Objectives

Play Video

The objectives of this chapter are to:

- Review real-time applications that require priority
- Overview edge-triggered interrupts
- Use the operating system to debounce switches
- Implement a priority scheduler
- Review of other real-time operating systems

To motivate the need for priority we will discuss some classic real-time system scenarios like Data Acquisition systems, Digital Signal Processing (DSP), and Real-Time Control systems. The level of detail provided here is not needed for the course, but we believe it will give you a context for the kinds of systems you may encounter as a practitioner in the RTOS domain.

4.1. Scenarios[edit]

4.1.1. Data Acquisition Systems
[edit]

Data Acquisition Systems

Play Video

Figure 4.1 illustrates the integrated approach to data acquisition systems. In this section, we begin with the clear understanding of the problem. We can use the definitions in this section to clarify the design parameters as well as to report the performance specifications.

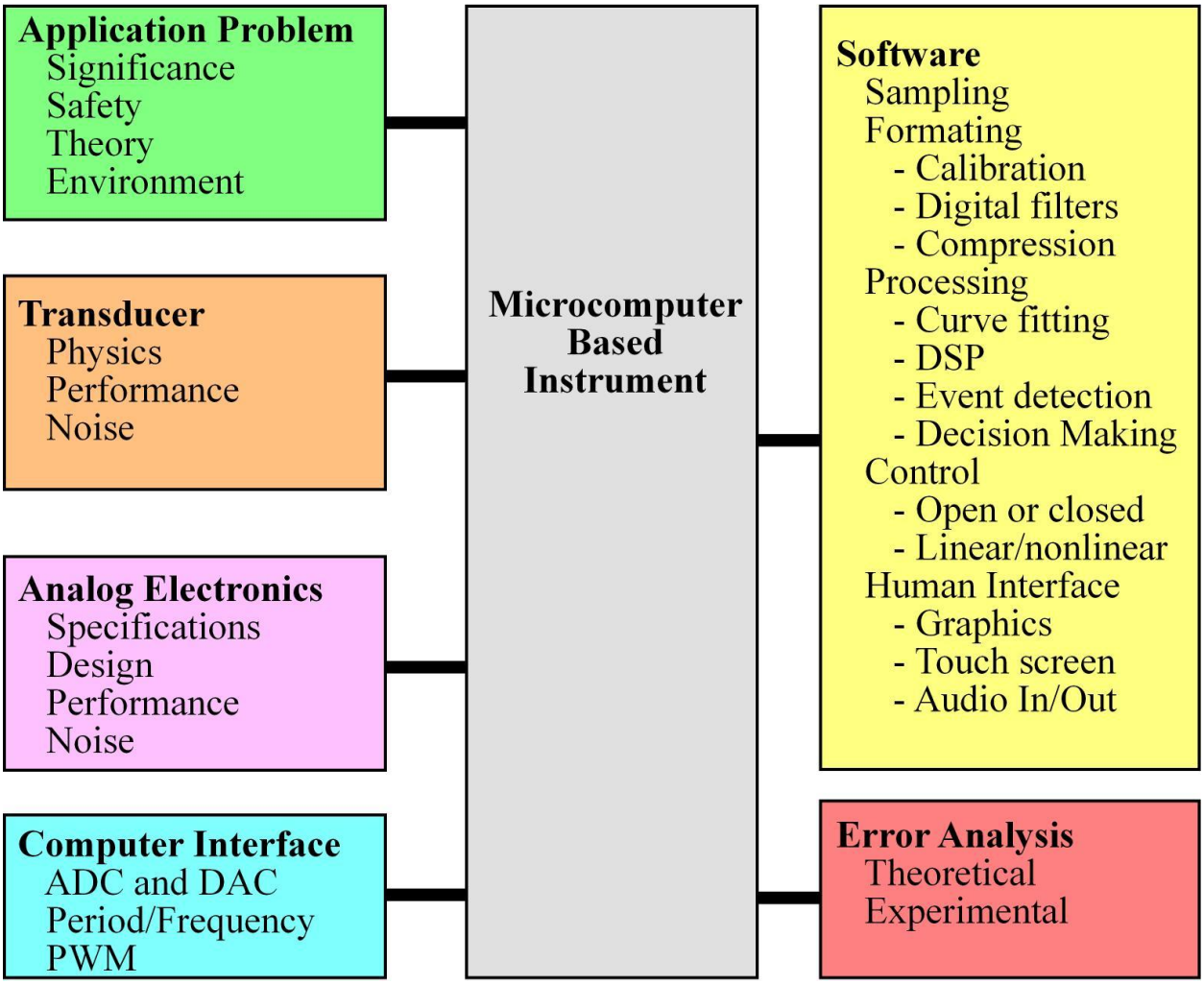

Figure 4.1. Individual components are integrated into a data acquisition system.

The **measurand** is the physical quantity, property, or condition that the instrument measures. See Figure 4.2. The measurand can be inherent to the object (like position, mass, or color), located on the surface of the object (like the human EKG, or surface temperature), located within the object (e.g., fluid pressure, or internal temperature), or separated from the object (like emitted radiation.) In general, a **transducer** converts one energy type into another. In the context of this section, the transducer converts the measurand into an electrical signal that can be processed by the microcontroller-based instrument. Typically, a transducer has a primary sensing element and a variable conversion element. The primary sensing element interfaces directly to the object and converts the measurand into a more convenient energy form. The output of the variable conversion element is an electrical signal that depends on the measurand. For example, the primary sensing element of a pressure transducer is the diaphragm, which converts pressure into a displacement of a plunger. The variable conversion element is a strain gauge that converts the plunger displacement into a change in electrical resistance. If the strain gauge is placed in a bridge circuit, the voltage output is directly proportional to the pressure. Some transducers perform a direct conversion without having a separate primary sensing

element and variable conversion element. The system contains signal processing, which manipulates the transducer signal output to select, enhance, or translate the signal to perform the desired function, usually in the presence of disturbing factors. The signal processing can be divided into stages. The **analog signal processing** consists of instrumentation electronics, isolation amplifiers, amplifiers, analog filters, and analog calculations. The first analog processing involves calibration signals and preamplification. Calibration is necessary to produce accurate results. An example of a calibration signal is the reference junction of a thermocouple. The second stage of the analog signal processing includes filtering and range conversion. The analog signal range should match the ADC analog input range. Examples of analog calculations include: RMS calculation, integration, differentiation, peak detection, threshold detection, phase lock loops, AM FM modulation/demodulation, and the arithmetic calculations of addition, subtraction, multiplication, division, and square root. When period, pulse width, or frequency measurement is used, we typically use an analog comparator to create a digital logic signal to measure. Whereas the Figure 4.1 outlined design components, Figure 4.2 shows the data flow graph for a data acquisition system or control system. The **control system** uses an actuator to drive a parameter in the real world to a desired value while the data acquisition system has no actuator because it simply measures the parameter in a nonintrusive manner.

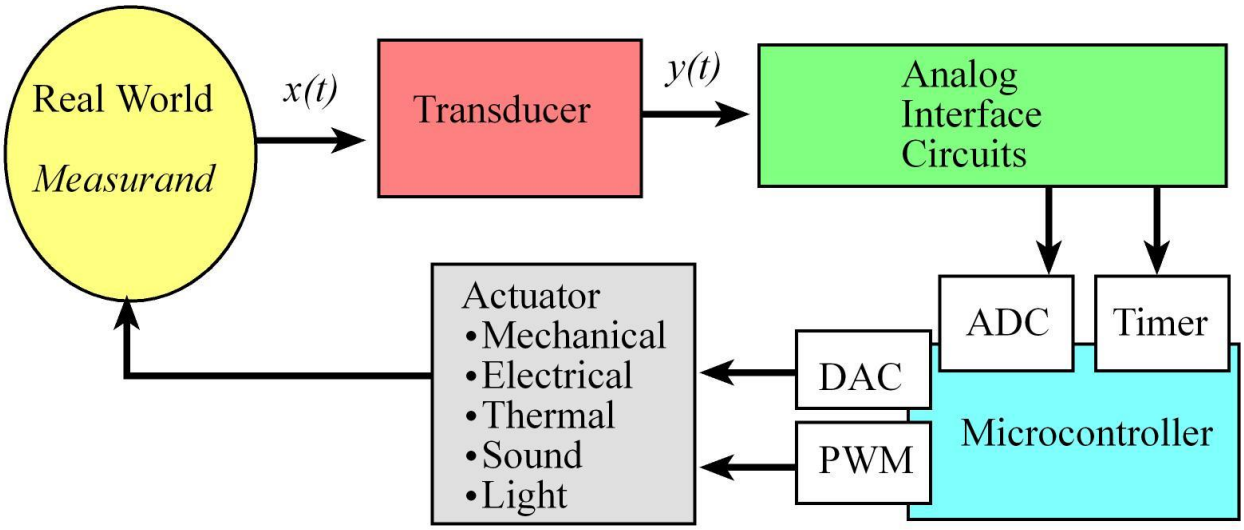

Figure 4.2. Signal paths for a data acquisition system without an actuator; the control system includes an actuator so the system can use feedback to drive the real-world parameter to a desired state.

The **data conversion element** performs the conversion between the analog and digital domains. This part of the instrument includes: hardware and software computer interfaces, ADC, DAC, and calibration references. The **analog to digital converter** (ADC) converts the analog signal into a digital number. The **digital to analog converter** (DAC) converts a digital number to an analog output.

In many systems the input could be digital rather than analog. For these systems measuring period, pulse width, and/or frequency provides a low-cost high-precision alternative to the traditional ADC. Similarly the output of the system could be digital. The **pulse width modulator** (**PWM**) is a digital output with a constant period, but variable duty cycle. The software can adjust the output of the actuator by setting the duty cycle of the PWM output.

The **digital signal processing** includes: data acquisition (sampling the signal at a fixed rate), data formatting (scaling, calibration), data processing (filtering, curve fitting, FFT, event detection, decision making, analysis), control algorithms (open or closed loop). The **human interface** includes the input and output which is available to the human operator. The advantage of computer-based instrumentation is that, devices that are sophisticated but easy to use and understand are possible. The **inputs** to the instrument can be audio (voice), visual (light pens, cameras), or tactile (keyboards, touch screens, buttons, switches, joysticks, roller balls). The **outputs** from the instrument can be numeric displays, CRT screens, graphs, buzzers, bells, lights, and voice. If the system can deliver energy to the real world then it is classified as a control system.

4.1.2. Real-time Digital Signal Processing[edit]

DSP

Play Video

Digital signal processing (DSP) is beyond the scope of this class. However, we would like to introduce the field of DSP in order to better understand how real-time operating systems will be deployed to process digital signals. In particular, sampling data will be an important task, so we will take special care to sample accurately and with low jitter.

Back in the last chapter we presented FIFO queues as a solution to the data flow problem with producers and consumers. We defined a **producer** as a thread that created data and stored it in the FIFO. If this data is created at a regular rate (e.g., in Lab 3 we sample the ADC at a fixed frequency of 1000 Hz) we can consider the data as a sampled digital signal. We defined a consumer as a thread that removed data from the FIFO and processed it. With DSP we will modify the data structure to allow access to multiple elements, using a multiple-access-circular-queue (MACQ), as shown in Figure 4.3.

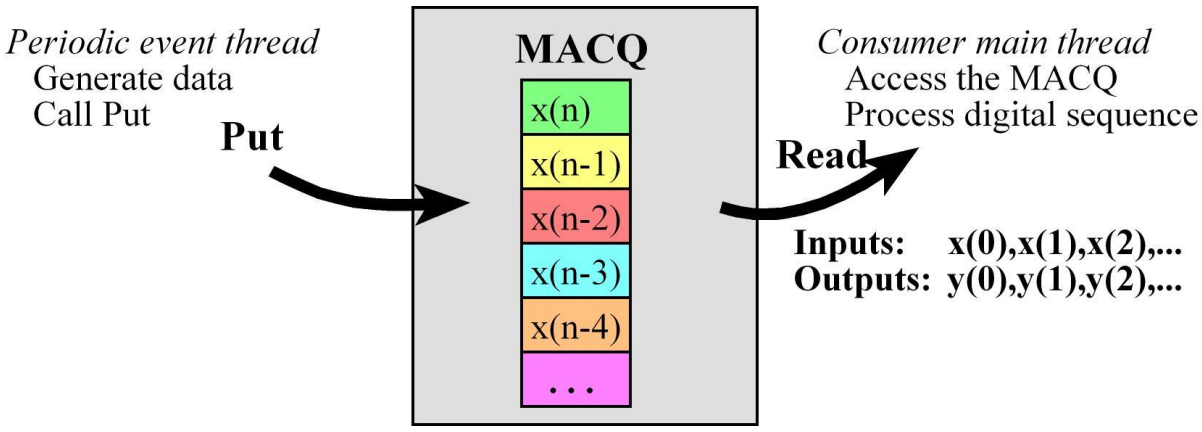

Figure 4.3. Data flow for digital signal processing.

We consider the sampled data as sequence of digital numbers. Let xc(t) be the continuous analog signal to be digitized. $x_c(t)$ is the analog input to the ADC converter. If f_s is the sample rate, then the computer samples the ADC every T seconds. ($T = 1/f_s$). Let $x(0),x(1),\ldots,x(n),\ldots$ be the ADC output sequence, where

$x(n) = x_c(nT)$ with $-\infty < n < +\infty$.

There are two types of approximations associated with the sampling process. Because of the finite precision of the ADC, amplitude errors occur when the continuous signal, $x_c(t)$, is sampled to obtain the digital sequence, $x(n)$. The second type of error occurs because of the finite sampling frequency. The **Nyquist Theorem** states that the digital sequence, $x(n)$, properly represents the DC to $\frac{1}{2}f_s$ frequency components of the original signal, $x(t)$. There are two important assumptions that are necessary to make when using digital signal processing:

1. We assume the signal has been sampled at a fixed and known rate, f_s
2. We assume aliasing has not occurred.

We can guarantee the first assumption by using a hardware clock to start the ADC at a fixed and known rate. A less expensive but not as reliable method is to implement the sampling routine as a high priority periodic interrupt process. If the time jitter is Δt then we can estimate the voltage error by multiplying the time jitter by the slew rate of the input, $dV/dt*\Delta t$. By establishing a high priority of the interrupt handler, we can place an upper bound on the interrupt latency, guaranteeing that ADC sampling is occurring at an almost fixed and known rate. We can observe the ADC input with a spectrum analyzer to prove there are no significant signal components above $\frac{1}{2}f_s$. "No significant signal components" is defined as having an ADC input voltage $|Z|$ less than the ADC resolution, Δz,

$|Z| \le \Delta z$ for all $f \ge \frac{1}{2}f_s$

A causal digital filter calculates $y(n)$ from $y(n-1), y(n-2),\ldots$ and $x(n), x(n-1), x(n\ 2),\ldots$ Simply put, a causal filter cannot have a nonzero output until it is given a nonzero input. The output of a causal filter, $y(n)$, cannot depend on future data (e.g., $y(n+1), x(n+1)$ etc.)

