EE443 - Embedded Systems

Lecture 8
Applications

Contents:

8.1 The Main Program

Simple embedded systems can function without an operating system. The main program written for such a system starts execution right after the system power is turned on or following a catastrophic failure that requires a complete restart. The master reset signal indicating these conditions direct the processor to the beginning of the main program. The main program is responsible for initializing all required system functions. Initialization involves setting operation parameters of peripheral units (i.e. programming of timers, interrupt controller, I/O modules) and system tasks (i.e. setting initial values, clearing buffers).

Once the system is fully functional, the processor is kept alive in a loop that runs forever monitoring the events that require system's response. In one extreme case, the main program performs all monitoring and response tasks in this loop using the polling methods. We have seen that a processor performing all operations in a polling loop may not give a timely response to all events and its timing accuracy is limited. In another extreme case, all system functions can be moved to the interrupt service routines that are activated automatically whenever it is necessary to respond to an external event as in the example given below:

```
void Main(void)
// All operations are performed in the interrupt service routines.
{
  <Initialize peripherals>
  <Initialize tasks>
  <Enable interrupts>
  while (TRUE)  // Main Loop
  { <do nothing>
  }  // end of Main Loop
}

void ISR1(void)
{
  <Give response to event-1>
  <Receive related data>
  <Process event-1 data>
}

void ISR2(void)
{
  <Give response to event-2>
  <Receive related data>
  <Process event-2 data>
}
```

In this extreme case, the management of interrupts (setting priorities, sharing resources, etc.) can be difficult or even impossible in some applications. Balancing the processor load between the main operation loop and the ISRs can be the best solution in these embedded systems applications.

```
// Main program:
void Main(void)
// Processor load is shared between main program and ISRs
{
  <Initialize peripherals>
  <Initialize tasks>
  <Enable interrupts>
  while (TRUE)  // Main Loop
  { if (<queue buffer-1 has data>)
    { <Process event-1 data> }
    if (<semaphore-2 is set>)
    { <Process event-2 data> }
    if (<event-3 requires response>)
    { • • • }
  }  // end of Main Loop
}

void ISR1(void)
{
  <Give immediate response to event-1>
  <Store data in queue buffer-1>
}

void ISR2(void)
{
  <Give immediate response to event-2>
  <Store data in memory and set semaphore-2>
}
```

ISRs perform the tasks that are required immediately after an event, such as reading data from an I/O unit or restarting a timer. The remaining low-priority processing can be done in the main operation loop. The data transfers between the main loop and the ISRs can be synchronized using queue buffers or semaphores. Note that, monitoring of low-priority or rarely occurring events can be done in the main polling loop among other low-priority processing tasks.

8.2 Finite State Machines

A finite state machine (FSM) can be implemented easily on a processor. The following code segment uses the **switch** statement to implement a basic FSM.

```
  • • •
  switch (CurrentState)
  {
  case STATE1:
    NextState = <function of CurrentState and FSMin>;
    FSMout = <function of CurrentState and FSMin>
  case STATE2:
    NextState = <function of CurrentState and FSMin>;
    FSMout = <function of CurrentState and FSMin>
  • • •
  • • •
  default:
// Case for initialization or handling error conditions:
    NextState = <default next state for undefined state>;
    FSMout = <FSM outputs for undefined state>
  }  // end of switch (CurrentState)
  <processing common for all states>
  CurrentState = NextState;  // update the state information
  • • •
```

Operation cycle of a FSM running on a processor is not much different than a FSM implemented on dedicated hardware. The next state and the FSM outputs are calculated every time the FSM code is executed. The calculated values are functions of the current state and the FSM inputs.

8.2.1 Handling Real-Time Inputs

Timing of the input signals can be critical in all digital systems. It is important to know when the inputs may change for implementation of a FSM or any other decision algorithm that relies on external inputs. We cannot assume that the external inputs will remain unchanged during the execution of an algorithm, unless it is guaranteed by the system hardware. If an algorithm requires a single input value throughout the execution, then the program cannot read the input source more than once. As an example, consider a program that takes its input from **PortA** and generates an output using the present and past input values.

```
  • • •
  DiffInput = PortA - PortAprev;  // calculate difference
  Fout = <function of PortA and DiffInput>
  • • •
  PortAprev = PortA;  // store input to be used next time
  • • •
```

This program reads **PortA** three times; 1) to obtain **DiffInput**, 2) to calculate **Fout**, and 3) to store the input value for the next execution cycle. If the **PortA** input changes between the first two read operations, then the calculated **Fout** value will be invalid. If the **PortA** input changes before the third read operation, then the stored value will be different, and the **DiffInput** calculated in the next execution cycle will be invalid. The solution is to save the **PortA** input in a buffer storage as shown below.

```
• • •
PortAprev = PortAsave;  // store input from the previous cycle
PortAsave = PortA;      // read PortA only once
DiffInput = PortAsave - PortAprev;  // calculate difference
Fout = <function of PortAsave and DiffInput>
• • •
```

This improved program accesses **PortA** just once during an execution cycle and stores the input in **PortAsave**. All problems that may result from the inputs changing during processing are eliminated by using the value stored in **PortAsave** throughout the program.

A similar problem occurs in the basic FSM structure given above. If the input, **FSMin**, changes during execution, then the calculated **NextState** and **FSMout** values may produce contradicting results. The solution is to save the **FSMin** in a temporary storage and to use the stored value in all calculations.

ADCs, I/O interfaces, and several other microcontroller peripherals have buffers that hold the data. Changes in external signals do not affect the buffers immediately, but the buffer contents may still change unexpectedly during processing as described in the following examples.

- Processor starts working on an ADC sample and the result in the buffer will not change unless a new conversion is started. However, an ISR may cut in and start another ADC conversion that replaces the data in the ADC buffer.
- Processor starts working on a byte received through a serial interface and stored in the receive buffer. Processing takes only 10 µs and another byte cannot arrive any sooner than 1 ms. One can assume that the received byte is safe, since the available time is 100 times longer than the time required for processing. However, if other interrupts are received and served during the processing, then it may take longer than 1 ms to complete the task.

In some cases, disabling or suspending the operation of the peripheral units can be a solution to prevent problems due to unexpected buffer updates. Most of the time, the easiest and the most reliable solution is to copy the data into a local variable that cannot be accessed by any other program module.

8.3 Look-up Tables

Look-up tables are used for reducing processor load at the expense of storing reasonable amount of data in the memory. Typical examples of look-up table usage:

- Function calculation or waveform generation that takes a lot of processor time without the help from a floating point processor.
- Conversion of non-linear sensor or actuator response to a linear function.
- Correction of non-linear distortions using calibration data.

