Tutorial Introduction

PURPOSE

- To explain how to configure and use the Analog-to-Digital Converter Module in common applications

OBJECTIVES:

- Identify the steps to set up and configure the Analog-to-Digital Converter Module.
- Identify techniques for maximizing the accuracy of analog-to-digital conversions.
- Write a program to configure the Analog-to-Digital Module and to take measurements.

CONTENT:

- 19 pages
- 2 questions
- 1 off-line programming exercise

LEARNING TIME:

- 30 minutes

PREREQUESITE:

- 68HC08 CPU training module

Welcome to this tutorial on the 68HC08 Analog-to-Digital Converter (ADC). This tutorial describes the features of 68HC08 ADC Modules and provides detailed training for the 68HC908GP32 ADC Module. Please note that on subsequent pages, you will find reference buttons in the upper right of the content frame that access additional content.

Upon completion of this tutorial, you'll be able to set up and configure the ADC Module, identify techniques for maximizing the accuracy of analog-to-digital conversions, and write a program to take ADC measurements.

The recommended prerequisite for this tutorial is the 68HC08 CPU training module. Click the Forward arrow when you're ready to begin the tutorial.

ADC Module Features

- 8-bit or 10-bit resolution
- Linear successive approximation with monotonicity
- Single conversion and continuous conversion modes
- Conversion complete indication by flag or interrupt
- Selectable ADC input clock

Let‘s begin this tutorial with a review of the ADC Module features.

The 68HC08 microcontroller unit (MCU) includes either an 8-bit or 10-bit resolution ADC. Some examples of 68HC08 MCUs that have an 8-bit ADC are the 68HC908GP, GR, JL, JK, and KX product families. Examples of 68HC08 MCUs with a 10-bit ADC are the 68HC908MR and SR product families.

All 68HC08 ADC Modules use the successive approximation principle. A discussion of this principle can be found in the 68HC11 Reference Manual M68HC11RM/AD (see http://www.Freescale.com)

All ADC Modules support both single conversion mode and continuous conversion mode. In single conversion mode, one conversion is completed between writes to the ADC status and control register. In continuous conversion mode, the ADC analog input is continually converted and written to the ADC data register. In this mode, data from the previous conversion is overwritten regardless of whether that data had been read or not.

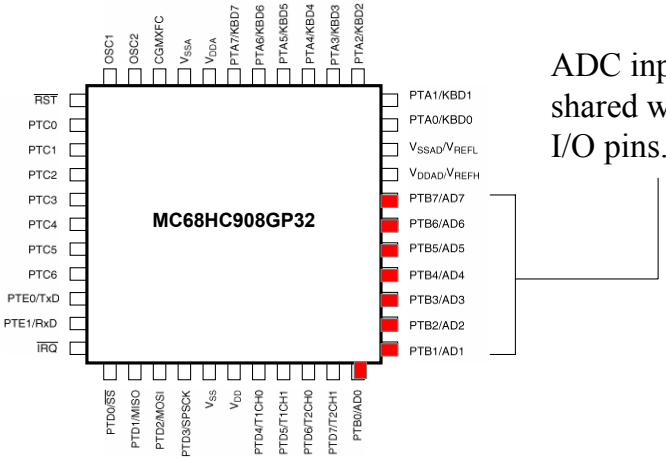
The ADC offers two different ways to monitor the conversion complete status. Depending on the conversion mode, you can use software to poll a flag value or you can configure the ADC to generate an interrupt signal when the conversion is complete.

The ADC has a selectable input clock. You can use the input clock to optimize ADC conversions for different crystal frequencies and to accomodate 68HC08 MCUs with phase-locked loop (PLL).

E02c (no content change)

The remainder of this tutorial covers the configuration and operation of the 68HC908GP32 MCU ADC Module.

ADC Module Pins



ADC input pins are shared with Port B I/O pins.

The figure shows the pin assignment of the 68HC908GP32 MCU 44-pin quad flat pack (QFP). This MCU includes an 8-bit ADC Module with eight input channels. Other 68HC08 derivatives contain a different number of input channels. Check the product documentation for specific details about a particular model.

In the 68HC908GP32, the eight input channels are multiplexed to the ADC Module. In all 68HC08 MCUs, the ADC input pins are shared with general purpose I/O pins. The eight input pins of the 68HC908GP32 ADC, pins AD7 - AD0, are shared with the general purpose I/O pins of port B (pins PTB7 - PTB0).