A **linear filter** is constructed from a linear equation. A **nonlinear filter** is constructed from a nonlinear equation. An example of a nonlinear filter is the median. To calculate the **median** of three numbers, one first sorts the numbers according to magnitude, then chooses the middle value. Other simple nonlinear filters include maximum, minimum, and square. A finite impulse response filter (FIR) relates y(n) only in terms of *x(n), x(n-1), x(n 2),...* If the sampling rate is 360 Hz, this simple FIR filter will remove 60 Hz noise:

$$y(n) = (x(n)+x(n-3))/2$$

An **infinite impulse response** filter (IIR) relates *y(n)* in terms of both *x(n), x(n-1),...*, and *y(n 1), y(n-2),...* This simple IIR filter has averaging or low-pass behavior:

$$y(n) = (x(n)+y(n-1))/2$$

The step signal represents a sharp change (like an edge in a photograph). We will analyze three digital filters.

The FIR is $y(n) = (x(n)+x(n-1))/2$.
The IIR is $y(n) = (x(n)+y(n-1))/2$.
The nonlinear filter is $y(n) = median(x(n), x(n-1), x(n-2))$.

The median can be performed on any odd number of data points by sorting the data and selecting the middle value. The median filter can be performed recursively or nonrecursively. A nonrecursive 3-wide median filter is implemented in Program 4.1.

```
uint8_t Median(uint8_t u1,uint8_t u2,uint8_t u3){
uint8_t result;
  if(u1>u2)
    if(u2>u3) result = u2; // u1>u2,u2>u3 u1>u2>u3
    else
    if(u1>u3) result = u3; // u1>u2,u3>u2,u1>u3 u1>u3>u2
    else      result = u1; // u1>u2,u3>u2,u3>u1 u3>u1>u2
  else
    if(u3>u2) result = u2; // u2>u1,u3>u2 u3>u2>u1
    else
    if(u1>u3) result = u1; // u2>u1,u2>u3,u1>u3 u2>u1>u3
    else      result = u3; // u2>u1,u2>u3,u3>u1 u2>u3>u1
  return(result);
}
```

Program 4.1: The median filter is an example of a nonlinear filter.

For a nonrecursive median filter, the original data points are not modified. For example, a 5-wide nonrecursive median filter takes as the filter output the median of {x(n), x(n-1), x(n-2), x(n–3), x(n-4)} On the other hand, a recursive median filter replaces the sample point with the filter output. For example, a 5-wide recursive median filter takes as the filter output the median of

{x(n), y(n-1), y(n-2), y(n-3), y(n-4)} where y(n-1), y(n-2),... are the previous filter outputs. A median filter can be applied in systems that have impulse or speckle noise. For example the noise every once in a while causes one sample to be very different than the rest (like a speck on a piece of paper) then the median filter will completely eliminate the noise. Except for the delay, the median filter passes a step without error. The step responses of the three filters are (Figure 4.4):

FIR ..., 0, 0, 0, 0.5, 1, 1, 1, ...
IIR ..., 0, 0, 0, 0.5, 0.75, 0.88, 0.94, 0.97, 0.98, 0.99, ...
median ..., 0, 0, 0, 0, 1, 1, 1, 1, 1, ...

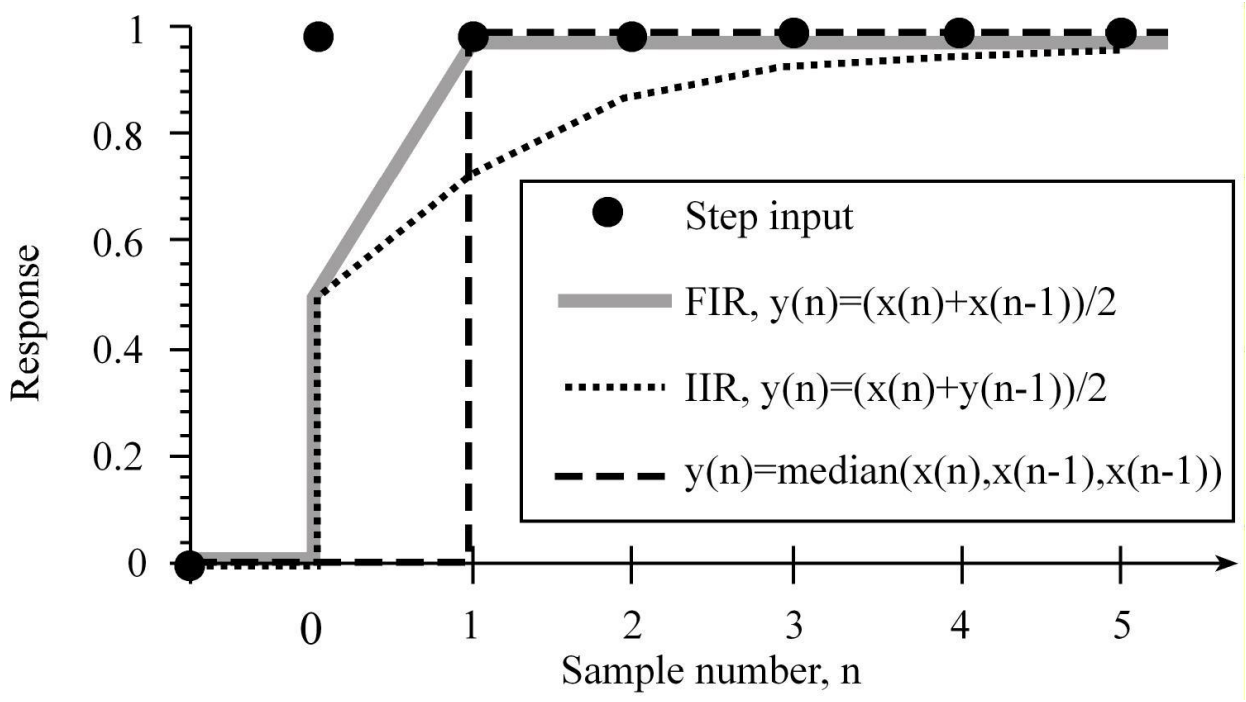

Figure 4.4. Step response of three simple digital filters.

The impulse represents a noise spike (like spots on a Xerox copy). The impulse response of a filter is defined as h(n). The median filter completely removes the impulse. The impulse responses of the three filters are (Figure 4.5):

FIR ..., 0, 0, 0, 0.5, 0.5, 0, 0, 0, ...
IIR ..., 0, 0, 0, 0.5, 0.25, 0.13, 0.06, 0.03, 0.02, 0.01, ...
median ..., 0, 0, 0, 0, 0, 0, 0, 0, 0, ...

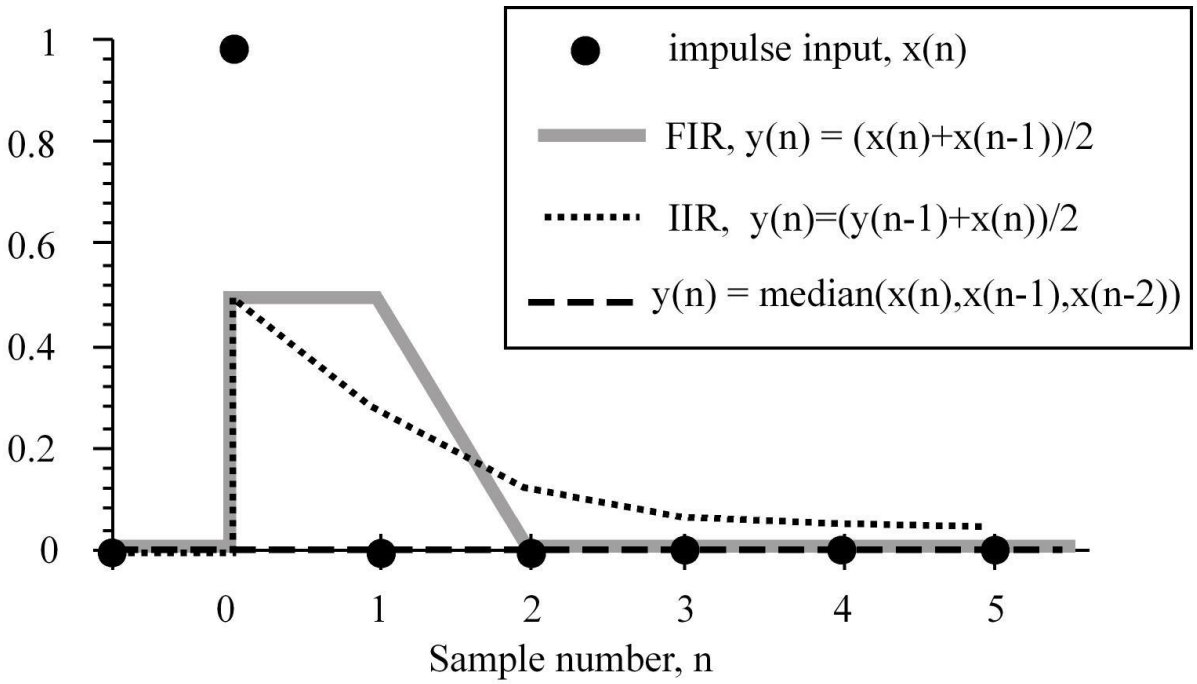

Figure 4.5. Impulse response of three simple digital filters. Note that the median filter preserves the sharp edges and removes the spike or impulsive noise.

Note that the median filter preserves the sharp edges and removes the spike or impulsive noise. Impulse and Step responses are important aspects in control theory as they give us the behavior of the output signal (in time) when presented with a sudden input (impulse) or when the inputs suddenly change from zero to 1 (step) respectively.

4.1.3. Real-time signal processing[edit]

Real-time signal processing

Play Video

In this section we will develop a simple example that samples an analog signal and calculates the derivative of that signal in real time. A transducer is used to convert the measurand into an electrical signal, a circuit is used to convert the input into the voltage range of the ADC, and the ADC is sampled at 1000 Hz. A periodic event task samples the ADC and stores the input into a multiple access circular queue (**MACQ**). A MACQ is a fixed length order preserving data structure, see Figure 4.6. The source process (ADC sampling software) places information into the MACQ. Once initialized, the MACQ is always full. The oldest data is discarded when the newest data is **Put** into a MACQ. The consumer process can read any of the data from the MACQ. The **Read** function is non-destructive. This means that the MACQ is not changed by the Read operation. In this MACQ, the newest sample, *x(n)* is stored in element **x[0],** *x(n-1)* is stored in element **x[1]**, *x(n-2)* is stored in element **x[2]**, and *x(n-3)* is stored in element **x[3]**.

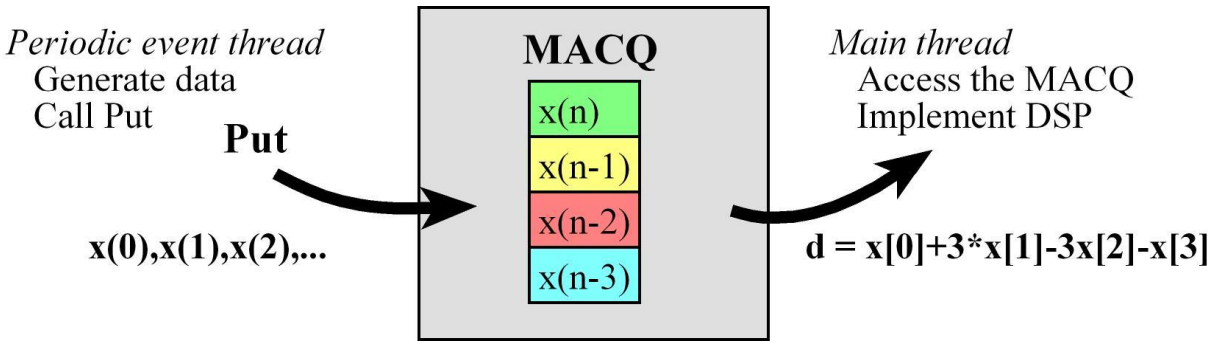

Figure 4.6. Data is sampled at 1 kHz with an event thread. MACQ contains the most recent four samples. A mailbox-like synchronization is used to pass data from producer to consumer.

To Put data into this MACQ, four steps are followed, as shown in Figure 4.7. First, the data is shifted down (steps 1, 2, 3), and then the new data is entered into the x[0] position (step 4).

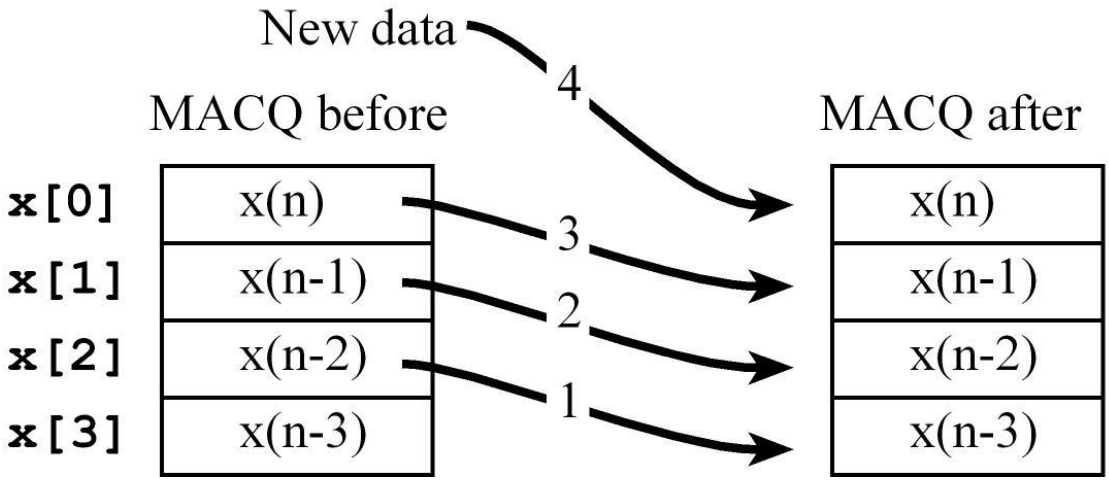

Figure 4.7. When data is put into a multiple access circular queue, the oldest data is lost.

The drawing in Figure 4.7 shows the position in memory of *x(n), x(n-1),…* does not move when one data sample is added. Notice however, the data itself does move. As time passes the data gets older, the data moves down in the MACQ. In this example the ADC sampling is triggered every 1 ms. *x(n)* will refer to the current sample, and *x(n-1)* will be the sample 1 ms ago. There are a couple of ways to implement a discrete time derivative. The simple approach is

$$d(n) = (x(n) - x(n-1))/\Delta t$$

In practice, this first order equation is quite susceptible to noise. An approach generating less noise calculates the derivative using a higher order equation like

$$d(n) = (x(n) + 3x(n-1) - 3x(n-2) - x(n-3))/(6\Delta t)$$

The C implementation of this discrete derivative uses a MACQ (Program 4.2). Since Δt is 1 ms, we simply consider the derivative to have units 6mV/ms and not actually execute the divide by 6Δt operation. Signed arithmetic is used because the slope may be negative.

```
int32_t x[4];         // MACQ (mV)
int32_t d;            // derivative(V/s)
int32_t DataReady;    // semaphore, initially 0
int32_t Derivative(void){ // called every 1 ms
  return x[0]+3*x[1]-3*x[2]-x[3]; // in 6V/s
}
void MACQ_Put(int32_t in){
  x[3] = x[2]; // shift data
  x[2] = x[1]; // units of mV
  x[1] = x[0];
  x[0] = in;
}
void RealTimeSampling(void){ // event thread at 1 kHz, every 1 ms
int32_t sample; // 0 to 1023
int32_t mV;     // -1650 to 1650 mV
  BSP_Microphone_Input(&sample); // sample is 0 to 1023
  mV = 1650*(sample-512)/512;    // in mV
  MACQ_Put(mV);                  // save in MACQ
  OS_Signal(&DataReady);
}
void DigitalSignalProcesing(void){ // main thread
  while(1){
    OS_Wait(&DataReady);
    d = Derivative();
  }
}
```

Program 4.2. Software implementation of first derivative using a multiple access circular queue.