8.3.1 Example: Waveform Generation

The following code segment is an example of sinusoidal waveform generation utilizing a look-up table. Only one quadrant of ***sin*(x)** function (for **x** = 0º, 10º, .. 90º) is stored in the table. The remaining quadrants are synthesized using the symmetry properties.

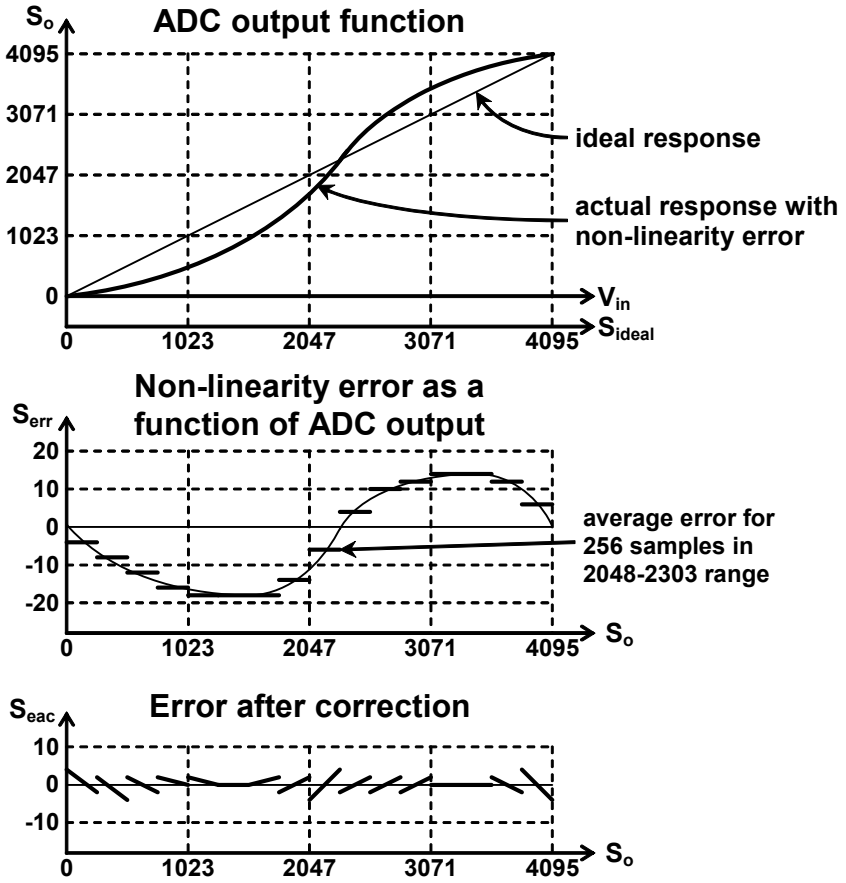
```
signed char SinSynth(void)
// Generates a sinusoidal waveform at 10 degree steps.
// Returns an output value in -127..+127 range.
{
// Function is sampled in one quadrant at every 10 degrees.
//                                  0, 10, 20, 30, 40, 50, 60, 70, 80, 90
static signed char  SineLUT[10] = {0, 22, 43, 64, 82, 97,110,119,125,127};
static signed char  LUTindex = 0;
static signed char  LUTinc = 1;   // index increment
static signed char  LUTpol = 0;   // output polarity: 1=>+ve, 0=>-ve

  SineOut = SineLUT[LUTindex];
  if (LUTpol == 0)
    SineOut = -SineOut;
  if (LUTindex == 9)
    LUTinc = -1;            // next time decrement the index
  else            // What difference it makes; "if" or "else if"?
    if (LUTindex == 0)
    { LUTinc = 1;           // next time increment the index
      LUTpol = 1 - LUTpol;  // reverse the output polarity
    }
  LUTindex += LUTinc;
  return SineOut;
}
```

Variables declared in a function are **local** variables that cannot be accessed outside the function. Ordinary local variables declared without the "**static**" attribute are assigned to the memory locations that can be used by other functions. Therefore, value of an ordinary local variable may be changed by other functions. The look-up table array and other variables of the **SinSynth** function are declared with the "**static**" attribute. These variables are placed at dedicated memory locations, and they preserve their values between the function calls. For instance, the last value assigned to **LUTindex** will remain unchanged when the **SinSynth** function is called again. Initial value of the **static** variables can be specified in the declaration. C compiler automatically generates the initialization code and appends it before the main program so that the **static** variables are initialized only once after a reset condition.

8.3.2 Example: Non-linearity Correction

The non-linearity error can be characterized using a precision voltmeter in a test setup. ADC samples and voltmeter readings are recorded throughout the ADC input range. The following graph shows the response of a 12-bit ADC as a function of the analog input and the corresponding ideal sample values.



The purpose is to obtain a more accurate measurement by subtracting the non-linearity error corresponding to the ADC samples. It is not practical to store an error value for every possible ADC output, unless 4 Kbytes of memory is available for a look-up table at no extra cost. In this example, ADC sample range is divided into 16 equal segments and an average non-linearity error is determined for each segment. The initial +/-18 LSB non-linearity error is reduced to +/-2 LSB after the correction as shown above. The averaged non-linearity errors are stored in a look-up table that has only 16 elements. Time consuming multiplication and division operations are avoided to obtain an efficient correction procedure.

```
signed short int CorrectADCNL(signed short int ADCsample)
// Applies non-linearity correction to the ADC output values using a LUT.
{
// Index to LUT is the 4 most significant bits of 12-bit ADC sample.
static signed short int  ADCerrLUT[16] = {-4,-8,-12,-15,-18,-18,-17,-14,
                                          -6, 5, 10, 12, 14, 14, 12,  6};
unsigned char      LUTindex;
signed short int   ADCcorrect;
// Use the 4 MSBs of 12-bit ADC sample as index.
  LUTindex = ADCsample >> 8;
  ADCcorrect = ADCsample - ADCerrLUT[LUTindex];
  return ADCcorrect;
}
```

The most significant byte of the ADC samples can also be used as an index for the look-up table. The repeated right-shift operation used above can be eliminated by using the following alternative code.

```
signed short int CorrectADCNL(signed short int ADCsample)
// Applies non-linearity correction to the ADC output values using a LUT.
{
// Index to LUT is the 4 most significant bits of 12-bit ADC sample.
static signed short int  ADCerrLUT[16] = {-4,-8,-12,-15,-18,-18,-17,-14,
                                          -6, 5, 10, 12, 14, 14, 12,  6};
// Use the most significant byte of 12-bit ADC sample as index.
unsigned char     *pMSByte = (unsigned char *)(&ADCsample);
signed short int   ADCcorrect;
  ADCcorrect = ADCsample - ADCerrLUT[*pMSByte];
  return ADCcorrect;
}
```

Similar algorithms can be applied for correction of non-linearity errors introduced by sensors, actuators, or signal conditioning circuitry. In any case, a precision measurement device is required to obtain the calibration data as a function of the measured quantity or the controlled output. For example, a precision thermometer is required to calibrate the response of a temperature sensor.

