EE443 - Embedded Systems

Lecture 7
Interrupts

Contents:

The first section of this lecture describes how and when the interrupts are preferable over polling techniques. The following sections explain how the interrupt mechanism works. The second half of the lecture notes, starting with the section "Management of Interrupts," describes the common problems and their solutions in typical microcontroller applications that utilize interrupts.

7.1 Polling Versus Interrupts

A microcontroller working in an embedded system is required to interact with the environment. These interactions require managing three types of tasks besides performing routine arithmetic, logic, and I/O operations on a microcontroller:

Responding to external events: Usually external events can happen anytime. The microcontroller has neither any control on these events nor any prior information about their timing.

Timing arrangements: These tasks require setting frequency, time interval, or duty cycle of some microcontroller actions with the required timing precision.

Time-related measurements: These tasks require timing precision depending on the system functions.

The program outline given below attempts to manage several of these tasks in a loop that is continuously repeated. The processor checks the status of all events one by one in every loop cycle, and it does not do anything else until an I/O device needs attention. This type of programming method for managing external interactions is called "**polling**" or "**software driven I/O**".

```
Main Program:
  Initialize peripherals
  Initialize tasks
  PollingLoop:
      Check keypad/keyboard:
        If key pressed
            Wait until the key signal is stabilized
            Process user entry
      Check position encoder:
        If encoder output changed from 0 to 1
            Increment pulse counter
      Check USART buffer:
        If data/command received
            Process the received data/command
      Check LoopCounter for ADC sampling time:
        If need to sample ADC
            Start ADC
            Wait for ADC conversion
            Read ADC
      Check motor speed:
        If encoder output is toggled
            Calculate speed error
            Calculate and set the driver output
      Increment LoopCounter
  Jump PollingLoop
```

If all of these operations are performed in a polling loop, then we face the following difficulties:

1. Processor may not respond to all events on time. It is not possible to assign priorities to the events. If an event occurs right after the processor checks for the related activity, then the response to that event will wait until the entire polling loop is executed one more time.

Examples:

`Check position encoder`: The processor is required to catch all the pulses received from the encoder. If the pulse rate is 1000/s, then the processor should check the encoder at least twice in every millisecond time period.

`Check USART buffer`: If the data rate is 1KByte/s, then the processor should check the UART at least once every millisecond. In case the USART has an input buffer, the processor should read the received data before the buffer gets full.

2. Processor time is wasted. If a significant part of the processing power is used for polling or generating delays, then we may end up using a faster processor just to catch up with the polling requirements.

Examples:

`Wait until the key signal is stabilized`: The processor will wait several milliseconds in a delay loop. If the **`LoopCounter`** is used for the delayed operation then this will require additional computation.

`Wait for ADC conversion`: The processor keeps checking ADC status or waits in an idle loop without doing anything else.

3. Timing accuracy will be limited. We will struggle with counting instructions and timing calculations to arrange the timing of system operations. Cycling through the polling loop will take longer when one or more of the events require processor response.

Examples:

`Check LoopCounter for ADC sampling time:` Sampling intervals cannot be arranged accurately while the loop cycle time changes randomly.

`Check motor speed:` If better than 100 μs accuracy is required in measurement of motor revolution time, then the processor should check the encoder output at least once every 100 μs. If a timer is used for the measurement, then how long does it take to restart the timer? How much does the response time vary depending on the polling loop activity?

4. Processor always needs full power. Some microcontrollers can be put in a "*sleep mode*" or "*power-save mode*" to reduce the power consumption by several orders of magnitude. The microcontroller can be programmed to wake up and resume normal operation mode after an external event. If all external interactions rely on the polling loop activity, then the microcontroller must be kept awake always.

As an alternative to the polling method, the processor hardware can be arranged to respond to **interrupts** that are external asynchronous signals. The interrupt response mechanism temporarily redirects the processor to a different location in the program memory. The processor starts to execute a procedure to perform the required task in response to the interrupt request. This response procedure is referred to as **interrupt service routine (ISR)** or **interrupt handler**. The processor returns back to the point where the interrupt request is received and resumes the regular program execution after it executes the "**`RETURN`**" instruction that terminates the ISR. A regular procedure or function is activated with a "**`CALL`**" operation at a predetermined location in the program. On the other hand, execution of an ISR can be triggered by an interrupt signal anytime or anywhere during the regular program flow. Due to this fact, an ISR cannot receive any parameters or it cannot return a result. Data transfers to/from ISRs are arranged through memory buffers or similar data structures.