When the MACQ holds many data points it can be implemented using a pointer or index to the newest data. In this way, the data need not be shifted each time a new sample is added. The disadvantage of this approach is that address calculation is required during the Read access. For example, we could implement a 16-element averaging filter. More specifically, we will calculate the average of the last 16 samples, see Program 4.3.

Entering data into this MACQ is a three step process (Figure 4.8). First, the pointer is decremented. If necessary, the pointer is wrapped such that it is always pointing to elements x[0] through **x[15]**. Second, new data is stored into the location of the pointer. Third, a second copy of the new data is stored 16 elements down from the pointer.

Because the pointer is maintained within the first 16 elements, ***Pt** to ***(Pt+15)** will always point to valid data within the MACQ. Let m be an integer from 0 to 15. In this MACQ, the data element x(n-m) can be found using *(Pt+m).

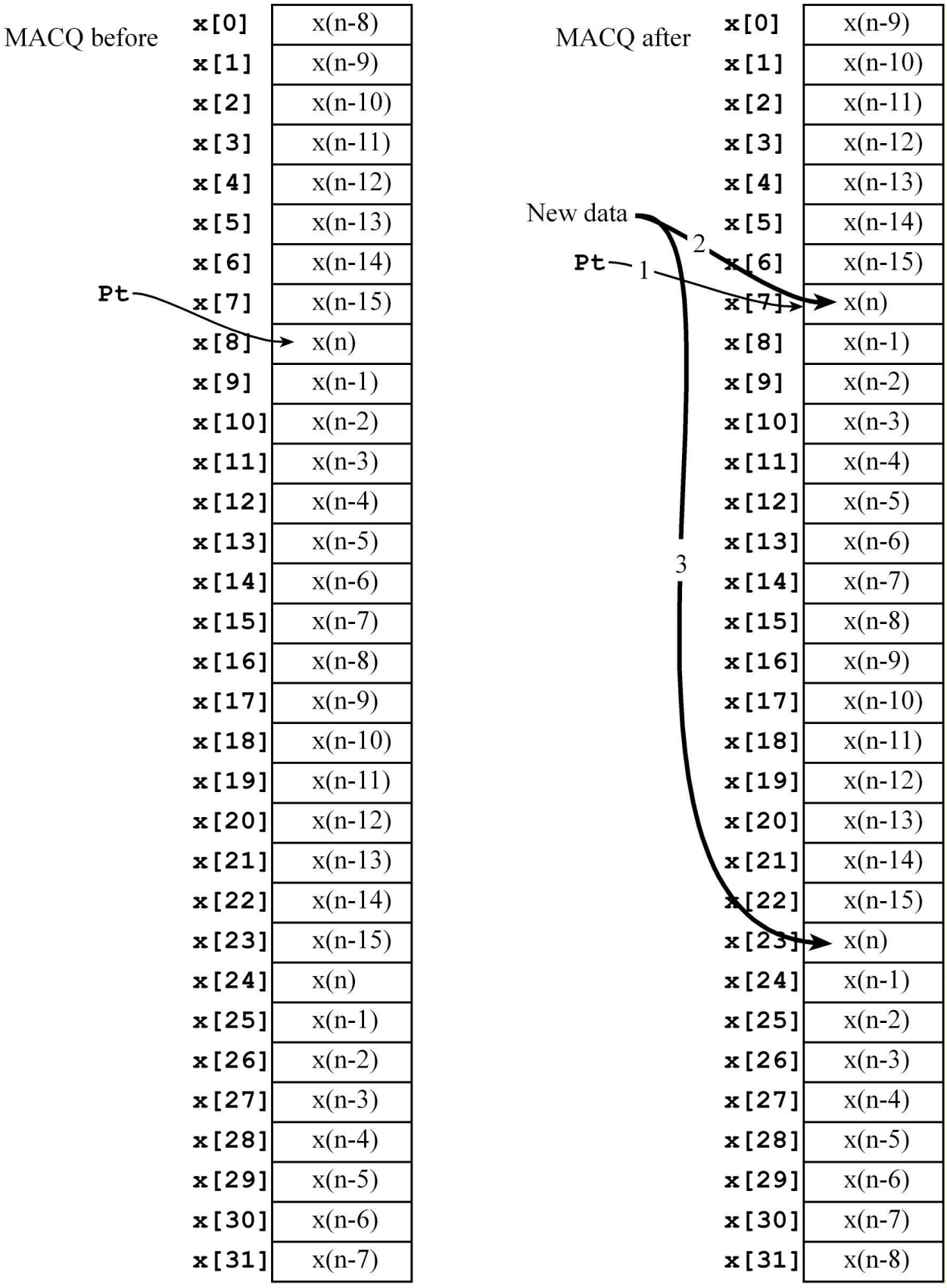
MACQ before
x[0] x(n-8)
x[1] x(n-9)
x[2] x(n-10)
x[3] x(n-11)
x[4] x(n-12)
x[5] x(n-13)
x[6] x(n-14)
Pt
x[7] x(n-15)
x[8] x(n)
x[9] x(n-1)
x[10] x(n-2)
x[11] x(n-3)
x[12] x(n-4)
x[13] x(n-5)
x[14] x(n-6)
x[15] x(n-7)
x[16] x(n-8)
x[17] x(n-9)
x[18] x(n-10)
x[19] x(n-11)
x[20] x(n-12)
x[21] x(n-13)
x[22] x(n-14)
x[23] x(n-15)
x[24] x(n)
x[25] x(n-1)
x[26] x(n-2)
x[27] x(n-3)
x[28] x(n-4)
x[29] x(n-5)
x[30] x(n-6)
x[31] x(n-7)
MACQ after
x[0] x(n-9)
x[1] x(n-10)
x[2] x(n-11)
x[3] x(n-12)
x[4] x(n-13)
New data
x[5] x(n-14)
2
Pt
1
x[6] x(n-15)
x[7] x(n)
x[8] x(n-1)
x[9] x(n-2)
x[10] x(n-3)
x[11] x(n-4)
x[12] x(n-5)
x[13] x(n-6)
3
x[14] x(n-7)
x[15] x(n-8)
x[16] x(n-9)
x[17] x(n-10)
x[18] x(n-11)
x[19] x(n-12)
x[20] x(n-13)
x[21] x(n-14)
x[22] x(n-15)
x[23] x(n)
x[24] x(n-1)
x[25] x(n-2)
x[26] x(n-3)
x[27] x(n-4)
x[28] x(n-5)
x[29] x(n-6)
x[30] x(n-7)
x[31] x(n-8)

Figure 4.8. When data is put into a multiple access circular queue, the oldest data is lost.

The drawing in Figure 4.8 shows the labels *x(n), x(n-1),...* moving from before to after. Notice however, the data itself does not move. What moves is the significance (or meaning) of the data. The data grows older as time passes. The passage of time is produced by decrementing the pointer.

Observation: It is possible to implement a pointer-based MACQ that keeps just one copy of the data. Time to access data would be slower, but half as much storage would be needed.

```
uint16_t x[32]; // two copies
uint16_t *Pt;   // pointer to current
uint16_t Sum;   // sum of the last 16 samples
void LPF_Init(void){
  Pt = &x[0]; Sum = 0;
}
// calculate one filter output
// called at sampling rate
// Input: new ADC data
// Output: filter output, DAC data
uint16_t LPF_Calc(uint16_t newdata){
  Sum = Sum - *(Pt+16); // subtract the one 16 samples ago
  if(Pt == &x[0]){
    Pt = &x[15];        // wrap, always within 0,1,...,15
  } else{
    Pt--;               // make room for data
  }
  *Pt = *(Pt+16) = newdata; // two copies of the new data
  return Sum/16;
}
```

Program 4.3. Digital low pass filter implemented by averaging the previous 16 samples (cutoff = $f_s/32$).

4.1.4. Real-time control systems[edit]

Control Systems

Play Video

A **control system** is a collection of mechanical and electrical devices connected for the purpose of commanding, directing, or regulating a **physical plant** (see Figure 4.9). The **real state variables** are the properties of the physical plant that are to be controlled. The **sensor** and **state estimator** comprise a data acquisition system. The goal of this data acquisition system is to

estimate the state variables. A **closed-loop** control system uses the output of the state estimator in a feedback loop to drive the errors to zero. The control system compares these **estimated state variables**, *X'(t)*, to the **desired state variables**, *X*(t)*, in order to decide appropriate action, *U(t)*. The **actuator** is a transducer that converts the control system commands, *U(t)*, into driving forces, *V(t)*, that are applied to the physical plant. In general, the goal of the control system is to drive the real state variables to equal the desired state variables. In actuality though, the controller attempts to drive the estimated state variables to equal the desired state variables. It is important to have an accurate state estimator, because any differences between the estimated state variables and the real state variables will translate directly into controller errors. If we define the error as the difference between the desired and estimated state variables:

$$e(t) = X^*(t) - X'(t)$$

then the control system will attempt to drive *e(t)* to zero. In general control theory, *X(t), X'(t), X*(t), U(t), V(t)* and *e(t)* refer to vectors, but the examples in this chapter control only a single parameter. Even though this chapter shows one-dimensional systems, and it should be straight-forward to apply standard multivariate control theory to more complex problems. We usually evaluate the effectiveness of a control system by determining three properties: steady state controller error, transient response, and stability. The **steady state controller error** is the average value of *e(t)*. The **transient response** is how long does the system take to reach 99% of the final output after *X** is changed. A system is **stable** if steady state (smooth constant output) is achieved. The error is small and bounded on a stable system. An **unstable** system oscillates, or it may saturate.

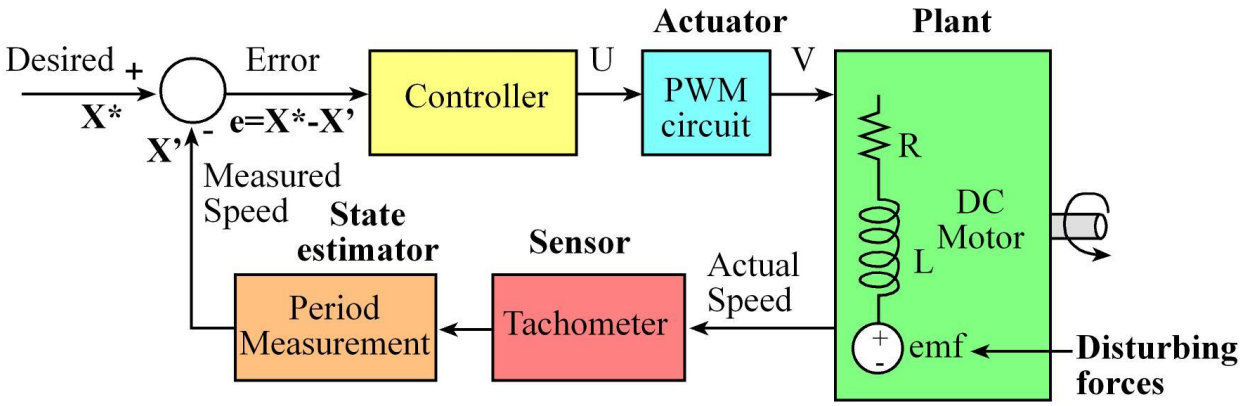

Figure 4.9. Block diagram of a microcomputer-based closed-loop control system.

An **open-loop** control system does not include a state estimator. It is called open loop because there is no feedback path providing information about the state variable to the controller. It will be difficult to use open-loop with the plant that is complex because the disturbing forces will have a significant effect on controller error. On the other hand, if the plant is well-defined and the disturbing forces have little effect, then an open-loop approach may be feasible. Because an

open-loop control system does not know the current values of the state variables, large errors can occur. Stepper motors are often used in open loop fashion.

In order to make a fast and accurate system, we can use linear control theory to develop the digital controller. There are three components of a proportional integral derivative **PID controller**.

The error, *E(t)*, is defined as the present set-point, *X*(t)*, minus the measured value of the controlled variable, *X'(t)*. The PID controller calculates its output by summing three terms. The first term is proportional to the error. The second is proportional to the integral of the error over time, and the third is proportional to the rate of change (first derivative) of the error term. The values of K_p, K_i and K_d are design parameters and must be properly chosen in order for the control system to operate properly. The proportional term of the PID equation contributes an amount to the control output that is directly proportional to the current process error. The gain term K_p adjusts exactly how much the control output response should change in response to a given error level. The larger the value of K_p, the greater the system reaction to differences between the set-point and the actual state variable. However, if K_p is too large, the response may exhibit an undesirable degree of oscillation or even become unstable. On the other hand, if K_p is too small, the system will be slow or unresponsive. An inherent disadvantage of proportional-only control is its inability to eliminate the steady state errors (offsets) that occur after a set-point change or a sustained load disturbance.

The integral term converts the first order proportional controller into a second order system capable of tracking process disturbances. It adds to the controller output a factor that takes corrective action for any changes in the load level of the system. This integral term is scaled to the sum of all previous process errors in the system. As long as there is a process error, the integral term will add more amplitude to the controller output until the sum of all previous errors is zero. Theoretically, as long as the sign of K_i is correct, any value of K_i will eliminate offset errors. But, for extremely small values of Ki, the controlled variables will return to the set-point very slowly after a load upset or set-point change occurs. On the other hand, if K_i is too large, it tends to produce oscillatory response of the controlled process and reduces system stability. The undesirable effects of too much integral action can be avoided by proper tuning (adjusting) the controller or by including derivative action which tends to counteract the destabilizing effects.

The derivative action of a PID controller adds a term to the controller output scaled to the slope (rate of change) of the error term. The derivative term "anticipates" the error, providing a greater control response when the error term is changing in the wrong direction and a dampening

response when the error term is changing in the correct direction. The derivative term tends to improve the dynamic response of the controlled variable by decreasing the process setting time, the time it takes the process to reach steady state. But if the process measurement is noisy, that is, if it contains high-frequency random fluctuations, then the derivative of the measured (controlled) variable will change wildly, and derivative action will amplify the noise unless the measurement is filtered.

We can also use just some of the terms. For example a proportional/integrator (PI) controller drops the derivative term. We will analyze the digital control system in the frequency domain, see Figure 4.10. Let *X(s)* be the Laplace transform of the state variable *x(t)*. Let *X*(s)* be the Laplace transform of the desired state variable *x*(t)*. Let *E(s)* be the Laplace transform of the error

$$E(s) = X^*(s) - X(s)$$

Let *G(s)* be the transfer equation of the PID linear controller. PID controllers are unique in this aspect. In other words we cannot write a transfer equation for a bang-bang, incremental or fuzzy logic controller.