8.3.3 Example: FSM Implementation with LUT

A finite state machine (FSM) calculates the next state and the FSM outputs based on the present state and the input values. These calculations require a long sequence of conditional branch instructions to implement the algorithm outlined in the previous section. If all possible next state values and FSM outputs are stored in a LUT then simple array indexing operations can replace the FSM calculations. The number of LUT elements is given by the number of FSM states multiplied by the number of possible input settings. The FSM implemented below has 16 states, 2 binary inputs (4 possible input settings), and 4 binary outputs.

```
unsigned char LUT_FSM(unsigned char FSMin)
// Implements a finite state machine (FSM) using a LUT. Two LSBs of FSMin are
// the FSM inputs. Four LSBs of the returned value are the FSM outputs.
{
// Four MSBs are the next state, four LSBs are the FSM outputs in the LUT:
static unsigned char  LUTarray[64] = {0x00,0x15,0x13, . . . . . .
                                      // rest of the table is not shown
static unsigned char  FSMstate = 0;  // initial state is 0
unsigned char         LUTindex;

  LUTindex = (FSMstate >> 2) | FSMin;    // calculate the LUT index
  FSMstate = LUTarray[LUTindex] & 0xF0;  // set the next state
  return LUTarray[LUTindex] & 0x0F;      // return the FSM outputs
}
```

The function given above is an efficient FSM implementation. Execution time of this function remains the same for all FSM cycles whereas the execution time of the previous FSM algorithm changes depending on the present state in each cycle. Having a fixed execution time can be advantageous when the FSM is part of a time-sensitive processing task.

8.4 Noise Reduction

8.4.1 Example: Moving Average Calculation

Normally an analog filter is placed at the input of an ADC to remove the high-frequency noise from the input signal. If the ADC sampling frequency is set at the Nyquist rate then further noise reduction is not possible without loosing valuable signal bandwidth. Filtering after sampling can be useful in the following cases:

- A variable ADC sampling rate is required in the application. The input analog filter works well at the maximum sampling frequency, but further noise reduction is necessary when lower sampling frequencies are used.
- Random noise is introduced during analog to digital conversion after analog filtering. This problem is common in MCUs with on-chip ADC units where ADC operation can be affected by thousands of surrounding digital signals.

The moving average (also called rolling average) algorithm is an efficient filtering method that does not require long multiplication operations. The division operation for averaging can be avoided if the number of averaged samples is chosen as a power of two. The following function calculates the moving average of four consecutive ADC samples.

```
signed short int FilterADC1(signed short int ADCsample)
/* Calculates the moving average of 4 consecutive ADC samples. */
static signed short int   Smp0 = 0;
static signed short int   Smp1 = 0;
static signed short int   Smp2 = 0;
static signed short int   Smp3 = 0;
static signed short int   SmpSum = 0;
signed short int          FilterOut;
// Add the input sample and subtract the oldest sample.
{ SmpSum = SmpSum + ADCsample - Smp0;
// Shift the stored samples.
  Smp0 = Smp1;
  Smp1 = Smp2;
  Smp2 = Smp3;
  Smp3 = ADCsample;
// Divide the sum of four samples by 4.
  FilterOut = SmpSum >> 2;
  return FilterOut;
}
```

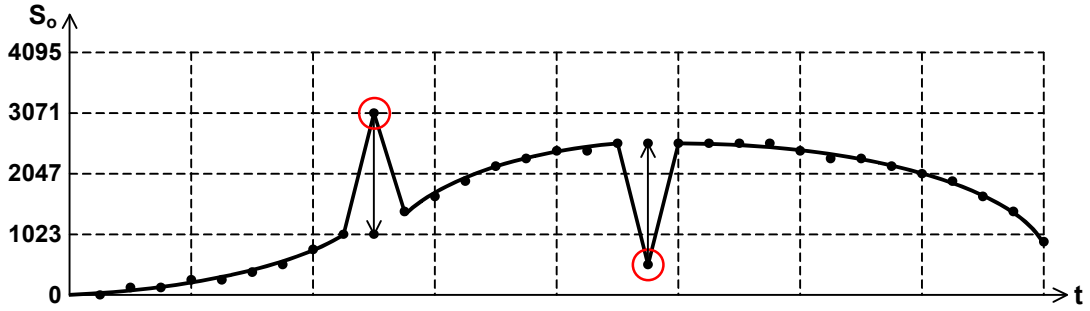
The function given above is an efficient way of calculating a moving average of four samples, but shifting a large number of stored samples takes too much time. Use of a circular buffer can speed up the calculations when a large number of samples are taken into account. The following procedure calculates a moving average over 32 samples.

```
signed short int FilterADC2(signed short int ADCsample)
/* Calculates the moving average of 32 consecutive ADC samples. */
static signed short int   SmpBfr[32] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
                                        0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
static unsigned char      Ksmp = 32;
static signed short int   SmpSum = 0;
static signed short int  *pSmp = SmpBfr;
signed short int          FilterOut;
// Add the input sample and subtract the oldest sample.
{ SmpSum = SmpSum + ADCsample - *pSmp;
// Store the received sample.
```

```
  *pSmp = ADCsample;
  Ksmp --;
// Go back to the first buffer element if the end is reached.
  if (Ksmp == 0)
  { Ksmp = 32;
    *pSmp = SmpBfr;
  }
  else
    pSmp ++;  // increment the buffer pointer
// Divide the sum of 32 samples by 32.
  FilterOut = SmpSum >> 5;
  return FilterOut;
}
```

8.4.2 Example: Eliminating Glitches

A glitch is a single data point that contains a large amount of error. Glitches are usually result from the coupling between the sampled analog signal and strong switching signals such as PWM waveforms used for motor control. A glitch filter detects these samples and replaces them with estimated values based on the previous sample history.

