPOL. The SPI data transfer formats are shown in Figure 13-3 and Figure 13-4 on page 137. Data bits are shifted out and latched in on opposite edges of the SCK signal, ensuring sufficient time for data signals to stabilize. This is clearly seen by summarizing Table 13-2 and Table 13-3, as done below:

Table 13-5. CPOL Functionality

	Leading Edge	Trailing Edge	SPI Mode
CPOL=0, CPHA=0	Sample (rising)	Setup (falling)	0
CPOL=0, CPHA=1	Setup (rising)	Sample (falling)	1
CPOL=1, CPHA=0	Sample (falling)	Setup (rising)	2
CPOL=1, CPHA=1	Setup (falling)	Sample (rising)	3

The 4-bit counter can be both read and written via the data bus, and can generate an overflow interrupt. Both the USI data register and the counter are clocked simultaneously by the same clock source. This allows the counter to count the number of bits received or transmitted and generate an interrupt when the transfer is complete. Note that when an external clock source is selected the counter counts both clock edges. In this case the counter counts the number of edges, and not the number of bits. The clock can be selected from three different sources: The USCK pin, timer/counter0 compare match or from software.

The Two-wire clock control unit can generate an interrupt when a start condition is detected on the two-wire bus. It can also generate wait states by holding the clock pin low after a start condition is detected, or after the counter overflows.

14.3 Functional Descriptions

14.3.1 Three-wire Mode

The USI three-wire mode is compliant to the serial peripheral interface (SPI) mode 0 and 1, but does not have the slave select ($\overline{SS}$) pin functionality. However, this feature can be implemented in software if necessary. Pin names used by this mode are: DI, DO, and USCK.

Figure 14-2. Three-wire Mode Operation, Simplified Diagram

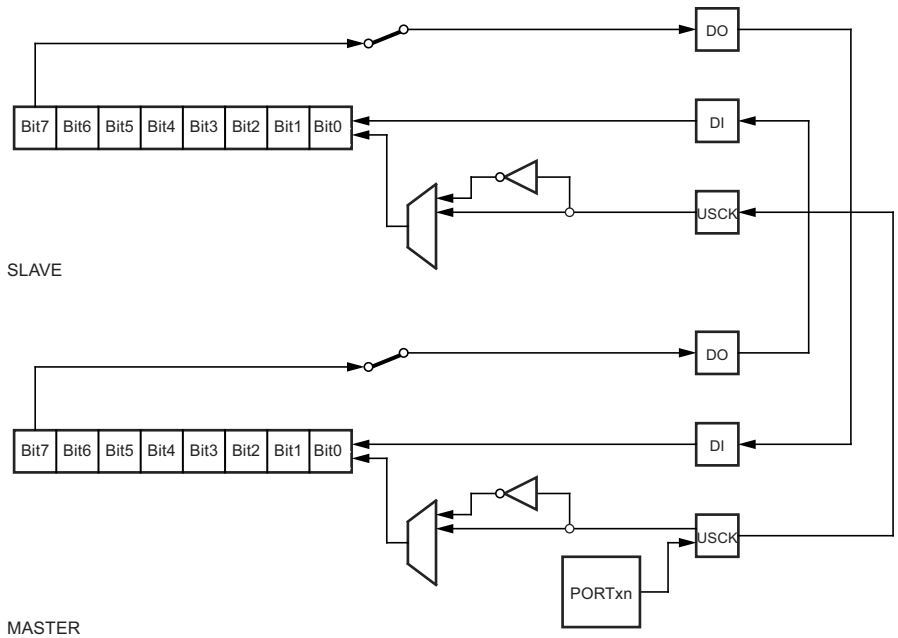
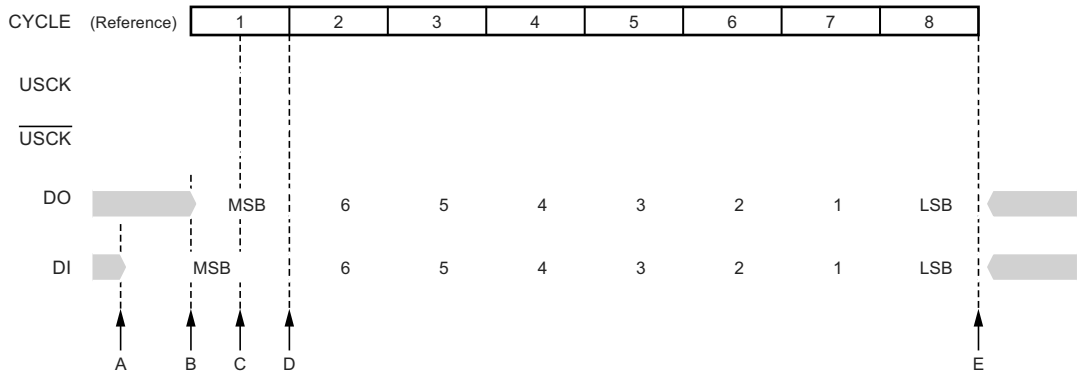


Figure 14-2 shows two USI units operating in three-wire mode, one as master and one as slave. The two USI data register are interconnected in such way that after eight USCK clocks, the data in each register are interchanged. The same clock also increments the USI's 4-bit counter. The counter overflow (interrupt) flag, or USIOIF, can therefore be used to determine when a transfer is completed.