Let *H(s)* be the transfer equation of the physical plant. If we assume the physical plant (e.g., a DC motor) has a simple single pole behavior, then we can specify its response in the frequency domain with two parameters. m is the DC gain and τ is its time constant. The transfer function of this simple motor is

The overall gain of the control system is

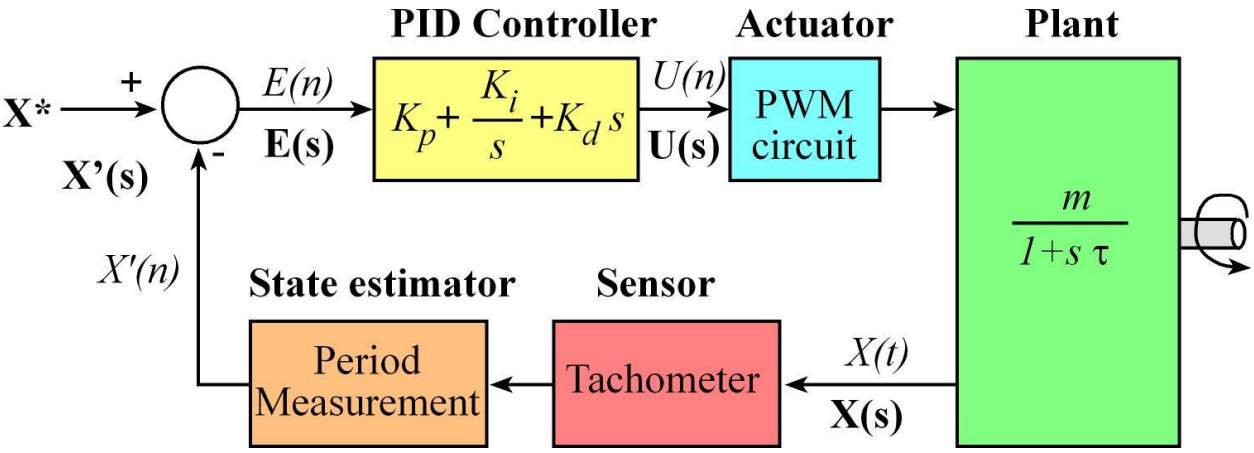

Figure 4.10. Block diagram of a linear control system in the frequency domain.

Theoretically we can choose controller constants, K_p K_i and K_d, to create the desired controller response. Unfortunately it can be difficult to estimate m and τ. If a load is applied to the motor, then m and τ will change.

To simplify the PID controller implementation, we break the controller equation into separate proportion, integral and derivative terms. I.e., let

$U(t) = P(t) + I(t) + D(t)$

where $U(t)$ is the actuator output, and $P(t)$, $I(t)$ and $D(t)$ are the proportional, integral and derivative components respectively. The proportional term makes the actuator output linearly related to the error. Using a proportional term creates a control system that applies more energy to the plant when the error is large. To implement the proportional term we simply convert it to discrete time.

where the index "n" refers to the discrete time input of $E(n)$ and output of $P(n)$.

***Observation**: In order to develop digital signal processing equations, it is imperative that the control system be executed on a regular and periodic rate.*

***Common error**: If the sampling rate varies, then controller errors will occur.*

The integral term makes the actuator output related to the integral of the error. Using an integral term often will improve the steady state error of the control system. If a small error accumulates for a long time, this term can get large. Some control systems put upper and lower bounds on this term, called anti-reset-windup, to prevent it from dominating the other terms.

The implementation of the integral term requires the use of a discrete integral or sum. If *I(n)* is the current control output, and *I(n-1)* is the previous calculation, the integral term is simply

where Δt is the sampling rate of *E(n)*.

The derivative term makes the actuator output related to the derivative of the error. This term is usually combined with either the proportional and/or integral term to improve the transient response of the control system. The proper value of K_d will provide for a quick response to changes in either the set point or loads on the physical plant. An incorrect value may create an overdamped (very slow response) or an underdamped (unstable oscillations) response. There are a couple of ways to implement the discrete time derivative. The simple approach is

In practice, this first order equation is quite susceptible to noise. Figure 4.11 shows a sequence of *E(n)* with some added noise. Notice that huge errors occur when the above equation is used to calculate derivative.

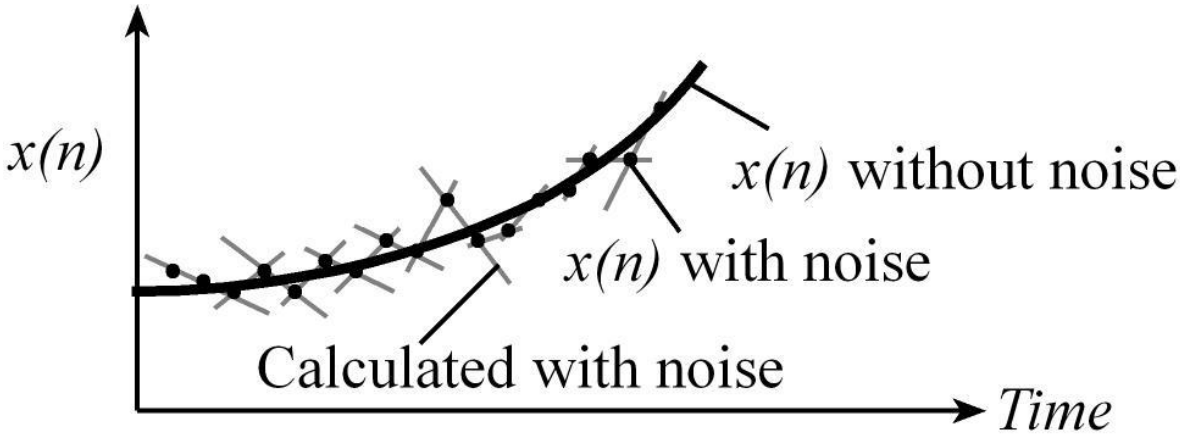

Figure 4.11. Illustration of the effect noise plays on the calculation of discrete derivative.

In most practical control systems, the derivative is calculated using the average of two derivatives calculated across different time spans. For example

that simplifies to

Linear regression through multiple points can yield the slope and yet be immune to noise.

The goal of this example is to spin a DC motor at a constant speed. A tachometer is used to measure the current speed in rotations per minute (**Speed**). The operator selects the desired speed, **Xstar** also in rpm. The motor time constant is defined as the time it takes the motor to reach 63% of the final speed after the delivered power is changed. Typically we run the controller ten times faster than its time constant. For this motor, the time constant is 100 ms, so we run the digital control loop every 10 ms.

```
void PIControlLoop(void){ // event thread
  E = Xstar-Speed;
  P = (5250*E)/1000;   // Kp = 5.250
  I = I+(158*E)/1000;  // KiDt = 0.158
  if(I < -500) I=-500; // anti-reset windup
  if(I > 40000) I=40000;
  U = P+I;             // PI controller
  if(U < 100) U=100;   // Constrain output
  if(U>39900) U=39900; // 100 to 39900
  Actuator(U);         // output
}
```

Program 4.4. Proportional-integral motor controller.

4.2 Edge-triggered Interrupts[edit]

4.2.1. Edge-triggered Interrupts on the TM4C123[edit]

Edge-triggered interrupts

Play Video

Synchronizing software to hardware events requires the software to recognize when the hardware changes states from busy to done. Many times the busy to done state transition is signified by a rising (or falling) edge on a status signal in the hardware. For these situations, we connect this status signal to an input of the microcontroller, and we use edge-triggered interfacing to configure the interface to set a flag on the rising (or falling) edge of the input. Using edge-triggered interfacing allows the software to respond quickly to changes in the external world. If we are using busy-wait synchronization, the software waits for the flag. If we are using interrupt synchronization, we configure the flag to request an interrupt when set. Each

of the digital I/O pins on the TM4C family can be configured for edge triggering. Table 4.1 lists some the registers available for Port A. For more details, refer to the datasheet for your specific microcontroller. Any or all of digital I/O pins can be configured as an edge-triggered input. When writing C code using these registers, include the header file for your particular microcontroller (e.g., tm4c123gh6pm.h).

To use any of the features for a digital I/O port, we first enable its clock in the **SYSCTL_RCGCGPIO_R**. For each bit we wish to use we must set the corresponding **DEN** (Digital Enable) bit. To use a pin as regular digital input or output, we clear its **AFSEL** (Alternate Function Select) bit. Setting the **AFSEL** will activate the pin's special function (e.g., UART, I2C, CAN etc.) For regular digital input/output, we clear **DIR** (Direction) bits to make them input, and we set DIR bits to make them output.

There are four additional registers for the TM4C. We clear bits in the **AMSEL** register to use the port for digital I/O. **AMSEL** bits exist for those pins which have analog functionality. We set the alternative function using both **AFSEL** and **PCTL** registers. We need to unlock PD7 and PF0 if we wish to use them. Because PC3-0 implements the JTAG debugger, we will never unlock these pins. Pins PC3-0, PD7 and PF0 are the only ones that implement the **CR** bits in their commit registers, where 0 means the pin is locked and 1 means the pin is unlocked. To unlock a pin, we first write 0x4C4F434B to the **LOCK** register, and then we write zeros to the **CR** register.

Address	7	6	5	4	3	2	1	0	Name
$4000.43FC	DATA	DATA	DATA	DATA	DATA	DATA	DATA	DATA	GPIO_PORTA_DATA_R
$4000.4400	DIR	DIR	DIR	DIR	DIR	DIR	DIR	DIR	GPIO_PORTA_DIR_R
$4000.4404	IS	IS	IS	IS	IS	IS	IS	IS	GPIO_PORTA_IS_R
$4000.4408	IBE	IBE	IBE	IBE	IBE	IBE	IBE	IBE	GPIO_PORTA_IBE_R
$4000.440C	IEV	IEV	IEV	IEV	IEV	IEV	IEV	IEV	GPIO_PORTA_IEV_R
$4000.4410	IME	IME	IME	IME	IME	IME	IME	IME	GPIO_PORTA_IM_R
$4000.4414	RIS	RIS	RIS	RIS	RIS	RIS	RIS	RIS	GPIO_PORTA_RIS_R
$4000.4418	MIS	MIS	MIS	MIS	MIS	MIS	MIS	MIS	GPIO_PORTA_MIS_R
$4000.441C	ICR	ICR	ICR	ICR	ICR	ICR	ICR	ICR	GPIO_PORTA_ICR_R
$4000.4420	SEL	SEL	SEL	SEL	SEL	SEL	SEL	SEL	GPIO_PORTA_AFSEL_R
$4000.4500	DRV2	DRV2	DRV2	DRV2	DRV2	DRV2	DRV2	DRV2	GPIO_PORTA_DR2R_R
$4000.4504	DRV4	DRV4	DRV4	DRV4	DRV4	DRV4	DRV4	DRV4	GPIO_PORTA_DR4R_R
$4000.4508	DRV8	DRV8	DRV8	DRV8	DRV8	DRV8	DRV8	DRV8	GPIO_PORTA_DR8R_R
$4000.450C	ODE	ODE	ODE	ODE	ODE	ODE	ODE	ODE	GPIO_PORTA_ODR_R
$4000.4510	PUE	PUE	PUE	PUE	PUE	PUE	PUE	PUE	GPIO_PORTA_PUR_R
$4000.4514	PDE	PDE	PDE	PDE	PDE	PDE	PDE	PDE	GPIO_PORTA_PDR_R
$4000.4518	SLR	SLR	SLR	SLR	SLR	SLR	SLR	SLR	GPIO_PORTA_SLR_R
$4000.451C	DEN	DEN	DEN	DEN	DEN	DEN	DEN	DEN	GPIO_PORTA_DEN_R
$4000.4524	CR	CR	CR	CR	CR	CR	CR	CR	GPIO_PORTA_CR_R
$4000.4528	AMSEL	AMSEL	AMSEL	AMSEL	AMSEL	AMSEL	AMSEL	AMSEL	GPIO_PORTA_AMSEL_R
	31-28	27-24	23-20	19-16	15-12	11-8	7-4	3-0	
$4000.452C	PMC7	PMC6	PMC5	PMC4	PMC3	PMC2	PMC1	PMC0	GPIO_PORTA_PCTL_R
$4000.4520	LOCK (write 0x4C4F434B to unlock, other locks) (reads 1 if locked, 0 if unlocked)								GPIO_PORTA_LOCK_R

Table 4.1. Port A registers for the TM4C.

To configure an edge-triggered pin, we first enable the clock on the port and configure the pin as a regular digital input. Clearing the **IS** (Interrupt Sense) bit configures the bit for edge triggering. If the **IS** bit were to be set, the trigger occurs on the level of the pin. Since most busy

to done conditions are signified by edges, we typically trigger on edges rather than levels. Next we write to the **IBE** (Interrupt Both Edges) and **IEV**(Interrupt Event) bits to define the active edge. We can trigger on the rising, falling, or both edges, as listed in Table 4.2. We clear the **IME**(Interrupt Mask Enable) bits if we are using busy-wait synchronization, and we set the **IME** bits to use interrupt synchronization.

DIR	AFSEL	IS	IBE	IEV	IME	Port mode
0	0	0	0	0	0	Input, falling edge trigger, busy wait
0	0	0	0	1	0	Input, rising edge trigger, busy wait
0	0	0	1	-	0	Input, both edges trigger, busy wait
0	0	0	0	0	1	Input, falling edge trigger, interrupt
0	0	0	0	1	1	Input, rising edge trigger, interrupt
0	0	0	1	-	1	Input, both edges trigger, interrupt

Table 4.2. Edge-triggered modes.

The hardware sets an **RIS** (Raw Interrupt Status) bit (called the trigger) and the software clears it (called the acknowledgement). The triggering event listed in Table 4.2 will set the corresponding **RIS** bit in the **GPIO_PORTA_RIS_R** register regardless of whether or not that bit is allowed to request a controller interrupt. In other words, clearing an **IME** bit disables the corresponding pin's interrupt, but it will still set the corresponding **RIS** bit when the interrupt would have occurred. The software can acknowledge the event by writing ones to the corresponding **IC** (Interrupt Clear) bit in the **GPIO_PORTA_IC_R** register. The **RIS** bits are read only, meaning if the software were to write to this registers, it would have no effect. For example, to clear bits 2, 1, and 0 in the **GPIO_PORTA_RIS_R** register, we write a 0x07 to the **GPIO_PORTA_IC_R** register. Writing zeros into **IC** bits will not affect the **RIS** bits.

For input signals we have the option of adding either a pull-up resistor or a pull-down resistor. If we set the corresponding **PUE** (Pull-Up Enable) bit on an input pin, the equivalent of a 50 to 110 kΩ resistor to +3.3 V power is internally connected to the pin. Similarly, if we set the corresponding **PDE** (Pull-Down Enable) bit on an input pin, the equivalent of a 55 to 180 kΩ resistor to ground is internally connected to the pin. We cannot have both pull-up and a pull-down resistor, so setting a bit in one register automatically clears the corresponding bit in the other register.

A typical application of pull-up and pull-down mode is the interface of simple switches. Using these modes eliminates the need for an external resistor when interfacing a switch. The switch interfaces for the two switches on the LaunchPad are illustrated in Figure 4.12. The Port F interfaces employ software-configured internal resistors, implementing negative logic switch inputs.