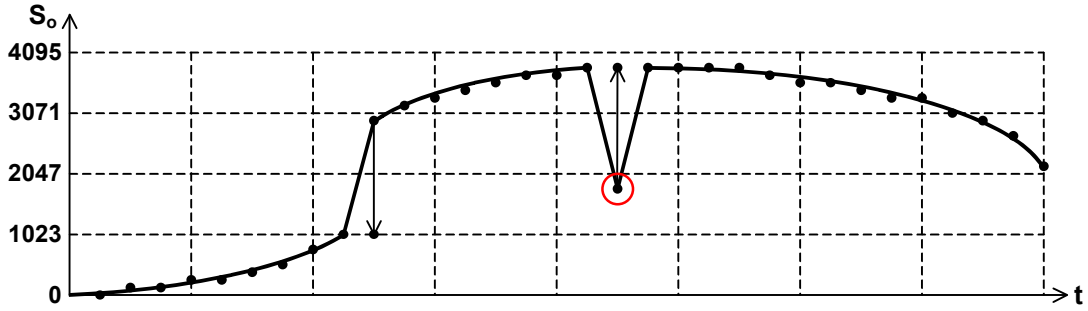


The following glitch filter function works with a moving average filter. Every incoming ADC sample is compared with the previously calculated moving average. The ADC sample is replaced with the average value when there is an unexpected jump in the sample value. The glitches are identified by detecting +/- changes greater than a previously defined **GLITCH_LIMIT**.

```
static signed short int   FilterOut = 0;
signed short int          ADCjump;  // absolute change in ADC output
// Check if the ADC sample has an unexpected jump.
{ ADCjump = ADCsample - FilterOut;
  if (ADCjump < 0)  // take absolute value
    ADCjump = -ADCjump;
  if (ADCjump > GLITCH_LIMIT)
// Replace the ADC sample with the current average value.
    ADCsample = FilterOut;
// Add the input sample and subtract the oldest sample.
  SmpSum = SmpSum + ADCsample - *pSmp;
  <continue with calculation of moving average>
  • • •
```

In this function, there is a risk of taking the large step changes in the input signal as glitch errors if the actual signal varies as shown below. The function takes also the subsequent samples as glitches since it never updates the calculated moving average value. The following glitch filter function uses a more complex glitch detection algorithm. Every time a sample is treated as a glitch, this information is stored and used when making decisions for the following samples. The first sample

of a large step change in the input signal is still treated as a glitch, but the algorithm realizes that the following samples are part of the actual input signal.




```
static signed short int   FilterOut = 0;
signed short int          ADCjump;   // absolute change in ADC output
signed char               JumpSign;  // sign of ADCjump
static signed char        GlitchState = 0;  // previous glitch detection
// Check if the ADC sample has an unexpected jump.
{ ADCjump = ADCsample - FilterOut;
  if (ADCjump < 0}  // take absolute value
  { ADCjump = -ADCjump;
    JumpSign = -1;
  }
  else
    JumpSign = 1;
  if (ADCjump > GLITCH_LIMIT)
// Replace the ADC sample with the current average value if the jump
// in ADC samples is not consistent.
  { if (JumpSign != GlitchState)
      ADCsample = FilterOut;
    GlitchState = JumpSign;
  }
  else
    GlitchState = 0;
// Add the input sample and subtract the oldest sample.
  SmpSum = SmpSum + ADCsample - *pSmp;
  <continue with calculation of moving average>
  • • •
```

One should always remember that the best solutions eliminate the problems right at the source. The ***band-aid*** solutions like the glitch filter described above rely on several assumptions related to the glitch amplitude and the signal shape. Signal coupling problems can be prevented by using isolation and proper grounding techniques in most cases. As an alternative, gating methods can be used to avoid simultaneous operation of system components. For example, a switching power signal can be suspended while an ADC is active, or ADC timing can be adjusted to avoid switching time instants.

8.5 I/O Buffers

8.5.1 Example: Data Logging

This example involves two interrupt service routines that work together to send data to a master computer. ISR1 writes the incoming data into a buffer. ISR2 is activated when the master computer asks for data. Every time ISR2 is executed, it first sends the number of data elements stored in the buffer and then it sends the buffer contents. The processing of incoming data cannot be delayed while ISR2 is sending data. Therefore, ISR1 has a higher priority and it can be activated during execution of ISR2.

First data logging implementation:

```
signed short int   OutBfr[BUFFER_SIZE];  // buffer storage
unsigned char      NDlog = 0;       // number of data in the buffer

void   ISR1(void)
/* Receives data and stores in the output buffer. */
signed short int           DataIn;
static signed short int   *pStore;
{
  • • •
  <get DataIn from source>
  if (NDlog == 0)  // check if the buffer is empty
    pStore = OutBfr;  // initialize the buffer write pointer
  else
    pStore ++;
  *pStore = DataIn;
  NDlog ++;
  • • •
}

void   ISR2(void)
/* Activated when the master computer asks for data.  Sends the number
   of data elements followed by the series of stored data. */
signed short int          *pSend;
unsigned char              k;
{
  • • •
  <send NDlog>
  pSend = OutBfr;  // initialize the buffer read pointer
  for (k = NDlog; k > 0; k--)
  { <send data at pSend>
    pSend ++;
  }
  NDlog = 0;
  • • •
}
```

In this first implementation there are couple of things that can go wrong:

1. ISR1 has higher priority, and it may store new data after ISR2 executes the statement "**`k = NDlog;`**" at the beginning of the **`for`** loop. The new data stored will be lost when the buffer status is reset with the statement "**`NDlog = 0;`**" in ISR2.

2. If the master computer is lazy, then the buffer may be filled up. ISR1 will keep incrementing the pointer, **`pStore`**, and it will end up writing into the other memory locations outside the reserved buffer storage.

In the following implementation we added another variable to indicate the buffer overrun conditions that occur in ISR1. The index, **k**, is compared to the number of stored elements, **NDlog**, in every cycle of the **while** loop.

