8 Interfacing

Contents

Introduction

An embedded system is normally designed to interact with the external world. They sometimes need to provide a human-machine interface for simple input / output operations. They also may need to measure analog quantities and output analog quantities. The following sections look at various techniques of interfacing to our microcontroller.

8.1 Input Switches

8.1.1 Interfacing a Switch to the Microcontroller

Several simple switch interfaces are shown below:

Simple switch interfaces

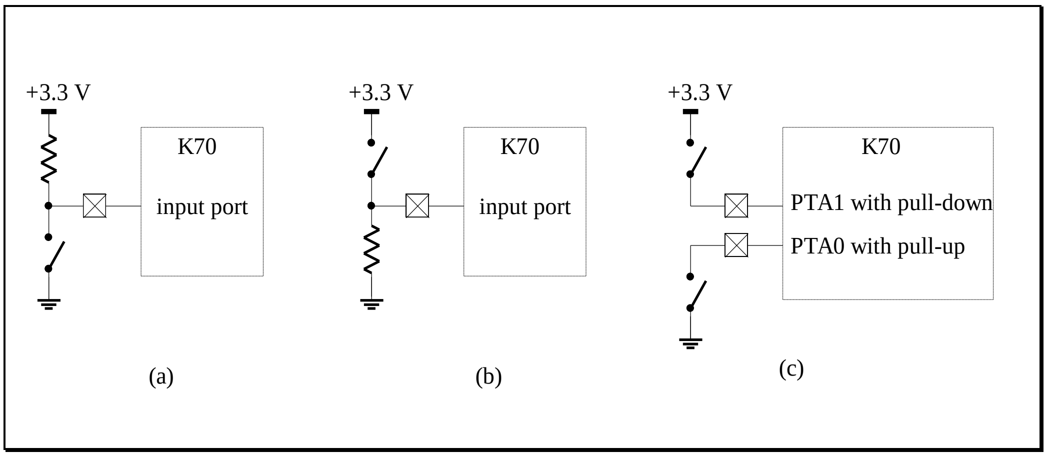


Figure 8.1

CMOS digital logic can use either pull-up or pull-down resistors, and the supply is typically 1.8 V, 2.5 V or 3.3V. In Figure 8.1 (a), a pull-up resistor is used to convert the mechanical signal into an electrical signal. When the switch is open, the input port is pulled to +3.3 V. When the switch is closed, the input port is forced to 0V.

Figure 8.1 (b) shows a pull-down circuit. When the switch in this circuit is open, the input is pulled to 0 V. When the switch is closed, the output is forced to +3.3 V. Notice the logic level of the switch input is reversed in the pull-down interface as compared to the pull-up case.

All ports on the K70 support both internal pull-ups and pull-downs. That is, either of the first two circuits in Figure 8.1 could be implemented on the K70 without the resistor, as shown in Figure 8.1 (c).

The software initialization for using a port sets the pull enable (`PE`) bit in the `PORTx_PCRn` register to enable pull-up or pull-down. For each port pin that is enabled for pull-up or pull-down, the corresponding pull select (`PS`) bit in the `PORTx_PCRn` register determines if it is pull-up (`1`) or pull-down (`0`).

EXAMPLE 8.1 Switch Interfacing with Internal Pull

Suppose we wish to initialize Port A for the circuit shown in Figure 8.1 (c). The software below will initialize Port A with the appropriate pull-up and pull-down.

```
// Port A Bit 0 is connected through a switch to 0 V
//   and uses internal pull-up
// Port A Bit 1 is connected through a switch to +3.3 V
//   and uses internal pull-down

void PortA_Init(void)
{
  // Enable clock gate for Port A to enable pin routing
  SIM_SCGC5 |= SIM_SCGC5_PORTA_MASK;

  // Set Port A Bit 0 as an input
  GPIOA_PDDR &= ~0x00000001;
  // PORTA_PCR0: ISF=0, MUX=1, PE = 1, PS = 1
  PORTA_PCR0 = PORT_PCR_ISF_MASK | PORT_PCR_MUX(1)
                 | PORT_PCR_PE_MASK | PORT_PCR_PS_MASK;

  // Set Port A Bit 1 as an input
  GPIOA_PDDR &= ~0x00000002;
  // PORTA_PCR1: ISF=0, MUX=1, PE = 1, PS = 0
  PORTA_PCR1 = PORT_PCR_ISF_MASK | PORT_PCR_MUX(1)
                 | PORT_PCR_PE_MASK;
}
```

8.1.2 Hardware Debouncing Using a Capacitor

Switch bounce causes multiple input changes on an input pin

Most inexpensive switches mechanically “bounce” when touched and when released. Typical bounce times range from 1 ms to 25 ms. Ideally, the switch resistance is zero (actually about 0.1Ω) when closed and infinite when open. This gives rise to the following switch timing:

Switch timing showing bounce on touch and release

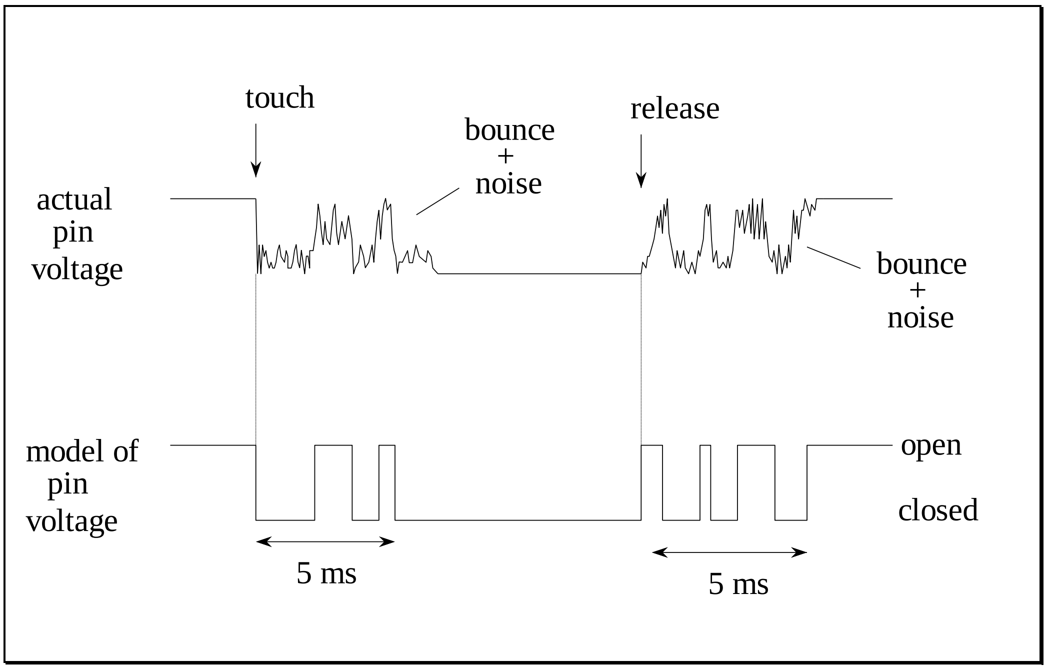


Figure 8.2

Hence, the electrical output “bounces” when using inexpensive switches and circuits having just a pull-up or pull-down resistor. It may or may not be important to debounce the switch. For example, if we are entering data via a keyboard, then we want to record only individual key presses. On the other hand, if the switch position specifies some static condition, and the operator sets the switch before turning on the microcontroller, then debouncing is not necessary.