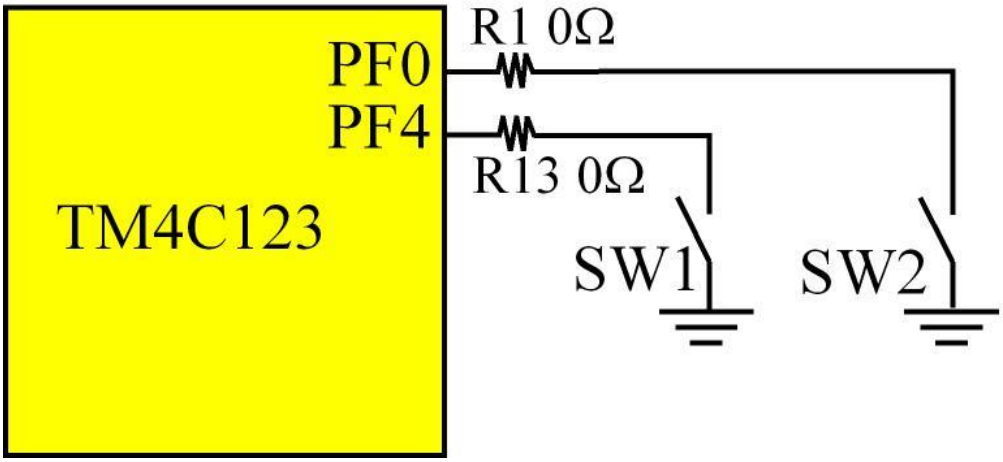

Figure 4.12. Edge-triggered interfaces can generate interrupts on a switch touch. These negative logic switches require internal pullup resistors. R1 and R13 are 0-ohm resistors can could be desoldered to disconnect the switches from the microcontroller.

Using edge triggering to synchronize software to hardware centers around the operation of the trigger flags, **RIS**. A busy-wait interface will read the appropriate **RIS** bit over and over, until it is set. When the **RIS** bit is set, the software will clear the **RIS** bit (by writing a one to the corresponding **IC** bit) and perform the desired function. With interrupt synchronization, the initialization phase will arm the trigger flag by setting the corresponding **IME** bit. In this way, the active edge of the pin will set the **RIS** and request an interrupt. The interrupt will suspend the main program and run a special interrupt service routine (ISR). This ISR will clear the **RIS** bit and perform the desired function. At the end of the ISR it will return, causing the main program to resume. In particular, five conditions must be simultaneously true for an edge-triggered interrupt to be requested:

- The trigger flag bit is set (**RIS**)
- The arm bit is set (**IME**)
- The level of the edge-triggered interrupt must be less than **BASEPRI**
- The edge-triggered interrupt must be enabled in the **NVIC_EN0_R**
- The edge-triggered interrupt must be disabled in the **NVIC_DIS0_R**
- Bit 0 of the special register **PRIMASK** is 0

Table 4.1 listed the registers for Port A. The other ports have similar registers. We will begin with a simple example that counts the number of falling edges on Port F bits 4,0 (Program 4.5). The initialization requires many steps. (a) The clock for the port must be enabled. (b) The global variables should be initialized. (c) The appropriate pins must be enabled as inputs. (d) We must specify whether to trigger on the rise, the fall, or both edges. In this case we will trigger on the fall of PF4,PF0. (e) It is good design to clear the trigger flag during initialization so that the first interrupt occurs due to the first rising edge after the initialization has been run. We do not wish to trigger on a falling edge that might have occurred during the power up phase of the system. (f) We arm the edge-trigger by setting the corresponding bits in the **IM** register. (g) We

establish the priority of Port F by setting bits 23 – 21 in the **NVIC_PRI7_R** register. We activate Port F interrupts in the NVIC by writing a one to bit 30 in the **NVIC_EN0_R** register ("IRQ number"). In most systems we would not enable interrupts in the device initialization. Rather, it is good design to initialize all devices in the system, then enable interrupts.

```
int32_t SW1,SW2 = 0;
void Switch_Init(void){
  SYSCTL_RCGCGPIO_R |= 0x20;  // (a) activate clock for Port F
  SW1 = SW2 = 0;              // (b) initialize counters
  GPIO_PORTF_LOCK_R = 0x4C4F434B; // unlock GPIO Port F
  GPIO_PORTF_CR_R = 0x1F;     // allow changes to PF4-0
  GPIO_PORTF_DIR_R = 0x02;    // (c) make PF4,PF0 in and PF1 is out
  GPIO_PORTF_DEN_R |= 0x13;   // enable digital I/O on PF4,PF0, PF1
  GPIO_PORTF_PUR_R |= 0x11;   // pullups on PF4,PF0
  GPIO_PORTF_IS_R &= ~0x11;   // (d) PF4,PF0 are edge-sensitive
  GPIO_PORTF_IBE_R &= ~0x11;  // PF4,PF0 are not both edges
  GPIO_PORTF_IEV_R &= ~0x11;  // PF4,PF0 falling edge event
  GPIO_PORTF_ICR_R = 0x11;    // (e) clear flags
  GPIO_PORTF_IM_R |= 0x11;    // (f) arm interrupt on PF4,PF0
  NVIC_PRI7_R = (NVIC_PRI7_R&0xFF00FFFF)|0x00A00000; // (g) priority 5
  NVIC_EN0_R = 0x40000000;    // (h) enable interrupt 30 in NVIC
}
void GPIOPortF_Handler(void){
  if(GPIO_PORTF_RIS_R&0x10){  // poll PF4
    GPIO_PORTF_ICR_R = 0x10;  // acknowledge flag4
    OS_Signal(&SW1);          // signal SW1 occurred
  }
  if(GPIO_PORTF_RIS_R&0x01){  // poll PF0
    GPIO_PORTF_ICR_R = 0x01;  // acknowledge flag0
    OS_Signal(&SW2);          // signal SW2 occurred
  }
}
```

Program 4.5. Interrupt-driven edge-triggered input that counts falling edges of PF4,PF0.

4.2.3. Debouncing a switch[edit]

Debugging

Play Video

One of the problems with switches is called switch **bounce**. Many inexpensive switches will mechanically oscillate for up to a few milliseconds when touched or released. It behaves like an underdamped oscillator. These mechanical oscillations cause electrical oscillations such that a port pin will oscillate high/low during the bounce. Contact bounce is a typical problem when interfacing switches. Figure 4.14 shows an actual voltage trace occurring when a negative logic

switch is touched. On both a touch and release, there can be from 0 to 2 ms of extra edges, called switch bounce. However, sometimes there is no bounce.

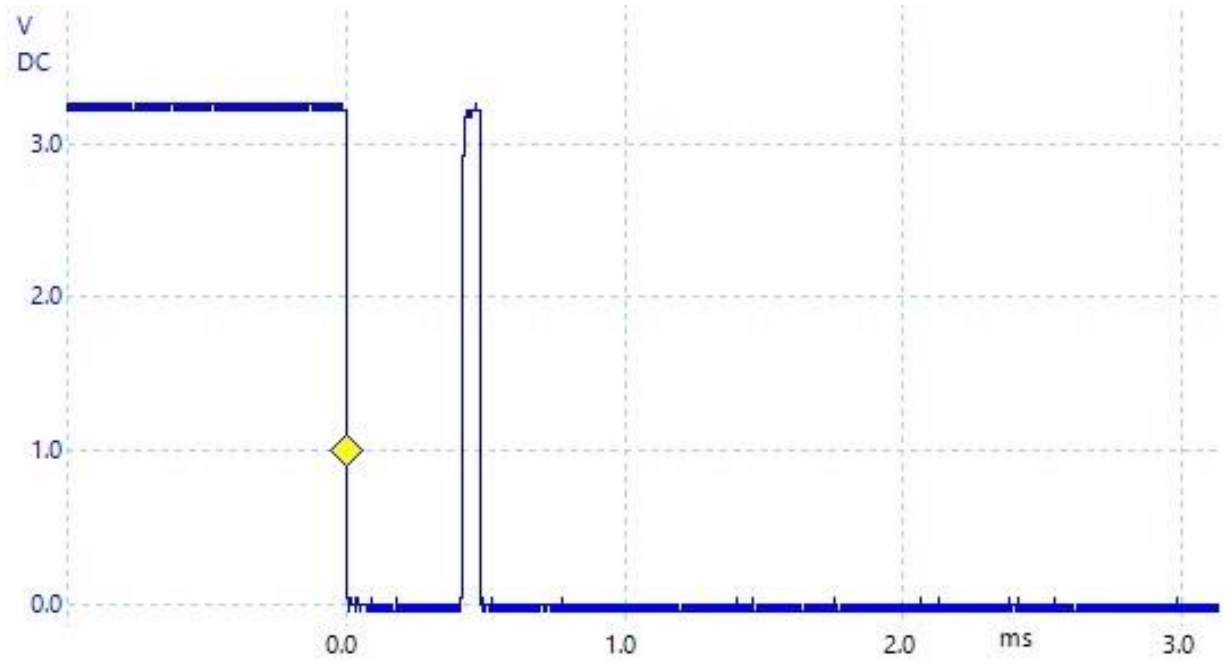

Figure 4.14. Because of the mass and spring some switches bounce.

This bounce is a problem when the system uses the switch to trigger important events. There are two problems to solve: 1) remove the bounce so there is one software event attached to the switch touch; 2) remove the bounce in such a way that there is low latency between the physical touch and the execution of the associated software task.

In some cases, this bounce should be removed. To remove switch bounce we can ignore changes in a switch that occur within 10 ms of each other. In other words, recognize a switch transition, disarm interrupts for 10ms, and then rearm after 10 ms.

Alternatively, we could record the time of the switch transition. If the time between this transition and the previous transition is less than 10ms, ignore it. If the time is more than 10 ms, then accept and process the input as a real event.

Another method for debouncing the switch is to use a periodic interrupt with a period greater than the bounce, but less than the time the switch is held down. Each interrupt we read the switch, if the data is different from the previous interrupt the software recognizes the switch event.

There are three approaches to debouncing a switch in hardware. 1) if you have a double throw switch (3 wires) you can use a set/reset flip flop; 2) you can use a capacitor and a Schmidt trigger; or 3) you can use a capacitor, diode, and Schmidt trigger. Jack Ganssle posted a guide to

debouncing that shows circuits for these three approaches. This is a course on operating systems, so we will show you how to use the OS to debounce the switch.

4.2.4. Debouncing a switch on TM4C123[edit]

Debouncing on the TM4C123

Play Video

If we have a RTOS we can perform a similar sequence. In particular, we will modify Program 4.5 to signal a semaphore. In order to run the user task immediately on touch we will configure it to trigger an interrupt on both edges. However, there can be multiple falling and rising edges on both a touch and a release, see Figure 4.15. A low priority main thread will wait on that semaphore, sleep for 10ms and then read the switch. The interrupt occurs at the start of the bouncing, but the reading of the switch occurs at a time when the switch state is stable. We will disarm the interrupt during the ISR, so the semaphore is incremented once per touch or once per release. We will rearm the interrupt at the stable time. Program 4.7 shows one possible solution that executes **Touch1** when the switch SW1 is touched, and it executes **Touch2** when switch SW2 is touched.

The main thread can be low priority because it needs to run before we release the switch. So if the bounce is 3 ms and the time we hold the switch is at least 50 ms (touching the switch slower than 10 times per second), the main thread needs to finish sleeping within 50 ms.

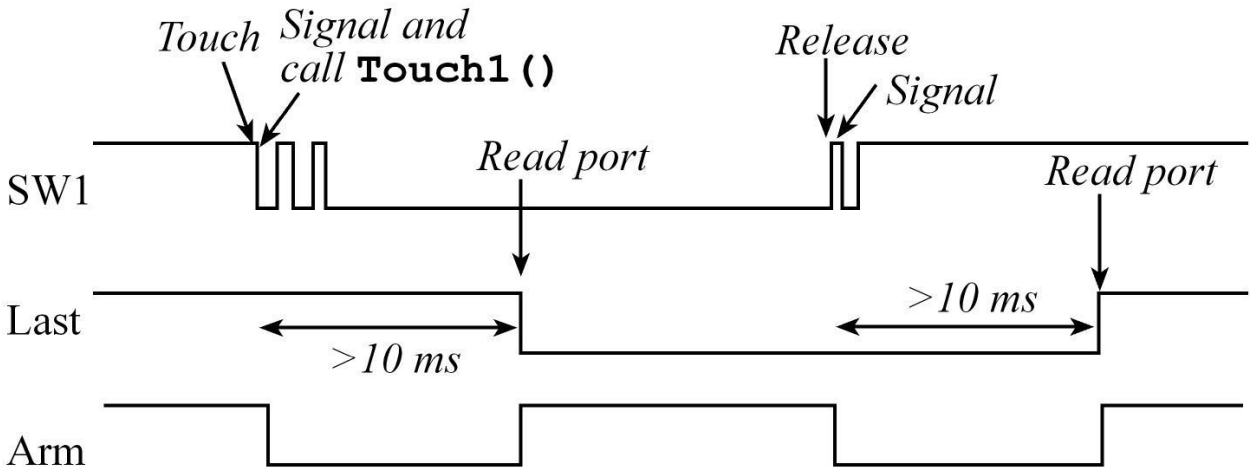

Figure 4.15. Touch and release both cause the ISR to run. The port is read during the stable time

```
int32_t SW1,SW2;
uint8_t last1,last2;
void Switch_Init(void){
  SYSCTL_RCGCGPIO_R |= 0x20; // activate clock for Port F
```

```
  OS_InitSemaphore(&SW1,0);  // initialize semaphores
  OS_InitSemaphore(&SW2,0);
  GPIO_PORTF_LOCK_R = 0x4C4F434B; // unlock GPIO Port F
  GPIO_PORTF_CR_R = 0x1F;         // allow changes to PF4-0
  GPIO_PORTF_DIR_R &= ~0x11; // make PF4,PF0 in
  GPIO_PORTF_DEN_R |= 0x11;  // enable digital I/O on PF4,PF0
  GPIO_PORTF_PUR_R |= 0x11;  // pullup on PF4,PF0
  GPIO_PORTF_IS_R &= ~0x11;  // PF4,PF0 are edge-sensitive
  GPIO_PORTF_IBE_R |= 0x11;  // PF4,PF0 are both edges
  GPIO_PORTF_ICR_R = 0x11;   // clear flags
  GPIO_PORTF_IM_R |= 0x11;   // arm interrupts on PF4,PF0
  NVIC_PRI7_R = (NVIC_PRI7_R&0xFF00FFFF)|0x00A00000; // priority 5
  NVIC_EN0_R = 0x40000000;   // enable interrupt 30 in NVIC
}
void GPIOPortF_Handler(void){
  if(GPIO_PORTF_RIS_R&0x10){  // poll PF4
    GPIO_PORTF_ICR_R = 0x10;  // acknowledge flag4
    OS_Signal(&SW1);          // signal SW1 occurred
    GPIO_PORTF_IM_R &= ~0x10; // disarm interrupt on PF4
  }
  if(GPIO_PORTF_RIS_R&0x01){  // poll PF0
    GPIO_PORTF_ICR_R = 0x01;  // acknowledge flag0
    OS_Signal(&SW2);          // signal SW2 occurred
    GPIO_PORTF_IM_R &= ~0x81; // disarm interrupt on PF0
  }
  OS_Suspend();
}
void Switch1Task(void){  // high priority main thread
  last1 = GPIO_PORTF_DATA_R&0x10;
  while(1){
    OS_Wait(&SW1);        // wait for SW1 to be touched/released
    if(last1){            // was previously not touched
      Touch1();           // user software associated with touch
    }else{
      Release1();         // user software associated with release
    }
    OS_Sleep(10);         // wait for bouncing to be over
    last1 = GPIO_PORTF_DATA_R&0x10;
    GPIO_PORTF_IM_R |= 0x10; // rearm interrupt on PF4
    GPIO_PORTF_ICR_R = 0x10; // acknowledge flag4
  }
}
void Switch2Task(void){  // high priority main thread
  last2 = GPIO_PORTF_DATA_R&0x01;
  while(1){
    OS_Wait(&SW2);       // wait for SW2 to be touched/released
    if(last2){           // was previously not touched
      Touch2();          // user software associated with touch
    }else{
```

```
      Release2();       // user software associated with release
    }
    OS_Sleep(10);       // wait for bouncing to be over
    last2 = GPIO_PORTF_DATA_R&0x01;
    GPIO_PORTF_IM_R |= 0x01; // rearm interrupt on PF0
    GPIO_PORTF_ICR_R = 0x01; // acknowledge flag0
  }
}
```

Program 4.7. Interrupt-driven edge-triggered input that calls Touch1() on the falling edge of PF4, calls Release1() on the rising edge of PF4, calls Touch2() on the falling edge of PF0 and calls Release2() on the rising edge of PF0.