The clock is generated by the master device software by toggling the USCK pin via the PORT register or by writing a one to the USITC bit in USICR.

Figure 14-3. Three-wire Mode, Timing Diagram



The three-wire mode timing is shown in Figure 14-3 at the top of the figure is a USCK cycle reference. One bit is shifted into the USI data register (USIDR) for each of these cycles. The USCK timing is shown for both external clock modes. In external clock mode 0 (USICS0 = 0), DI is sampled at positive edges, and DO is changed (data register is shifted by one) at negative edges. External clock mode 1 (USICS0 = 1) uses the opposite edges versus mode 0, i.e., samples data at negative and changes the output at positive edges. The USI clock modes corresponds to the SPI data mode 0 and 1.

Referring to the timing diagram (Figure 14-3), a bus transfer involves the following steps:

1. The slave device and master device sets up its data output and, depending on the protocol used, enables its output driver (mark A and B). The output is set up by writing the data to be transmitted to the USI data register. Enabling of the output is done by setting the corresponding bit in the port data direction register. Note that point A and B does not have any specific order, but both must be at least one half USCK cycle before point C where the data is sampled. This must be done to ensure that the data setup requirement is satisfied. The 4-bit counter is reset to zero.
2. The master generates a clock pulse by software toggling the USCK line twice (C and D). The bit value on the slave and master's data input (DI) pin is sampled by the USI on the first edge (C), and the data output is changed on the opposite edge (D). The 4-bit counter will count both edges.
3. Step 2. is repeated eight times for a complete register (byte) transfer.
4. After eight clock pulses (i.e., 16 clock edges) the counter will overflow and indicate that the transfer is completed. The data bytes transferred must now be processed before a new transfer can be initiated. The overflow interrupt will wake up the processor if it is set to idle mode. Depending of the protocol used the slave device can now set its output to high impedance.

14.3.2 SPI Master Operation Example

The following code demonstrates how to use the USI module as a SPI master:

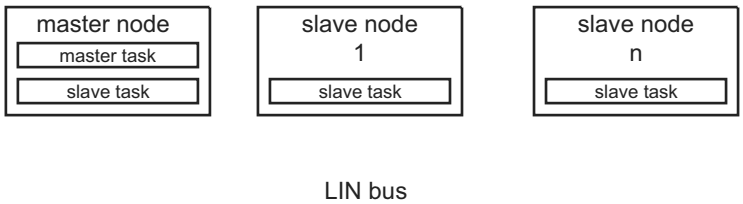
```
SPITransfer:
    sts    USIDR,r16
    ldi    r16,(1<<USIOIF)
    sts    USISR,r16
    ldi    r16,(1<<USIWM0)|(1<<USICS1)|(1<<USICLK)|(1<<USITC)
SPITransfer_loop:
    sts    USICR,r16
    lds    r16, USISR
    sbrs   r16, USIOIF
    rjmp   SPITransfer_loop
    lds    r16,USIDR
    ret
```

15.3 LIN Protocol

15.3.1 Master and Slave

A LIN cluster consists of one master task and several slave tasks. A master node contains the master task as well as a slave task. All other nodes contain a slave task only.

Figure 15-1. LIN Cluster with One Master Node and “n” Slave Nodes



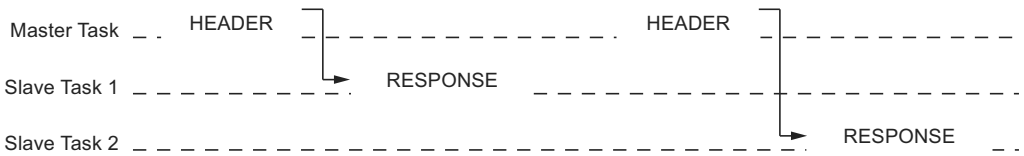
The master task decides when and which frame shall be transferred on the bus. The slave tasks provide the data transported by each frame. Both the master task and the slave task are parts of the frame handler

15.3.2 Frames

A frame consists of a header (provided by the master task) and a response (provided by a slave task).