These are the major programming tasks required for utilization of the interrupt mechanism:

1. Write an ISR for every interrupt input, that performs the operations required for the corresponding interrupt condition.
2. Make the necessary arrangements so that the processor can call the ISR corresponding to each interrupt condition. The details related to interrupt vectors are explained in the next section.
3. Arrange the priority of interrupt requests and other options by using the interrupt controller or other programming methods.

7.2 Interrupt Calls

An ISR is written for every interrupt input available in the microcontroller. Interrupts may occur at any time during the regular code execution. The processor must automatically call the ISR written for a particular interrupt input after the interrupt is activated. An **interrupt controller** is necessary to manage these automatic call operations. The interrupt controller is a hardware unit that has several functions:

- Receive all interrupt requests and generate the necessary control signals for the processor to initiate the interrupt calls with proper timing.
- Determine the target address of the interrupt call.
- Make decisions on when to initiate an interrupt call depending on the interrupt priorities and other conditions. The processor may disable the interrupt calls during execution of critical instructions. An interrupt request must wait while the processor is busy serving another interrupt.
- Track the status of pending interrupts. An interrupt request may not be served immediately if the interrupt calls are disabled or if the processor is busy because of some other reason. The interrupt controller starts the call for a pending interrupt when it is allowed.

The interrupt controller should have a mechanism to store the addresses of all ISRs and give the correct ISR address to the processor quickly and efficiently. **Interrupt vectors** are collection of target addresses or collection of simple branch instructions that are used to direct the code execution to the target ISRs during interrupt calls. The name "*interrupt vector*" is used since it can be seen as an arrow that points to the beginning address of the ISR code written for a particular interrupt. There are two common methods for a processor to obtain the target address of an ISR.

1. Interrupt Vectors Stored by the Interrupt Controller:

The interrupt controller keeps the addresses of ISRs and provides the target address to the processor whenever an interrupt call is required.

Initialization: Processor writes the target addresses of all ISRs to the interrupt controller during system initialization. Processor may also send interrupt priority settings and other options to the interrupt controller as part of the initialization.

Interrupt Call:
1. Interrupt controller initiates the interrupt call, and sends the target ISR address to the processor.
2. Processor saves the return address and directly calls the ISR at the target address provided by the interrupt controller.

2. Interrupt Vectors Stored in Predetermined Addresses:

The target addresses of interrupt calls are **hardwired** (determined by hardware, and cannot be changed by software) in this method. Interrupt controller is not responsible for storing ISR addresses. All interrupt calls are directed to a reserved memory section that is usually located at the very beginning of the program memory (i.e. starting at **0x0000**). A few bytes of program memory are reserved for every

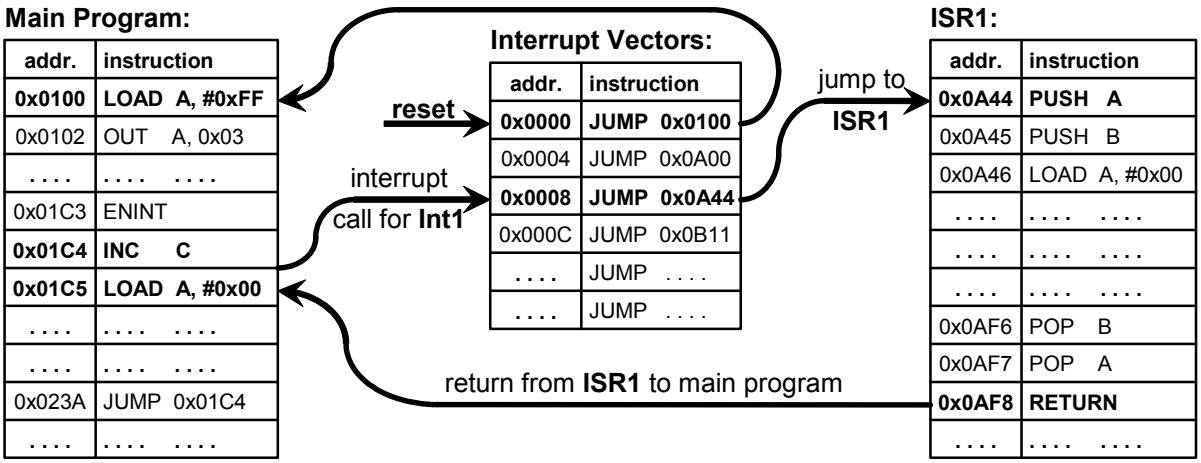
interrupt call just enough to write a **JUMP** or **GOTO** instruction that redirects the call to the actual ISR address.

