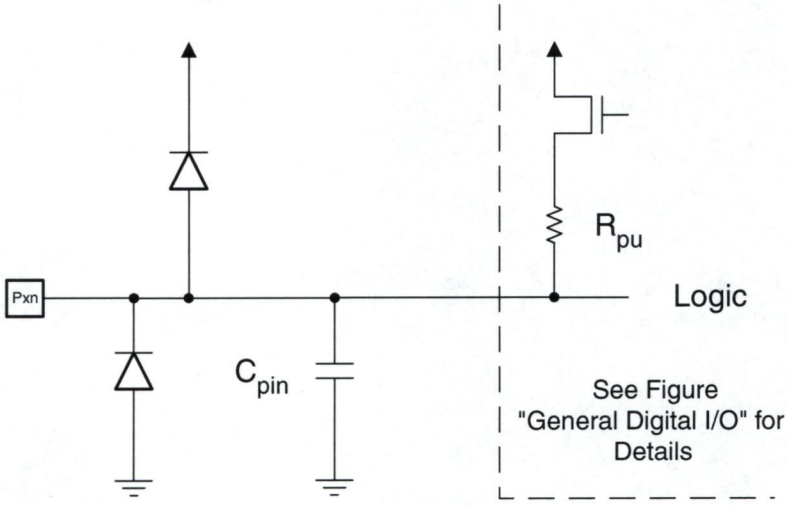
I/O Ports

Introduction

All AVR ports have true Read-Modify-Write functionality when used as general digital I/O ports. This means that the direction of one port pin can be changed without unintentionally changing the direction of any other pin with the SBI and CBI instructions. The same applies when changing drive value (if configured as output) or enabling/disabling of pull-up resistors (if configured as input). Each output buffer has symmetrical drive characteristics with both high sink and source capability. The pin driver is strong enough to drive LED displays directly. All port pins have individually selectable pull-up resistors with a supply-voltage invariant resistance. All I/O pins have protection diodes to both V_{CC} and Ground as indicated in Figure 21. Refer to "Electrical Characteristics" on page 242 for a complete list of parameters.

Figure 21. I/O Pin Equivalent Schematic



All registers and bit references in this section are written in general form. A lower case "x" represents the numbering letter for the port, and a lower case "n" represents the bit number. However, when using the register or bit defines in a program, the precise form must be used (i.e., PORTB3 for bit 3 in Port B, here documented generally as PORTxn). The physical I/O Registers and bit locations are listed in "Register Description for I/O Ports" on page 65.

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

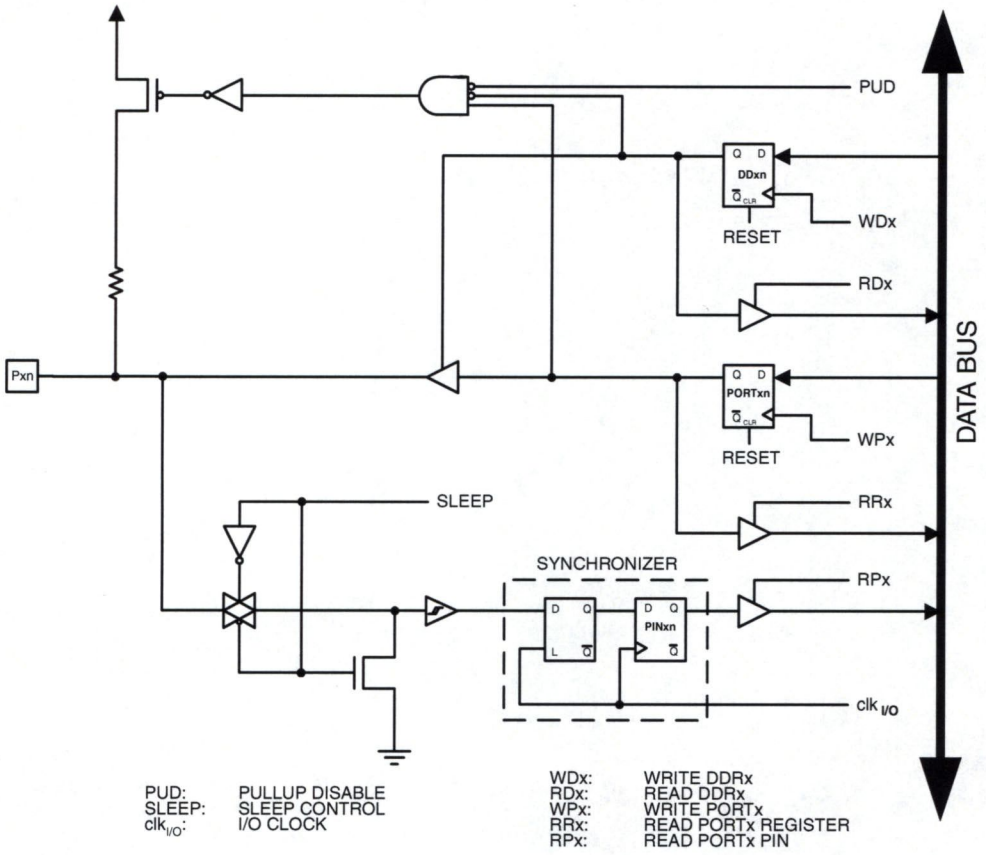
Using the I/O port as General Digital I/O is described in "Ports as General Digital I/O" on page 51. Most port pins are multiplexed with alternate functions for the peripheral features on the device. How each alternate function interferes with the port pin is described in "Alternate Port Functions" on page 56. Refer to the individual module sections for a full description of the alternate functions.

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

Ports as General Digital I/O

The ports are bi-directional I/O ports with optional internal pull-ups. Figure 22 shows a functional description of one I/O port pin, here generically called Pxn.

Figure 22. General Digital I/O[1]



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

Configuring the Pin

Each port pin consists of 3 Register bits: DDxn, PORTxn, and PINxn. As shown in "Register Description for I/O Ports" on page 65, the DDxn bits are accessed at the DDRx I/O address, the PORTxn bits at the PORTx I/O address, and the PINxn bits at the PINx I/O address.

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

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

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

When switching between tri-state ({DDxn, PORTxn} = 0b00) and output high ({DDxn, PORTxn} = 0b11), an intermediate state with either pull-up enabled ({DDxn, PORTxn} = 0b01) or output low ({DDxn, PORTxn} = 0b10) must occur. Normally, the pull-up enabled state is fully acceptable, as a high-impedant environment will not notice the difference between a strong high driver

and a pull-up. If this is not the case, the PUD bit in the SFIOR Register can be set to disable all pull-ups in all ports.

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

Table 20 summarizes the control signals for the pin value.

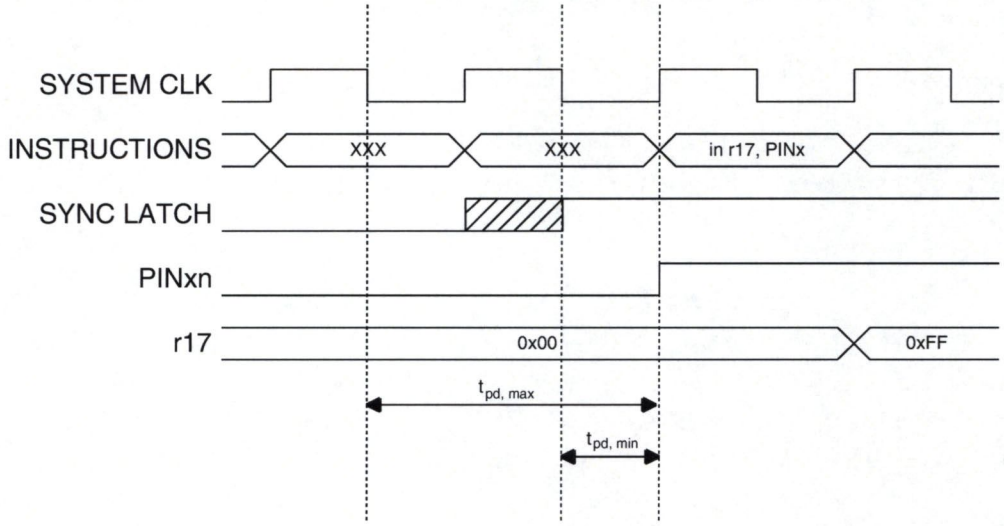
Table 20. Port Pin Configurations

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

Reading the Pin Value

Independent of the setting of Data Direction bit DDxn, the port pin can be read through the PINxn Register Bit. As shown in Figure 22, the PINxn Register bit and the preceding latch constitute a synchronizer. This is needed to avoid metastability if the physical pin changes value near the edge of the internal clock, but it also introduces a delay. Figure 23 shows a timing diagram of the synchronization when reading an externally applied pin value. The maximum and minimum propagation delays are denoted $t_{pd,max}$ and $t_{pd,min}$, respectively.

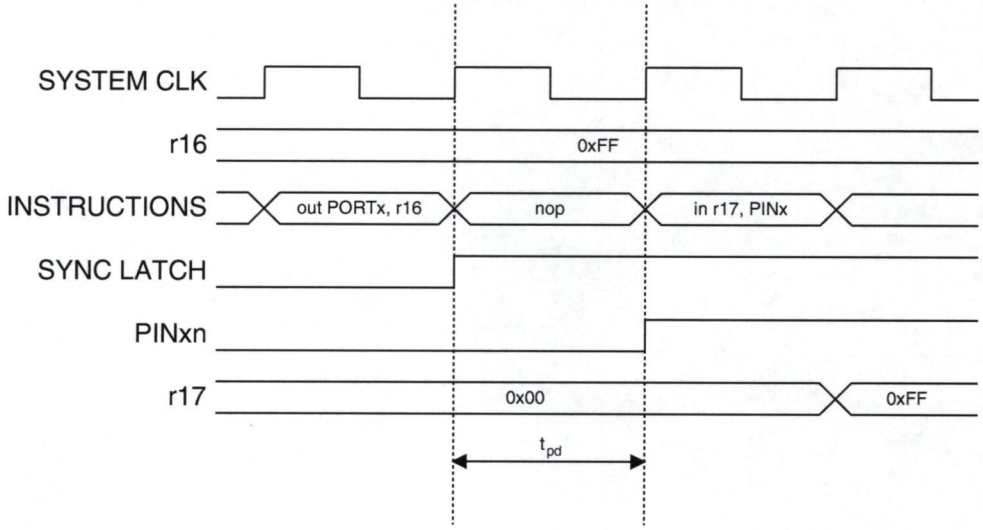
Figure 23. Synchronization when Reading an Externally Applied Pin Value



Consider the clock period starting shortly *after* the first falling edge of the system clock. The latch is closed when the clock is low, and goes transparent when the clock is high, as indicated by the shaded region of the "SYNC LATCH" signal. The signal value is latched when the system clock goes low. It is clocked into the PINxn Register at the succeeding positive clock edge. As indicated by the two arrows $t_{pd,max}$ and $t_{pd,min}$, a single signal transition on the pin will be delayed between ½ and 1-½ system clock period depending upon the time of assertion.