You can select any ADC pin as the analog input pin using the channel select bits of the ADC status and control register (ADCH4-ADCH0). The remaining ADC pins will be controlled by the port I/O logic and used for general purpose I/O.

The analog portion of the 68HC908GP32 ADC Module uses the V_{DDAD} pin for power and the V_{SSAD} pin as the ground pin.

Next, we'll look at how to configure the ADC Module for voltage conversion.

Configuring the ADC

- Configure V_{DDAD}
  - Connect V_{DDAD} to the same voltage as V_{DD}.
  - Use external filtering to ensure a clean power voltage
  - Route V_{DDAD} carefully and place bypass capacitors close to the package.
- Connect V_{SSAD} to the same voltage as V_{SS}.
- Separate V_{REFH} and V_{REFL}.

E04 (display bullet list as shown)To configure the 68HC908GP32 ADC Module for voltage conversion, first configure the power pin, V_{DDAD}. You will usually connect this pin to the same voltage potential as the V_{DD} pin. To achieve good ADC results, you may need to use external filtering to ensure a clean V_{DDAD}. For maximum noise immunity, route V_{DDAD} carefully and place bypass capacitors as close as possible to the package.

The ADC uses the V_{SSAD} pin as the ground pin. You should connect this pin to the same voltage potential as V_{SS}.

Note that in this configuration, the high reference voltage pin, V_{REFH}, is separated from the low reference voltage pin, V_{REFL} pin. The V_{REFH} pin is shared with the power pin (V_{DDAD}) and V_{REFL} is shared with the ground pin (V_{SSAD}).

ADC Voltage Conversion

- V_{ADIN} should not exceed the analog supply voltages.
- If $V_{REFH} \leq V_{ADIN} \leq V_{REFL}$, the ADC performas a linear conversion.
- If $V_{ADIN} = V_{REFH}$, the result is $FF.
- If $V_{ADIN} = V_{REFL}$, the result is $00.
- The conversion is monotonic and has no missing codes.

Once you've configured the ADC Module as described, the Module is ready to convert the input voltage, V_{ADIN}, to a digital value. The input voltage signal is read from the ADC input channel selected in the ADC status and control register. The conversion result depends on the value of V_{ADIN}.

Recall that the high reference voltage, V_{REFH}, is connected to the ADC analog power pin, V_{DDAD}, and the low reference voltage, V_{REFL}, is conneced to the ADC analog ground pin, V_{SSAD}. Therefore, V_{ADIN} should not exceed the analog supply voltages.

If the value of V_{ADIN} is between V_{REFH} & V_{REFL}, the ADC converts the voltage using a linear conversion. The result is one of 256 digital values ranging from $00 to $FF. If V_{ADIN} equals V_{REFH}, the ADC converts the signal to $FF. If V_{ADIN} equals V_{REFL}, the ADC converts it to $00.

The conversion process is monotonic and has no missing codes. This means that if the input voltage is increasing, the ADC converts the signal to all values between $00 and $FF. In this case, the next conversion result will always be higher then the previous one.

ADC Module Registers

- ADC Status and Control Register (ADSCR)
- ADC Data Register (ADR)
- ADC Clock Register (ADCLK)

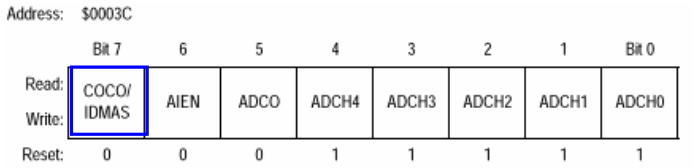
You can control and monitor ADC operations using the three ADC Module registers.

Using the ADC status and control register (ADSCR), you can configure the analog input channel and the conversion mode, and monitor the conversion complete status. After the input voltage is converted, the ADC writes the results to the ADC data register (ADR). You can configure the ADC input clock using the ADC clock register (ADCLK).

All of the ADC registers are memory mapped. For the 68HC908GP32 ADC, the ADSCR is at memory location $003C, the ADR is at memory location $003D, and the ADCLK is at memory location $003E.

Next, we'll look at each register in detail beginning with the ADSCR.

ADC Status and Control Register (ADSCR)



COCO — Conversions Complete
1 = Conversion completed (AIEN = 0)
0 = Conversion not completed (AIEN = 0)/CPU interrupt (AIEN = 1)

The figure shows the eight bits of the ADC status and control register.