Initialization: Processor program includes the branch instructions that target ISR addresses at the memory section reserved for the interrupt vectors. The assembler or the compiler determines the ISR addresses, and it places the **JUMP** instructions that will serve as interrupt vectors when the machine code is generated. Hence, the target ISR addresses are readily available when the executable code is downloaded into the program memory.

Interrupt Call:
1. An interrupt request results in an interrupt call to the predetermined address corresponding to the interrupt source (i.e. Int0 => 0x0004, Int1 => 0x0008, Int2 => 0x000C, **. . .**).
3. Processor saves the return address, and it executes the first instruction at the predetermined address. This instruction is a **JUMP** or **GOTO** instruction that **redirects the interrupt call** to the actual ISR address.

The following figure shows the program memory organization for redirection of interrupt calls. The first interrupt vector at the address **0x0000** is usually reserved for the reset condition. This JUMP instruction takes the processor to the beginning of the main program for initialization of all system functions after the reset. In the figure, **Int1** interrupt is received during execution of the instruction at **0x01C4**. The return address (**0x01C5**) is saved on the stack and the program counter is set to the predetermined target address (**0x0008**) automatically. The ISR written for the **Int1** interrupt starts after execution of the **JUMP 0x0A44** instruction.

Memory Organization for Redirection of Interrupt Calls



The redirection of interrupt calls is more common in microcontrollers. Addressing of ISRs can be handled by the compiler or the assembler, and the interrupt vectors are readily available as part of the generated machine code. In that case, a program developer is not required to arrange ISR addresses and write the code to initialize the interrupt vectors stored with an interrupt controller.

Several alterations of these methods can be found in the large variety of microcontroller architectures available in the market. For example, a single predetermined address can be assigned to all interrupts. The processor executes a ***master ISR*** that checks the status of all interrupt inputs to determine the interrupt source first. Once the interrupt source is identified, the processor can redirect the interrupt call to the target ISR corresponding to the interrupt source.

7.3 Interrupt Sources

This section gives a list of common interrupt sources and typical operations performed in the ISRs related to service tasks.

1. Counting external events: An interrupt input can be utilized for counting pulses or rising/falling edges generated by an external device. This device can be an encoder used for position sensing or a sensor switch. The ISR simply increments or decrements a variable at every interrupt call. We can eliminate some hardware components necessary for the system operations by replacing these components with interrupt functions.

2. Keypad/keyboard: An interrupt is received every time a key is pressed. ISR reads the related information from the keypad or keyboard (i.e. the code that identifies the key that was activated). The microcontrollers designed to support a keypad interface provide a "**pin-change interrupt**" feature. A pin-change interrupt call occurs when there is a status change on one of several input pins that are connected to the keypad.

3. Timer used in programmed mode: The processor sets the count limit or initializes the counter and enables the count operation for a programmed delay period. The timer generates an interrupt when the count limit is reached at the end of the delay period. The ISR can perform any task that requires a timing arrangement, such as, activating ADC sampling or toggling an output signal. Usually the timer is re-programmed or re-started in the same ISR to initiate another delay period.

4. Timer used in gated or triggered mode: The timer is used to measure the duration of an external event, such as, revolution time of a motor or duration of a time-encoded signal. The timer generates an interrupt at the end of the measured period. The ISR reads the timer count, re-starts the timer, and calculates an output response for the measured input, if necessary.

5. External interrupt for time measurement: In the previous example, a timer receives the external signal at its gate or trigger input and the processor receives an interrupt from the timer. You can make the same measurement when the processor receives an interrupt directly from the external signal. The ISR reads the timer count and re-starts the timer as in the previous example. In this case, the timer is used as a ***stop-watch*** where the timer operation is controlled by the processor commands instead of an external gate or trigger signal.

6. ADC: If the microcontroller has a slow ADC, processor time is wasted while checking ADC status continuously until each conversion is finished. ADC can issue an interrupt when the conversion result is ready, and the ISR serving the ADC interrupt reads the ADC output data. The same ISR can process the ADC sample or it can store the sample in memory, making it available to other program modules.

7. Universal Synchronous/Asynchronous Receiver Transmitter (USART): A USART supports communication through the well-known serial interfaces such as RS-232 and RS-422. A USART interface can generate an interrupt every time a single byte is received or a portion of the input FIFO buffer is filled with incoming data. The ISR reads the USART buffer and it can store the incoming data in a larger memory buffer. The critical function of the ISR is to grab the data stored in the

USART buffer before it becomes completely full. Otherwise, the USART buffer contents may be lost when they are overwritten by new incoming data.