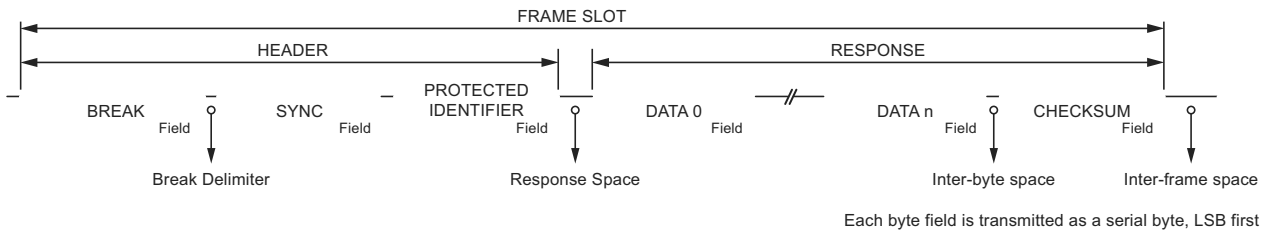
The header consists of a BREAK and SYNC pattern followed by a PROTECTED IDENTIFIER. The identifier uniquely defines the purpose of the frame. The slave task appointed for providing the response associated with the identifier transmits it. The response consists of a DATA field and a CHECKSUM field.

Figure 15-2. Master and Slave Tasks Behavior in LIN Frame



The slave tasks waiting for the data associated with the identifier receives the response and uses the data transported after verifying the checksum.

Figure 15-3. Structure of a LIN Frame



15.3.3 Data Transport

Two types of data may be transported in a frame; signals or diagnostic messages.

- Signals
  Signals are scalar values or byte arrays that are packed into the data field of a frame. A signal is always present at the same position in the data field for all frames with the same identifier.
- Diagnostic messages
  Diagnostic messages are transported in frames with two reserved identifiers. The interpretation of the data field depends on the data field itself as well as the state of the communicating nodes.

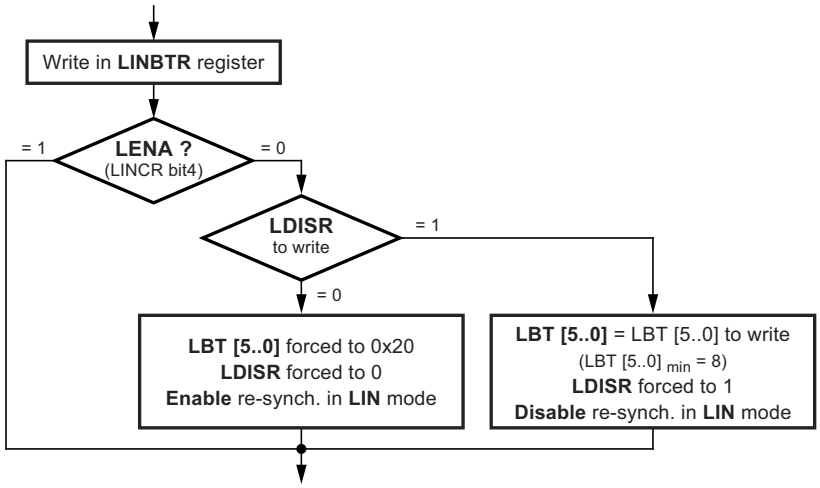
15.5.6.3 Handling LBT[5..0]

LDISR bit of LINBTR register is used to:

- Disable the re-synchronization (for instance in the case of LIN MASTER node),
- To enable the setting of LBT[5..0] (to manually adjust the baud rate especially in the case of UART mode). A minimum of 8 is required for LBT[5..0] due to the sampling operation.

Note that the LENA bit of LINCR register is important for this handling (see Figure 15-8).

Figure 15-8. Handling LBT[5..0]



15.5.7 Data Length

Section 15.4.6 “LIN Commands” on page 154 describes how to set or how are automatically set the LRXDL[3..0] or LTXDL[3..0] fields of LINDLR register before receiving or transmitting a response.

In the case of Tx response the LRXDL[3..0] will be used by the hardware to count the number of bytes already successfully sent.

In the case of Rx response the LTXDL[3..0] will be used by the hardware to count the number of bytes already successfully received.

If an error occurs, this information is useful to the programmer to recover the LIN messages.

15.5.7.1 Data Length in LIN 2.1

- If LTXDL[3..0]=0 only the CHECKSUM will be sent,
- If LRXDL[3..0]=0 the first byte received will be interpreted as the CHECKSUM,
- If LTXDL[3..0] or LRXDL[3..0] >8, values will be forced to 8 after the command setting and before sending or receiving of the first byte.

15.6.6 LIN Baud Rate Register - LINBRR

Bit	7	6	5	4	3	2	1	0	
	LDIV7	LDIV6	LDIV5	LDIV4	LDIV3	LDIV2	LDIV1	LDIV0	LINBRRL
	-	-	-	-	LDIV11	LDIV10	LDIV9	LDIV8	LINBRRH
Bit	15	14	13	12	11	10	9	8	
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 15:12 - Reserved Bits**

These bits are reserved for future use. For compatibility with future devices, they must be written to zero when LINBRR is written.