When reading back a software assigned pin value, a *nop* instruction must be inserted as indicated in Figure 24. The *out* instruction sets the "SYNC LATCH" signal at the positive edge of the clock. In this case, the delay t_{pd} through the synchronizer is 1 system clock period.

Figure 24. 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[1]

```
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*/
_NOP();
/* 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.

Digital Input Enable and Sleep Modes

As shown in Figure 22, the digital input signal can be clamped to ground at the input of the Schmitt-trigger. The signal denoted SLEEP in the figure, is set by the MCU Sleep Controller in Power-down mode, Power-save mode, and Standby mode to avoid high power consumption if some input signals are left floating, or have an analog signal level close to $V_{CC}/2$.

SLEEP is overridden for port pins enabled as External Interrupt pins. If the External Interrupt Request is not enabled, SLEEP is active also for these pins. SLEEP is also overridden by various other alternate functions as described in "Alternate Port Functions" on page 56.

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

Register Description for I/O Ports

The Port B Data Register – PORTB

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

The Port B Data Direction Register – DDRB

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

The Port B Input Pins Address – PINB

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

The Port C Data Register – PORTC

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

The Port C Data Direction Register – DDRC

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

The Port C Input Pins Address – PINC

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

The Port D Data Register – PORTD

Bit	7	6	5	4	3	2	1	0	
	PORTD7	PORTD6	PORTD5	PORTD4	PORTD3	PORTD2	PORTD1	PORTD0	PORTD
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

The Port D Data Direction Register – DDRD

Bit	7	6	5	4	3	2	1	0	
	DDD7	DDD6	DDD5	DDD4	DDD3	DDD2	DDD1	DDD0	DDRD
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

The Port D Input Pins Address – PIND

Bit	7	6	5	4	3	2	1	0	
	PIND7	PIND6	PIND5	PIND4	PIND3	PIND2	PIND1	PIND0	PIND
Read/Write	R	R	R	R	R	R	R	R	
Initial Value	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	

Analog-to-Digital Converter

Features

- 10-bit Resolution
- 0.5 LSB Integral Non-linearity
- ± 2 LSB Absolute Accuracy
- 13 - 260 µs Conversion Time
- Up to 15 kSPS at Maximum Resolution
- 6 Multiplexed Single Ended Input Channels
- 2 Additional Multiplexed Single Ended Input Channels (TQFP and QFN/MLF Package only)
- Optional Left Adjustment for ADC Result Readout
- 0 - V_{CC} ADC Input Voltage Range
- Selectable 2.56V ADC Reference Voltage
- Free Running or Single Conversion Mode
- Interrupt on ADC Conversion Complete
- Sleep Mode Noise Canceler

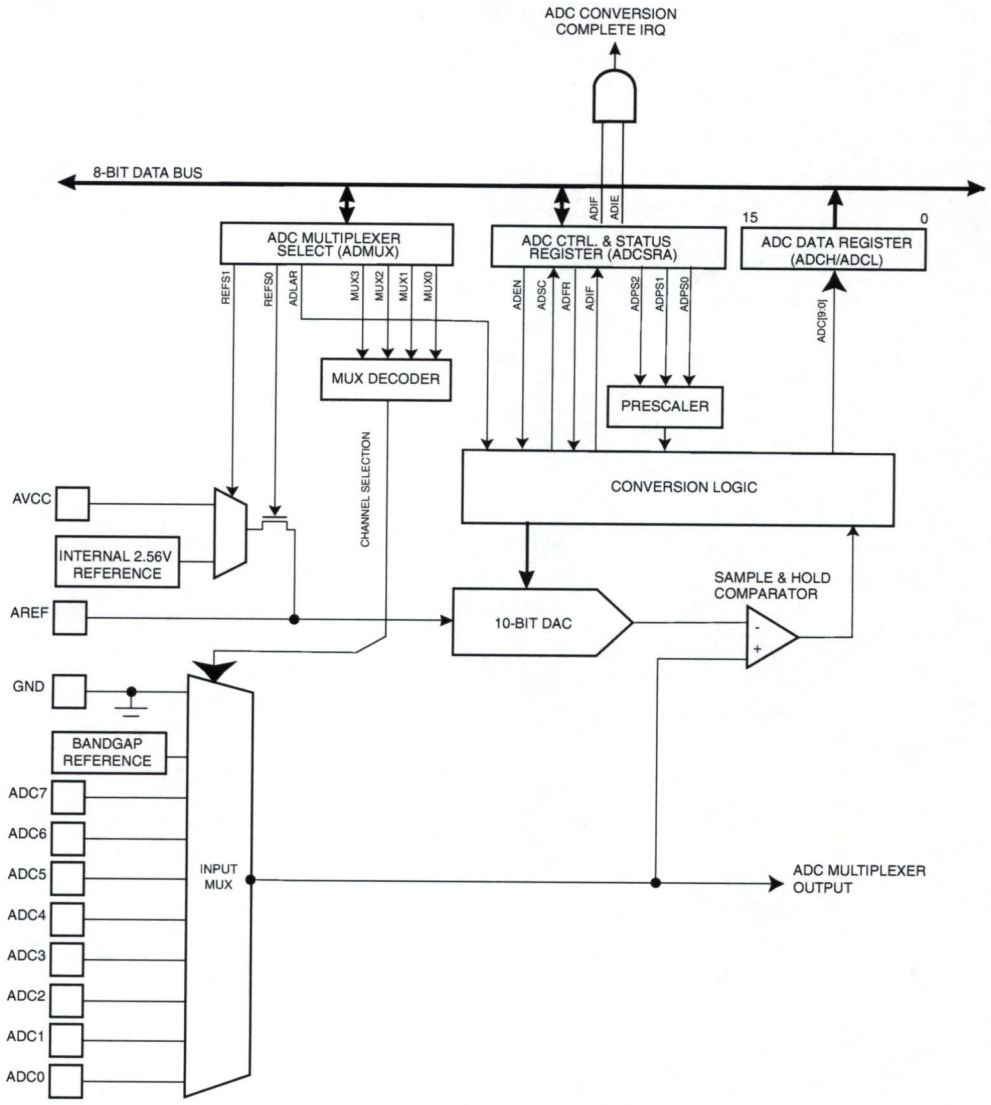
The ATmega8 features a 10-bit successive approximation ADC. The ADC is connected to an 8-channel Analog Multiplexer which allows eight single-ended voltage inputs constructed from the pins of Port C. The single-ended voltage inputs refer to 0V (GND).

The ADC contains a Sample and Hold circuit which ensures that the input voltage to the ADC is held at a constant level during conversion. A block diagram of the ADC is shown in Figure 90.

The ADC has a separate analog supply voltage pin, AV_{CC}. AV_{CC} must not differ more than ± 0.3V from V_{CC}. See the paragraph "ADC Noise Canceler" on page 201 on how to connect this pin.

Internal reference voltages of nominally 2.56V or AV_{CC} are provided On-chip. The voltage reference may be externally decoupled at the AREF pin by a capacitor for better noise performance.

Figure 90. Analog to Digital Converter Block Schematic Operation



The ADC converts an analog input voltage to a 10-bit digital value through successive approximation. The minimum value represents GND and the maximum value represents the voltage on the AREF pin minus 1 LSB. Optionally, AV_{CC} or an internal 2.56V reference voltage may be connected to the AREF pin by writing to the REFSn bits in the ADMUX Register. The internal voltage reference may thus be decoupled by an external capacitor at the AREF pin to improve noise immunity.

The analog input channel is selected by writing to the MUX bits in ADMUX. Any of the ADC input pins, as well as GND and a fixed bandgap voltage reference, can be selected as single ended inputs to the ADC. The ADC is enabled by setting the ADC Enable bit, ADEN in ADCSRA. Voltage reference and input channel selections will not go into effect until ADEN is set. The ADC does not consume power when ADEN is cleared, so it is recommended to switch off the ADC before entering power saving sleep modes.

The ADC generates a 10-bit result which is presented in the ADC Data Registers, ADCH and ADCL. By default, the result is presented right adjusted, but can optionally be presented left adjusted by setting the ADLAR bit in ADMUX.

If the result is left adjusted and no more than 8-bit precision is required, it is sufficient to read ADCH. Otherwise, ADCL must be read first, then ADCH, to ensure that the content of the Data Registers belongs to the same conversion. Once ADCL is read, ADC access to Data Registers is blocked. This means that if ADCL has been read, and a conversion completes before ADCH is read, neither register is updated and the result from the conversion is lost. When ADCH is read, ADC access to the ADCH and ADCL Registers is re-enabled.

The ADC has its own interrupt which can be triggered when a conversion completes. When ADC access to the Data Registers is prohibited between reading of ADCH and ADCL, the interrupt will trigger even if the result is lost.

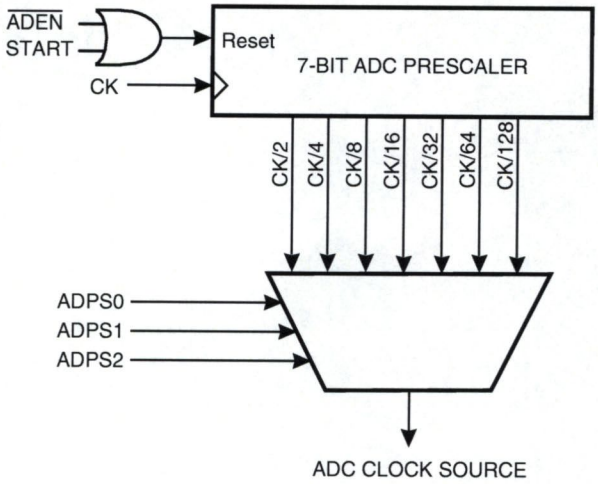
Starting a Conversion

A single conversion is started by writing a logical one to the ADC Start Conversion bit, ADSC. This bit stays high as long as the conversion is in progress and will be cleared by hardware when the conversion is completed. If a different data channel is selected while a conversion is in progress, the ADC will finish the current conversion before performing the channel change.

In Free Running mode, the ADC is constantly sampling and updating the ADC Data Register. Free Running mode is selected by writing the ADFR bit in ADCSRA to one. The first conversion must be started by writing a logical one to the ADSC bit in ADCSRA. In this mode the ADC will perform successive conversions independently of whether the ADC Interrupt Flag, ADIF is cleared or not.

Prescaling and Conversion Timing