The conversion complete flag, COCO, is a read-only that indicates when a conversion is complete. In single conversion mode, the ADC sets the COCO flag to 1 after each conversion is completed. In continuous conversion mode, the ADC sets the COCO flag to 1 after the first conversion is completed.

In 68HC08 MCUs with direct-memory access (DMA), you can use bit seven to control DMA operation. In this configuration, bit seven of the ADSCR is referred to as IDMAS. You should only write to this bit position if your model contains DMA. Writing to this bit position in an MCU that doesn‘t contain DMA will mask ADC interrupts and cause unwanted results.

Setting the interrupt enable flag, AIEN, to 1 configures the ADC to generate an interrupt signal when the conversion is complete. The ADC generated interrupt signal is cleared when the data register is read or the status and control register is written.

You can select the ADC conversion mode with the continuous conversion flag, ADCO. Set this bit to 1 to select continuous conversion m ode. When this bit is reset, the ADC is configured for single conversion mode.

The remaining five bits of the ADSCR contain the ADC channel select bits, ADCH4 - ADCH0. In the 68HC908GP32 ADC, you can use these bits to select one of eight input channels (AD0 - AD7) and to verify ADC operations.

Input Channel Selection

Mux Channel Select
Analog-to-Digital Converter (ADC)

ADCH4	ADCH3	ADCH2	ADCH1	ADCH0	Input Select
0	0	0	0	0	PTB0/AD0
0	0	0	0	1	PTB1/AD1
0	0	0	1	0	PTB2/AD2
0	0	0	1	1	PTB3/AD3
0	0	1	0	0	PTB4/AD4
0	0	1	0	1	PTB5/AD5
0	0	1	1	0	PTB6/AD6
0	0	1	1	1	PTB7/AD7
↓	↓	↓	↓	↓	Reserved
1	1	0	1	1	Reserved
1	1	1	0	0	Reserved
1	1	1	0	1	V_{REFH}
1	1	1	1	0	V_{SSAD}
1	1	1	1	1	ADC power off

NOTE: If any unused channels are selected, the resulting ADC conversion will be unknown or reserved.

Next, let's review the ADC channel select bits in more detail.

The table shows the different bit codes for the 68HC908GP32 ADC channel select bits. The table shows the bit codes to select one of the eight input channels. You can also select V_{REFH} and V_{SSAD} as the input signal to verify ADC operations. When the ADC is not needed, you can turn the ADC power off by setting all of the ADC channel select bits to 1. This will help to minimize system power consumption.

Note that if you select an unused input channel or a reserved bit code, the ADC conversion result is unknown.

ADC Data Register (ADR)

Address: $0003D

	Bit 7	6	5	4	3	2	1	Bit 0
Read:	AD7	AD6	AD5	AD4	AD3	AD2	AD1	AD0
Write:								
Reset:	0	0	0	0	0	0	0	0

= Unimplemented

Next, let's look at the ADC data register (ADR). This is a read-only register that the ADC uses to store the conversion results. The ADC updates the ADR after each conversion is completed.

ADC Clock Register (ADCLK)

ADIV2–ADIV0 — ADC Clock Prescaler Bits

The last register is the ADC clock register (ADCLK).

Using the ADC clock prescaler bits, ADIV2 - ADIV0, you can select one of five divider values: 1, 2, 4, 8, or 16. The ADC generates the ADC clock frequency by dividing the clock source by the selected divider value.

You can select the ADC input clock source using the the ADC clock select bit, ADICLK. If the bit is reset, the ADC will use the external clock CGMXCLK as the input clock. If you set the bit to 1, the ADC will use the 68HC08 PLL-generated internal bus clock as the input clock.

You should select the input clock source based on the clock rate. The ADC module has been designed to operate best with an input clock rate of 1 MHz. If the external clock rate is greater than or equal to 1 MHz, use the external clock as the input source. Otherwise, use the internal bus clock as the input source.

Calculating Conversion Time

- Conversion starts after a write to the ADSCR register
- One conversion will require between 16 and 17 ADC clock cycles to complete.

$$\text{Conversion Time} = \frac{\text{16 to 17 ADC cycles}}{\text{ADC frequency}}$$

$$= \text{16 µs or 17 µs with 1 MHz ADC clock}$$

Once you've selected the input clock source, you can calculate the amount of time it takes to complete a single conversion. First determine the number of clock cycles it takes to complete the conversion and then divide this value by the input clock frequency.