A hardware method to debounce a switch places a capacitor across the switch to limit the rise time, followed by an inverter with hysteresis. With this circuit there is a significant delay from the release of the switch until the fall of the output.

A hardware circuit that removes switch bounce

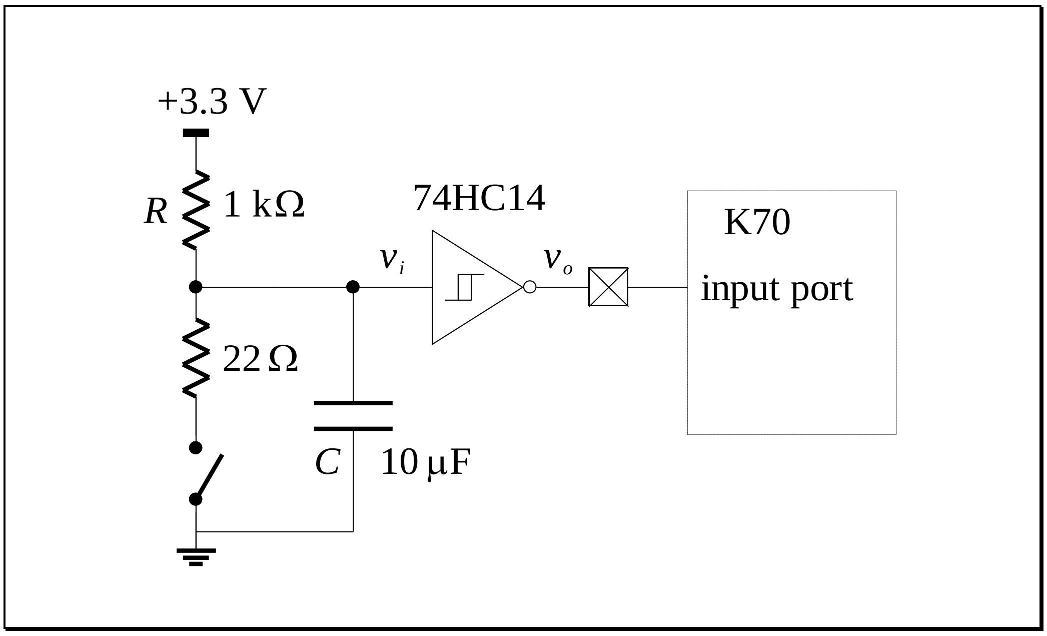


Figure 8.3

If the input switch is closed, its resistance will be about 0.1Ω, and the output of the 74HC14 will be high (logic 1). If the input switch is open, its resistance will be infinite, and the output of the 74HC14 will be low (logic 0). The 22Ω is used to limit the discharge current when the switch is pressed (which causes sparks that produce carbon deposits to build up until the switch no longer works).

8.6

Switch touch bounce is removed by the capacitor

The touch timing with and without the capacitor is shown below:

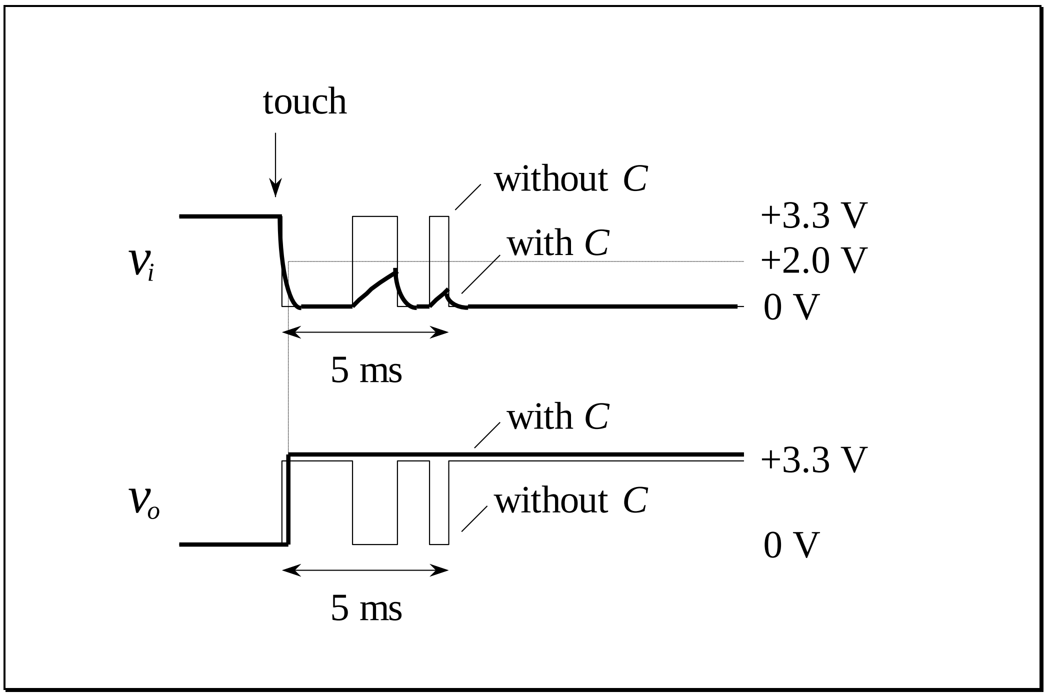


Figure 8.4

Notice that there is minimal delay between the touching of the switch and the transition of the Schmitt inverter output. This is because the capacitor is quickly discharged through the $22\,\Omega$ resistor.

With a capacitor-based debounced switch, there is minimal delay between the closing of the switch and the rising edge at the microcontroller input.	(9.1)

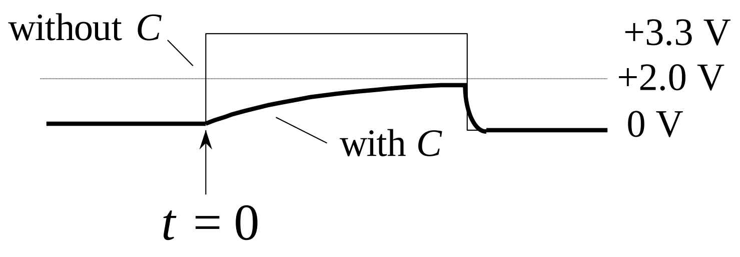
The voltage rise during a bounce interval when the switch is open is given by:

$$v(t) = V_{OH}\left(1 - e^{-t/RC}\right) \tag{9.2}$$

The capacitor is chosen such that the input voltage does not exceed the input high threshold voltage of the Schmitt trigger during the bouncing.

EXAMPLE 8.2 Choosing a Capacitor for Debouncing

In the example, $R = 1\,\text{k}\Omega$, and the bounce time is $\Delta t = 5\,\text{ms}$.