Figure 91. ADC Prescaler



By default, the successive approximation circuitry requires an input clock frequency between 50 kHz and 200 kHz to get maximum resolution. If a lower resolution than 10 bits is needed, the input clock frequency to the ADC can be higher than 200 kHz to get a higher sample rate.

The ADC module contains a prescaler, which generates an acceptable ADC clock frequency from any CPU frequency above 100 kHz. The prescaling is set by the ADPS bits in ADCSRA. The prescaler starts counting from the moment the ADC is switched on by setting the ADEN bit in ADCSRA. The prescaler keeps running for as long as the ADEN bit is set, and is continuously reset when ADEN is low.

When initiating a single ended conversion by setting the ADSC bit in ADCSRA, the conversion starts at the following rising edge of the ADC clock cycle. A normal conversion takes 13 ADC clock cycles. The first conversion after the ADC is switched on (ADEN in ADCSRA is set) takes 25 ADC clock cycles in order to initialize the analog circuitry.

The actual sample-and-hold takes place 1.5 ADC clock cycles after the start of a normal conversion and 13.5 ADC clock cycles after the start of an first conversion. When a conversion is complete, the result is written to the ADC Data Registers, and ADIF is set. In single conversion mode, ADSC is cleared simultaneously. The software may then set ADSC again, and a new conversion will be initiated on the first rising ADC clock edge.

In Free Running mode, a new conversion will be started immediately after the conversion completes, while ADSC remains high. For a summary of conversion times, see Table 73.

Figure 92. ADC Timing Diagram, First Conversion (Single Conversion Mode)

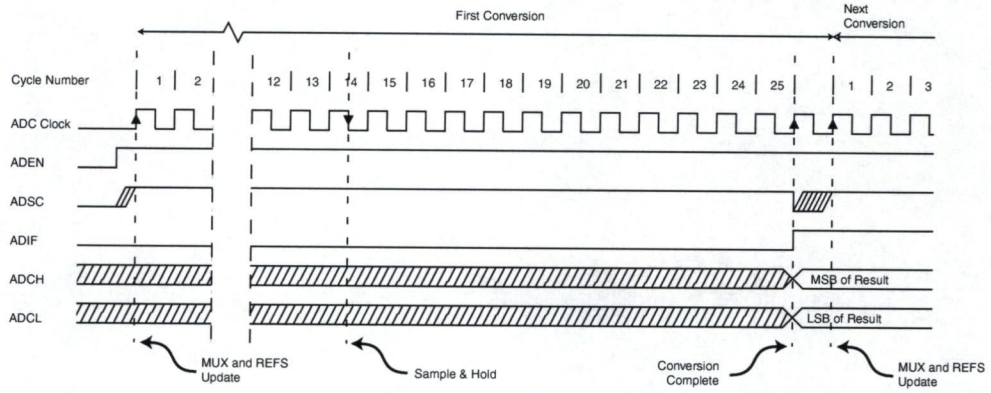


Figure 93. ADC Timing Diagram, Single Conversion

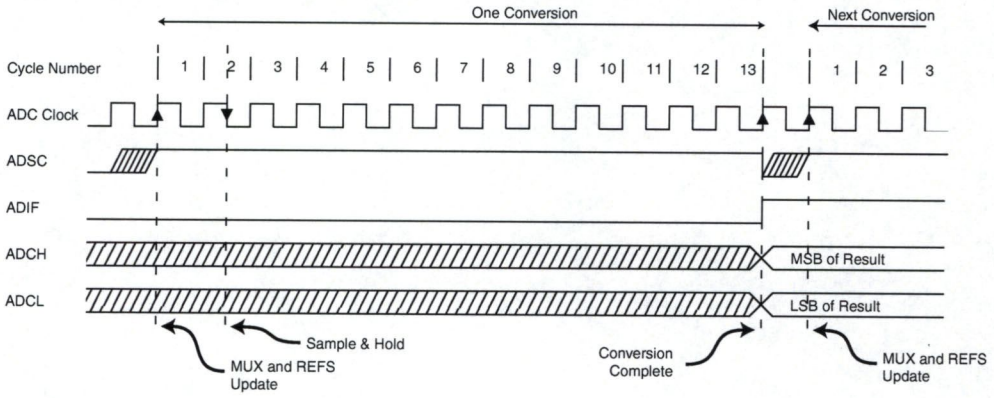


Figure 94. ADC Timing Diagram, Free Running Conversion

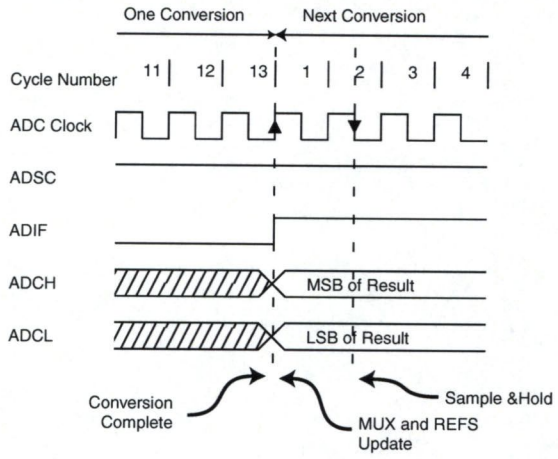


Table 73. ADC Conversion Time

Condition	Sample & Hold (Cycles from Start of Conversion)	Conversion Time (Cycles)
Extended conversion	13.5	25
Normal conversions, single ended	1.5	13

Changing Channel or Reference Selection

The MUXn and REFS1:0 bits in the ADMUX Register are single buffered through a temporary register to which the CPU has random access. This ensures that the channels and reference selection only takes place at a safe point during the conversion. The channel and reference selection is continuously updated until a conversion is started. Once the conversion starts, the channel and reference selection is locked to ensure a sufficient sampling time for the ADC. Continuous updating resumes in the last ADC clock cycle before the conversion completes (ADIF in ADCSRA is set). Note that the conversion starts on the following rising ADC clock edge after ADSC is written. The user is thus advised not to write new channel or reference selection values to ADMUX until one ADC clock cycle after ADSC is written.

If both ADFR and ADEN is written to one, an interrupt event can occur at any time. If the ADMUX Register is changed in this period, the user cannot tell if the next conversion is based on the old or the new settings. ADMUX can be safely updated in the following ways:

1. When ADFR or ADEN is cleared.
2. During conversion, minimum one ADC clock cycle after the trigger event.
3. After a conversion, before the Interrupt Flag used as trigger source is cleared.

When updating ADMUX in one of these conditions, the new settings will affect the next ADC conversion.

ADC Input Channels

When changing channel selections, the user should observe the following guidelines to ensure that the correct channel is selected:

In Single Conversion mode, always select the channel before starting the conversion. The channel selection may be changed one ADC clock cycle after writing one to ADSC. However, the simplest method is to wait for the conversion to complete before changing the channel selection.

In Free Running mode, always select the channel before starting the first conversion. The channel selection may be changed one ADC clock cycle after writing one to ADSC. However, the simplest method is to wait for the first conversion to complete, and then change the channel selection. Since the next conversion has already started automatically, the next result will reflect the previous channel selection. Subsequent conversions will reflect the new channel selection.

ADC Voltage Reference

The reference voltage for the ADC (V_{REF}) indicates the conversion range for the ADC. Single ended channels that exceed V_{REF} will result in codes close to 0x3FF. V_{REF} can be selected as either AV_{CC}, internal 2.56V reference, or external AREF pin.

AV_{CC} is connected to the ADC through a passive switch. The internal 2.56V reference is generated from the internal bandgap reference (V_{BG}) through an internal amplifier. In either case, the external AREF pin is directly connected to the ADC, and the reference voltage can be made more immune to noise by connecting a capacitor between the AREF pin and ground. V_{REF} can also be measured at the AREF pin with a high impedant voltmeter. Note that V_{REF} is a high impedant source, and only a capacitive load should be connected in a system.

If the user has a fixed voltage source connected to the AREF pin, the user may not use the other reference voltage options in the application, as they will be shorted to the external voltage. If no external voltage is applied to the AREF pin, the user may switch between AV_{CC} and 2.56V as reference selection. The first ADC conversion result after switching reference voltage source may be inaccurate, and the user is advised to discard this result.

ADC Noise Canceler

The ADC features a noise canceler that enables conversion during sleep mode to reduce noise induced from the CPU core and other I/O peripherals. The noise canceler can be used with ADC Noise Reduction and Idle mode. To make use of this feature, the following procedure should be used:

1. Make sure that the ADC is enabled and is not busy converting. Single Conversion mode must be selected and the ADC conversion complete interrupt must be enabled.
2. Enter ADC Noise Reduction mode (or Idle mode). The ADC will start a conversion once the CPU has been halted.
3. If no other interrupts occur before the ADC conversion completes, the ADC interrupt will wake up the CPU and execute the ADC Conversion Complete interrupt routine. If another interrupt wakes up the CPU before the ADC conversion is complete, that interrupt will be executed, and an ADC Conversion Complete interrupt request will be generated when the ADC conversion completes. The CPU will remain in Active mode until a new sleep command is executed.