- **Bits 11:0 - LDIV[11:0]: Scaling of $clk_{i/o}$ Frequency**

The LDIV value is used to scale the entering $clk_{i/o}$ frequency to achieve appropriate LIN or UART baud rate.

15.6.7 LIN Data Length Register - LINDLR

Bit	7	6	5	4	3	2	1	0	
	LTXDL3	LTXDL2	LTXDL1	LTXDL0	LRXDL3	LRXDL2	LRXDL1	LRXDL0	LINDLR
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 7:4 - LTXDL[3:0]: LIN Transmit Data Length**

In LIN mode, this field gives the number of bytes to be transmitted (clamped to 8 Max).

In UART mode this field is unused.

- **Bits 3:0 - LRXDL[3:0]: LIN Receive Data Length**

In LIN mode, this field gives the number of bytes to be received (clamped to 8 Max).

In UART mode this field is unused.

15.6.8 LIN Identifier Register - LINIDR

Bit	7	6	5	4	3	2	1	0	
	LP1	LP0	LID5 / LDL1	LID4 / LDL0	LID3	LID2	LID1	LID0	LINIDR
Read/Write	R	R	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bits 7:6 - LP[1:0]: Parity**

In LIN mode:

```
LP0 = LID4 ^ LID2 ^ LID1 ^ LID0
LP1 = ! ( LID1 ^ LID3 ^ LID4 ^ LID5 )
```

In UART mode this field is unused.

17. ADC – Analog to Digital Converter

17.1 Features

- 10-bit resolution
- 1.0 LSB integral non-linearity
- ±2 LSB absolute accuracy
- 13 - 260µs conversion time (low - high resolution)
- Up to 15kSPS at maximum resolution
- 11 multiplexed single ended input channels
- 8 differential input pairs with selectable gain
- Temperature sensor input channel
- Voltage from internal current source driving (ISRC)
- Optional left adjustment for ADC result readout
- 0 - AVcc ADC input voltage range
- Selectable 1.1V/2.56V ADC voltage reference
- Free running or single conversion mode
- ADC start conversion by auto triggering on interrupt sources
- Interrupt on ADC conversion complete
- Sleep mode noise canceler
- Unipolar/bipolar input mode
- Input polarity reversal mode

17.2 Overview

The Atmel® ATtiny87/167 features a 10-bit successive approximation ADC. The ADC is connected to a 11-channel analog multiplexer which allows 16 differential voltage input combinations and 11 single-ended voltage inputs constructed from the pins PA7..PA0 or PB7..PB4. The differential input is equipped with a programmable gain stage, providing amplification steps of 8x or 20x on the differential input voltage before the A/D conversion. The single-ended voltage inputs refer to 0V (AGND).

The ADC contains a sample and hold circuit which ensures that the input voltage to the ADC is held at a constant level during conversion. A block diagram of the ADC is shown in Figure 17-1 on page 177.

Internal reference voltages of nominally 1.1V or 2.56V are provided On-chip. Alternatively, AVcc can be used as reference voltage for single ended channels. There are also options to output the internal 1.1V or 2.56V reference voltages or to input an external voltage reference and turn-off the internal voltage reference. These options are selected using the REFS[1:0] bits of the ADMUX control register and using AREFEN and XREFEN bits of the AMISCR control register.

19.4 Software Break Points

DebugWIRE supports program memory break points by the AVR® BREAK instruction. Setting a break point in Atmel® AVR Studio® will insert a BREAK instruction in the program memory. The instruction replaced by the BREAK instruction will be stored. When program execution is continued, the stored instruction will be executed before continuing from the program memory. A break can be inserted manually by putting the BREAK instruction in the program.

The flash must be re-programmed each time a break point is changed. This is automatically handled by AVR Studio through the debugWIRE interface. The use of break points will therefore reduce the flash data retention. Devices used for debugging purposes should not be shipped to end customers.

19.5 Limitations of DebugWIRE

The debugWIRE communication pin (dW) is physically located on the same pin as external reset (RESET). An external reset source is therefore not supported when the debugWIRE is enabled.

The debugWIRE system accurately emulates all I/O functions when running at full speed, i.e., when the program in the CPU is running. When the CPU is stopped, care must be taken while accessing some of the I/O registers via the debugger (AVR Studio).

A programmed DWEN fuse enables some parts of the clock system to be running in all sleep modes. This will increase the power consumption while in sleep. Thus, the DWEN Fuse should be disabled when debugWire is not used.

19.6 DebugWIRE Related Register in I/O Memory

The following section describes the registers used with the debugWire.