We choose C so that the voltage rise doesn't pass the Schmitt trigger input high threshold of $V_{T+} = 2\,\text{V}$ until $5\,\text{ms}$ has passed:

Timing used to calculate the capacitor value

$$\begin{aligned}
V_{T+} &\ge V_{OH}\left(1 - e^{-\Delta t/RC}\right) \\
1 - \frac{V_{T+}}{V_{OH}} &\le e^{-\Delta t/RC} \\
\ln\left(1 - \frac{V_{T+}}{V_{OH}}\right) &\le \frac{-\Delta t}{RC} \\
C &\ge \frac{-\Delta t}{R\ln\left(1 - \frac{V_{T+}}{V_{OH}}\right)} \\
&\ge \frac{-5\times10^{-3}}{1\times10^{3}\ln\left(1 - \frac{2}{3.3}\right)} \\
&\ge 5.367\ \mu\text{F}
\end{aligned}$$

Therefore, choose $C = 10\ \mu\text{F}$.

Switch release bounce is also removed by the capacitor

The release timing with and without the capacitor is shown below:

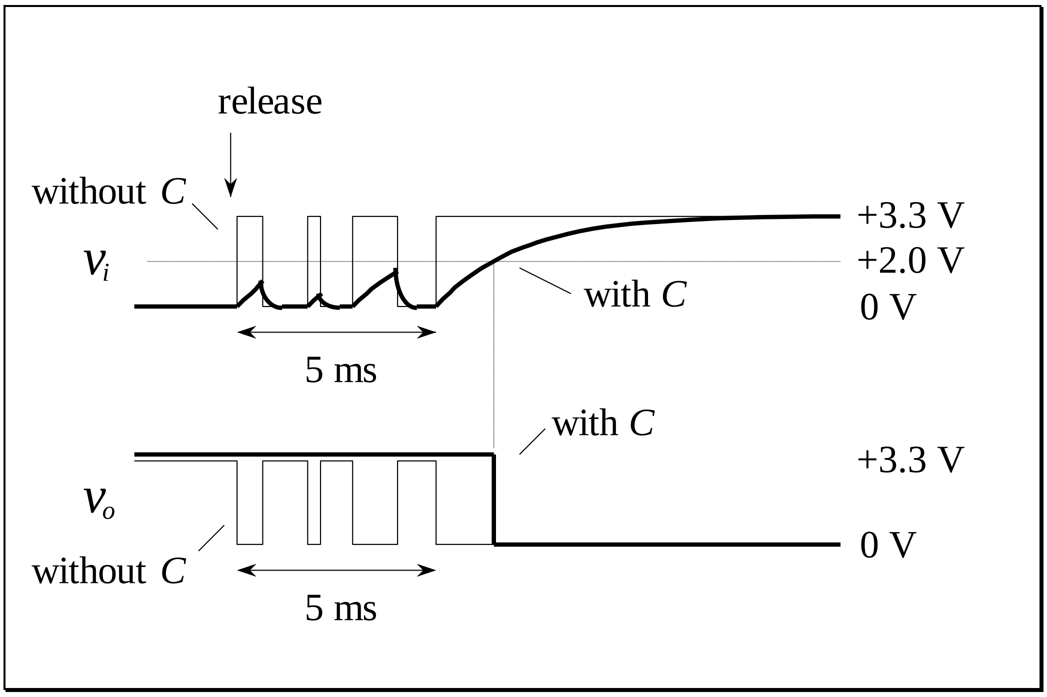


Figure 8.5

There is a significant delay from the release of the switch until the fall of the output, since the capacitor charges up slowly through the $1\,\text{k}\Omega$ resistor.

With a capacitor-based debounced switch, there is a large delay between the opening of the switch and the falling edge at the microcontroller input. (9.3)

Hysteresis is required on the inverter logic gate because the capacitor causes the "logic" input to rise very slowly. Thus, while the input voltage is in the transition region between "low" and "high", a regular logic gate will be operating in its linear region, and the output will be undefined. Furthermore, any noise on the input whilst in the transition region would cause a regular gate to toggle with the noise. The hysteresis removes the extra transitions that might occur with a regular gate:

Timing showing why a logic gate with hysteresis is used instead of a regular logic gate

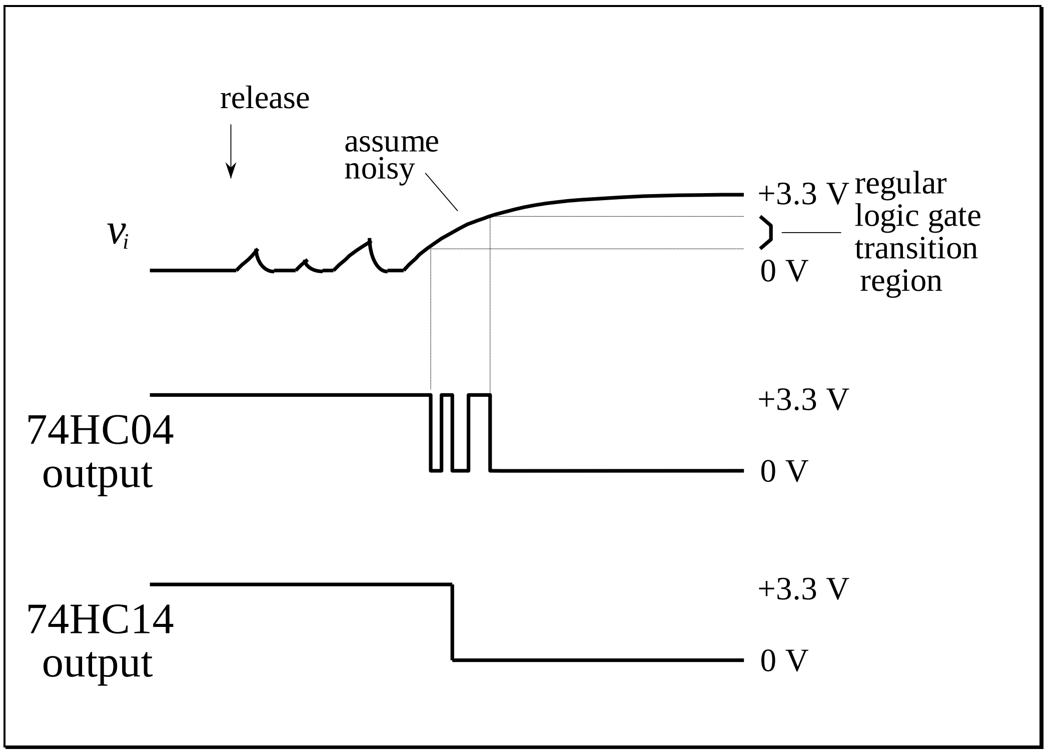


Figure 8.6

8.1.3 Software Debouncing

It is less expensive to remove switch bounce using software methods. It is appropriate to use a software approach because the software is fast compared to the bounce time. Typically we use a pull-up resistor to convert the switch position into a CMOS-level digital signal.