A USART interrupt can be utilized for data transmission as well as data reception. Sending one byte of data takes nearly **1 ms** at **9600** baud rate. Similar to the slow ADC example given above, a processor can easily waste 10,000 instructions if it waits for **1 ms** before sending another byte through the USART. A sequence of bytes can be queued in a buffer instead of waiting while each byte is transmitted. The USART module generates an interrupt when it is ready to transmit the next byte, and the corresponding ISR sends the queued data one byte at a time. If necessary, the processor can disable the USART transmit interrupt when there are no more data queued for transmission. The transmit interrupt can be enabled again when more data are queued in the buffer.

7.3.1 Edge-Sensitive and Level-Sensitive Interrupts

All interrupt inputs are received by the interrupt controller in a microcontroller. The choice between edge-sensitivity and level-sensitivity determines the way an interrupt signal is treated internally. The interrupt mechanism of a microcontroller provides interrupt flags indicating the status of interrupt requests. The processor can check the status of interrupt requests by reading interrupt flags without actually serving the requests. These flag bits can be put together in one or more interrupt status registers or they may be stored separately in SFRs that are part of the peripheral units. The flag behavior and the processor's response to interrupt requests depend on the sensitivity type as explained below.

Level-Sensitive: An interrupt call can be started while the interrupt input is at the active level. An active-low (0 means interrupt) or an active-high (1 means interrupt) input can be recognized as long as the interrupt source keeps the signal at the active level. A level-sensitive interrupt flag is cleared when the external interrupt signal becomes inactive. One should be aware of the following issues related to level-sensitive interrupts:

1. The interrupt source may cancel the interrupt request before it is served by the processor. The processor may miss a level-sensitive interrupt if it is canceled before it is served.

2. A level-sensitive interrupt must be cleared at the source. Otherwise, interrupt requests are repeated as long as the interrupt signal remains active.

Edge-Sensitive: The internal interrupt flag of an edge-sensitive interrupt is typically the output of a flip-flop or a latch that is set right after a rising or falling edge is received at the external interrupt input. The active edge is determined by the interrupt hardware as indicated in the microcontroller specification or one of the two options can be selected depending on the programmable features. An edge-sensitive interrupt flag remains active until the interrupt flag is cleared by the processor or the corresponding ISR is called.

The following table compares the interrupt response for the two sensitivity options after a request is sent by an external interrupt source:

Action	Level-Sensitive:	Edge-sensitive:
Interrupt request	The source must keep the request line at the active level until the interrupt is served.	Interrupt request is stored after the active edge. A short pulse is sufficient to initiate an interrupt call.
Setting interrupt flag	Set as long as the request line from the interrupt source is active.	Set after the active edge, and it is not cleared until the interrupt is served.
Serve the interrupt	ISR must perform the required service or at least clear the status at the interrupt source to remove the interrupt condition.	ISR may proceed without performing the service task if it is not required. If the source still needs service then it should send a request again.
Clearing interrupt flag	Internal interrupt flag is cleared right after the external interrupt signal becomes inactive.	Interrupt controller or processor must clear the internal interrupt flag.

7.3.2 Multiple Sources On a Single Interrupt Input

A single interrupt input can be shared by more than one interrupt source as long as the ISR serving the interrupt is properly written to respond all possible interrupt requests. The first step is to determine the origin of the interrupt among the sources driving the interrupt input. The processor polls the possible interrupt sources in the ISR and performs the necessary operations to respond the interrupt request(s) according to the origin of the interrupt.

Polling all possible interrupt sources in a single ISR is not as efficient as using a separate ISR for each source, but it is certainly better than polling continuously without any interrupt mechanism. If we merge multiple interrupt signals to drive a single interrupt input, then polling takes place only when one of the interrupt sources needs service.

7.4 Management of Interrupts

So far we have seen the good things about interrupts:

- ✓ Immediate response to external events.
- ✓ No more wasting of processing time while polling for service demands. The same system performance can be obtained using a simpler and/or slower processor.
- ✓ Better timing precision in applications using timers.
- ✓ Reduction of hardware components utilizing interrupt mechanism for counting, data buffering, and similar purposes.
- ✓ Arrangement of sleep or power-save modes in a microcontroller through the interrupt mechanism.