19.6.1 DebugWIRE Data Register – DWDR

Bit	7	6	5	4	3	2	1	0	
	DWDR[7:0]								DWDR
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

The DWDR register provides a communication channel from the running program in the MCU to the debugger. This register is only accessible by the debugWIRE and can therefore not be used as a general purpose register in the normal operations.

20. Flash Programming

The device provides a self-programming mechanism for downloading and uploading program code by the MCU itself. The self-programming can use any available data interface (i.e. LIN, USART, ...) and associated protocol to read code and write (program) that code into the program memory.

The program memory is updated in a page by page fashion. Before programming a page with the data stored in the temporary page buffer, the page must be erased. The temporary page buffer is filled one word at a time using SPM and the buffer can be filled either before the page erase command or between a page erase and a page write operation:

- Alternative 1, fill the buffer before a page erase
  - Fill temporary page buffer
  - Perform a page erase
  - Perform a page write
- Alternative 2, fill the buffer after page erase
  - Perform a page erase
  - Fill temporary page buffer
  - Perform a page write

If only a part of the page needs to be changed, the rest of the page must be stored (for example in the temporary page buffer) before the erase, and then be re-written. When using alternative 1, the boot loader provides an effective read-modify-write feature which allows the user software to first read the page, do the necessary changes, and then write back the modified data. If alternative 2 is used, it is not possible to read the old data while loading since the page is already erased. The temporary page buffer can be accessed in a random sequence. It is essential that the page address used in both the page erase and page write operation is addressing the same page.

20.1 Self-programming the Flash

20.1.1 Performing Page Erase by SPM

To execute page erase, set up the address in the Z-pointer, write “00000011_b” to SPMCSR and execute SPM within four clock cycles after writing SPMCSR. The data in R1 and R0 is ignored. The page address must be written to PCPAGE in the Z-register. Other bits in the Z-pointer will be ignored during this operation.

- The CPU is halted during the page erase operation.

20.1.2 Filling the Temporary Buffer (Page Loading)

To write an instruction word, set up the address in the Z-pointer and data in R1:R0, write “00000001_b” to SPMCSR and execute SPM within four clock cycles after writing SPMCSR. The content of PCWORD in the Z-register is used to address the data in the temporary buffer. The temporary buffer will auto-erase after a page write operation or by writing the CTPB bit in SPMCSR. It is also erased after a system reset. Note that it is not possible to write more than one time to each address without erasing the temporary buffer.

If the EEPROM is written in the middle of an SPM page load operation, all data loaded will be lost.

20.1.3 Performing a Page Write

To execute page write, set up the address in the Z-pointer, write “00000101_b” to SPMCSR and execute SPM within four clock cycles after writing SPMCSR. The data in R1 and R0 is ignored. The page address must be written to PCPAGE. Other bits in the Z-pointer must be written to zero during this operation.

- The CPU is halted during the Page Write operation.

Table 21-5. Fuse Low Byte

Fuse Low Byte	Bit No	Description	Default Value
CKDIV8[4]	7	Divide clock by 8	0 (programmed)
CKOUT[3]	6	Clock output	1 (unprogrammed)
SUT1	5	Select start-up time	1 (unprogrammed)[1]
SUT0	4	Select start-up time	0 (programmed)[1]
CKSEL3	3	Select Clock source	0 (programmed)[2]
CKSEL2	2	Select Clock source	0 (programmed)[2]
CKSEL1	1	Select Clock source	1 (unprogrammed)[2]
CKSEL0	0	Select Clock source	0 (programmed)[2]

Notes:
1. The default value of SUT1..0 results in maximum start-up time for the default clock source. See Table 4-4 on page 28 for details.
2. The default setting of CKSEL3..0 results in internal RC Oscillator @ 8 MHz. See Table 4-3 on page 28 for details.
3. The CKOUT Fuse allows the system clock to be output on PORTB5. See Section 4.2.7 “Clock Output Buffer” on page 32 for details.
4. See Section 4.4 “System Clock Prescaler” on page 38for details.

21.2.1 Latching of Fuses

The fuse values are latched when the device enters programming mode and changes of the fuse values will have no effect until the part leaves programming mode. This does not apply to the EESAVE Fuse which will take effect once it is programmed. The fuses are also latched on power-up in normal mode.

21.3 Signature Bytes

All Atmel® microcontrollers have a three-byte signature code which identifies the device. This code can be read in both serial and parallel mode, also when the device is locked. The three bytes reside in a separate address space.

Table 21-6. Signature Bytes

Device	Address	Value	Signature Byte Description
ATtiny87	0	0x1E	Indicates manufactured by Atmel
	1	0x93	Indicates 8KB flash memory
	2	0x87	Indicates ATtiny87 device when address 1 contains 0x93
ATtiny167	0	0x1E	Indicates manufactured by Atmel
	1	0x94	Indicates 16KB flash memory
	2	0x87	Indicates ATtiny167 device when address 1 contains 0x94

21.4 Calibration Byte

The Atmel ATtiny87/167 has a byte calibration value for the internal RC oscillator. This byte resides in the high byte of address 0x000 in the signature address space. During reset, this byte is automatically written into the OSCCAL register to ensure correct frequency of the calibrated RC oscillator.