A switch interface for software debouncing

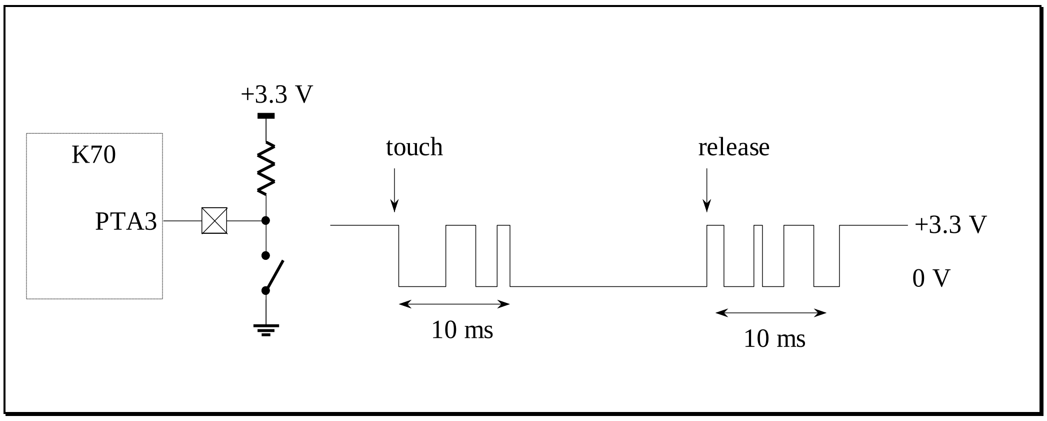


Figure 8.7

There are several ways to implement software debouncing. In the examples below, it is assumed that the switch bounce is less than 10 ms.

EXAMPLE 8.3 Software Debouncing – Simple Time Delay

In this example, the microcontroller is dedicated to the interface and does not perform any other functions while the routines are running. The routine waits for the switch to be pressed (PTA3 low) and returns 10 ms after the switch is pressed.

We assume that FlexTimer module 0 (FTM0) has been set up to use a free-running counter, and the clock module is 50 MHz. We use Channel 1 as a simple output compare and poll its flag:

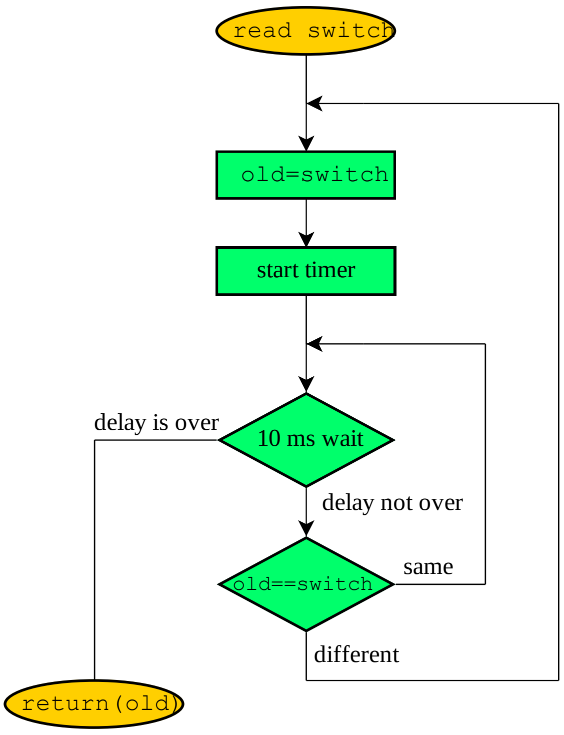
```
void WaitPress(void)
{
  // Loop here until switch is pressed
  while ((GPIOA_PDIR & 0x00000008) == 0x00000008);

  // Set the compare value 10 ms into the future
  FTM0_CnV(1) = FTM0_CNT + 15625;

  // Wait for switch to stop bouncing
  while ((FTM0_CnSC(1) & FTM_CnSC_CHF_MASK) == 0);
}

void WaitRelease(void)
{
  // Loop here until switch is released
  while ((GPIOA_PDIR & 0x00000008) == 0);

  // Set the compare value 10 ms into the future
  FTM0_CnV(1) = FTM0_CNT + 15625;

  // Wait for switch to stop bouncing
  while ((FTM0_CnSC(1) & FTM_CnSC_CHF_MASK) == 0);
}

void PortA_Init(void)
{
  // Enable clock gate for Port A to enable pin routing
  SIM_SCGC5 |= SIM_SCGC5_PORTA_MASK;

  // Set Port A Bit 0 as an input
  GPIOA_PDDR &= ~0x00000001;
  // PORTA_PCR0: ISF=0, MUX=1, PE = 0, PS = 0
  PORTA_PCR0 = PORT_PCR_ISF_MASK | PORT_PCR_MUX(1);
}
```

8.12

```c
void FTM_Init(void)
{
  // Enable clock gate to FTM0 module
  SIM_SCGC6 |= SIM_SCGC6_FTM0_MASK;

  // Ensure the counter is a free-running counter
  FTM0_CNTIN = 0;
  FTM0_MOD = 0xffff;
  FTM0_CNT = 0;

  // Use the system clock for the counter (50 MHz)
  // and set the prescale to 32
  FTM0_SC |= FTM_SC_CLKS(1) | FTM_SC_PS(5);

  // Set Channel 1 as an output compare
  FTM0_CnSC(1) |= FTM_CnSC_MSA_MASK;

  // Enable the timer module in FTM mode
  FTM0_MODE |= FTM_MODE_FTMEN_MASK;
}
```

EXAMPLE 8.4 Software Debouncing - Waiting for Stability

In this example, the microcontroller reads the current value of the switch. If the switch is currently bouncing, it will wait for stability.



A return value of 0 means pressed (`PTA3 = 0`), and 1 means not pressed (`PTA3 = 1`). Notice that the software always waits in a "do nothing" loop for 10 ms. This inefficiency can be eliminated by placing the switch I/O in a foreground interrupt-driven thread.

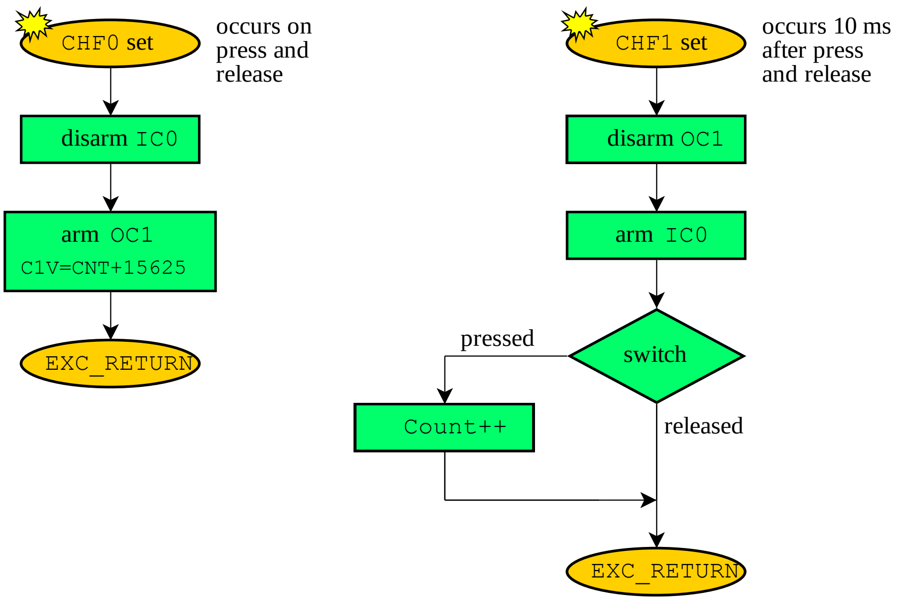
```
uint32_t ReadPTA3(void)
{
  uint32_t old;

  // Get current value
  old = (GPIOA_PDIR & 0x00000008);
  // Set the compare value 10 ms into the future
  FTM0_CnV(1) = FTM0_CNT + 15625;
  // Unchanged for 10 ms?
  while ((FTM0_CnSC(1) & FTM_CnSC_CHF_MASK) == 0)
  {
    // Changed?
    if (old != (GPIOA_PDIR & 0x00000008))
    {
      old = (GPIOA_PDIR & 0x00000008); // new value
      FTM0_CnV(1) = FTM0_CNT + 15625;  // restart delay
    }
  }
  return old;
}

void PortA_Init(void)
{
  // As before...
}

void FTM_Init(void)
{
  // As before...
}
```