The main trouble in using interrupts is the fact that processor behavior is no longer deterministic. In other words, processor does not follow a well-defined sequence of instructions as it does in a simple polling loop. Interrupts redirect program flow to ISRs, and this redirection can occur at any time while the interrupts are enabled. **An interrupt call may occur between any pair of machine instructions.** A programmer should always keep in mind that a simple statement written in a high-level language may correspond to a number of instructions in the machine code. For example, the increment statement given below,

```
Count ++;  // increment Count by 1
```

will be replaced by three machine instructions in the executable code generated for a typical processor:

```
LOAD   A, Count;  // load the variable into register-A
ADD    A, #1;     // increment register-A
STORE  A, Count;  // write the result back to memory
```

If two program modules access a common variable then interruption of operations involving that variable will cause errors. Two examples of these type of errors are given in the following.

Example 1: Main program and an ISR modify a one-byte variable
In this example, **`Count`** is a one-byte variable modified in the main program and in an ISR.

Main program:

```
- - -
Count ++;
- - -
- - -
```

ISR for Int1:

```
- - -
if (Count > 0)
{ Count --;
- - -
```

Assume that the initial value of **`Count`** is **5**. If both of the modules increment and decrement **`Count`** only once then the final value of **`Count`** is expected to be **5** again. An error will occur if the sequence of events is as follows.

- **`Count`** is loaded into a register-**A** in the main program.
- An interrupt is received before the main program stores the incremented **`Count`** value back to the memory.
- ISR for Int1 starts, loads the initial **`Count`** value from the memory, decrements it, and writes back **4** to the **`Count`** location in memory.
- ISR restores the register contents and returns back to main program.

- Main program resumes execution while the **Count** value in register-**A** is still **5**.
- Main program increments register-**A** and stores **6** back to the memory.

The final value of **Count** stored in memory is **6** instead of **5** because the main program ignores the decrement operation in the ISR and overwrites the calculated result.

Example 2: Main program reads a two-byte variable and an ISR modifies it
In this case, consider the addition of **16-bit** numbers with the following statement.

```
Sum += Count;  // add Count to Sum
```

An 8-bit processor performs two addition operations to complete a 16-bit addition:

```
LOAD   A, SumL;    // load the least significant 8 bits into register-A
ADD    A, CountL;  // add the least significant 8 bits of Count
STORE  A, SumL;    // write the result back to memory
LOAD   A, SumH;    // load the most significant 8 bits into register-A
ADDwC  A, CountH;  // add the most significant 8 bits of Count together
                   // with the carry generated in the previous addition
STORE  A, SumH;    // write the result back to memory
```

In the following example, **Count** is a 16-bit common variable that is read in the main program and modified in an ISR.

Main program:

```
- - -
Sum += Count;
- - -
```

ISR for Int1:

```
- - -
Count ++;
- - -
```

Assume that initially **Sum** is **0**, and **Count** is **255** (**0x00FF**). If an interrupt is received after the addition in the main program then the result of **Sum += Count;** will be **255** (**0x00FF**). If an interrupt is received before the addition then the result will be **256** (**0x0100**). Both of these results are acceptable, but if an interrupt is received in the middle of the addition then an error will occur as described below.

- The least significant 8 bits are added and the result (**255=0xFF**) is stored back to **SumL** location in the memory.
- An interrupt is received before the main program adds the most significant 8 bits of **Count**.
- ISR increments **Count**, and stores **256** (**0x0100**) back to the memory.
- Main program resumes execution after the ISR returns, adds the most significant 8 bits of **Count**, and stores the result (**0x01**) back to **SumH** location in the memory.

The final 16-bit value of **Sum** stored in the memory is **511** (**0x01FF**) which is 100% larger than the expected result.

Interruption of the statements involving shared variables should be prevented. In the following code, **Int1** interrupt is disabled right before **Count** is read, and it is enabled again after the addition operation.

Main program:

```
- - -
<disable Int1 interrupt>
Sum += Count;
<enable Int1 interrupt>
- - -
```

ISR for Int1:

```
- - -
- - -
Count ++;
- - -
- - -
```

If an interrupt is received while it is disabled then execution of the corresponding ISR will be delayed until the interrupt is enabled again. More information on minimization of delays can be found in the section on *interrupt response time*.

The following are the typical problems that may originate from the processor behavior which is not deterministic as a result of interrupt calls.

1. Sharing memory: Remember that an ISR cannot accept any parameters and it cannot return any results. If a system uses interrupts then the only way to exchange data between ISRs and the other program modules is to have shared memory locations. The programmer had to make sure that there will not be any conflicts or data transfer failures while the program modules accessing the shared memory locations as in the examples given above.