C. Load Data Low Byte

1. Set XA1, XA0 to “0,1”. This enables data loading.
2. Set DATA = data low byte (0x00 - 0xFF).
3. Give XTAL1 a positive pulse. This loads the data byte.

D. Load Data High Byte

1. Set BS1 to “1”. This selects high data byte.
2. Set XA1, XA0 to “0,1”. This enables data loading.
3. Set DATA = data high byte (0x00 - 0xFF).
4. Give XTAL1 a positive pulse. This loads the data byte.

E. Latch Data

1. Set BS1 to “1”. This selects high data byte.
2. Give PAGEL a positive pulse. This latches the data bytes. (See Figure 21-3 on page 214 for signal waveforms)

F. Repeat B through E until the entire buffer is filled or until all data within the page is loaded.

While the lower bits in the address are mapped to words within the page, the higher bits address the pages within the FLASH. This is illustrated in Figure 21-2. Note that if less than eight bits are required to address words in the page (pagesize < 256), the most significant bit(s) in the address low byte are used to address the page when performing a page write.

G. Load Address High byte

1. Set XA1, XA0 to “0,0”. This enables address loading.
2. Set BS1 to “1”. This selects high address.
3. Set DATA = address high byte (0x00 - 0xFF).
4. Give XTAL1 a positive pulse. This loads the address high byte.

H. Program Page

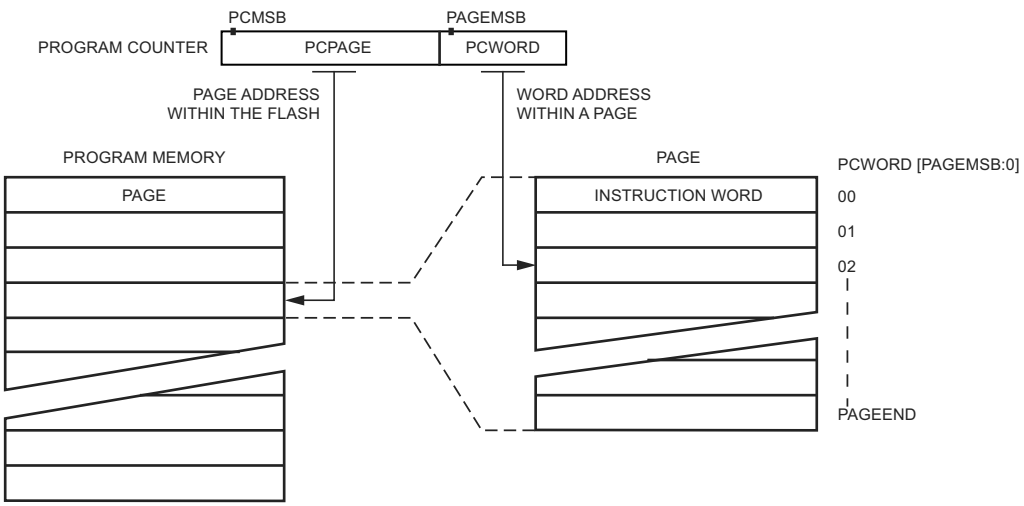
1. Give $\overline{WR}$ a negative pulse. This starts programming of the entire page of data. RDY/$\overline{BSY}$ goes low.
2. Wait until RDY/$\overline{BSY}$ goes high (See Figure 21-3 on page 214 for signal waveforms).

I. Repeat B through H until the entire flash is programmed or until all data has been programmed.

J. End Page Programming

1. 1. Set XA1, XA0 to “1,0”. This enables command loading.
2. Set DATA to “0000 0000$_b$”. This is the command for no operation.
3. Give XTAL1 a positive pulse. This loads the command, and the internal write signals are reset.

Figure 21-2. Addressing the Flash Which is Organized in Pages



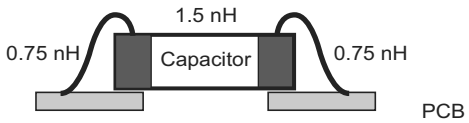
23. Decoupling Capacitors

The operating frequency (i.e. system clock) of the processor determines in 95% of cases the value needed for microcontroller decoupling capacitors.

The hypotheses used as first evaluation for decoupling capacitors are:

- The operating frequency (f_{op}) supplies itself the maximum peak levels of noise. The main peaks are located at f_{op} and $2 \times f_{op}$.
- An SMC capacitor connected to 2 micro-vias on a PCB has the following characteristics:
  - 1.5 nH from the connection of the capacitor to the PCB,
  - 1.5 nH from the capacitor intrinsic inductance.