Second data logging implementation:

```
signed short int   OutBfr[BUFFER_SIZE];  // buffer storage
unsigned char      NDlog = 0;       // number of data in the buffer
unsigned char      BfrOverRun = 0;  // number of missed data

void   ISR1(void)
/* Receives data and stores in the output buffer. */
signed short int           DataIn;
static signed short int   *pStore;
{
  • • •
  <get DataIn from source>
  if (NDlog == BUFFER_SIZE)  // check if the buffer is full
    BfrOverRun ++;   // increment number of missed data points
  else
  { if (NDlog == 0)  // check if the buffer is empty
      pStore = OutBfr;  // initialize the buffer write pointer
    else
      pStore ++;
    *pStore = DataIn;
    NDlog ++;
  }
  • • •
}

void   ISR2(void)
/* Activated when the master computer asks for data.  Sends the buffer
   overrun status and the number of data elements followed by the
   series of stored data. */
signed short int          *pSend;
unsigned char              k;
{
  • • •
  <send BfrOverRun>
  <send NDlog>
  pSend = OutBfr;  // initialize the buffer read pointer
  k = 0;
  while (k < NDlog)
  { <send data at pSend>
    pSend ++;
    k ++;
  }
  NDlog = 0;
  BfrOverRun = 0;
  • • •
}
```

The second set of ISRs seem better than the first implementation, but there are still other problems:

1. ISR2 sends **NDlog** at the beginning. If ISR1 stores new data while ISR2 is executing the while loop, then the actual number of data transmitted will be more than the **NDlog** value received by the master computer.

2. There is still a chance of loosing data. If ISR1 is activated after termination of the while loop and before the statement "**NDlog = 0;**" in ISR2, then the stored data will be lost.

Any implementation that relies on the single linear buffer structure will have similar problems as long as ISR1 has higher priority. Disabling interrupts during data transmission is not an option, because ISR1 response cannot be delayed for a long time. One possible solution is to use two independent buffer arrays, so that ISR1 and ISR2 will not access the same buffer at any time. However, this solution requires twice as much buffer memory that may not be available. A good solution can be obtained by using a circular queue structure as shown in the third data logging implementation given below.

Third data logging implementation using circular buffer:

```
signed short int    OutBfr[BUFFER_SIZE];
signed short int   *pStore;       // enqueue pointer
signed short int   *pSend;        // dequeue pointer
signed short int   *EOBptr;       // pointer to end of the buffer
unsigned char       NDlog;        // number of stored data
unsigned char       BfrOverRun;   // number of missed data

void  InitBuffer(void)
// Initializes the I/O buffer.
{ pStore = Qbuffer;
  pSend = Qbuffer;
  EOBptr = Qbuffer + BUFFER_SIZE;
  NDlog = 0;
  BfrOverRun = 0;
}

void  ISR1(void)
/* Receives data and stores in the output buffer. */
signed short int    DataIn;
{
  • • •
  <get DataIn from source>
  if (NDlog == BUFFER_SIZE)  // check if the buffer is full
    BfrOverRun ++;
  else
  { *pStore = DataIn;
    pStore ++;  // increment enqueue pointer
    if (pStore == EOBptr)  // check if reached end of buffer
      pStore = OutBfr;  // go back to the first location
    NDlog ++;
  }
  • • •
}

void  ISR2(void)
/* Activated when the master computer asks for data.  Sends the buffer
   status and the number of data elements followed by the series of
   stored data. */
unsigned char   k;
{
  • • •
  <send NDlog>
  k = NDlog;
  <send BfrOverRun>
  BfrOverRun = 0;
  while (k > 0)
  { <send data at pSend>
    pSend ++;  // increment dequeue pointer
    if (pSend == EOBptr)  // check if reached end of buffer
      pSend = Qbuffer;  // go back to the first location
```

```
    k --;
    NDlog --;
  }
  • • •
}
```

Although the risk of inconsistent data transfers is reduced significantly, there is still a chance of failure in this implementation that is discussed in the following section.

8.5.2 Avoiding Race Conditions

Race conditions may occur when two procedures try to access the same memory location at the same time. We don't need to worry about race conditions when the programs follow a regular sequential flow pattern where a procedure cannot be activated before the other procedure finishes its task. We have a different picture, if one of the procedures can be activated by an interrupt while the other procedure is running.

Detecting all possible race conditions is not an easy job. First, we should list all shared memory locations. In the buffer example given above, the memory locations accessed by the two ISRs are:

```
signed short int    OutBfr[BUFFER_SIZE];
unsigned char       NDlog;        // number of stored data
unsigned char       BfrOverRun;   // number of missed data
```

Then we need to check if the procedure with the higher priority (ISR1 in this example) is accessing a shared memory location while it is being used by the other procedure. We had to make sure that the master computer receives consistent information regarding the number of stored and missed data elements.

Make the access to `OutBfr` safe: We prevent ISR1 writing to the same buffer element accessed by ISR2 using the variable **`NDlog`**. Decrementing **`NDlog`** in ISR2 releases a buffer element, and it should be done when ISR2 no longer needs that buffer element. (*What would happen, if* "**`NDlog --;`**" *is moved to the beginning of the while loop?*)

Make the access to `NDlog` safe: ISR2 sends an **`NDlog`** value to the master computer at the beginning:

```
<send NDlog>
k = NDlog;
```

If ISR1 stores a new data before ISR2 executes "**`k = NDlog`**", then the number of data master computer actually receives will be one more then the **`NDlog`** sent. Solution is to send the data count that is exactly the number of data that will be transmitted:

```
k = NDlog;
<send k>
```

ISR2 also reads and writes **`NDlog`** with the statement "**`NDlog --;`**". This statement will take three machine instructions on a typical 8-bit processor. If ISR1 is activated between these instructions then the **`NDlog`** value stored by ISR1 will be incorrect. The solution is to prevent access to **`NDlog`** during execution of these instructions:

```
<disable ISR1 interrupt>
NDlog --;
<enable ISR1 interrupt>
```

Make the access to `BfrOverRun` safe: ISR2 sends **`BfrOverRun`** at the beginning. There will be a problem in case ISR1 attempts to store new data before ISR2 sends out at least one data element from a full buffer. If ISR1 is activated before ISR2 executes "**`NDlog --;`**" then it will increment **`BfrOverRun`** one more time. One possible solution is to send **`BfrOverRun`** after the data sequence by placing the following statements after the while loop.

```
<send BfrOverRun>
BfrOverRun = 0;
```

Here we assume that a buffer overrun will not occur right after ISR2 transmits the buffer contents. If there is any chance of a buffer overrun occuring at the end of ISR2, then ISR1 should not be activated before ISR2 resets **`BfrOverRun`**:

```
<disable ISR1 interrupt>
k = BfrOverRun;
BfrOverRun = 0;
<enable ISR1 interrupt>
<send k>
```