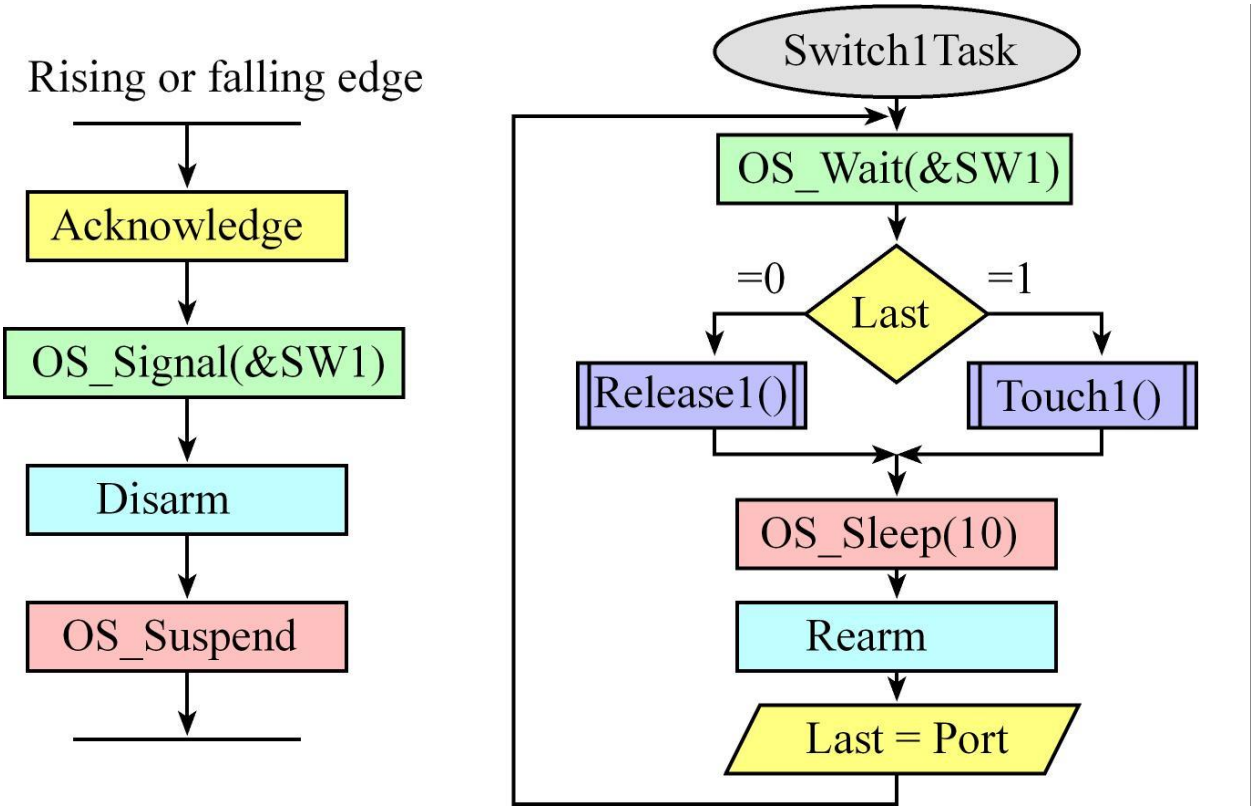

Figure 4.16. Flowchart of a RTOS-solution to switch bounce. Switch1Task is a high-priority main thread. Notice that Release1 is executed immediately after a release, and Touch1 is executed immediate after the switch is touched. However the global variable Last is set at a time the switch is guaranteed to be stable.

4.3. Priority Scheduler[edit]

4.3.1. Implementation[edit]

Priority Scheduler

Play Video

To implement priority, we add another field to the TCB. In this system we define 0 as the highest priority and 254 as the lowest. In some operating systems, each thread must have unique priority, but in Lab 4 multiple threads can have the same priority. If we have multiple threads with equal priority, these threads will be run in a round robin fashion.

```
struct tcb{
  int32_t *sp;       // pointer to stack (valid for threads not running
  struct tcb *next;  // linked-list pointer
  int32_t *BlockPt;  // nonzero if blocked on this semaphore
  uint32_t Sleep;    // nonzero if this thread is sleeping
  uint8_t Priority;  // 0 is highest, 254 lowest
};
```

Program 4.9. TCB for the priority scheduler.

The strategy will be to find the highest priority thread, which is neither blocked nor sleeping and run it as shown in Figure 4.18. If there are multiple threads at that highest priority that are not sleeping nor blocked, then the scheduler will run them in a round robin fashion. The statement, **pt = pt->next** guarantees that the same higher priority task is not picked again.

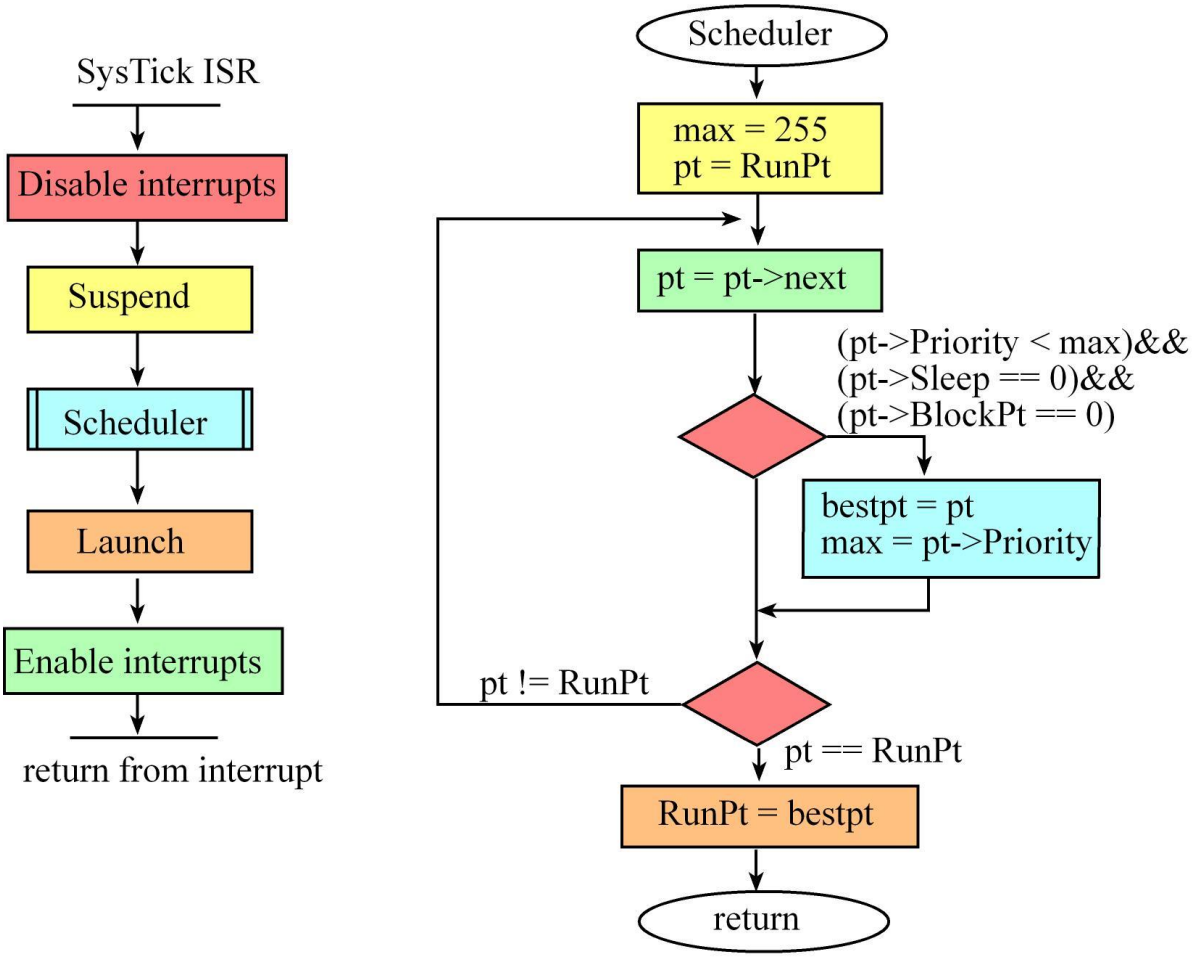

Figure 4.18. Priority scheduler finds the highest priority thread.

```
void Scheduler(void){ // every time slice
  uint32_t max = 255; // max
  tcbType *pt;
  tcbType *bestPt;
  pt = RunPt;         // search for highest thread not blocked or sleeping
  do{
    pt = pt->next;    // skips at least one
    if((pt->Priority < max)&&((pt->BlockPt)==0)&&((pt->Sleep)==0)){
      max = pt->Priority;
      bestPt = pt;
    }
  } while(RunPt != pt); // look at all possible threads
  RunPt = bestPt;
}
```

Program 4.10. One possible priority scheduler.

4.3.2. Multi-level Feedback Queue[edit]

MLFQ

Play Video

The priority scheduler in the previous section will be inefficient if there are a lot of threads. Because the scheduler must look at all threads, the time to run the scheduler grows linearly with the number of threads. One implementation that is appropriate for priority systems with many threads is called the multi-level feedback queue (MLFQ). MLFQ was introduced in 1962 by Corbato et al. and has since been adopted in some form by all the major operating systems, BSD Unix and variants, Solaris and Windows. Its popularity stems from its ability to optimize performance with respect to two metrics commonly used in traditional Operating Systems. These metrics are turnaround time, and response time. Turnaround time is the time elapsed from when a thread arrives till it completes execution. Response time is the time elapsed from when a thread arrives till it starts execution. Preemptive scheduling mechanisms like **Shortest Time-to-Completion First** (STCF) and **Round-Robin** (RR) are optimal at minimizing the average turnaround time and response time respectively. However, both perform well on only one of these metrics and show very poor performance with respect to the other. MLFQ fairs equally well on both these metrics. As the name indicates, MLFQ has multiple queues, one per priority level, with multiple threads operating at the same priority level. In keeping with our description of priority, we assume level 0 is the highest priority and higher levels imply lower priority. There will be a finite number of priority levels from 0 to *n*-1, see Figure 4.19. The rules that govern the processing of these queues by the scheduler are as follows:

1. Startup: All threads start at the highest priority. Start in queue at level 0.
2. Highest runs: If Priority(T_i) < Priority(T_j) then T_i is scheduled to run before T_j.
3. Equals take turns: If Priority(T_i) = Priority(T_j) then T_i and T_j are run in RR order.
4. True accounting: If a thread uses up its timeslice at priority m then its priority is reduced to m+1. It is moved to the corresponding queue.
5. Priority Boost: The scheduler does a periodic reset, where all threads are bumped to the highest priority.

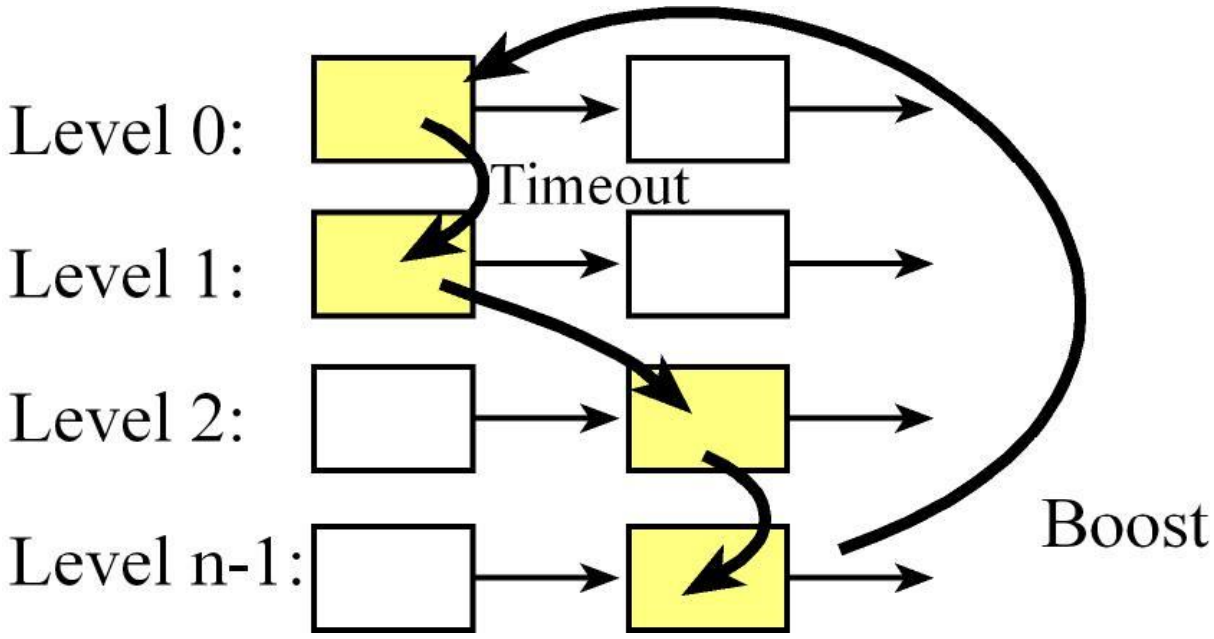

Figure 4.19. The shaded task in this figure begins in the level 0 (highest) priority queue. If it runs to the end of its 10-ms time slice (timeout), it is bumped to level 1. If it again runs to the end of its 10-ms time slice, it is bumped to level 2. Eventually, a thread that does not sleep or block will end up in the lower priority queue. Periodically the system will reset and place all threads back at level 0.