Note that the ADC will not be automatically turned off when entering other sleep modes than Idle mode and ADC Noise Reduction mode. The user is advised to write zero to ADEN before entering such sleep modes to avoid excessive power consumption.

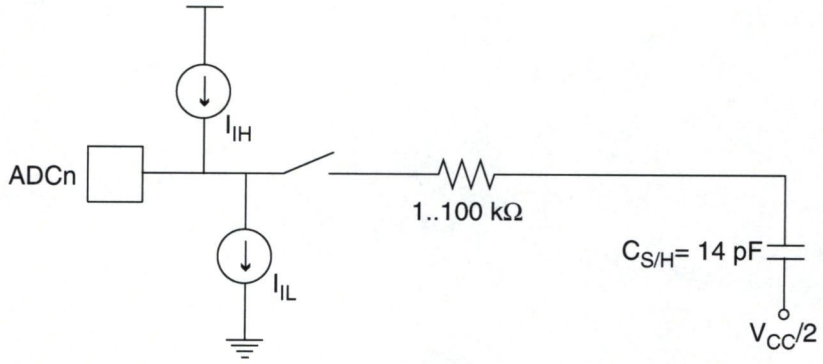
Analog Input Circuitry

The analog input circuitry for single ended channels is illustrated in Figure 95. An analog source applied to ADCn is subjected to the pin capacitance and input leakage of that pin, regardless of whether that channel is selected as input for the ADC. When the channel is selected, the source must drive the S/H capacitor through the series resistance (combined resistance in the input path).

The ADC is optimized for analog signals with an output impedance of approximately 10 kΩ or less. If such a source is used, the sampling time will be negligible. If a source with higher impedance is used, the sampling time will depend on how long time the source needs to charge the S/H capacitor, with can vary widely. The user is recommended to only use low impedant sources with slowly varying signals, since this minimizes the required charge transfer to the S/H capacitor.

Signal components higher than the Nyquist frequency ($f_{ADC}/2$) should not be present for either kind of channels, to avoid distortion from unpredictable signal convolution. The user is advised to remove high frequency components with a low-pass filter before applying the signals as inputs to the ADC.

Figure 95. Analog Input Circuitry

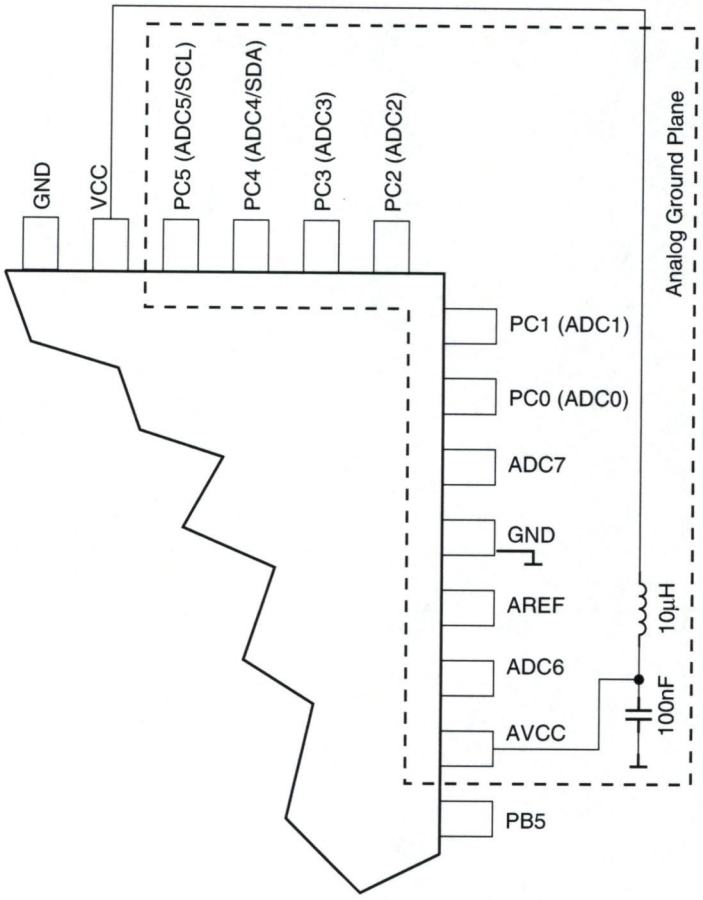


Analog Noise Canceling Techniques

Digital circuitry inside and outside the device generates EMI which might affect the accuracy of analog measurements. If conversion accuracy is critical, the noise level can be reduced by applying the following techniques:

1. Keep analog signal paths as short as possible. Make sure analog tracks run over the ground plane, and keep them well away from high-speed switching digital tracks.
2. The AV_{CC} pin on the device should be connected to the digital V_{CC} supply voltage via an LC network as shown in Figure 96.
3. Use the ADC noise canceler function to reduce induced noise from the CPU.
4. If any ADC [3..0] port pins are used as digital outputs, it is essential that these do not switch while a conversion is in progress. However, using the Two-wire Interface (ADC4 and ADC5) will only affect the conversion on ADC4 and ADC5 and not the other ADC channels.

Figure 96. ADC Power Connections



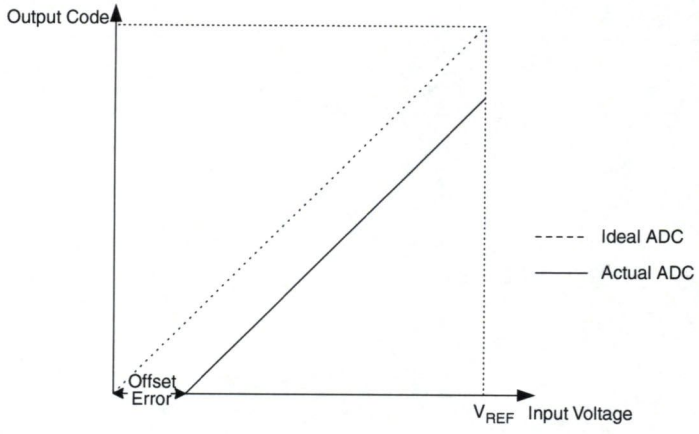
ADC Accuracy Definitions

An n-bit single-ended ADC converts a voltage linearly between GND and V_{REF} in 2^n steps (LSBs). The lowest code is read as 0, and the highest code is read as 2^n-1.

Several parameters describe the deviation from the ideal behavior:

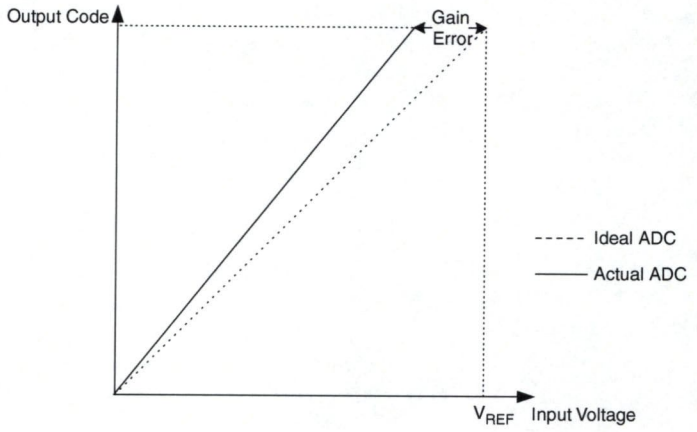
- Offset: The deviation of the first transition (0x000 to 0x001) compared to the ideal transition (at 0.5 LSB). Ideal value: 0 LSB.

Figure 97. Offset Error



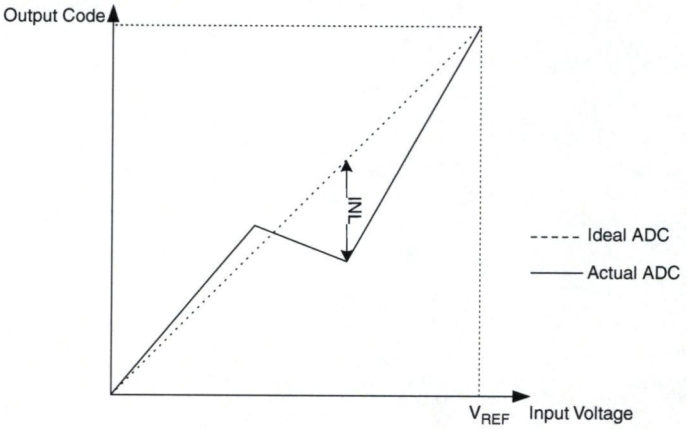
- Gain error: After adjusting for offset, the gain error is found as the deviation of the last transition (0x3FE to 0x3FF) compared to the ideal transition (at 1.5 LSB below maximum). Ideal value: 0 LSB

Figure 98. Gain Error



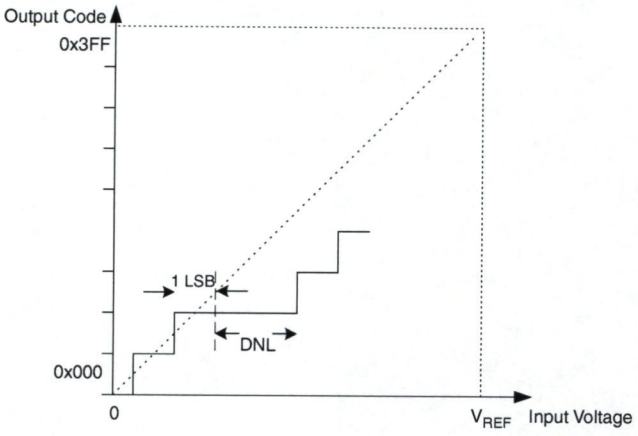
- Integral Non-linearity (INL): After adjusting for offset and gain error, the INL is the maximum deviation of an actual transition compared to an ideal transition for any code. Ideal value: 0 LSB.

Figure 99. Integral Non-linearity (INL)



- Differential Non-linearity (DNL): The maximum deviation of the actual code width (the interval between two adjacent transitions) from the ideal code width (1 LSB). Ideal value: 0 LSB.

Figure 100. Differential Non-linearity (DNL)



- Quantization Error: Due to the quantization of the input voltage into a finite number of codes, a range of input voltages (1 LSB wide) will code to the same value. Always ±0.5 LSB.
- Absolute accuracy: The maximum deviation of an actual (unadjusted) transition compared to an ideal transition for any code. This is the compound effect of offset, gain error, differential error, non-linearity, and quantization error. Ideal value: ±0.5 LSB.

ADC Conversion Result

After the conversion is complete (ADIF is high), the conversion result can be found in the ADC Result Registers (ADCL, ADCH).

For single ended conversion, the result is

$$ADC = \frac{V_{IN} \cdot 1024}{V_{REF}}$$

where V_{IN} is the voltage on the selected input pin and V_{REF} the selected voltage reference (see Table 74 on page 205 and Table 75 on page 206). 0x000 represents ground, and 0x3FF represents the selected reference voltage minus one LSB.

ADC Multiplexer Selection Register – ADMUX

Bit	7	6	5	4	3	2	1	0	
	REFS1	REFS0	ADLAR	–	MUX3	MUX2	MUX1	MUX0	ADMUX
Read/Write	R/W	R/W	R/W	R	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bit 7:6 – REFS1:0: Reference Selection Bits**

These bits select the voltage reference for the ADC, as shown in Table 74. If these bits are changed during a conversion, the change will not go in effect until this conversion is complete (ADIF in ADCSRA is set). The internal voltage reference options may not be used if an external reference voltage is being applied to the AREF pin.

Table 74. Voltage Reference Selections for ADC

REFS1	REFS0	Voltage Reference Selection
0	0	AREF, Internal V_{ref} turned off

Table 74. Voltage Reference Selections for ADC

REFS1	REFS0	Voltage Reference Selection
0	1	AV_{CC} with external capacitor at AREF pin
1	0	Reserved
1	1	Internal 2.56V Voltage Reference with external capacitor at AREF pin

- **Bit 5 – ADLAR: ADC Left Adjust Result**

The ADLAR bit affects the presentation of the ADC conversion result in the ADC Data Register. Write one to ADLAR to left adjust the result. Otherwise, the result is right adjusted. Changing the ADLAR bit will affect the ADC Data Register immediately, regardless of any ongoing conversions. For a complete description of this bit, see "The ADC Data Register – ADCL and ADCH" on page 208.

- **Bits 3:0 – MUX3:0: Analog Channel Selection Bits**

The value of these bits selects which analog inputs are connected to the ADC. See Table 75 for details. If these bits are changed during a conversion, the change will not go in effect until this conversion is complete (ADIF in ADCSRA is set).

Table 75. Input Channel Selections

MUX3..0	Single Ended Input
0000	ADC0
0001	ADC1
0010	ADC2
0011	ADC3
0100	ADC4
0101	ADC5
0110	ADC6
0111	ADC7
1000	
1001	
1010	
1011	
1100	
1101	
1110	1.30V (V_{BG})
1111	0V (GND)