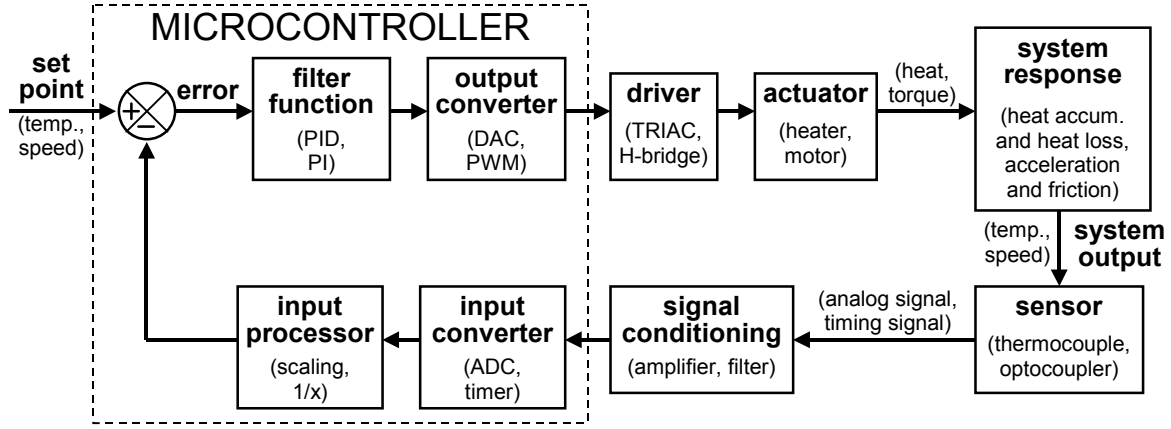
The following is a list of good programming practices to minimize the problems originating from race conditions and other conflicts related to shared memory access.

- Always remember that a simple statement written in a high-level language may correspond to several instructions in the machine code.
- Keep the number of common and global variables at minimum. More shared memory locations means more trouble. It is advisable to make some local variables accessible through simple functions instead of allowing global access to these variables.
- Use semaphores, buffer pointers and counters carefully. First check if a buffer element is available before writing anything into that buffer element. Make sure that the data in a buffer element is not required any longer before releasing that buffer element.
- Minimize the duration of access to shared memory. If processing of the data stored in a shared variable takes a long time then make a copy and use the copied data in the process.

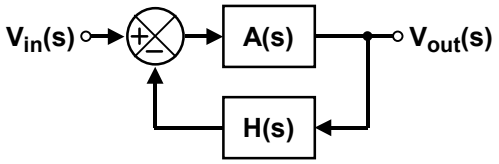
8.6 Process Control

Major components of a closed-loop process control system is shown in the block diagram given below. A sensor is used to digitize the system output that can be the temperature of a container, speed of a motor, or some other measurable property. The common examples of the components and the signal flow are listed in parenthesis for two cases. In the first case, the controlled system parameter is the temperature. In the second case, rotational speed of a motor is controlled. In a typical control system, the microcontroller contains the input and output converters such as, ADCs, DACs, and timers. The processor reads the **system output** detected by a sensor, and calculates an **error** according to the **set point** entered by a user or a master computer. The output response is calculated for every input sample using a filter algorithm such as, **PID** (proportional-integral-derivative) or **PI** (proportional-integral) controller.

In both of the example cases, the system response is similar to a lossy integrator. The temperature of a container increases as it accumulates the heat generated by the actuator. The system time constant is given by the heat capacity of the container and the thermal resistance between the container and the surrounding medium. In case of the motor control example, the electrical power is accumulated as the motor speed increases. The system time constant depends on inertia and frictional losses of the rotor and the load it drives.



Overall accuracy of a closed loop control system is mainly determined by the accuracy of the feedback path. Consider a basic feedback control system characterized by the forward path and feedback path frequency domain transfer functions **A(s)** and **H(s)**:



$$\frac{V_{out}(s)}{V_{in}(s)} = \frac{A(s)}{1 + H(s)A(s)}$$

Remember that the effect of forward path distortions (i.e. gain variation, non-linearity) on the overall system response are reduced by a factor of $1/(1+|H(s)A(s)|)$. In practical terms, if the output power of the actuator is less than expected, then this may slow down the system response, but the feedback mechanism eventually compensates for this deficiency. On the other hand, compensation of deficiencies in the feedback path is not possible. Having **3ºC** error on the reading from a

temperature sensor is not any different than entering a wrong temperature setting with **3ºC** error.

When it comes to choosing the components of a feedback control system, several specifications such as, linearity, offset, drift, resolution, and reference stability determine the accuracy of the system. The specifications of components in the feedback path (sensor, signal conditioning electronics, ADC) are in general more significant than accuracy of components in the forward path (DAC, output driver, actuator). However, having a monotonic response is equally important for the forward and feedback paths. If the actuator output decreases while the controller is trying to increase the output, then oscillations are likely to appear at the system output. In case a DAC is used as the output converter, the specified differential nonlinearity should be better than +/-1 LSB to assure a monotonic response. If the differential nonlinearity is more than +/-1 LSB, then one or more of the low order input bits should be discarded to obtain a monotonic response. Effective resolution of a DAC or ADC is determined by its differential nonlinearity as well as the number of input bits.

8.6.1 PID Controller Implementation

PID controllers are used frequently in ordinary process control applications. The **filter function** of a PID controller has the general form given below. The filter function calculates the output response as a function of the error at the input, time integral of the error, and derivative of the error.

$$F_{out}(t) = G_p\, E(t) + G_i \int_{t'=0}^{t} E(t)\, dt' + G_d \frac{dE(t)}{dt}$$

The output response is determined by the gain factors, G_p, G_i, and G_d, of the proportional, integral and derivative parts, respectively. The error signal, **E(t)**, is the difference between the set point and the feedback received from the sensor. The proportional part responds to instantaneous changes in the system output. Main function of the integral part is to provide a relatively large DC level at the output in most of the common control systems. For example, if temperature of an oven is under control, then the DC level at the output corresponds to the steady heat losses. In case of motor speed control, the DC level provides the power required to overcome average mechanical load and friction. The derivative component is mostly used for compensation purposes and it slows down the controller response.