The conversion process starts after the ADSCR is written to. A typical conversion takes 16 ADC clock cycles to complete. When there is a one clock synchronization delay between the CPU clock and the ADC clock, the conversion will take 17 clock cycles to complete. A one clock synchronization delay is possible if the A/D clock is different than the CPU clock. With the input clock frequency set to 1 MHz, a typical conversion takes about 16-17 µs.

Improving Conversion Accuracy

- Reduce noise in the A/D subsystem:
  - Separate noisy signals from the A/D signals.
  - Reduce the noise coupling.
  - See application note *AN1059/D*, *System Design and Layout Techniques for Noise Reduction in MCU Based Systems.*
- Allow for A/D on-current stabilization.
- Minimize source impedance.
- Average the results of multiple conversions.

Next, we'll look at some techniques you can use to improve analog-to-digital conversion accuracy.

The most effective method for improving accuracy is to reduce the noise that is introduced into the A/D subsystem using careful layout. Where possible, separate noisy signals from sensitive A/D signals. If complete separation is not cost-effective, reduce the noise coupling as much as possible. For more information about noise reduction, see application note AN1059/D, *System Design and Layout Techniques for Noise Reduction in MCU Based Systems* (see http://www.Freescale.com).

After turning on the A/D subsystem, allow time for the A/D on-current to stabilize before starting conversions. The ADC power-up time is provided in the technical data book. You should also verify that the source impedance is not too large. Errors caused by A/D input current leakage increases in proportion to the source impedance.

If bandwidth permits, you can reduce the impact of injected noise by taking multiple conversions and then averaging the results.

Question

If the input voltage, V_{ADIN}, is equal to V_{REFL}, what is the conversion result? Click on your BEST choice.

a) $00
b) $FF
c) All values between $00 and $FF
d) none of the above

Let's review what we've discussed so far with a couple of questions to check your understanding of the material. If the input voltage, V_{ADIN}, is equal to V_{REFL}, what is the conversion result? Click on your best choice.

Correct. When V_{ADIN}, equals V_{REFL}, the ADC converts the signal to the digital value $00.

Incorrect. When V_{ADIN}, equals V_{REFL}, the ADC converts the signal to the digital value $00.

Question

Which of these steps minimizes the noise injected into the A/D converter? Click on your BEST choice.

a) Separate noisy signals from A/D inputs
b) Reduce noise coupling
c) Stabilize the A/D on-current
d) Minimize source impedance
e) All of the above

Which of these steps minimizes the noise injected into the A/D converter? Click on your best choice.

Correct. All of the items in the list should be used to minimize noise. For more information, see application note AN1059/D, *System Design and Layout Techniques for Noise Reduction in MCU Based Systems* (see http://www.Freescale.com).

Incorrect. All of the items in the list should be used to minimize noise. For more information, see application note AN1059/D, *System Design and Layout Techniques for Noise Reduction in MCU Based Systems* (see http://www.Freescale.com).

Programming Exercise

- Write a program to configure the 68HC908GP32 ADC for measurements.
  - Use ADC channel 0 for input.
  - Set the ADCLK to 1MHz ADC clock.
  - Turn off the COP and LVI.
  - Display results on a set of LEDs.
  - Use Port D to drive the LEDs.
- Write a subroutine to perform the ADC measurement.
  - Identify ADC input channel in accumulator.
  - Verify that the port DDR bit is cleared.
  - Return conversion result in accumulator.

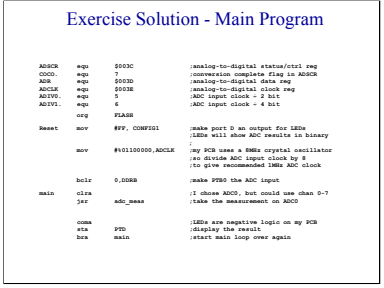
Now that we've discussed how to configure the ADC, let's write a program to configure the ADC Module and take some measurements. For this exercise, write a program to configure the 68HC908GP32 ADC.

Your main program should read input from one ADC channel and display the results on a set of 8 LEDs. After completing a measurement, your program should repeat the process.

In your program initialization, configure the ADCLK to 1MHz ADC clock assuming an 8 MHz system clock. Use port D to drive the LEDs. Use ADC channel 0 for input. Make sure you turn off the Computer Operating Properly Module (COP) and Low Voltage Inhibit Module (LVI).