Figure 23-1. Capacitor description



According to the operating frequency of the product, the decoupling capacitances are chosen considering the frequencies to filter, f_{op} and $2 \times f_{op}$.

The relation between frequencies to cut and decoupling characteristics are defined by:

$$fop = \frac{1}{2\pi\sqrt{LC_1}} \text{ and } 2 \times fop = \frac{1}{2\pi\sqrt{LC_2}}$$

where:

- L: the inductance equivalent to the global inductance on the Vcc/Gnd lines.
- C_1 and C_2: decoupling capacitors ($C_1 = 4 \times C_2$).

Then, in normalized value range, the decoupling capacitors become:

Table 23-1. Decoupling Capacitors versus Frequency

f_{op} , operating frequency	C_1	C_2
16MHz	33nF	10nF
12MHz	56nF	15nF
10MHz	82nF	22nF
8MHz	120nF	33nF
6MHz	220nF	56nF
4MHz	560nF	120nF

These decoupling capacitors must to be implemented as close as possible to each pair of power supply pins:

- 16-17 for logic sub-system,
- 5-6 for analogical sub-system.

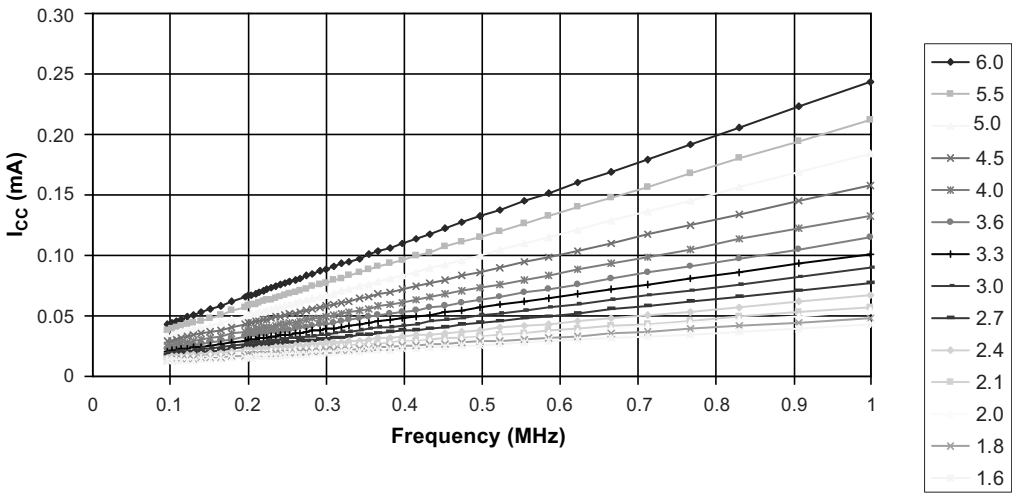
Nevertheless, a bulk capacitor of 10-47µF is also needed on the power distribution network of the PCB, near the power source.

For further information, please refer to application notes AVR® 040 “EMC design considerations” and AVR042 “hardware design considerations” on the Atmel® web site.

Figure 24-19. Calibrated 8.0MHz RC Oscillator Frequency versus OSCCAL Value

24.8 Current Consumption in Reset

Figure 24-20. Reset Supply Current versus Vcc, Frequencies 0.1 - 1.0MHz (Excluding Current Through the Reset Pull-up)