Discrete domain representation of the PID controller can be obtained by using an accumulator replacing the integration in the continuous domain response:

$$Acc(k) = Acc(k\text{-}1) + Err(k)$$

$$Fout(k) = K_p\, Err(k) + K_i\, Acc(k) + K_d \left[Err(k) - Err(k\text{-}1)\right]$$

The discrete domain gain factors are given by,

$$K_p = G_p\,, \quad K_i = G_i\, \Delta t\,, \quad K_d = G_d/\Delta t$$

where, **Δt** is the sampling time interval. The calculation of the PID controller response can be simplified by writing,

$$\begin{aligned} Fout(k) - Fout(k\text{-}1) = {} & K_p \left[Err(k) - Err(k\text{-}1)\right] + \\ & K_i \left[Acc(k) - Acc(k\text{-}1)\right] + \\ & K_d \left[Err(k) - 2\, Err(k\text{-}1) + Err(k\text{-}2)\right] \end{aligned}$$

The PID controller output can be calculated based on the output value obtained in the previous sampling interval. The accumulator block can be eliminated simply by replacing the term, $\mathbf{[Acc(k) - Acc(k-1)]}$, with **Err(k)**:

$$\begin{aligned}\mathbf{Fout(k)} = \; & \mathbf{Fout(k-1)} + \\ & \mathbf{[K_p + K_i + K_d]\ Err(k)} + \\ & \mathbf{[-K_p - 2\ K_d]\ Err(k-1)} + \\ & \mathbf{K_d\ Err(k-2)}\end{aligned}$$

This final form is the common implementation of a PID controller for basic process control applications. The three multiplication factors, $\mathbf{M_1 = [K_p + K_i + K_d]}$, $\mathbf{M_2 = [-K_p - 2\ K_d]}$, and $\mathbf{M_3 = K_d}$, are calculated whenever the controller parameters, $\mathbf{G_p}$, $\mathbf{G_i}$, $\mathbf{G_d}$, and $\mathbf{\Delta t}$, are modified. The arithmetic operations performed periodically in every $\mathbf{\Delta t}$ interval are simplified as shown in the following expression and the processor load is minimized.

$$\mathbf{Fout(k) = Fout(k-1) + M_1\ Err(k) + M_2\ Err(k-1) + M_3\ Err(k-2)}$$

$\mathbf{M_1}$, $\mathbf{M_2}$, and $\mathbf{M_3}$ remain constant after the optimum parameter values are determined according to the process requirements.

8.6.2 Handling Overflow

In the PID controller described above, assume that **Fout(k)** drives a DAC or some other kind of actuator that accepts 16-bit numbers at its input. An overflow will occur when the process conditions require saturation of **Fout(k)** at one of the two limits determined by the number format. Right after the saturation point, **Fout(k)** ***wraps around*** taking a value near the opposite limit:

Signed 2's complement:		**Unsigned numbers:**	
32767	0111...11	65535	1111...11
+ 1	+ 0000...01	+ 1	+ 0000...01
-32768	1000...00	0	0000...00

If the overflow condition is not handled properly, then the microcontroller drives the actuator towards the opposite extreme. As a result, the actuator power can be zero while the system still requires the maximum power. The system output moves away from the set point and the error increases. Overflow conditions should be detected in order to prevent such drastic errors. Following are the important reminders for handling overflow conditions.

- An overflow may occur during any addition, subtraction or multiplication. Overflow checks must be performed after every stage of calculations. In the PID controller implementation given above, overflow conditions may occur in any of the multiplication operations $\mathbf{M_1}$x**Err(k)**, $\mathbf{M_2}$x**Err(k-1)**, or $\mathbf{M_3}$x**Err(k-3)**.
- Properly determine the number format according to the range of operands in an arithmetic operation. In some cases, using an additional byte to extend the number range can be easier and faster than checking for overflow, depending on the capabilities of the processor.
- Alternatively, the range of operands can be restricted to eliminate the possibility of overflow in a calculation.

- All number representations have a lower limit as well as an upper limit. If an unsigned variable is forced to have a negative value, then it will wrap around to the upper limit resulting in drastic errors.

As an example consider a control system where the filter calculations are limited by 16-bit resolution because of the timing reuirements and slow processor speed. The number ranges for the input/output variables are specified below.

- 10-bit numbers at the set point and feedback inputs.
- Controller output is a 16-bit unsigned number.
- $\mathbf{M_1}$, $\mathbf{M_2}$, and $\mathbf{M_3}$ are stored as signed 8-bit numbers.

Calculation of Err(k): The difference between the set point and feedback inputs should be stored as an 11-bit number to avoid overflow. The initial error in the system output can be large during the transient response (i.e. after power-up), but the expected error during steady state operation of a typical system is usually much smaller. The calculated error values can be restricted to the range, -128...+127, resulting in an 8-bit number instead of 11 bits required to cover the entire error range. The restricted error range slows down the transient response of the system, but the steady state accuracy is preserved.

Calculation of M_1x$Err(k)$, M_2x$Err(k-1)$, and M_3x$Err(k-3)$: If the full error range is used, then overflow checks are required in the multiplication operations. There is no possibility of overflow when the error range is restricted to -128...+127.

Calculation of $Fout(k) = Fout(k-1) + M_1\ Err(k) + M_2\ Err(k-1) + M_3\ Err(k-2)$: The processor adds 16-bit numbers to produce a 16-bit result. Therefore, checking for overflow condition is necessary after each addition operation. A single overflow check can be sufficient if the intermediate results are stored as 17-bit numbers. The order of addition operations should be arranged as follows to minimize the possibility of overflow:

$$\mathbf{Fout(k) = Fout(k-1) + \left[\left(M_1\ Err(k) + M_2\ Err(k-1) \right) + M_3\ Err(k-2) \right]}$$

The possibility of overflow in addition of $\mathbf{M_1\ Err(k)}$ and $\mathbf{M_2\ Err(k-1)}$ is low since the two operands are likely to have opposite signs. Note that the controller output is expected to be a 16-bit unsigned number. The filter output, **Fout(k)**, can be stored as a signed number to avoid complications resulting from combining unsigned and signed operands in the final addition. The unsigned output range can be obtained simply by inverting the most significant bit of **Fout(k)**.

8.6.3 Saturation Recovery

Mathematical models based on frequency domain representations of controller and system response assume ideal operating conditions, where there is no saturation limit for any of the system variables. In practice, an overflow condition in a digital control system results in the saturation of the system variables. In analog systems, saturation occurs when a system variable reaches the maximum or minimum value determined by the supply voltages. Recovery from saturation conditions is an important factor especially for the control systems that have integrator or accumulator components.