ADC Control and Status Register A – ADCSRA

Bit	7	6	5	4	3	2	1	0	
	ADEN	ADSC	ADFR	ADIF	ADIE	ADPS2	ADPS1	ADPS0	ADCSRA
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bit 7 – ADEN: ADC Enable**

Writing this bit to one enables the ADC. By writing it to zero, the ADC is turned off. Turning the ADC off while a conversion is in progress, will terminate this conversion.

- **Bit 6 – ADSC: ADC Start Conversion**

In Single Conversion mode, write this bit to one to start each conversion. In Free Running mode, write this bit to one to start the first conversion. The first conversion after ADSC has been written after the ADC has been enabled, or if ADSC is written at the same time as the ADC is enabled, will take 25 ADC clock cycles instead of the normal 13. This first conversion performs initialization of the ADC.

ADSC will read as one as long as a conversion is in progress. When the conversion is complete, it returns to zero. Writing zero to this bit has no effect.

- **Bit 5 – ADFR: ADC Free Running Select**

When this bit is set (one) the ADC operates in Free Running mode. In this mode, the ADC samples and updates the Data Registers continuously. Clearing this bit (zero) will terminate Free Running mode.

- **Bit 4 – ADIF: ADC Interrupt Flag**

This bit is set when an ADC conversion completes and the Data Registers are updated. The ADC Conversion Complete Interrupt is executed if the ADIE bit and the I-bit in SREG are set. ADIF is cleared by hardware when executing the corresponding interrupt Handling Vector. Alternatively, ADIF is cleared by writing a logical one to the flag. Beware that if doing a Read-Modify-Write on ADCSRA, a pending interrupt can be disabled. This also applies if the SBI and CBI instructions are used.

- **Bit 3 – ADIE: ADC Interrupt Enable**

When this bit is written to one and the I-bit in SREG is set, the ADC Conversion Complete Interrupt is activated.

- **Bits 2:0 – ADPS2:0: ADC Prescaler Select Bits**

These bits determine the division factor between the XTAL frequency and the input clock to the ADC.

Table 76. ADC Prescaler Selections

ADPS2	ADPS1	ADPS0	Division Factor
0	0	0	2
0	0	1	2
0	1	0	4
0	1	1	8
1	0	0	16
1	0	1	32
1	1	0	64
1	1	1	128

The ADC Data Register – ADCL and ADCH

ADLAR = 0

Bit	15	14	13	12	11	10	9	8	
	–	–	–	–	–	–	ADC9	ADC8	ADCH
	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2	ADC1	ADC0	ADCL
	7	6	5	4	3	2	1	0	
Read/Write	R	R	R	R	R	R	R	R	
	R	R	R	R	R	R	R	R	
Initial Value	0	0	0	0	0	0	0	0	
	0	0	0	0	0	0	0	0	

ADLAR = 1

Bit	15	14	13	12	11	10	9	8	
	ADC9	ADC8	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2	ADCH
	ADC1	ADC0	–	–	–	–	–	–	ADCL
	7	6	5	4	3	2	1	0	
Read/Write	R	R	R	R	R	R	R	R	
	R	R	R	R	R	R	R	R	
Initial Value	0	0	0	0	0	0	0	0	
	0	0	0	0	0	0	0	0	

When an ADC conversion is complete, the result is found in these two registers.

When ADCL is read, the ADC Data Register is not updated until ADCH is read. Consequently, if the result is left adjusted and no more than 8-bit precision is required, it is sufficient to read ADCH. Otherwise, ADCL must be read first, then ADCH.

The ADLAR bit in ADMUX, and the MUXn bits in ADMUX affect the way the result is read from the registers. If ADLAR is set, the result is left adjusted. If ADLAR is cleared (default), the result is right adjusted.

- **ADC9:0: ADC Conversion result**

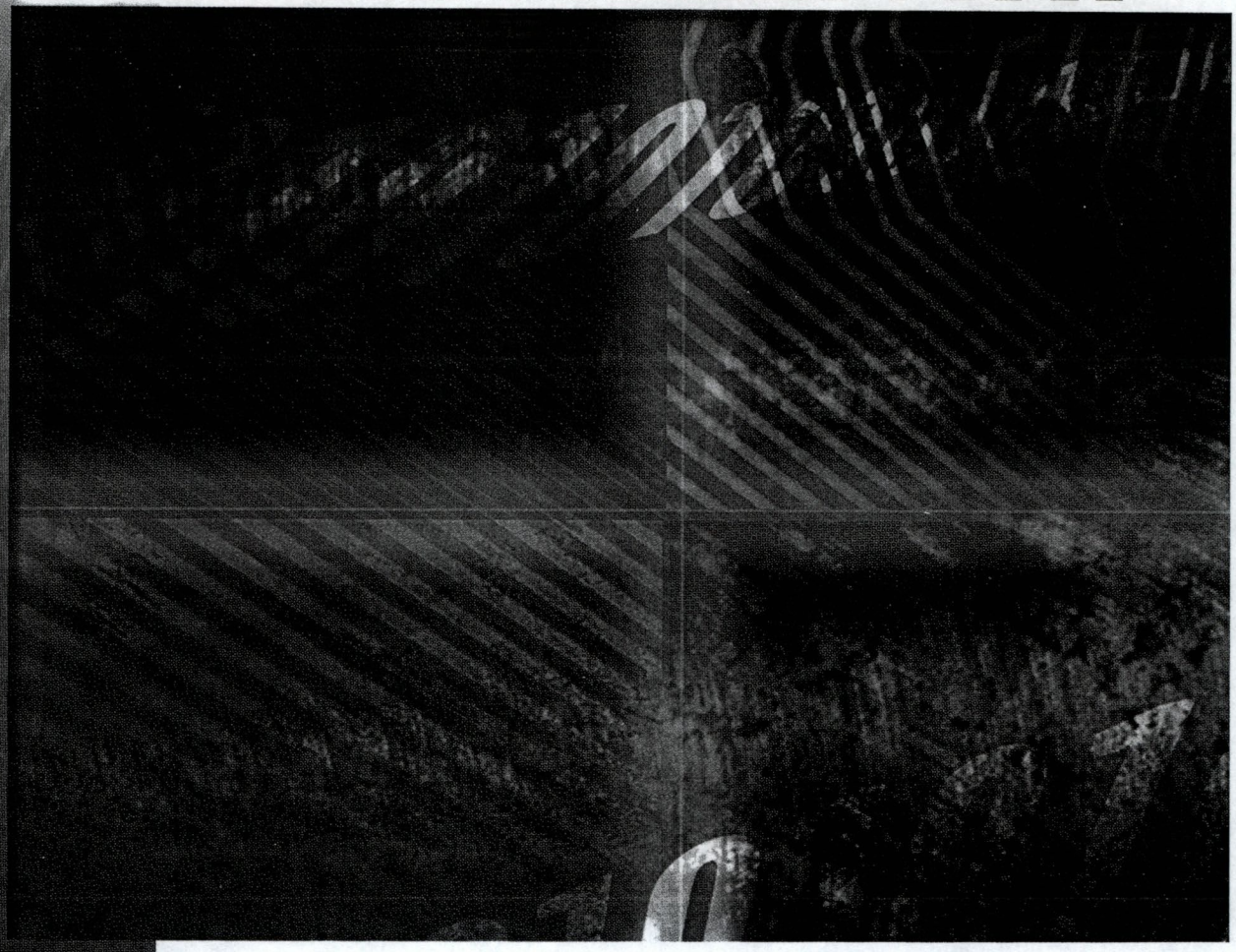
These bits represent the result from the conversion, as detailed in "ADC Conversion Result" on page 205.

THE C PROGRAMMING LANGUAGE

Brian W. Kernighan • Dennis M. Ritchie

THOMSON
DELMAR LEARNING

Embedded C Programming and the Atmel AVR



2nd Edition

Barnett,
Cox and
O'Cull

Embedded C Language Tutorial

1.1 OBJECTIVES

At the conclusion of this chapter, you should be able to

- Define, describe, and identify variable and constant types, their scope, and uses
- Construct variable and constant declarations for all sizes of numeric data and for strings
- Apply enumerations to variable declarations
- Assign values to variables and constants by means of the assignment operator
- Evaluate the results of all of the operators used in C
- Explain the results that each of the control statements has on program flow
- Create functions that are composed of variables, operators, and control statements to complete tasks
- Apply pointers, arrays, structures, and unions as function variables
- Create C programs that complete tasks using the concepts in this chapter

1.2 INTRODUCTION

This chapter provides a baseline course in the C programming language as it applies to embedded microcontroller applications. The chapter includes extensions to the C language that are a part of the CodeVisionAVR® C language. You will go from beginning concepts through writing complete programs, with examples that can be implemented on a microcontroller to further reinforce the material.

The information is presented somewhat in the order that it is needed by a programmer:

- Declaring variables and constants
- Simple I/O, so that programs can make use of the parallel ports of the microcontroller

- Assigning values to the variables and constants, and doing arithmetic operations with the variables
- C constructs and control statements to form complete C programs

The final sections cover the more advanced topics, such as pointers, arrays, structures, and unions, and their use in C programs. Advanced concepts such as real-time programming and interrupts complete the chapter.

1.3 BEGINNING CONCEPTS

Writing a C program is, in a sense, like building a brick house: A foundation is laid, sand and cement are used to make bricks, these bricks are arranged in rows to make a course of blocks, and the courses are then stacked to create a building. In an embedded C program, sets of instructions are put together to form functions; these functions are then treated as higher-level operations, which are then combined to form a program.

Every C language program must have at least one function, namely *main()*. The function *main()* is the foundation of a C language program, and it is the starting point when the program code is executed. All functions are invoked by *main()* either directly or indirectly. Although functions can be complete and self-contained, variables and parameters can be used to cement these functions together.

The function *main()* is considered to be the lowest-level task, since it is the first function called from the system starting the program. In many cases, *main()* will contain only a few statements that do nothing more than initialize and steer the operation of the program from one function to another.

An embedded C program in its simplest form appears as follows:

```
void main()
{
      while(1)     // do forever..
          ;
}
```

The program shown above will compile and operate perfectly, but you will not know that for sure, because there is no indication of activity of any sort. We can embellish the program such that you can actually see life, review its functionality, and begin to study the syntactical elements of the language:

```
#include <stdio.h>

void main()
{
      printf("HELLO WORLD"); /* the classic C test program.. */
      while(1)           // do forever..
          ;
}
```

This program will print the words "HELLO WORLD" to the standard output, which is most likely a serial port. The microcontroller will sit and wait, forever or until the microcontroller is reset. This demonstrates one of the major differences between a personal computer program and a program that is designed for an embedded microcontroller: namely, that the embedded microcontroller applications contain an infinite loop. Personal computers have an operating system, and once a program has executed, it returns control to the operating system of the computer. An embedded microcontroller, however, does not have an operating system and cannot be allowed to fall out of the program at any time. Hence, every embedded microcontroller application has an infinite loop built into it somewhere, such as the *while(1)* in the example above. This prevents the program from running out of things to do and doing random things that may be undesirable. The **while** construct will be explained in a later section.

The example program also provides an instance of the first of the common preprocessor compiler directives. **#include** tells the compiler to include a file called stdio.h as a part of this program. The function *printf()* is provided for in an external library, and it is made available to us because its definition is located in the stdio.h file. As we continue, these concepts will come together quickly.

These are some of the elements to take note of in the previous examples:

;	A semicolon is used to indicate the end of an expression. An expression in its simplest form is a semicolon alone.
{ }	Braces "{}" are used to delineate the beginning and the end of the function's contents. Braces are also used to indicate when a series of statements is to be treated as a single block.
"text"	Double quotes are used to mark the beginning and the end of a text string.
// or /* . . . */	Slash-slash or slash-star/star-slash are used as comment delimiters.

Comments are just that, a programmer's notes. Comments are critical to the readability of a program. This is true whether the program is to be read by others or by the original programmer at a later time. The comments shown in this text are used to explain the function of each line of the code in the example. The comments should always explain the actual *function* of the line in the program, and not just echo the specific instructions that are used on the line.

The traditional comment delimiters are the slash-star (/*), star-slash (*/) configuration. Slash-star is used to create block comments. Once a slash-star (/*) is encountered, the compiler will ignore the subsequent text, even if it encompasses multiple lines, until a star-slash (*/) is encountered. Refer to the first line of the *main()* function in the previous program example for an example of these delimiters.

The slash-slash (//) delimiter, on the other hand, will cause the compiler to ignore the comment text only until the end of the line is reached. These are used in the second line of the *main()* function of the example program.

As we move into the details, a few syntactical rules and some basic terminology should be remembered:

- An identifier is a variable or function name made up of a letter or underscore (_), followed by a sequence of letters and/or digits and/or underscores.
- Identifiers are case sensitive.
- Identifiers can be any length, but some compilers may recognize only a limited number of characters, such as the first thirty-two. So beware!
- Particular words have special meaning to the compiler and are considered reserved words. These reserved words must be entered in lowercase and should never be used as identifiers. Table 1–1 lists reserved words.

auto	defined	float	long	static	while
break	do	for	register	struct	
bit	double	funcused	return	switch	
case	eeprom	goto	short	typedef	
char	else	if	signed	union	
const	enum	inline	sizeof	unsigned	
continue	extern	int	sfrb	void	
default	flash	interrupt	sfrw	volatile	

Table 1–1 *Reserved Word List*

- Since C is a free-form language, "white space" is ignored unless delineated by quotes. This includes blank (space), tab, and new line (carriage return and/or line feed).

1.4 VARIABLES AND CONSTANTS

It is time to look at data stored in the form of variables and constants. Variables, as in algebra, are values that can be changed. Constants are fixed. Variables and constants come in many forms and sizes; they are stored in the program's memory in a variety of forms that will be examined as we go along.

1.4.1 VARIABLE TYPES

A variable is declared by the reserved word indicating its type and size followed by an identifier:

```
unsigned char Peabody;
int dogs, cats;
long int total_dogs_and_cats;
```

Variables and constants are stored in the limited memory of the microcontroller, and the compiler needs to know how much memory to set aside for each variable without wasting

memory space unnecessarily. Consequently, a programmer must declare the variables, specifying both the size of the variable and the type of the variable. Table 1–2 lists variable types and their associated sizes.

Type	Size (Bits)	Range
bit	1	0, 1
char	8	−128 to 127
unsigned char	8	0 to 255
signed char	8	−128 to 127
int	16	−32768 to 32767
short int	16	−32768 to 32767
unsigned int	16	0 to 65535
signed int	16	−32768 to 32767
long int	32	−2147483648 to 2147483647
unsigned long int	32	0 to 4294967295
signed long int	32	−2147483648 to 2147483647
float	32	±1.175e-38 to ±3.402e38
double	32	±1.175e-38 to ±3.402e38

Table 1–2 *Variable Types and Sizes*

1.4.2 VARIABLE SCOPE

As noted above, constants and variables must be declared prior to their use. The *scope* of a variable is its accessibility within the program. A variable can be declared to have either *local* or *global* scope.

Local Variables

Local variables are memory spaces allocated by the function when the function is entered, typically on the program stack or a compiler-created heap space. These variables are not accessible from other functions, which means their scope is limited to the functions in which they are declared. The local variable declaration can be used in multiple functions without conflict, since the compiler sees each of these variables as being part of that function only.

Global Variables

A global or external variable is a memory space that is allocated by the compiler and can be accessed by all the functions in a program (unlimited scope). A global variable can be modified by any function and will retain its value to be used by other functions.

Global variables are typically cleared (set to zero) when *main()* is started. This operation is most often performed by startup code generated by the compiler, invisible to the programmer.

An example piece of code is shown below to demonstrate the scope of variables:

```
unsigned char globey;    //a global variable

void function_z (void)  //this is a function called from main()
{
    unsigned int tween;          //a local variable

    tween = 12;                  //OK because tween is local
    globey = 47;                 //OK because globey is global
    main_loc = 12;       // This line will generate an error
                         // because main_loc is local to main.
}

void main()
{
    unsigned char main_loc;  //a variable local to main()
    globey = 34;                 //Ok because globey is a global
    tween = 12;          //will cause an error - tween is local
                           // to function_z

    while(1)             // do forever..
        ;
}
```

When variables are used within a function, if a local variable has the same name as a global variable, the local will be used by the function. The value of the global variable, in this case, will be inaccessible to the function and will remain untouched.

1.4.3 CONSTANTS

As described earlier in the text, constants are fixed values—they may not be modified as the program executes. Constants in many cases are part of the compiled program itself, located in read-only memory (ROM), rather than an allocated area of changeable random access memory (RAM). In the assignment

```
x = 3 + y;
```

the number *3* is a constant and will be coded directly into the addition operation by the compiler. Constants can also be in the form of characters or a string of text:

```
printf("hello world");
            // The text "hello world" is placed in program memory
            // and never changes.
x = 'B';    // The letter 'B' is permanently set
            // in program memory.
```

You can also declare a constant by using the reserved word **const** and indicating its type and size. An identifier and a value are required to complete the declaration:

```
const char c = 57;
```

Identifying a variable as a constant will cause that variable to be stored in the program code space rather than in the limited variable storage space in RAM. This helps preserve the limited RAM space.

Numeric Constants

Numeric constants can be declared in many ways by indicating their numeric base and making the program more readable. Integer or long integer constants may be written in

- Decimal form without a prefix (such as 1234)
- Binary form with **0b** prefix (such as 0b101001)
- Hexadecimal form with **0x** prefix (such as 0xff)
- Octal form with **0** prefix (such as 0777)

There are also modifiers to better define the intended size and use of the constant:

- Unsigned integer constants can have the suffix **U** (such as 10000U).
- Long integer constants can have the suffix **L** (such as 99L).
- Unsigned long integer constants can have the suffix **UL** (such as 99UL).
- Floating point constants can have the suffix **F** (such as 1.234F).
- Character constants must be enclosed in single quotation marks, **'a'** or **'A'**.

Character Constants

Character constants can be printable (like 0–9 and A–Z) or non-printable characters (such as new line, carriage return, or tab). Printable character constants may be enclosed in single quotation marks (such as 'a'). A backslash followed by the octal or hexadecimal value in single quotes can also represent character constants:

```
't'   can be represented by   '\164'      (octal)
                   or
't'   can be represented by   '\x74'      (hexadecimal)
```

Table 1–3 lists some of the "canned" non-printable characters that are recognized by the C language.

Backslash (\) and single quote (') characters themselves must be preceded by a backslash to avoid confusing the compiler. For instance, '\' ' is a single quote character and '\\' is a backslash. BEL is the bell character and will make a sound when it is received by a computer terminal or terminal emulator.

1.4.4 ENUMERATIONS AND DEFINITIONS

Readability in C programs is very important. Enumerations and definitions are provided so that the programmer can replace numbers with names or other more meaningful phrases.

1.5 I/O OPERATIONS

Embedded microcontrollers must interact directly with other hardware. Therefore, many of their input and output operations are accomplished using the built-in parallel ports of the microcontroller.

Most C compilers provide a convenient method of interacting with the parallel ports through a library or header file that uses the **sfrb** and **sfrw** compiler commands to assign labels to each of the parallel ports and other I/O devices. These commands will be discussed in a later section, but the example below will serve to demonstrate the use of the parallel ports:

```
#include <MEGA8535.h>        // register definition header file for
                             // an Atmel ATMega8535

unsigned char z;             // declare z

void main(void)
{
    DDRB = 0xff;          // set all bits of port B for output

    while(1)
    {
       z = PINA;          // read the binary value on the
                          // port A pins (i.e., input from port A)
       PORTB = z + 1;     // write the binary value read from port A
                          // plus 1 to port B
    }
}
```

The example above shows the methods to both read and write a parallel port. z is declared as an unsigned character size variable because the port is an 8-bit port and an unsigned character variable will hold 8-bit data. Notice that the labels for the pins of port A and output port B are all capitalized because these labels must match the way the labels are defined in the included header file MEGA8535.h.

The DDRx register is used to determine which bits of port x (A, B, and so on depending on the processor) are to be used for output. Upon reset, all of the I/O ports default to input by the microcontroller writing a 0 into all the bits of the DDRx registers. The programmer then sets the DDRx bits depending on which bits are to be used for output. For example,

```
DDRB = 0xc3;    // set the upper 2 and lower 2 bits
                // of port B for output
```

This example configures the upper two bits and the lower two bits to be used as output bits.

the WDE bit must be cleared as the next instruction. An example is shown below:

```
WDTCR = 0x18;  /*set WDE and WDTOE simultaneously*/
WDTCR = 0c00;  /*set WDE to 0 immediately*/
```

Normally the watchdog timer, once enabled, is never disabled because the purpose of enabling it to start with is to protect the processor against errant or erratic processing.

2.6 PARALLEL I/O PORTS

The parallel I/O ports are the most general-purpose I/O devices. Each of the parallel I/O ports has three associated I/O registers: the *data direction register*, DDRx (where "x" is A, B, C, and so on depending on the specific processor and parallel port being used), the *port driver register*, typically called PORTx, and the *port pins register*, PINx.

The data direction register's purpose is to determine which bits of the port are used for input and which bits are used for output. The input and output bits can be mixed as desired by the programmer. A processor reset clears all of the data direction register bits to logic 0, setting all of the port's bits for input. Setting any bit of the data direction register to a logic 1 sets the corresponding port bit for output mode. For instance, setting the least significant two bits of DDRA to a logic 1 and the other bits to a logic 0 sets the least significant two bits of port A for output and the balance of the bits for input.

Writing to the output bits of port A is accomplished as follows:

```
PORTA = 0x2;    /*sets the second bit of port A*/
                /*and clear the other seven bits*/
```

Reading from the input bits of port A is accomplished as follows:

```
x = PINA;  /*reads all 8 pins of port A*/
```

In this latter example, "x" would contain the value from all of the bits of port A, both input and output, because the PIN register reflects the value of all of the bits of the port.

Input port pins are floating, that is, there is not necessarily a pull-up resistor associated with the port pin. The processor can supply the pull-up resistor, if desired, by writing a logic 1 to the corresponding bit of the port driver register as shown below:

```
DDRA = 0xC0;  /*upper 2 bits as output, lower 6 as input*/
PORTA = 0x3;  /*enable internal pull-ups on lowest 2 bits*/
```

In general, although this varies by the specific processor, the port pins are capable of sinking 20 mA, but they can source much less current. This means that the ports can directly drive LEDs, provided that the port pin is sinking the current as shown in Figure 2–10.

Figure 2–15 shows a program that uses the least significant four bits of port C as input and the most significant four bits as output. Pull-up resistors are enabled on the input pins of the least significant two bits of the port.

Additional information and examples relative to the I/O ports are shown in Section 1.5, "I/O Operations," in Chapter 1, "Embedded C Language Tutorial."

```
#include <mega16.h>

void main (void)
{
   DDRC = 0xf0;  /*set upper 4 bits of PORTC as output*/
   PORTC = 0x03; /*enable pull-ups on the 2 least significant*/
                 /*bits of PORTC*/

   while(1)
   {
      PORTC = (PINC << 4);  /*read the lower 4 bits of POTRC*/
                            /*shift them 4 bits left and output*/
                            /*them to the upper 4 bits of PORTC*/
   }
}
```

Figure 2–15 *Parellel Port Example*

EXAMPLE

Chapter 2 Example Project: Part B

This is the second part of the Chapter 2 example program to create a system to collect operational data for a stock car racer. In the first part, it was determined that the system has sufficient memory resources to do the task. In this part, the overall code structure and user interface will be created.

It is important to determine which peripheral resources are to be used for each measurement, input, and output. If you were creating the data collection system from scratch, you could determine the resources to use, and, as you assigned the resources, you would be sure that the processor had sufficient resources. In this case, however, you need to determine which peripherals are being used for each measurement so you can write your program accordingly.

Investigation, and perhaps a little circuit tracing, has shown that the system connections are as follows:

- The Start button is connected to the INT1 pin on the Mega163. Pressing it pulls INT1 low.
- The Upload button is connected to Port A, bit 0. Pressing it pulls the bit low.
- The engine rpm pulses are connected to the ICP (input capture pin).
- The drive shaft rpm pulses are connected to the INT0 pin.
- The engine temperature signal is connected to the ADC3 (analog-to-digital converter input #3) pin.

A separate index (in this case, *sndcntr*) is used to retrieve the data from the queue. As each byte is retrieved, this index is incremented. Finally, when the two indices are equal, the program knows that the queue has been emptied.

Actually, the CodeVisionAVR C language compiler can provide a transmitter queue, a receiver queue, or both using the "CodeWizard" code generator feature. This example is provided to demonstrate how the queue works for educational reasons.

Getting back to the example program, the *sendmsg()* function called from the switch statement puts the message into the queue and starts the transmit function. The switch statement passes a pointer (an address) to the appropriate message when it calls the function. The function first puts the CR and LF characters into the queue and then puts the message to be transmitted into the queue using the pointer. Finally, the function writes the first byte in the queue into the UDR to start the transmission process. After this character has been transmitted, the TXC interrupt occurs and the ISR loads the next character from the queue into the UDR, and so the cycle continues until the queue is empty, as indicated by the two indices being equal. A more elaborate form of the queue function with additional explanation may be found in Chapter 3, "Standard I/D and Preprocessor Functions."

2.9 ANALOG INTERFACES

In spite of the prevalence of digital devices, the world is still actually analog by nature. A microcontroller is able to handle analog data by first converting the data to digital form. An AVR microcontroller includes both an analog-to-digital conversion peripheral and an analog comparator peripheral. Each of these analog interfaces will be covered in this section, along with a brief background on analog-to-digital conversion.

Microcontrollers use analog-to-digital converters to convert analog quantities such as temperature and voltage (for example, a low-battery monitor), to convert audio to digital formats, and to perform a host of additional functions.

2.9.1 ANALOG-TO-DIGITAL BACKGROUND

Analog-to-digital conversion (as well as digital-to-analog conversion) is largely a matter of *proportion*. That is, the digital number provided by the analog-to-digital converter (ADC) relates to the proportion that the input voltage is of the full voltage range of the converter. For instance, applying 2 V to the input of an ADC with a full-scale range of 5 V will result in a digital output that is 40 percent of the full range of the digital output (2 V / 5 V = 0.4).

ADCs are available that have a variety of input voltage ranges and output digital ranges. The output digital ranges are usually expressed in terms of bits, such as 8 bits or 10 bits. The number of bits at the output determines the range of numbers that can be read from the output of the converter. An 8-bit converter will provide outputs from 0 up to $2^8 - 1$ or 255, and a 10-bit converter will provide outputs from 0 up to $2^{10} - 1$ or 1023.

In the previous example, in which 2 V was applied to a converter with a full-scale range of 5 V, an 8-bit converter would read 40 percent of 255, or 102. The proportion/conversion

factor is summarized in the following formula:

$$\frac{V_{in}}{V_{fullscale}} = \frac{x}{2^n - 1}$$

In the formula above, "x" is the digital output and "n" is the number of bits in the digital output. Using this formula and solving for x, you can calculate the digital number read by the computer for any given input voltage. Using the formula within a program where you have x, you can use the formula to solve for the voltage being applied. This is useful when you might be trying to display the actual voltage on an LCD readout.

An important issue buried in this formula is the issue of resolution. The resolution of measurement, or the finest increment that can be measured, is calculated as follows:

$$V_{resolution} = \frac{V_{fullscale}}{2^n - 1}$$

For an 8-bit converter that has a full-scale input range of 5 V, the resolution would be calculated as follows:

$$V_{resolution} = 5V/(2^8 - 1) = 5V/255 = 20\ mV \ \text{(approx.)}$$

Therefore, the finest voltage increment that can be measured in this situation is 20 mV. It would be inappropriate to attempt to make measurements that are, for example, accurate to within 5 mV with this converter.

2.9.2 ANALOG-TO-DIGITAL CONVERTER PERIPHERAL

The ADC peripheral in the AVR microcontrollers is capable of 10-bit resolution and can operate at speeds as high as 15 kSPS (kilo-samples per second). It can read the voltage on one of eight different input pins of the microcontroller, meaning that it is capable of reading from eight different analog sources.

Two registers control the analog-to-digital converter: The *ADC control and status register* (ADCSRA), controls the functioning of the ADC, and the *ADC multiplexer select register* (ADMUX), controls which of the eight possible inputs are being measured. Figure 2–41 shows the bit definitions for the ADC control and status register.

The ADC requires a clock frequency in the range of 50 kHz to 200 kHz to operate at maximum resolution. Higher clock frequencies are allowed but at decreased resolution. The ADC clock is derived from the system clock by means of a prescaler in a manner similar to the timers. The least significant three bits of ADCSRA control the prescaler division ratio. These bits must be set so that the system clock, when divided by the selected division ratio, provides an ADC clock between 50 kHz and 200 kHz. The selection bits and division ratios are shown in Figure 2–42.

Although it could be done by trial and error, the most direct method for choosing the ADC

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0

Bit	Description
ADEN	ADC Enable bit. Set to enable the ADC.
ADSC	ADC Start Conversion bit. Set to start a conversion.
ADATE	ADC Free Running Select bit. Set to enable free run mode.
ADIF	ADC Interrupt Flag bit. Is set by hardware at the end of a conversion cycle.
ADIE	ADC Interrupt Mask bit. Set to allow the interrupt to occur at the end of a conversion.
ADPS2	ADC Prescaler select bits.
ADPS1	
ADPS0	

Figure 2–41 *ADCSRA Bit Definitions*

preselector factor is to divide the system clock by 200 kHz and then choose the next higher division factor. This will ensure an ADC clock that is as fast as possible but under 200 kHz.

The ADC, like the serial USART, is somewhat slower than the processor. If the processor were to wait for each analog conversion to be complete, it would be wasting valuable time. As a result, the ADC is usually used in an interrupt-driven mode.

Although the discussion that follows uses the more common interrupt-driven mode, it is also possible for the ADC to operate in free-running mode, in which it continuously does conversions as fast as possible. When reading the ADC output in free-running mode, it is

ADPS2	ADPS1	ADPS0	Division Factor
0	0	0	2
0	0	1	2
0	1	0	4
0	1	1	8
1	0	0	16
1	0	1	32
1	1	0	64
1	1	1	128

Figure 2–42 *ADC Preselector Division Ratios*

necessary to disable the interrupts or stop the free-running conversions, read the result, and then re-enable the interrupts and free-running mode. These steps are necessary to ensure that the data read is accurate, in that the program will not be reading the data during the time that the processor is updating the ADC result registers.

The ADC is usually initialized as follows:

1. Set the three lowest bits of ADCSR for the correct division factor.
2. Set ADIE high to enable interrupt mode.
3. Set ADEN high to enable ADC.
4. Set ADSC to immediately start a conversion.

For a division factor of 8, the following lines of code would initialize the ADC to read the analog voltage on the ADC2 pin:

```
ADMUX = 2;          /*read analog voltage on ADC2*/
ADCSR = 0xcb;       /*ADC on, interrupt mode, /8, & started*/
```

The initialization above sets up the ADC, enables it, and starts the first conversion all at once. This is useful because the first conversion cycle after the ADC is enabled is an extra-long cycle to allow for the setup time of the ADC. The long cycle, then, occurs during the balance of the program initialization, and the ADC interrupt will occur immediately after the global interrupt enabled bit is set. Notice that the ADMUX register is loaded with the number of the ADC channel to be read.

Figures 2–43 and 2–44 show the hardware and software, respectively, for a limit detector system based on the analog input voltage to ADC channel 3. Briefly, the system lights the red LED if the input voltage exceeds 3 V, lights the yellow LED if the input voltage is below 2 V, or lights the green LED if the input voltage is within the range of 2 V to 3 V.

The limit detector program in Figure 2–44 shows a typical application for the ADC. The ADC is initialized and started in *main()* by setting ADCSRA to 0xCE. ADC channel 3 is selected by setting ADMUX to 3. This starts the process so the ADC interrupt will occur at the end of the first conversion. Checking the ADC output to see which LED to light and lighting the appropriate LED are all handled in the ADC interrupt ISR.

Notice that the 10-bit output from the ADC is read by reading the data from ADCW. ADCW is a pseudo-register provided by CodeVisionAVR that allows retrieving the data from the two ADC result registers, ADCL and ADCH, at once. Other compilers would more likely require the programmer to read both ADC registers (in the correct order, even) and combine them in the 10-bit result as a part of the program.

Also notice that the programmer has used the analog-to-digital conversion formula in the ISR. The compiler will do the math and create a constant that will actually be used in the program. You will find that this technique can be used to your advantage in several other places, such as loading the UBRR for the USART.

The ADC peripheral in the AVR microcontrollers varies somewhat according to the spe-

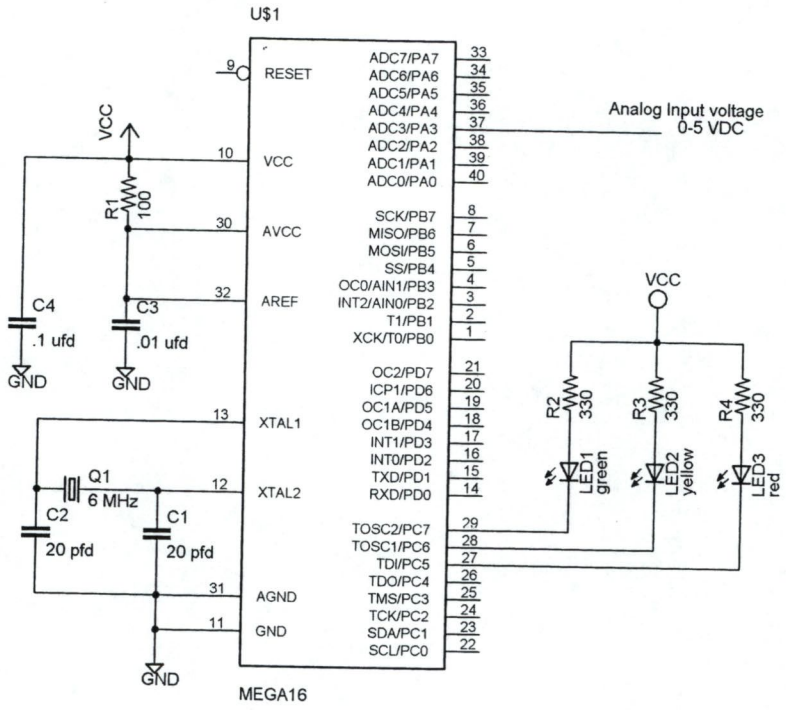


Figure 2–43 *ADC Example Hardware*

```
#include <mega16.h>

/*Define output port and light types*/
#define LEDs PORTC
#define red 0b11011111
#define green 0b01111111
#define yellow 0b10111111

/*ADC ISR, reads ADC, sets lights*/
interrupt [ADC_INT] void adc_isr(void)
{
   unsigned int adc_data;    /*variable for ADC results*/
   adc_data = ADCW;  /*read all 10 bits into variable*/
   if (adc_data >(3*1023)/5)
   {
      LEDs = red;          /*too high (>3V)*/
   }
   else if (adc_data < (2*1023)/5)
   {
```

Figure 2–44 *ADC Example Software (Continues)*

```
        LEDs = yellow;      /*too low (<2V)*/
    }
    else
    {
        LEDs = green;  /*must be just right - like Goldilock's porridge*/
    }
    ADCSRA = ADCSRA | 0x40;  /*start the next conversion    */
}

void main(void)
{
    DDRC = 0xe0;      /*most significant 3 bits for output*/
    ADMUX = 0x3;    /*select to read only channel 3*/
    ADCSRA=0xCE;     /*ADC on, /64, interrupt unmasked, and started*/

    #asm("sei")     /*global interrupt enable bit*/

    while (1)
    {
        ;      /*do nothing but wait on ADC interrupt*/
    }
}
```

Figure 2–44 *ADC Example Software (Continued)*

cific microcontroller in use. All of the ADCs require some noise suppression on the ADC Vcc connection (see Figure 2–43 or 2–46). Some also have a built-in noise canceller function, and some have the ability to control Vref internally. You will need to check the specification for your particular microcontroller when using the ADC.

2.9.3 ANALOG COMPARATOR PERIPHERAL

The analog comparator peripheral is a device that compares two analog inputs: AIN0, the positive analog comparator input, and AIN1, the negative analog comparator input. If AIN0 > AIN1, then the *analog comparator output* bit (ACO) is set. When the ACO bit changes state, either positive-going, negative-going, or both, it will cause an interrupt to occur, provided that the analog comparator interrupt is unmasked, or else the change of state may be set to cause an input capture to occur on timer/counter 1.

The *analog comparator control and status register* (ACSR) shown in Figure 2–45, is used to control the analog comparator functions.

As a simple example of analog comparator functioning, consider the system shown in Figure 2–46. A battery is used to power the system, so it is important to know when the battery voltage becomes dangerously low.