Use a subroutine to perform the ADC measurement. Your main program should use the accumulator to pass the desired ADC input channel to the subroutine. In the subroutine, verify that one port DDR bit has been cleared for the ADC input channel. The subroutine should return the conversion result in the accumulator.

Take a moment to review the exercise instructions. When you are finished writing your program, click the Forward arrow to continue the tutorial and review the exercise solution.



Exercise Solution - Main Program

```
ADSCR   equ     $003C           ;analog-to-digital status/ctrl reg
COCO.   equ     7               ;conversion complete flag in ADSCR
ADR     equ     $003D           ;analog-to-digital data reg
ADCLK   equ     $003E           ;analog-to-digital clock reg
ADIV0.  equ     5               ;ADC input clock ÷ 2 bit
ADIV1.  equ     6               ;ADC input clock ÷ 4 bit
        org     FLASH

Reset   mov     #FF, CONFIG1    ;make port D an output for LEDs
                                ;LEDs will show ADC results in binary
                                ;
        mov     #%01100000,ADCLK ;my PCB uses a 8MHz crystal oscillator
                                ;so divide ADC input clock by 8
                                ;to give recommended 1MHz ADC clock

        bclr    0,DDRB          ;make PTB0 the ADC input

main    clra                    ;I chose ADC0, but could use chan 0-7
        jsr     adc_meas        ;take the measurement on ADC0

        coma                    ;LEDs are negative logic on my PCB
        sta     PTD             ;display the result
        bra     main            ;start main loop over again
```

Compare your main program to the one provided in the solution.

The main program initializes the 68HC908GP32 ADC Module out of reset and then calls a subroutine to take A/D measurements. On reset, the first step is to set up the hardware configuration register. It's best to do this explicitly, even if the default state is what you want.

Note that the assembler equate statements are not required for the program to run properly. These statements are used to make the source code more readable. They also make it easier to translate the program to run on different parts. For example, if the ADSCR is found in a different memory location on a different MCU, only the ADSCR equate statement needs to be changed to make the program run on the new part. If the actual hex location of the ADSCR had been used everywhere it was needed in the code, the program would have to be edited in multiple places. In an actual application program, the equate statements would typically be in a separate file and would be incorporated into the code using an "include" statement.

The program runs out of FLASH memory on this chip, so we need to tell the compiler where that is located in memory. The "org" directive does this. When the program is compiled into an S-record, the first line of that record will contain this information and the FLASH programmer will use it to put the program in the right place.

No RAM variables are used in this simple program, so the next step is to initialize the variables that only have to be set once. Port B0 will always be the input port, Port D will always be the output port, and the ADC clock will always run at the same frequency. Therefore, we set these variables before the main program loop.

Exercise Solution - Main Program

```
ADSCR   equ     $003C           ;analog-to-digital status/ctrl reg
COCO.   equ     7               ;conversion complete flag in ADSCR
ADR.    equ     $003D           ;analog-to-digital data reg
ADCLK   equ     $003E           ;analog-to-digital clock reg
ADIV0.  equ     5               ;ADC input clock - 2 bit
ADIV1.  equ     6               ;ADC input clock - 4 bit

        org     FLASH

Reset   mov     #FF, CONFIG1    ;make port D an output for LEDs
                                ;LEDs will show ADC results in binary
                                ;
        mov     #%01100000,ADCLK ;my PCB uses a 8MHz crystal oscillator
                                ;so divide ADC input clock by 8
                                ;to give recommended 1MHz ADC clock

        bclr    0,DDRB          ;make PTB0 the ADC input

main    clra                    ;I chose ADC0, but could use chan 0-7
        jsr     adc_meas        ;take the measurement on ADC0
                                ;
        coma                    ;LEDs are negative logic on my PCB
        sta     PTD             ;display the result
        bra     main            ;start main loop over again
```

The main program loop calls a subroutine to take a measurement and displays the measurement on the LEDs. The subroutine handles all ADC related activities. The subroutine requires that the clock is set. It also needs a working port pin and an input channel. The ADC clock and port set up were done on reset, so we just need to tell the subroutine what channel to use. This is done by loading our choice onto the accumulator.

When the subroutine returns, the 8-bit measurement is on the accumulator. As it turns out, the printed-circuit board is wired so that each LED turns on when its pin on port D goes low. A reading of zero would turn on all of the LEDs. This can be fixed by complimenting the value on the accumulator before writing it out to the port. All of the LEDs are off when it's a zero and on when we are reading full-scale. Store the value using port D and loop back to the beginning.