2. Sharing resources: The same considerations regarding memory access are also applicable to the resources (i.e. ADC, DAC, USART) that are shared by other ISRs or other program modules. For example, an ADC operation started in the main program may be overwritten by an ISR if an interrupt is received before the main program reads the ADC output. In a similar way, data sent through a USART may become corrupt if an ISR takes the USART control while another program module is in the middle of a transmission.

3. An interrupt is received while executing an ISR: If the second interrupt request is very urgent, then the processor should make another interrupt call while it is in the middle of the first ISR. Otherwise, the second interrupt call will wait until the first ISR returns. The necessary priority management methods should be established, so that the processor or the interrupt controller can make decisions depending on how urgent is an interrupt request.

4. It takes too long to execute an ISR: Nesting several interrupt calls (starting ISRs while running ISRs) or having too many pending interrupts (interrupts waiting to be serviced) is not desirable. These may cause problems even if the interrupt priorities are arranged properly. Processing tasks that are not required to be completed in an ISR can be moved to the main program. If an ISR returns quickly, then it is less likely to receive another interrupt request during execution of the ISR.

7.4.1 Interrupt Priority

All microcontrollers provide some options or programmable features to set the priority of interrupt calls. In a typical microcontroller, all interrupt controller options are set by writing necessary commands into special function registers (SFRs). The following is a list of common settings that allow management of the interrupt related operations in a microcontroller:

1. Enable / Disable global interrupts: Disabling global interrupts blocks all interrupt calls until they are enabled again. Some processors have dedicated instructions to enable/disable interrupts independent of the interrupt controller. Others require setting or clearing a bit in an SFR provided for the interrupt controller. Disabling interrupts does not mean that the interrupt requests are ignored forever. Interrupt requests are received while the interrupts are disabled, but the ISR calls are delayed until the interrupts are enabled again.

2. Mask individual interrupts: An interrupt controller may enable/disable interrupt calls for each interrupt independent of the other interrupts. This is achieved by setting an SFR bit for an interrupt input that masks the corresponding interrupt

activity. In some devices the interrupt input can be completely ignored while it is being masked. As an alternative, an interrupt condition can be latched to initiate the interrupt call after the mask is removed.

3. Set interrupt priority: An interrupt controller is likely to have several options and several limitations for setting interrupt priorities. The programmer must read the microcontroller specification and/or technical manual to find out these options and limitations. This information is also critical for design of the system hardware while assigning interrupt inputs to external devices. It is necessary to connect every interrupt source to the correct interrupt input according to the priority requirements. Several options for setting interrupt priorities that can be found in a variety of microcontrollers are described below.

- Significance of interrupt priority may vary from one microcontroller to another. Interrupt priority determines the interrupt that will be served first when there are two or more pending interrupts. In some microcontrollers, an interrupt with higher priority can also interrupt a lower priority ISR. If this is not supported by the interrupt hardware, a programmer can use one of the software options given below to allow other interrupt calls in an ISR.
- The default priority levels assigned to all interrupts are hardwired (determined by the hardware). In some microcontrollers, interrupt priorities can be assigned by software. Usually, there is an SFR bit for each interrupt input that selects between high and low priority levels.
- In simple Microchip PIC microcontrollers, there is a single target ISR address for all interrupts. A *master ISR* at this address can determine the interrupt source and it can make decisions on the priority of interrupt calls. All priority options can be controlled manually by the programmer.
- In Atmel AVR microcontrollers, interrupt controller automatically disables all interrupts after an interrupt call is initiated. Interrupts are enabled again when a special "*Return From Interrupt*" instruction is executed at the end of an ISR. As a result, other interrupts are not allowed by default until the currently executed ISR returns. The programmer can manually enable interrupts in the middle of an ISR to allow other interrupt calls after that point.
- Interrupt flag bits indicate the status of interrupt requests. Clearing an interrupt flag indicates that the interrupt request has been served, but it does not necessarily mean the ISR execution is complete. If an interrupt controller makes decisions based on the state of interrupt flags then other interrupt calls can be allowed clearing the flag of an ISR. Once the processor clears the interrupt flag, this may allow other interrupt calls even if they have lower priority.

There are alternative methods for managing interrupts as described above. In simple systems the processor itself can make the decisions related to interrupts looking at the state of interrupt flags provided in a single SFR. Complex interrupt controllers on the other hand, are equipped with several SFRs indicating pending interrupts (interrupts waiting to be served), interrupts that are currently being served, and other interrupt status and control information.