An obvious precondition to choosing a thread is to make sure it is "ready", that is, it is not blocked on a resource or sleeping. This rule is implicit and hence not listed here. Rules 2, and 3 are self-explanatory as MLFQ attempts to schedule the highest priority ready thread at any time. Rule 1 makes sure that every thread gets a shot at executing as quickly as possible, the first time it enters the system. Rule 4 is what determines when a thread is moved from one level to another. Further, whether a thread uses up its timeslice at one shot or over multiple runs, true accounting requires that the accumulated time for the thread at a given priority level be considered. There are versions of MLFQ that let a thread remain at a priority level with its accrued time towards the timeslice reset to zero, if it blocked on a resource. These versions allowed the possibility of gaming the scheduler. Without rule 5, MLFQ eventually reduces to RR after running for a while with all threads operating at the lowest priority level. By periodically boosting all threads to the highest priority, rule 5 causes a scheduler reset that lets the scheduler adapt to changes in thread behavior.

4.3.3. Starvation and aging[edit]

Starvation and aging

One disadvantage of a priority scheduler on a busy system is that low priority threads may never be run. This situation is called **starvation**. For example, if a high priority thread never sleeps or blocks, then the lower priority threads will never run. It is the responsibility of the user to assign priorities to tasks. As mentioned earlier, as processor utilization approaches one, there will not be a solution. In general, starvation is not a problem of the RTOS but rather a result of a poorly designed user code.

One solution to starvation is called **aging**. In this scheme, threads have a permanent fixed priority and a temporary working priority. The permanent priority is assigned according to the rules of the previous paragraph, but the temporary priority is used to actually schedule threads. Periodically the OS increases the temporary priority of threads that have not been run in a long time. For example, the **Age** field is incremented once every 1ms if the thread is not blocked or not sleeping. For every 10 ms the thread has not been run, its **WorkingPriority** is reduced. Once a thread is run, its temporary priority is reset back to its permanent priority. When the thread is run, the Age field is cleared and the **FixedPriority**is copied into the **WorkingPriority**.

```
struct tcb{
  int32_t *sp;              // pointer to stack (valid for threads not running
  struct tcb *next;         // linked-list pointer
  int32_t *BlockPt;         // nonzero if blocked on this semaphore
  uint32_t Sleep;           // nonzero if this thread is sleeping
  uint8_t WorkingPriority; // used by the scheduler
  uint8_t FixedPriority;    // permanent priority
  uint32_t Age;             // time since last execution
};
```

Program 4.11. TCB for the priority scheduler.

4.3.4. Priority inversion and inheritance[edit]

Real example

Play Video

Another problem with a priority scheduler is priority inversion, a condition where a high-priority thread is waiting on a resource owned by a low-priority thread. For example, consider the case where both a high priority and low priority thread need the same resource. Assume the low-priority thread asks for and is granted the resource, and then the high-priority thread asks for it and blocks. During the time the low priority thread is using the resource, the high-priority thread essentially becomes low priority. The scenario in Figure 4.20 begins with a low priority meteorological task asking for and being granted access to a shared memory using the **mutex** semaphore. The second step is a medium priority communication task runs for a long time. Since communication is higher priority than the meteorological task, the communication task runs but the meteorological task does not run. Third, a very high priority

task starts but also needs access to the shared memory, so it calls wait on **mutex**. This high priority task, however, will block because **mutex** is 0. Notice that while the communication task is running, this high priority task effectively runs at low priority because it is blocked on a semaphore captured previously by the low priority task.

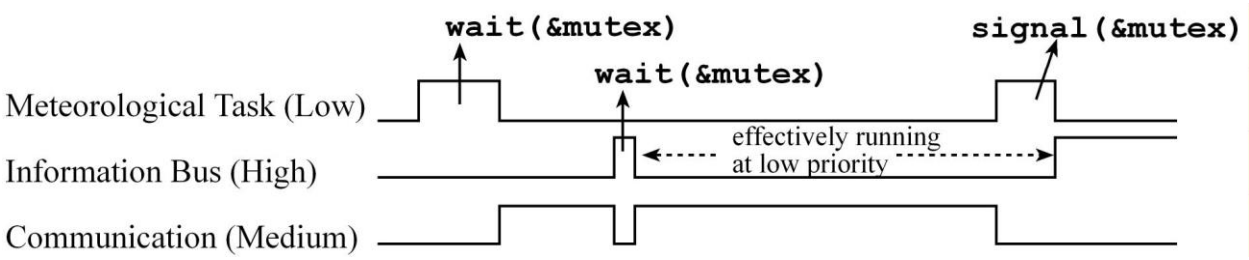

Figure 4.20. Priority inversion as occurred with Mars Pathfinder.

http://research.microsoft.com/en-us/um/people/mbj/Mars_Pathfinder/Mars_Pathfinder.html

One solution to priority inversion is **priority inheritance**. With priority inheritance, once a high-priority thread blocks on a resource, the thread holding that resource is granted a temporary priority equal to the priority of the high-priority blocked thread. Once the thread releases the resource, its priority is returned to its original value.

A second approach is called **priority ceiling**. In this protocol each semaphore is assigned a priority ceiling, which is a priority equal to the highest priority of any task which may block on a semaphore for that resource. With priority ceiling, once a high-priority thread blocks on a resource, the thread holding that resource is granted a temporary priority equal to the priority of the priority ceiling. Just like inheritance, once the thread releases the resource, its priority is returned to its original value.

Note: none of the labs in this class will require you to implement aging or priority inheritance. We introduce the concepts of priority inheritance because it is a feature available in most commercial priority schedulers.

4.4. Running Event Threads[edit]

High priority threads[edit]

High priority main thread

Play Video

In Labs 2 and 3, we ran time-critical tasks (event tasks) directly from the interrupt service routine. Now that we have a priority scheduler, we can place time-critical tasks as high priority main

threads. We will block these time-critical tasks waiting on an event (semaphore), and when the event occurs we signal its semaphore. Because we now have a high priority thread not blocked, the scheduler will run it immediately. In Program 4.12, we have a periodic interrupt that simply signals a semaphore and invokes the scheduler. If we assign the program Task0 as a high priority main thread, it will be run periodically with very little jitter.

It may seem like a lot of trouble to run a periodic task. One might ask why not just put the time-critical task in the interrupt service routine. A priority scheduler is flexible in two ways. First, because it implements priority we can have layers of important, very important and very very important tasks. Second, we can use this approach for any triggering event, hardware or software. We simply make that triggering event call OS_Signal and **OS_Suspend**. One of the advantages of this approach is the separation of the user/application code from the OS code. The OS simply signals the semaphore on the appropriate event and the user code runs as a main thread.

```
int32_t TakeSoundData; // binary semaphore
void RealTimeEvents(void){
  OS_Signal(&TakeSoundData);
  OS_Suspend();
}
void Task0(void){
  while(1){
    OS_Wait(&TakeSoundData); // signaled every 1ms
    TExaS_Task0();           // toggle virtual logic analyzer
    Profile_Toggle0();       // viewed by the logic analyzer to know Task0 started
// time-critical software
  }
}
int main(void){
  OS_Init();
// other initialization
  OS_InitSemaphore(&TakeSoundData,0);
  OS_AddThreads(&Task0,0,&Task1,1,&Task2,2, &Task3,3,
      &Task4,3, &Task5,3, &Task6,3, &Task7,4);
  BSP_PeriodicTask_InitC(&RealTimeEvents,1000,0);
  TExaS_Init(LOGICANALYZER, 1000);          // initialize the logic analyzer
  OS_Launch(BSP_Clock_GetFreq()/THREADFREQ); // doesn't return
  return 0; // this never executes
}
```

Program 4.12. Running time-critical tasks as high priority event threads.

4.5. Available RTOS[edit]

4.5.1. Micrium uC/OS-II[edit]

We introduced several concepts that common in real-time operating systems but ones we don't implement in our simple RTOS. To complete this discussion, we explore some of the popular RTOSs (for the ARM Cortex-M) in commercial use and how they implement some of the features we covered.

Micrium µC/OS-II is a portable, ROMable, scalable, preemptive, real-time deterministic multitasking kernel for microprocessors, microcontrollers and DSPs (for more information, see http://micrium.com/rtos/ucosii/overview/). Portable means user and OS code written on one processor can be easily shifted to another processor. ROMable is a standard feature of most compilers for embedded systems, meaning object code is programmed into ROM, and variables are positioned in RAM. Scalable means applications can be developed on this OS for 10 threads, but the OS allows expansion to 255 threads. Like most real-time operating systems, high priority tasks can preempt lower priority tasks. Because each thread in Micrium µC/OS-II has a unique priority (no two threads have equal priority), the threads will run in a deterministic pattern, making it easy to certify performance. In fact, the following lists the certifications available for Micrium µC/OS-II

- MISRA-C:1998
- DO178B Level A and EUROCAE ED-12B
- Medical FDA pre-market notification (510(k)) and pre-market approval (PMA)
- SIL3/SIL4 IEC for transportation and nuclear systems
- IEC-61508

As of September 2014, Micrium µC/OS-II is available for 51 processor architectures, including the Cortex M3 and Cortex M4F. Ports are available for download on http://micrium.com. Micrium µC/OS-II manages up to 255 application tasks. µC/OS-II includes: semaphores; event flags; mutual-exclusion semaphores that eliminate unbounded priority inversions; message mailboxes and queues; task, time and timer management; and fixed sized memory block management.

Micrium µC/OS-II's footprint can be scaled (between 5 kibibytes to 24 kibibytes) to only contain the features required for a specific application. The execution time for most services provided by µC/OS-II is both constant and deterministic; execution times do not depend on the number of tasks running in the application. To provide for stability and protection, this OS runs user code with the PSP and OS code with the MSP. The way in which the Micrium µC/OS supports many processor architectures is to be layered. Only a small piece of the OS code is processor specific. It also provides a Board Support Package (BSP) so the user code can also be layered, see Figure 4.21.

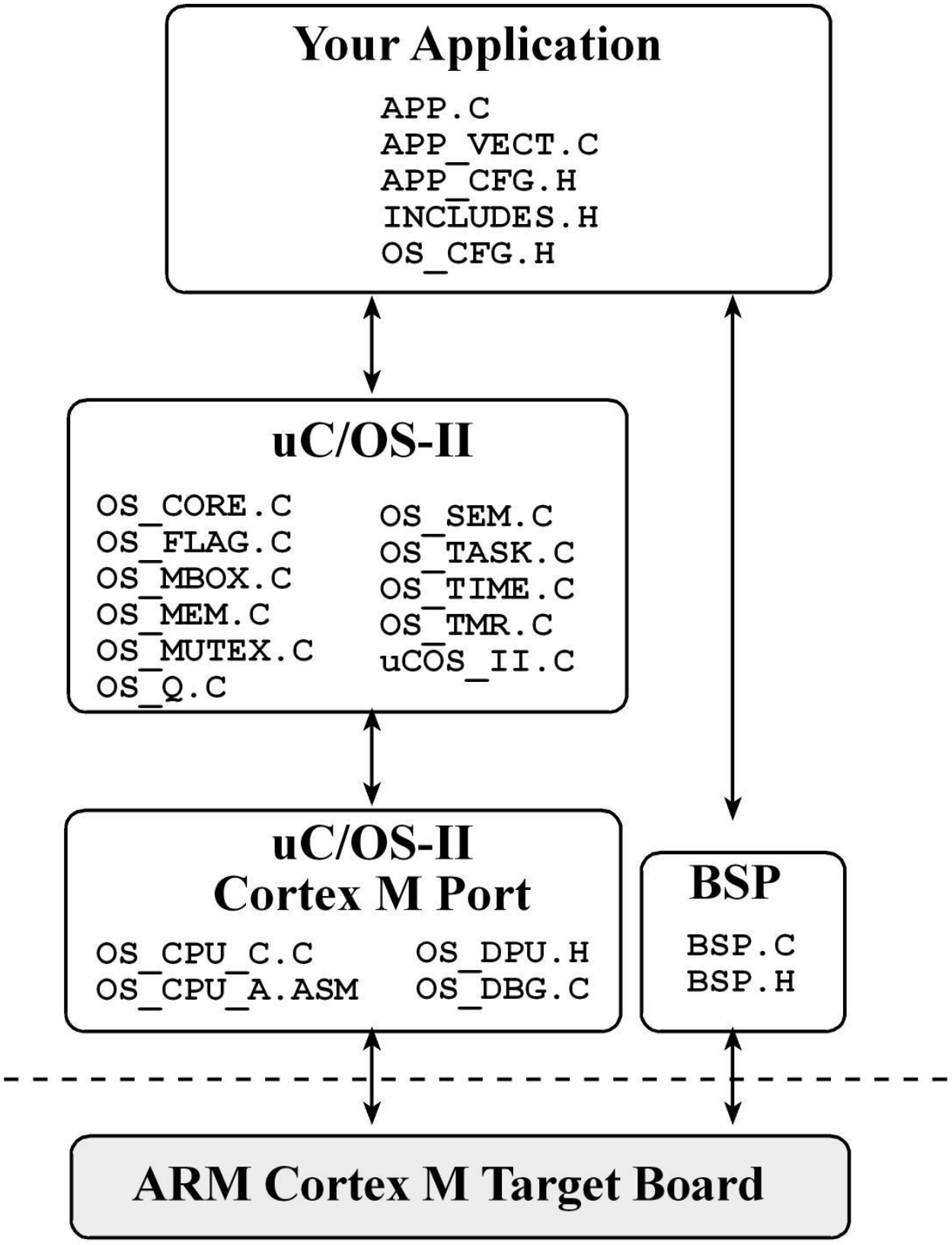

Figure 4.21. Block diagram of the Micrium uC/OSII.

To illustrate the operation of Micrium µC/OS-II, Program 4.25 shows the thread-switch code. PendSV is an effective method for performing context switches with Cortex-M because the Cortex-M saves R0-R3,R12,LR,PC,PSW on any exception, and restores the same on return from exception. So only saving of R4-R11 is required and fixing up the stack pointers. Using the PendSV exception this way means that context saving and restoring is identical whether it is initiated from a thread or occurs due to an interrupt or exception. On entry into PendSV handler 1) xPSR, PC, LR, R12, R0-R3 have been saved on the process stack (by the processor); 2) Processor mode is switched to Handler mode (from Thread mode); 3) The stack is now the Main stack (switched from Process stack); 3) OSTCBCur points to the OS_TCB of the task to suspend; and 4) OSTCBHighRdy points to the OS_TCB of the task to resume. There nine steps for switching a thread:

1. Get the process SP, if 0 then go to step 4. the saving part (first switch);
2. Save remaining regs R4-R11 on process stack;
3. Save the process SP in its TCB, OSTCBCur->OSTCBStkPtr = SP;
4. Call OSTaskSwHook();
5. Get current high priority, OSPrioCur = OSPrioHighRdy;
6. Get current ready thread TCB, OSTCBCur = OSTCBHighRdy;
7. Get new process SP from TCB, SP = OSTCBHighRdy->OSTCBStkPtr;
8. Restore R4-R11 from new process stack;
9. Perform exception return which will restore remaining context.

```
OS_CPU_PendSVHandler
    CPSID   I                   ; Prevent interruption during context switch
    MRS     R0, PSP             ; PSP is process stack pointer
    CBZ     R0, OS_CPU_PendSVHandler_nosave       ; Skip first time

    SUBS    R0, R0, #0x20       ; Save remaining regs R4-11 on process stack
    STM     R0, {R4-R11}

    LDR     R1, =OSTCBCur       ; OSTCBCur->OSTCBStkPtr = SP;
    LDR     R1, [R1]
    STR     R0, [R1]            ; R0 is SP of process being switched out

; At this point, entire context of process has been saved
OS_CPU_PendSVHandler_nosave
    PUSH    {R14}               ; Save LR exc_return value
    LDR     R0, =OSTaskSwHook ; OSTaskSwHook();
    BLX     R0
    POP     {R14}

    LDR     R0, =OSPrioCur   ; OSPrioCur = OSPrioHighRdy;
    LDR     R1, =OSPrioHighRdy
    LDRB    R2, [R1]
    STRB    R2, [R0]

    LDR     R0, =OSTCBCur    ; OSTCBCur  = OSTCBHighRdy;
```

```
    LDR     R1, =OSTCBHighRdy
    LDR     R2, [R1]
    STR     R2, [R0]

    LDR     R0, [R2]  ; R0 is new PSP; SP = OSTCBHighRdy->OSTCBStkPtr;
    LDM     R0, {R4-R11}    ; Restore R4-11 from new process stack
    ADDS    R0, R0, #0x20
    MSR     PSP, R0         ; Load PSP with new process SP
    ORR     LR, LR, #0x04   ; Ensure exception return uses process stack
    CPSIE   I
    BX      LR          ; Exception return will restore remaining context
```

Program 4.25. Thread switch code on the Micrium uC/OSII (The thread switching software in this course were derived from this OS).