With a software-based debounced switch, the signal arrives at the microcontroller input without delay, but software delays may occur at either touch or release. (9.4)

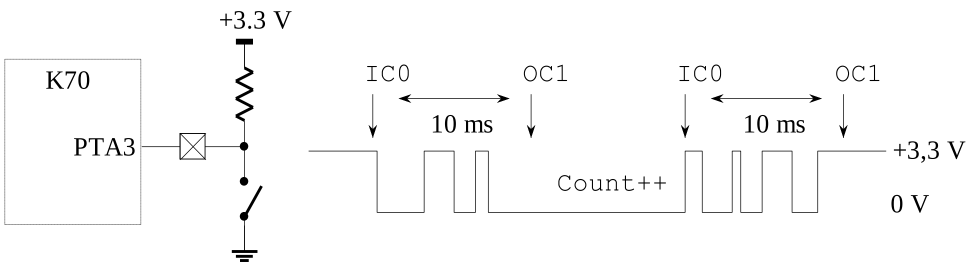
Input capture is a convenient mechanism to detect changes on the digital signal. The input capture can be configured to interrupt either on the rise , the fall or both the rise and fall. Because of the bounce, any of these modes will generate an interrupt request when the key is touched or released. A combination of input capture and output compare interrupts allows the switch interface to be performed in the background.

EXAMPLE 8.5 Software Debouncing - Interrupts

This example simply counts the number of times the switch is pressed. The `IC0` interrupt occurs immediately after the switch is pressed and released. Because the `IC0` handler disarms itself, the bounce will not cause additional interrupts. The `OC1` interrupt occurs 10 ms after the switch is pressed and 10 ms after the switch is released. At this time the switch position is stable (no bounce).



The first `IC0` interrupt occurs when the switch is first touched. The first `OC1` interrupt occurs 10 ms later. At this time the global variable `Count` is incremented. The second `IC0` interrupt occurs when the switch is released. The second `OC1` interrupt does not increment the `Count` but simply rearms the input capture system. The initialization routine initializes the system with `IC0` armed and `OC1` disarmed.



8.16

```
// Counts the number of button pushes
// Button connected to PTA3 = Ch0 of FTM0

uint32_t Count;              // Number of button pushes
const uint16_t WAIT = 15625; // The bounce wait time

void PortA_Init(void)
{
  // Enable clock gate for Port A to enable pin routing
  SIM_SCGC5 |= SIM_SCGC5_PORTA_MASK;

  // Set up PTA3 to be an input on FTM0 Ch0
  // PORTA_PCR3: ISF=0, MUX=3
  PORTA_PCR3 = PORT_PCR_MUX(3);
}

void FTM_Init(void)
{
  // Disable interrupts
  __DI();

  // Enable clock gate to FTM0 module
  SIM_SCGC6 |= SIM_SCGC6_FTM0_MASK;

  // Ensure the counter is a free-running counter
  FTM0_CNTIN = 0;
  FTM0_MOD = 0xffff;
  FTM0_CNT = 0;

  // Use the system clock for the counter (50 MHz)
  // and set the prescale to 32
  FTM0_SC |= FTM_SC_CLKS(1) | FTM_SC_PS(5);

  // Set Channel 0 as an input capture
  FTM0_CnSC(0) &= ~FTM_CnSC_MSA_MASK
  // Capture on rising or falling edge
  FTM0_CnSC(0) |=FTM_CnSC_ELSB_MASK | FTM_CnSC_ELSA_MASK;
  // Clear the IC0 flag if it is set
  FTM0_CnSC(0) &= ~FTM_CnSC_CHF_MASK;
  // Enable interrupts on Channel 0
  FTM0_CnSC(0) |= FTM_CnSC_CHIE_MASK;

  // Set Channel 1 as an output compare - output pin disabled
  FTM0_CnSC(1) |= FTM_CnSC_MSA_MASK;
  // Disable interrupts on Channel 1
  FTM0_CnSC(1) &= ~FTM_CnSC_CHIE_MASK;

  // Enable the timer module in FTM mode
  FTM0_MODE |= FTM_MODE_FTMEN_MASK;

  // Reset counter to zero
  Count = 0;

  // Enable interrupts
  __EI();
}
```

```
void __attribute__ ((interrupt)) FTM0_ISR(void)
{
  uint8_t channelNb;

  // Respond to IC0 interrupt if it occurred
  if (FTM0_CnSC(0) & FTM_CnSC_CHIE_MASK)
  {
    if (FTM0_CnSC(0) & FTM_CnSC_CHF_MASK)
    {
      // Clear interrupt flag
      FTM0_CnSC(0) &= ~FTM_CnSC_CHF_MASK;
      // Turn IC0 interrupt off
      FTM0_CnSC(0) &= ~FTM_CnSC_CHIE_MASK;
      // Turn OC1 interrupt on
      FTM0_CnSC(1) |= FTM_CnSC_CHIE_MASK;
      // Wait for 10 ms
      FTM0_CnV(1) = FTM0_CNT + WAIT;
    }
  }

  // Respond to OC1 interrupt if it occurred
  if (FTM0_CnSC(1) & FTM_CnSC_CHIE_MASK)
  {
    if (FTM0_CnSC(1) & FTM_CnSC_CHF_MASK)
    {
      // Clear interrupt flag
      FTM0_CnSC(1) &= ~FTM_CnSC_CHF_MASK;
      // Turn OC1 interrupt off
      FTM0_CnSC(1) &= ~FTM_CnSC_CHIE_MASK;
      // Clear the IC0 flag if it is set
      FTM0_CnSC(0) &= ~FTM_CnSC_CHF_MASK;
      // Turn IC0 interrupt on
      FTM0_CnSC(0) |= FTM_CnSC_CHIE_MASK;
      // Increment counter if button is pressed
      if ((GPIOA_PDIR & 0x00000008) == 0)
        Count++;
    }
  }
}
```

In `vectors.c`:

```
(tIsrFunc)&Cpu_Interrupt,     /* 0x4D  CMP2 */
(tIsrFunc)&FTM0_ISR,          /* 0x4E  FTM0 */
(tIsrFunc)&Cpu_Interrupt,     /* 0x4F  FTM1 */
```