Controller parameters are determined for optimum system performance under steady state operating conditions. Normally, saturation of the system variables can

be avoided at steady state operation, but saturation conditions easily occur during the transient response of common control systems. A well-designed system recovers from saturation in the shortest possible time with minimal disturbance at the system output. This requirement should be taken into account in design of any control system, whether it is analog or digital. As an example consider the two possible calculation options for a proportional-integral (**PI**) controller response:

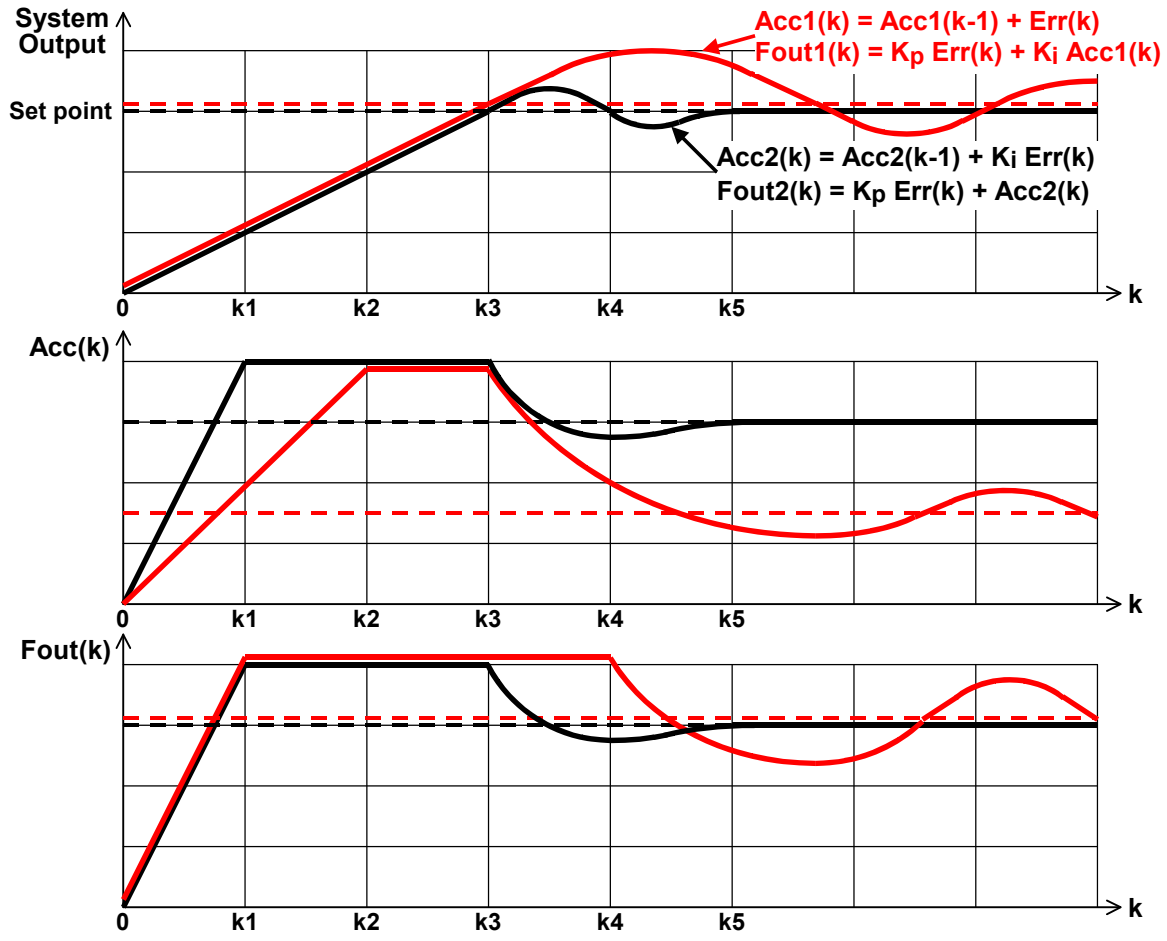
First implementation:

$$Acc1(k) = Acc1(k-1) + Err(k)$$
$$Fout1(k) = K_p\, Err(k) + K_i\, Acc1(k)$$

Second implementation:

$$Acc2(k) = Acc2(k-1) + K_i\, Err(k)$$
$$Fout2(k) = K_p\, Err(k) + Acc2(k)$$

These two implementations are mathematically identical assuming there are no limits for the variables. Multiplication with K_i is done at the accumulator input in the second calculation method, instead of applying the same gain factor at the accumulator output. However, the transient response obtained with the two implementations will be different when we take into account the possibility of saturation. The picture given below compares the controller response with the two methods of calculation. The second implementation has the advantage of faster saturation recovery.



Dashed lines in the picture show the target set point for the system output and the corresponding steady-state values of **Acc1(k)**, **Acc2(k)**, **Fout1(k)**, and **Fout2(k)**

for $\mathbf{K_i = 2}$. The system is turned on at **k=0**. Controller response in each time interval is explained below:

k=0..k1: There is a large error at the beginning. **Acc1(k)**, **Acc2(k)** ramp-up with the addition of error samples, and **Fout1(k)**, **Fout2(k)** follows.

k=k1: Overflow condition occurs at the controller output. **Fout1(k)** reaches saturation, but **Acc1(k)** continues to rise (red lines). **Acc2(k)** and **Fout2(k)** saturate at the same time (black lines).

k=k2: **Acc1(k)** reaches saturation. It takes twice as long compared to the second method, since **Acc2(k)** input is multiplied by $\mathbf{K_i = 2}$.

k=k3: System output reaches the set point at **k=k3**. Error polarity is reversed. **Acc2(k)** and **Fout2(k)** start to fall immediately. However, **Fout1(k)** remains at saturation, because $\mathbf{K_i}$***Acc1(k)** still result in an overflow, and the system response keeps rising. A lower overflow threshold can be set for **Acc1(k)**. This is not a practical solution since a different overflow threshold is required for each $\mathbf{K_i}$ setting.

k=k4: **Acc1(k)** drops to the point where **Fout1(k)** in no longer in saturation. **Fout1(k)** starts to fall, but there is a big overshoot at the system output controlled by **Fout1(k)** because of the additional recovery time.

k>k5: System output controlled by **Fout2(k)** is already at steady state. If the system is controlled by **Fout1(k)**, then it takes a long time to reach steady-state after the big overshoot.

Improper handling of saturation conditions will cause large overshoots during the transient response as seen above. Note that, having an unnecessarily high overflow limit for an accumulator may adversely effect the saturation recovery in a control system.

There is another advantage of using the second calculation method. If the first method is used, then changing $\mathbf{K_i}$ during steady state operation may cause a big jump at the controller output. These large transients may damage the system while somebody is adjusting gain factors trying to optimize controller response. If the second method is used, then changing $\mathbf{K_i}$ will affect the rate of accumulation but it will not cause a sudden change at the controller output.