Since PendSV is set to lowest priority in the system, we know that it will only be run when no other exception or interrupt is active, and therefore safe to assume that context being switched out was using the process stack (PSP). Micrium µC/OS-II provides numerous hooks within the OS to support debugging, profiling, and feature expansion. An example of a hook is the call to OSTaskSwHook(). The user can specify the action invoked by this call. Micrium µC/OS-III extends this OS with many features as more threads, round-robin scheduling, enhanced messaging, extensive performance measurements, and time stamps.

4.5.2. Texas Instruments RTOS[edit]

TI-RTOS scales from a real-time multitasking kernel to a complete RTOS solution including additional middleware components and device drivers. TI-RTOS is provided with full source code and requires no up-front or runtime license fees. TI-RTOS Kernel is available on most TI microprocessors, microcontrollers and DSPs. TI-RTOS middleware, drivers and board initialization components are available on select ARM® Cortex™-M4 Tiva-C, C2000™ dual core C28x + ARM Cortex-M3, MSP430 microcontrollers, and the SimpleLink™ WiFi® CC3200. For more information, see http://www.ti.com/tool/ti-rtos or search RTOS on www.ti.com. TI-RTOS combines a real-time multitasking kernel with additional middleware components including TCP/IP and USB stacks, a FAT file system, and device drivers, see Figure 4.22 and Table 4.5. TI-RTOS provides a consistent embedded software platform across TI's microcontroller devices, making it easy to port legacy applications to the latest devices.

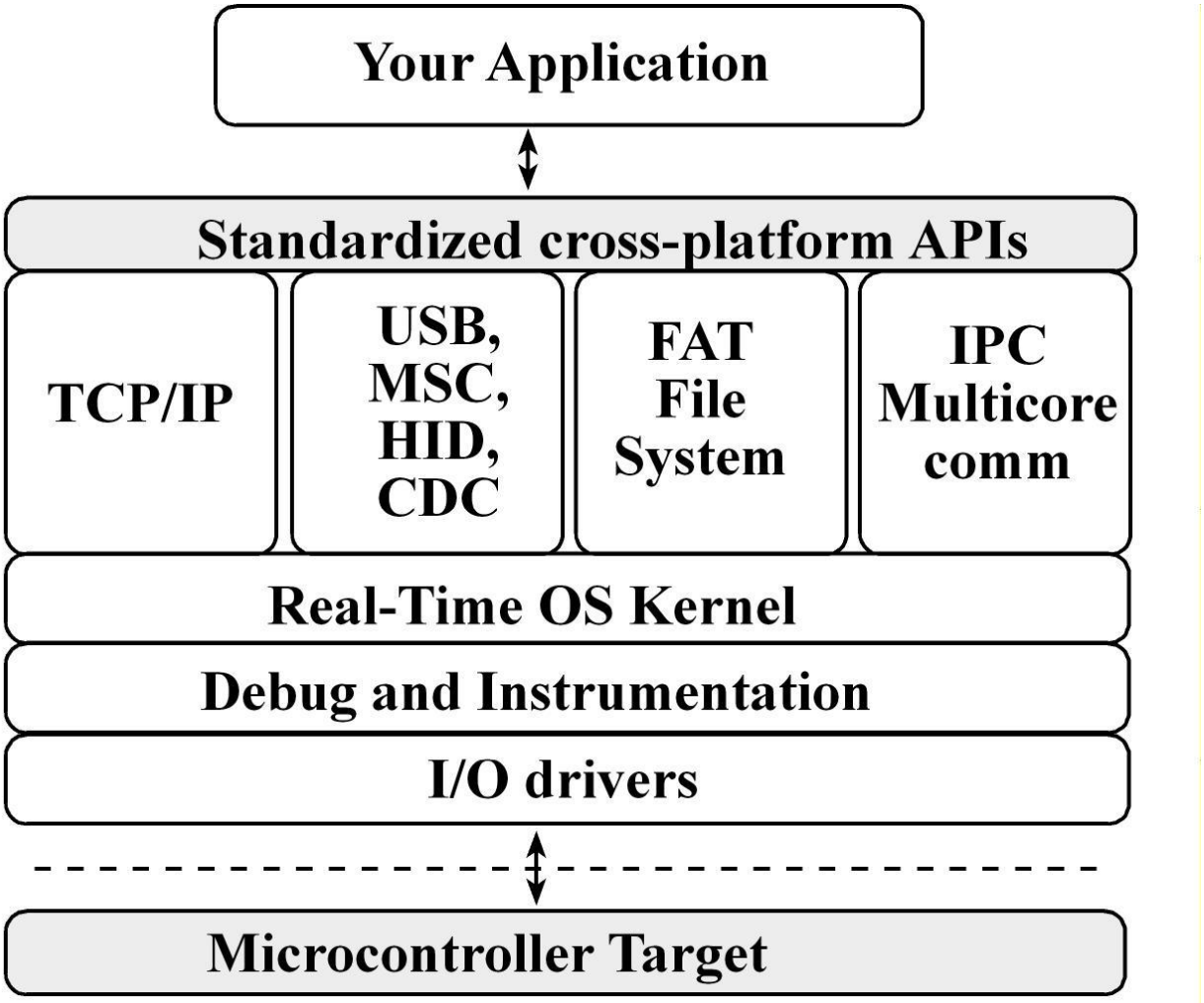

Figure 4.22. Block diagram of the Texas Instruments RTOS.

TI-RTOS Module	Description
TI-RTOS Kernel	TI-RTOS Kernel provides deterministic preemptive multithreading and synchronizatio handling. TI-RTOS Kernel is highly scalable down to a few KBs of memory.
TI-RTOS Networking	TI-RTOS Networking provides an IPv4 and IPv6-compliant TCP/IP stack along with ass and DHCP.
TI-RTOS File System	TI-RTOS File System is a FAT-compatible file system based on the open source Fatfs p
TI-RTOS USB	TI-RTOS USB provides both USB Host and Device stacks, as well as MSC, CDC, and HI TivaWare USB stack.
TI-RTOS IPC	The TI-RTOS IPC provides efficient interprocessor communication in multicore device
TI-RTOS Instrumentation	TI-RTOS Instrumentation allows developers to include debug instrumentation in their including context-switching, to be displayed by system-level analysis tools.
TI-RTOS Drivers and Board Initialization	TI-RTOS Drivers and Board Initialization provides a set of device driver APIs, such as E all devices, as well as initialization code for all supported boards. All driver and board MWare, or MSP430Ware libraries.

Table 4.5 Components of the TI RTOS.

4.5.3. ARM RTX Real-Time Operating System[edit]

The Keil RTX is a royalty-free, deterministic Real-Time Operating System designed for ARM and Cortex-M devices. For more information, search RTX RTOS on www.arm.com. It allows you to create programs that simultaneously perform multiple functions and helps to create applications which are better structured and more easily maintained. RTX is available royalty-free and includes source code. RTX is deterministic. It has flexible scheduling including round-robin, pre-emptive, and collaborative. It operates at high speed with low interrupt latency. It has a small footprint. It supports unlimited number of tasks each with 254 priority levels. It provides an unlimited number of mailboxes, semaphores, mutex, and timers. It includes support for multithreading and thread-safe operation. There is debugging support in MDK-ARM. It has a dialog-based setup using µVision Configuration Wizard. RTX allows up to 250 active tasks. The priority scheduler supports up to 254 priority levels. The OS will dynamically check for valid stacks for running tasks. It implements timeouts, interval timing, and user timers. Synchronization and inter-task communication are handled by signals/events, semaphores, mutexes, and mailboxes. A task switch, the Cortex M3 version shown as Program 4.26, requires 192 bus cycles. The STMDB instruction saves the current thread and the LDMIA instruction restores the context for the next thread.

```
__asm void PendSV_Handler (void) {
        BL      __cpp(rt_pop_req)   ; choose next thread to run
        LDR     R3,=__cpp(&os_tsk)
        LDM     R3,{R1,R2}              ; os_tsk.run, os_tsk.new
        CMP     R1,R2
        BEQ     Sys_Exit
        PUSH    {R2,R3}
        MOV     R3,#0
        STRB    R3,[R1,#TCB_RETUPD]     ; os_tsk.run->ret_upd = 0
        MRS     R12,PSP                 ; Read PSP
        STMDB   R12!,{R4-R11}           ; Save Old context
        STR     R12,[R1,#TCB_TSTACK]    ; Update os_tsk.run->tsk_stack
        BL      rt_stk_check            ; Check for Stack overflow
        POP     {R2,R3}
        STR     R2,[R3]                 ; os_tsk.run = os_tsk.new
        LDR     R12,[R2,#TCB_TSTACK]    ; os_tsk.new->tsk_stack
        LDMIA   R12!,{R4-R11}           ; Restore New Context
        MSR     PSP,R12                 ; Write PSP
        LDRB    R3,[R2,#TCB_RETUPD]     ; Update ret_val?
        CBZ     R3,Sys_Exit
        LDRB    R3,[R2,#TCB_RETVAL]     ; Write os_tsk.new->ret_val
        STR     R3,[R12]
Sys_Exit MVN    LR,#:NOT:0xFFFFFFFD     ; set EXC_RETURN value
        BX      LR                      ; Return to Thread Mode
}
```

Program 4.26. Thread switch code on the ARM RTX RTOS (see file HAL_CM3.c).

ARM's Cortex Microcontroller Software Interface Standard (CMSIS) is a standardized hardware abstraction layer for the Cortex-M processor series. The CMSIS-RTOS API is a generic RTOS interface for Cortex-M processor-based devices. You will find details of this standard as part of the Keil installation at **Keil\ARM\CMSIS\Documentation\RTOS\html**. CMSIS-RTOS provides a standardized API for software components that require RTOS functionality and gives therefore serious benefits to the users and the software industry.

- CMSIS-RTOS provides basic features that are required in many applications or technologies such as UML or Java (JVM).
- The unified feature set of the CMSIS-RTOS API simplifies sharing of software components and reduces learning efforts.
- Middleware components that use the CMSIS-RTOS API are RTOS agnostic. CMSIS-RTOS compliant middleware is easier to adapt.
- Standard project templates (such as motor control) of the CMSIS-RTOS API may be shipped with freely available CMSIS-RTOS implementations.

4.5.4. FreeRTOS[edit]

FreeRTOS is a class of RTOS that is designed to be small enough to run on a microcontroller. FreeRTOS only provides the core real-time scheduling functionality, inter-task communication, timing and synchronization primitives. This means it is more accurately described as a real-time kernel, or real-time executive. FreeRTOS is available for 35 processor architectures, with millions of product deployments. For more information on FreeRTOS, see their web site at http://www.freertos.org/RTOS-Cortex-M3-M4.html. The starter project for the LM3S811 can be easily recompiled to run an any of the Texas Instruments Cortex M microcontrollers.

FreeRTOS is licensed under a modified GPL and can be used in commercial applications under this license without any requirement to expose your proprietary source code. An alternative commercial license option is also available in cases that: You wish to receive direct technical support. You wish to have assistance with your development. You require legal protection or other assurances. Program 4.27 shows the PendSV handler that implements the context switch. Notice that this thread switch does not disable interrupts. Rather, the ISB instruction acts as an instruction synchronization barrier. It flushes the pipeline of the processor, so that all instructions following the ISB are fetched from cache or memory again, after the ISB instruction has been completed. Similar to Micrium µC/OS-II and ARM RTX, the FreeRTOS does run user threads with the process stack pointer (PSP).

```
__asm void xPortPendSVHandler( void ){
    extern uxCriticalNesting;
    extern pxCurrentTCB;
    extern vTaskSwitchContext;
    PRESERVE8
    mrs r0, psp
```

```
    isb
    ldr    r3, =pxCurrentTCB    /* Get the location of current TCB. */
    ldr    r2, [r3]
    stmdb r0!, {r4-r11}    /* Save the remaining registers. */
    str r0, [r2]    /* Save the new top of stack into the TCB. */
    stmdb sp!, {r3, r14}
    mov r0, #configMAX_SYSCALL_INTERRUPT_PRIORITY
    msr basepri, r0
    bl vTaskSwitchContext
    mov r0, #0
    msr basepri, r0
    ldmia sp!, {r3, r14}
    ldr r1, [r3]
    ldr r0, [r1] /* first item in pxCurrentTCB is task top of stack. */
    ldmia r0!, {r4-r11} /* Pop registers and critical nesting count. */
    msr psp, r0
    isb
    bx r14
    nop
}
```

Program 4.27. Thread switch code on FreeRTOS also uses PendSV for the Cortex M3.

4.5.5. Other Real Time Operating Systems[edit]

Other real time operating systems available for the Cortex M are listed in Table 4.6

Provider	Product
CMX Systems	CMX-RTX,CMX-Tiny
Expresslogic	ThreadX
Green Hills	Integrity®, µVelOSity
Mentor Graphics	Nucleus+®
Micro Digital	SMX®
RoweBots	Unison
SEGGER	embOS

Table 4.6 Other RTOS for the Cortex M (http://www.ti.com/lsds/ti/tools-software/rtos.page#arm)

Deployed in over 1.5 billion devices, VxWorks® by Wind River® is the world's leading real-time operating system (RTOS). It is listed here in the other category because it is deployed on such architectures as the X86, ARM Cortex-A series, and Freescale QorIQ, but not on the Cortex M microcontrollers like the TM4C123. VxWorks delivers hard real-time performance, determinism, and low latency along with the scalability, security, and safety required for aerospace and defense, industrial, medical, automotive, consumer electronics, networking, and other industries. VxWorks has become the RTOS of choice when certification is required. VxWorks supports the space, time, and resource partitioning required for IEC 62304, IEC 61508, IEC 50128, DO-178C,

and ARINC 653 certification. VxWorks customers can design their systems to the required level of security by picking from a comprehensive set of VxWorks security features. VxWorks is an important play in providing solutions for the Internet of Things (IoT), where connectivity, scalability, and security are required. For more information
see http://www.windriver.com/products/vxworks/