EXAMPLE 8.6 Software Debouncing – Interrupts with Low Latency

The latency of the previous example is defined as the time when the switch is touched until the time when the count is incremented. Because of the delay introduced by the `OC1` interrupt, the latency is 10 ms. If we assume the switch is not bouncing (currently being touched or released) at the time of the initialization, we can reduce this latency to less than $1\,\mu\text{s}$ by introducing a global Boolean variable called `SwitchPushed`. If `SwitchPushed` is false, then the switch is currently not pushed and the software is searching for a touch. If `SwitchPushed` is true, then the switch is currently pushed and the software is searching for a release.

```
// Counts the number of button pushes
// Button connected to PTA3 = Ch0 of FTM0

uint32_t Count;               // Number of button pushes
const uint16_t WAIT = 15625;  // The bounce wait time
BOOL SwitchPushed;            // State of button
  // bFALSE means open,  looking for a touch
  // bTRUE means closed, looking for release

void FTM_Init(void)
{
  // Disable interrupts
  __DI();

  // Enable clock gate to FTM0 module
  SIM_SCGC6 |= SIM_SCGC6_FTM0_MASK;

  // Ensure the counter is a free-running counter
  FTM0_CNTIN = 0;
  FTM0_MOD = 0xffff;
  FTM0_CNT = 0;

  // Use the system clock for the counter (50 MHz)
  // and set the prescale to 32
  FTM0_SC |= FTM_SC_CLKS(1) | FTM_SC_PS(5);

  // Set Channel 0 as an input capture
  FTM0_CnSC(0) &= ~FTM_CnSC_MSA_MASK
  // Capture on rising or falling edge
  FTM0_CnSC(0) |=FTM_CnSC_ELSB_MASK | FTM_CnSC_ELSA_MASK;
  // Clear the IC0 flag if it is set
  FTM0_CnSC(0) &= ~FTM_CnSC_CHF_MASK;
  // Enable interrupts on Channel 0
  FTM0_CnSC(0) |= FTM_CnSC_CHIE_MASK;

  // Set Channel 1 as an output compare - output pin disabled
  FTM0_CnSC(1) |= FTM_CnSC_MSA_MASK;
  // Disable interrupts on Channel 1
  FTM0_CnSC(1) &= ~FTM_CnSC_CHIE_MASK;
  // Enable the timer module in FTM mode
  FTM0_MODE |= FTM_MODE_FTMEN_MASK;

  // Reset counter to zero
  Count = 0;

  // Get initial state of button
  SwitchPushed = ((GPIOA_PDIR & 0x00000008) == 0);

  // Enable interrupts
  __EI();
}

void __attribute__ ((interrupt)) FTM0_ISR(void)
{
  uint8_t channelNb;

  // Respond to IC0 interrupt if it occurred
  if (FTM0_CnSC(0) & FTM_CnSC_CHIE_MASK)
  {
    if (FTM0_CnSC(0) & FTM_CnSC_CHF_MASK)
    {
      // Clear interrupt flag
      FTM0_CnSC(0) &= ~FTM_CnSC_CHF_MASK;
      // Turn IC0 interrupt off
      FTM0_CnSC(0) &= ~FTM_CnSC_CHIE_MASK;
      // Turn OC1 interrupt on
      FTM0_CnSC(1) |= FTM_CnSC_CHIE_MASK;
      // Wait for 10 ms
      FTM0_CnV(1) = FTM0_CNT + WAIT;
      // An edge occurred - toggle state
      SwitchPushed = !SwitchPushed;
      // If a touch occurred, increment the counter
      if (SwitchPushed)
        Count++;
    }
  }

  // Respond to OC1 interrupt if it occurred
  if (FTM0_CnSC(1) & FTM_CnSC_CHIE_MASK)
  {
    if (FTM0_CnSC(1) & FTM_CnSC_CHF_MASK)
    {
      // Clear interrupt flag
      FTM0_CnSC(1) &= ~FTM_CnSC_CHF_MASK;
      // Turn OC1 interrupt off
      FTM0_CnSC(1) &= ~FTM_CnSC_CHIE_MASK;
      // Clear the IC0 flag if it is set
      FTM0_CnSC(0) &= ~FTM_CnSC_CHF_MASK;
      // Turn IC0 interrupt on
      FTM0_CnSC(0) |= FTM_CnSC_CHIE_MASK;
    }
  }
}
```

8.20

In `vectors.c`:

```
(tIsrFunc)&Cpu_Interrupt,    /* 0x4D  CMP2 */
(tIsrFunc)&FTM0_ISR,         /* 0x4E  FTM0 */
(tIsrFunc)&Cpu_Interrupt,    /* 0x4F  FTM1 */
```

Now the latency is simply the time required for the microcontroller to recognize and process the input capture interrupt. Assuming there are no other interrupts, this time is less than 50 cycles.

8.2 Analog to Digital Conversion

An analog-to-digital converter (ADC) is used to quantize an external analog signal so as to represent it digitally. If the samples of an analog signal are taken at a sufficiently high rate, then the samples furnish enough information for the analog signal to be reconstructed exactly. Once the analog signal has been converted to a digital form, it can be filtered, manipulated, and processed. The processed signal can then be converted back to an analog signal through the use of a digital-to-analog converter (DAC).

8.2.1 ADC Module

The Analog-to-Digital Converter (ADC) module has the capability of sampling up to four pairs of differential and 24 single-ended analog channels with either 8-bit, 10-bit, 12-bit or 16-bit quantization. The output format is in 2's complement 16-bit sign extended for differential mode and unsigned 16-bit for single-ended modes. The signed notation is useful when converting shifted bipolar inputs (a shifted bipolar input will have its "zero" at exactly half way between the minimum and maximum conversion values). The ADC can also be set up to automatically sample a sequence of analog channels.

A simplified block diagram of the ADC module is shown below:

A simplified ADC block diagram

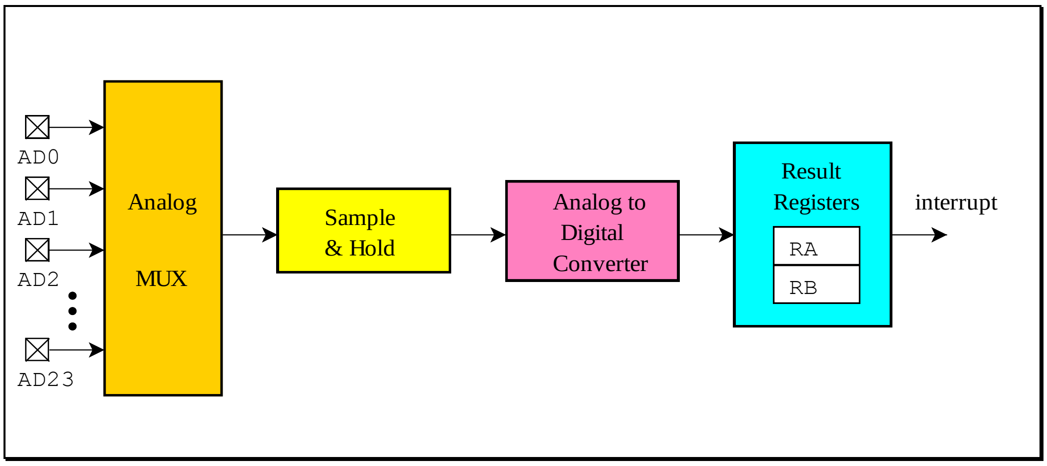


Figure 8.8

The analog multiplexer (MUX) is just an analog switch that connects one of the analog channels to the sample-and-hold (S/H) block.