26. Instruction Set Summary (Continued)

Mnemonics	Operands	Description	Operation	Flags	#Clocks
Data Transfer Instructions					
MOV	Rd, Rr	Move between registers	Rd ←Rr	None	1
MOVW	Rd, Rr	Copy register word	Rd+1:Rd ←Rr+1:Rr	None	1
LDI	Rd, K	Load immediate	Rd ← K	None	1
LD	Rd, X	Load indirect	Rd ← (X)	None	2
LD	Rd, X+	Load indirect and post-inc.	Rd← (X), X ← X + 1	None	2
LD	Rd, - X	Load indirect and pre-dec.	X ← X –1, Rd ← (X)	None	2
LD	Rd, Y	Load indirect	Rd ← (Y)	None	2
LD	Rd, Y+	Load indirect and post-inc.	Rd ← (Y), Y ←Y + 1	None	2
LD	Rd, - Y	Load indirect and pre-dec.	Y←Y –1, Rd ← (Y)	None	2
LDD	Rd,Y+q	Load indirect with displacement	Rd ← (Y + q)	None	2
LD	Rd, Z	Load indirect	Rd ← (Z)	None	2
LD	Rd, Z+	Load lidirect and post-inc.	Rd ← (Z), Z ← Z+1	None	2
LD	Rd, -Z	Load indirect and pre-dec.	Z ← Z–1, Rd ← (Z)	None	2
LDD	Rd, Z+q	Load indirect with displacement	Rd ← (Z + q)	None	2
LDS	Rd, k	Load direct from SRAM	Rd ← (k)	None	2
ST	X, Rr	Store indirect	(X) ← Rr	None	2
ST	X+, Rr	Store indirect and post-inc.	(X) ← Rr, X ← X + 1	None	2
ST	- X, Rr	Store indirect and pre-dec.	X ← X– 1, (X) ← Rr	None	2
ST	Y, Rr	Store indirect	(Y) ← Rr	None	2
ST	Y+, Rr	Store indirect and post-inc.	(Y) ← Rr, Y ←Y + 1	None	2
ST	- Y, Rr	Store indirect and pre-dec.	Y ←Y–1, (Y) ← Rr	None	2
STD	Y+q,Rr	Store indirect with displacement	(Y + q) ← Rr	None	2
ST	Z, Rr	Store indirect	(Z)← Rr	None	2
ST	Z+, Rr	Store indirect and post-inc.	(Z) ← Rr, Z ← Z + 1	None	2
ST	-Z, Rr	Store indirect and pre-dec.	Z ← Z – 1, (Z) ← Rr	None	2
STD	Z+q,Rr	Store indirect with displacement	(Z + q) ← Rr	None	2
STS	k, Rr	Store direct to SRAM	(k) ← Rr	None	2
LPM		Load program memory	R0 ← (Z)	None	3
LPM	Rd, Z	Load program memory	Rd ←(Z)	None	3
LPM	Rd, Z+	Load program memory and post-inc	Rd ← (Z), Z ← Z+1	None	3
SPM		Store program memory	(Z) ← R1:R0	None	-
IN	Rd, P	In port	Rd ← P	None	1
OUT	P, Rr	Out port	P ← Rr	None	1
PUSH	Rr	Push register on stack	STACK ← Rr	None	2
POP	Rd	Pop register from stack	Rd ← STACK	None	2
MCU Control Instructions					
NOP		No operation		None	1
SLEEP		sleep	(see specific descr. for sleep function)	None	1
WDR		Watchdog reset	(see specific descr. for WDR/timer)	None	1
BREAK		Break	For on-chip debug only	None	N/A

2. Gain control of the crystal oscillator

   The crystal oscillator (0.4 -> 16MHz) doesn't latch its gain control (CKSEL/CSEL[2..0] bits):

   a. The '*recover system clock source*' command doesn't returns CSEL[2..0] bits.
   b. The gain control can be modified on the fly if CLKSELR changes.

   Problem Fix/Workaround

   a. No workaround.
   b. As soon as possible, after any CLKSELR modification, re-write the appropriate crystal oscillator setting (CSEL[3]=1 and CSEL[2..0] / CSUT[1..0] bits) in CLKSELR.

   Code example:

```
; Select crystal oscillator ( 16MHz crystal, fast rising power)
   ldi    temp1,((    <<CSEL0)|(    <<CSUT0))
   sts    CLKSELR, temp1
; Enable clock source (crystal oscillator)
   ldi    temp2,( <<CLKCCE)
   ldi    temp3,(    <<CLKC0)   ; CSEL = "0010"
   sts    CLKCSR,temp2          ; Enable CLKCSR register access
   sts    CLKCSR,temp3          ; Enable crystal oscillator clock
; Clock source switch
   ldi    temp3,(    <<CLKC0)   ; CSEL = "0100"
   sts    CLKCSR,temp2          ; Enable CLKCSR register access
   sts    CLKCSR,temp3          ; Clock source switch
; Select watchdog clock ( 128KHz, fast rising power)
   ldi    temp3,((    <<CSEL0)|(    <<CSUT0))
   sts    CLKSELR, temp3        ; (*)
; (*) !!! Loose gain control of crystal oscillator !!!
; ==> WORKAROUND ...
   sts    CLKSELR, temp1
; ...
```

3. *Disable clock source*' command remains enabled

   In the dynamic clock switch module, the '*disable clock source*' command remains running after disabling the targeted clock source (the clock source is set in the CLKSELR register).

   Problem Fix/Workaround

   After a '*disable clock source*' command, reset the CLKCSR register writing 0x80.

   Code example:

```
; Select crystal oscillator
   ldi    temp1,(    <<CSEL0)
   sts    CLKSELR, temp1
; Disable clock source (crystal oscillator)
   ldi    temp2,( <<CLKCCE)
   ldi    temp3,(    <<CLKC0)   ; CSEL = "0001"
   sts    CLKCSR,temp2          ; Enable CLKCSR register access
   sts    CLKCSR,temp3          ; (*) Disable crystal oscillator clock
; (*) !!! At this moment, if any other clock source is selected by CLKSELR,
;         this clock source will also stop !!!
; ==> WORKAROUND ...
   sts    CLKCSR,temp2
```