In the context of this document, an interrupt that has ***higher priority*** can interrupt the ISR of a ***lower priority*** interrupt. This means, another interrupt call can be initiated while the lower priority ISR is running. The decisions related to management of interrupts are explained considering a relatively simple control mechanism. An interrupt flag is set when one of the interrupt inputs is activated, and

it remains high until the interrupt is served or it is cleared by the processor. Interrupt controller makes decisions according to the state of these flags and the incoming interrupt signals.

7.4.2 Sharing Resources

An embedded system needs several resources that can be a peripheral unit in the microcontroller or an external hardware device. Conflicts may occur when two program modules share a resource. In the example given below, two ISRs use an ADC for sampling analog signals received at different input channels.

ISR1 for Int1:

```
100  Push A
101  <select ADC input-1>
102  <start ADC>
103  <wait for conversion>
104  Read  A, ADCout
105  Store  A, Data1
...  <process the ADC sample
...   stored in Data1>
...  ---
130  <clear Int1 flag>
131  Pop A
132  Return
```

ISR2 for Int2:

```
200  Push A
201  <select ADC input-2>
202  <start ADC>
203  <wait for conversion>
204  Read  A, ADCout
205  Store  A, Data2
...  <process the ADC sample
...   stored in Data2>
...  ---
240  <clear Int2 flag>
241  Pop A
242  Return
```

We do not expect any conflicts if we know for sure that another interrupt call may not occur while running one of the ISRs. On the other hand, if the interrupt served by **ISR2** has a higher priority, then an interrupt call to **ISR2** may occur between any pair of instructions in **ISR1**. **ISR1** will read the wrong ADC sample if **Int2** interrupt is received at any point after line-**101** and before line-**104** in **ISR1**:

1. **ISR1** selects the ADC input-1 and starts ADC for the next conversion.
2. **Int2** interrupt starts **ISR2** before **ISR1** reads the conversion result.
3. **ISR2** selects the ADC input-2 for the next conversion and reads the ADC sample for input-2.
4. **ISR2** returns and **ISR1** resumes execution.
5. **ISR1** reads the ADC sample for input-2 instead of input-1.

The easiest way to prevent this conflict is to modify **ISR1** blocking the interrupts during execution of the critical instructions:

ISR1 for Int1:

```
100  Push A
101  DisInt   // disable interrupts
102  <select ADC input-1>
103  <start ADC>
104  <wait for conversion>
105  Read  A, ADCout
106  Store  A, Data1
107  EnInt    // enable interrupts
...  <process the ADC sample
...   stored in Data1>
...  ---
132  <clear Int1 flag>
133  Pop A
134  Return
```

No interrupt calls are allowed once the interrupts are disabled until they are enabled again. Note that **ISR2** will wait until **ISR1** reads the ADC sample, although **ISR2** has a higher priority. It is important to keep the duration of any disabled interrupts at the minimum to avoid unnecessary delays in serving high-priority interrupts. If the system uses other high-priority interrupts then it is preferable to disable only the interrupts that interfere with the ADC usage. In that case, the global interrupt control commands (**`DisInt`** and **`EnInt`**) should be replaced by the statements that disable and enable only the **Int2** interrupt.

The interrupt controller may not support interruption of an ISR by higher-priority interrupts depending on the microcontroller family being used. If the nested interrupt calls are not supported by the hardware, then **ISR2** interrupt calls can be enabled in the middle of **ISR1** by using other programming options as shown in the following two cases.

Case 1: Interrupts are disabled automatically when ISR1 is called

ISR1 for Int1:

```
100  Push A
101  <select ADC input-1>
102  <start ADC>
103  <wait for conversion>
104  Read  A, ADCout
105  Store  A, Data1
106  EnInt
...  <process the ADC sample
...   stored in Data1>
...  ---
131  <clear Int1 flag>
132  Pop A
133  Return
```

Case-2: Interrupt controller decides to call **ISR2** depending on **Int1** flag status

ISR1 for Int1:

```
100  Push A
101  <select ADC input-1>
102  <start ADC>
103  <wait for conversion>
104  Read  A, ADCout
105  Store  A, Data1
106  <clear Int1 flag>
...  <process the ADC sample
...   stored in Data1>
...  ---
132  Pop A
133  Return
```