The analog multiplexer is just a switch

The sample and hold block is used to charge a sample capacitor to a voltage very close to the applied analog voltage

The sample-and-hold circuit consists of a *sample capacitor* and a buffer. It is important to take into account the characteristics of the S/H block in the design of the analog interface hardware external to the microcontroller. The external hardware's output resistance and the sample capacitor form a first-order lowpass filter, also known as a single time constant (STC) circuit, or just a lowpass *RC* circuit. This first-order filter determines the amount of time that is needed to charge the sample capacitor to a voltage that is almost equal to the true analog voltage.

Analog input circuit, showing external resistance and sample capacitor that form a first-order lowpass filter

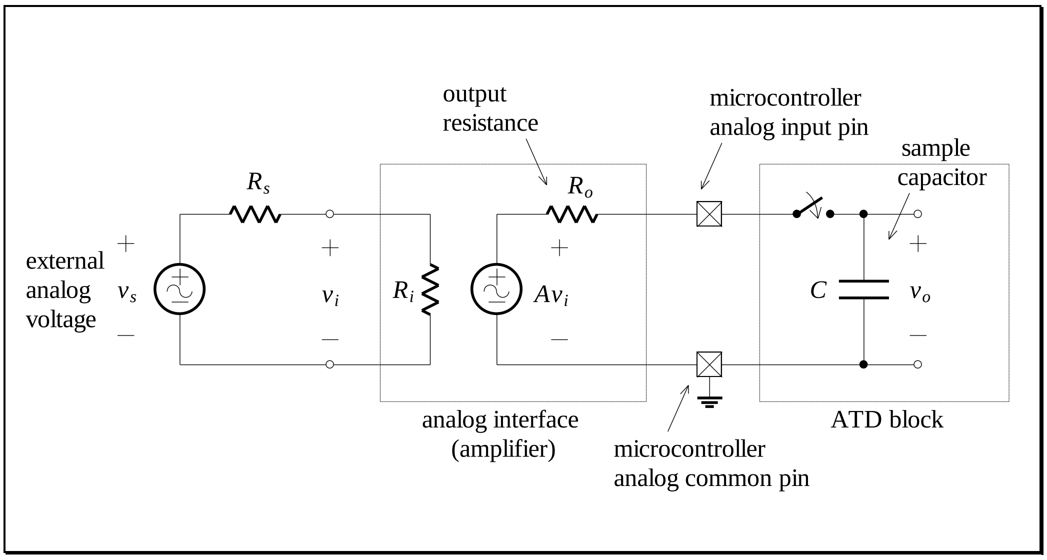


Figure 8.9

The voltage held on the sample capacitor is fed into the analog-to-digital converter. There are many types of ADC – the type used in the K70 series of microcontrollers is a successive approximation architecture (SAR). It functions by comparing the stored analog sample voltage with a series of digitally generated analog voltages. By following a binary search algorithm, the ADC locates the approximating voltage that is nearest to the sampled voltage.

The results of the analog-to-digital conversions are stored in separate result registers. The completion of an analog-to-digital conversion can be used to trigger an interrupt.

Further Information

A complete description of the ADC module can be found in Chapter 38 of Freescale's *K70 Sub-Family Reference Manual*.

8.3 Digital to Analog Conversion

The K70 has two on-board 12-bit low-power, general-purpose digital-to-analog converters. The output of the DAC can be placed on an external pin or set as one of the inputs to the analog comparators, or ADC.

A simple DAC can also be made from a pulse width modulated (PWM) waveform. If a PWM waveform is passed through a lowpass filter, and the PWM has a sufficiently high frequency, then the output of the filter will be a smooth analog waveform corresponding to the average value of the PWM taken over many periods.

8.3.1 Pulse Width Modulator

A pulse width modulator is a device which varies the duty cycle (the "on time" versus "total time", in percent) of a square wave. They can be used to turn transistors on and off in an external circuit to drive devices such as DC motors and 3-phase AC motors. They can also be used to create a simple digital-to-analog converter.

In the K70, PWM waveforms are generated by the FlexTimer module (FTM). A conceptual block diagram of the PWM functionality is shown below:

A simplified PWM block diagram

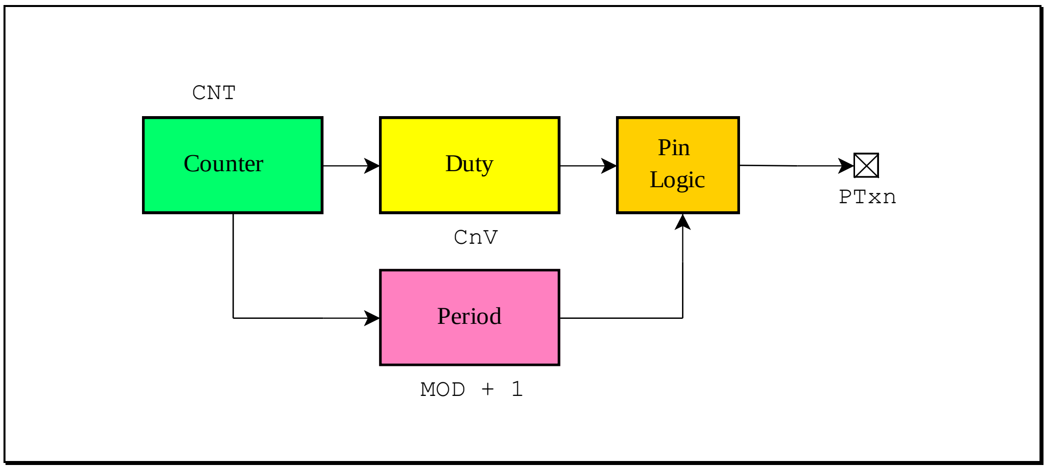


Figure 8.10

The PWM uses a duty value and a period value

Each of the 8 channels of the FTM can be set up as a PWM. Each channel uses the common 16-bit counter `CNT`, and the common modulus value held in `MOD`. For each channel, the counter compares to two values: a *duty* value held in `CnV`; and the common *period* value held in `MOD`. In its simplest mode of operation, known as edge-aligned PWM, the output is set high when the counter equals the period value, and the output is set low when the counter equals the duty value.