Exercise Solution - Subroutine

```
*****************************************************************
* SUBROUTINE NAME:  adc_meas              REVISED DATE: 01/02/2000  *
*                                                                   *
* PURPOSE:  perform A-to-D conversion on one channel of the ADC     *
*                                                                   *
* ENTRY CONDITIONS:    ADCLK register set to give 1MHz ADC clock    *
*                      Desired ADC input channel on accumulator     *
*                      One port DDR bit cleared for that channel    *
*                                                                   *
* EXIT CONDITIONS:     Conversion result on accumulator             *
*****************************************************************

adc_meas  ora       #%00100000          ;select continuous conversion mode
                                        ;  along with user-selected channel
 sta      ADSCR                         ;turn on ADC and wait for one whole
          brclr    COCO.,ADSCR,*        ;  conversion cycle to stabilize

 lda      ADR                           ;clear the COCO bit by reading ADR
          brclr    COCO.,ADSCR,*        ;this time take the real measurement

          mov      #%00011111,ADSCR     ;got it - now turn off the ADC
          lda      ADR                  ;retrieve the ADC measurement result
          rts                           ;return with result in accumulator
```

In this exercise, the ADC measurement is so simple that we could have easily used the main program loop to take the measurement. However, the subroutine is a good example of code reusability. It can be called from anywhere in a larger program. For example, suppose we had an application that polls the eight different temperature sensors on each of the port B pins. This routine can be used for all of them. The only input it needs is the identifier of the ADC input channel.

The ADC needs to be told whether to use single or continuous conversion mode. This is selected using bit 5 of the ADSCR. Next, combine the conversion select bit with the input channel choice into an 8-bit value. The best way to do this is with the logical OR operation. For example, let’s select continuous conversion mode using input channel 3. We use the accumulator to pass the subroutine a “3” for the input channel. If we "OR" the accumulator with $20 for continuous conversion mode, the result is $23.

As soon as we store this value in the ADSCR, the ADC Module starts converting. The ADC circuit hasn't stabilized yet, so we need to wait one conversion cycle for it to settle down. The most direct way to do this is to loop continuously while polling the conversion complete bit in the ADSCR. When this bit is set, the conversion is complete and the circuit has had time to stabilize. This step is not required if you start converting after the ADC Module is powered on.

Reading the ADR clears the COCO bit. We won't use the first measurement since the ADC circuit was unstable when it was taken. Instead, we’ll wait for the next measurement to complete. As soon as it's ready, the COCO bit gets set again and the we break out of the loop.

Turn the ADC Module off by setting the ADC channel select bits in the ADSCR. Then read the measurement from the ADR and return the result in the accumulator.

Exercise Solution - Enhancements

- Use an averaging algorithm to take measurements.
- Pass arguments using the stack.
- Enbable interrupt events to report conversion status.
- Include an error-handling routine.

Next, let's consider some enhancements we can make to this solution for real-world applications.

As we discussed earlier, we can use an averaging algorithm to filter the data. There are many ways to accomplish this depending on how you want to filter. One option you should always consider is using a hardware filter on the input. If you know that the real signal you're trying to measure can't change significantly in less than a second, there's no point in measuring changes to the nearest millisecond. Put a low-pass filter on the input channels and block the noise.

You might want to use the stack instead of the accumulator to pass arguments to the subroutine. If there is only one argument and the accumulator is available, there's nothing wrong with this method.

The loop and poll method in this subroutine may cause an infinite loop. If for some reason the COCO bit never gets set, the code will hang in the loop and never recover. Another problem with this method is that it wastes CPU time. An alternative method is to configure the ADC to trigger an interrupt event when the COCO bit is set. Using this method, your program can do other useful things while waiting for the ADC to finish its conversion.

With an interrupt-driven program, you can set the amount of time the program waits for an ADC conversion to complete. If the ADC has been converting for 20 clock cycles and the COCO bit still isn't set, the program could branch to an error-handling routine and inform you that something is wrong. This kind of fault-tolerant programming is beyond the scope of the exercise, but it forms an important part of an overall strategy to produce robust applications software.

Tutorial Completion

- ADC Module Configuration
- ADC Measurements
- A/D Conversion Accuracy

In this tutorial, you've had an opportunity to work with the 68HC08 ADC Module. You've learned how to configure the ADC and how to maximize A/D conversion accuracy. You've also written a program to configure the ADC and to take measurements.