PWM waveform generation

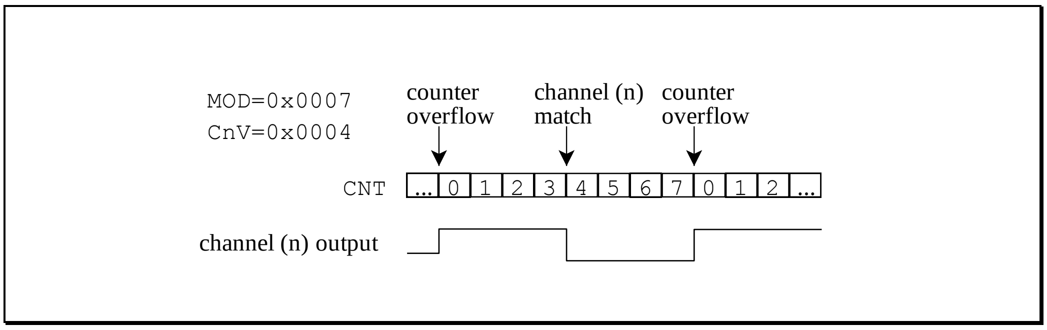


Figure 8.11

Various constant duty cycle waveforms are shown below:

Various constant duty cycle PWM waveforms

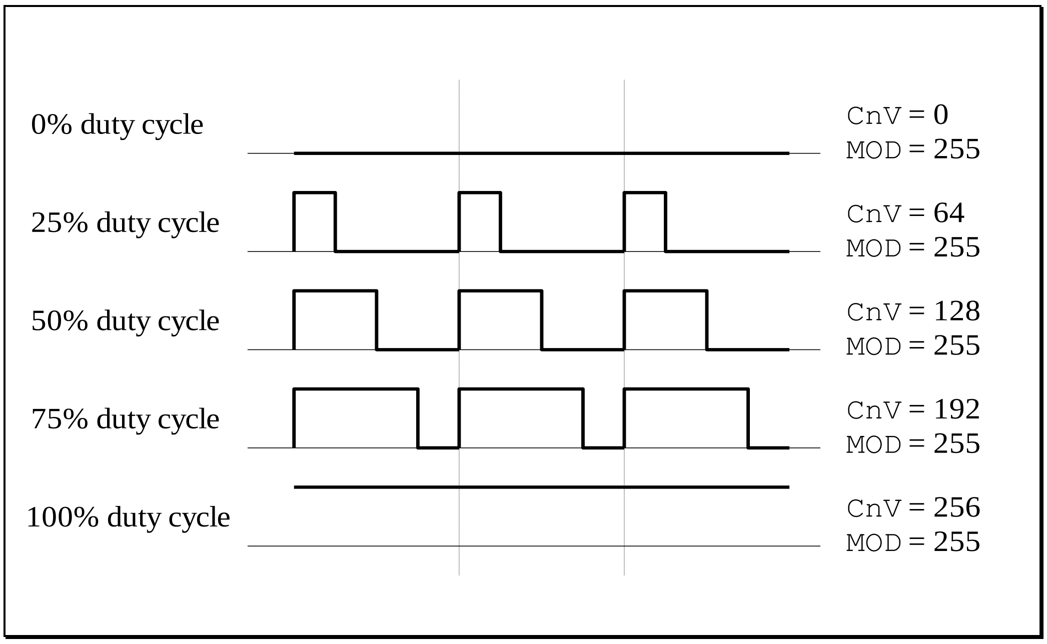


Figure 8.12

The generation of fixed duty cycle square waves is only one application of the PWM function. It is more generally used to *modulate* the pulse width of the square wave. Such a waveform is shown below:

A PWM waveform that approximates a sine wave

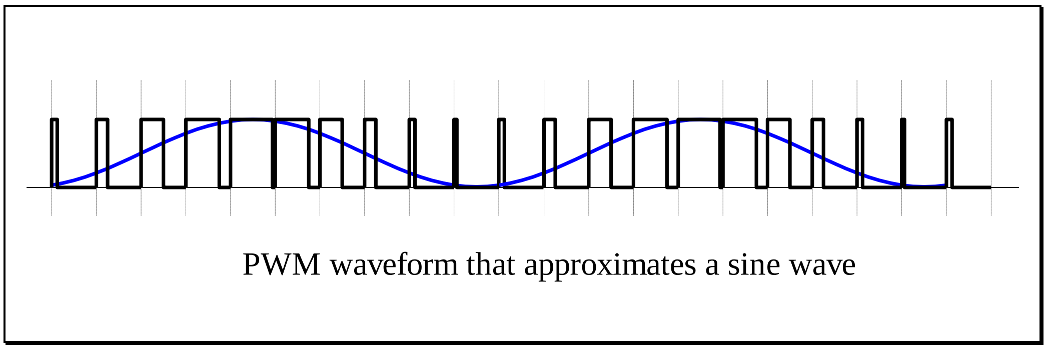


Figure 8.13

Such a PWM waveform can be used to generate a continually varying analog signal. The area under the PWM waveform approximates the area under the desired analog waveform. The PWM waveform needs to be filtered by external hardware to remove the high frequency components, and to leave just the fundamental and DC components. You can think of the analog output as responding to the *average* value of the PWM waveform (over a small time interval).

An external filter is usually not required when the device being driven provides an inherent filtering function. For example, a DC motor, which exhibits mechanical inertia, cannot respond to the rapid fluctuations of the PWM waveform – but it can respond to the slowly varying "average" value of the waveform. In this case, the DC motor speed would be seen to vary sinusoidally. There may also be an audible "hum" or ""buzz" due to the high frequency components being within the range of human hearing (20 Hz – 20 kHz). If such a hum is undesirable, then the designer can increase the frequency of the PWM wave so that the high frequency components are out of audio range.

Generation of an analog voltage via a PWM output

If an analog voltage is required, a simple buffered *RC* circuit can be used:

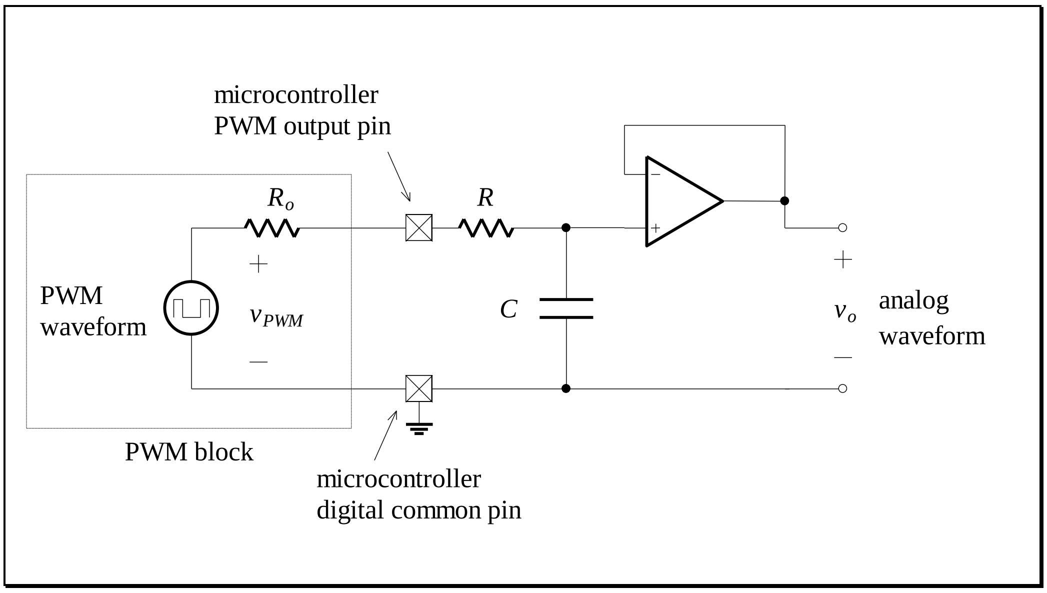


Figure 8.14

Further Information

A complete description of the FlexTimer Module can be found in Chapter 43 of Freescale's *K70 Sub-Family Reference Manual*.