In the first case, the interrupt controller automatically disables all interrupts after **ISR1** is called. If **Int2** interrupt is received after **ISR1** is called then by default, **ISR2** call is delayed until **ISR1** returns and the interrupts are enabled again. The **`EnInt`** instruction is added to enable other interrupt calls after that point. In the second case, the interrupt controller holds the **ISR2** call while the **Int1** flag is active. The **`<clear Int1 flag>`** statement is moved from end of **ISR1** to the point where **ISR1** no longer needs the ADC. The interrupt controller will allow other interrupt calls regardless of their priority after the **Int1** flag is cleared. If there is a pending **Int2** interrupt at that point then **ISR2** will be called. The **Int1** and **Int2** priorities assigned in the interrupt controller are not important in both cases. **ISR2** can be called before **ISR1** returns even if **ISR2** has a lower priority

7.4.3 Data Flow Control

ISRs must use shared variables or data buffers to transfer data to/from other program modules. In the example given above, **ISR1** processes the ADC sample after reading and storing it in memory. If the processing of ADC samples can wait until the microcontroller returns back to the main program, then we can remove this task from **ISR1** and place it into the main program as follows:

Main program:

```
...  ---
050  MainLoop:
...    ---
060    Load  A, Data1Ready
061    JumpIfZero NoData1
062      Load  A, Data1Low
063      Load  B, Data1High
...      <process the ADC sample
...       in A and B registers>
080      Load  A, #0x00
081      Store A, Data1Ready
082    NoData1:
...    ---
090  Jump MainLoop
```

ISR1 for Int1:

```
100  Push A
101  <select ADC input-1>
102  <start ADC>
103  <wait for conversion>
104  Read  A, ADCoutL
105  Store A, Data1Low
106  Read  A, ADCoutH
107  Store A, Data1High
108  Load  A, #0x01
109  Store A, Data1Ready
110  <clear Int1 flag>
111  Pop A
112  Return
```

ISR1 reads the ADC sample, stores it in memory, and then sets the **Data1Ready** flag. The **Data1Ready** flag is a simple one-byte variable that takes the value of either **0x00** or **0x01**. The main program checks the value of **Data1Ready** in the polling loop. If **Data1Ready** is set, then the main program reads the stored ADC sample and performs the necessary calculations. Main program clears **Data1Ready** to avoid making the calculations on the same ADC sample over and over again in the main loop. The flags like **Data1Ready** that are used for controlling data transfers between program modules are called "**semaphores**".

In the last example an ADC sample is two bytes that cannot be read or stored in a single memory cycle on an 8-bit microcontroller. The main program uses two load instructions to get the ADC sample into registers one byte at a time. If an **Int1** interrupt call occurs between lines **062** and **063**, then **ISR1** will get a new ADC sample before the main program loads the stored ADC sample completely. The main program will end up using the low and high bytes from two different ADC samples in the calculations. One possible solution is to block the **ISR1** interrupt call while the main program loads the stored ADC sample:

Main program:

```
...  ---
050  MainLoop:
...    ---
060    Load  A, Data1Ready
061    JumpIfZero NoData1
062      <disable Int1 interrupt>
063      Load  A, Data1Low
064      Load  B, Data1High
065      <enable Int1 interrupt>
...      <process the ADC sample
...       in A and B registers>
080      Load  A, #0x00
081      Store A, Data1Ready
082    NoData1:
...    ---
090  Jump MainLoop
```

ISR1 for Int1:

```
100  Push A
101  <select ADC input-1>
102  <start ADC>
103  <wait for conversion>
104  Read  A, ADCoutL
105  Store A, Data1Low
106  Read  A, ADCoutH
107  Store A, Data1High
108  Load  A, #0x01
109  Store A, Data1Ready
110  <clear Int1 flag>
111  Pop A
112  Return
```

The **Int1** interrupt is disabled while the main program safely loads the stored data. Note that the main program should not read **Data1Low** or **Data1High** after the **Int1** interrupt is enabled. The data loaded into A and B registers should be used

throughout the processing of ADC sample. Otherwise, the sample value may change in the middle of the processing when **ISR1** stores a new sample.

An alternative solution is obtained by checking the semaphore status in **ISR1**. The **Data1Ready** flag can be used to carry start/stop commands in two directions as shown below.

Main program:

```
...  ---
050  MainLoop:
...    ---
060    Load  A, Data1Ready
061    JumpIfZero NoData1
062      Load  A, Data1Low
063      Load  B, Data1High
...      <process the ADC sample
...       in A and B registers>
080      Load  A, #0x00
081      Store A, Data1Ready
082    NoData1:
...    ---
...    ---
090  Jump MainLoop
```

ISR1 for Int1:

```
100  Push A
101  Load  A, Data1Ready
102  JumpIfNotZero SkipISR1
103    <select ADC input-1>
104    <start ADC>
105    <wait for conversion>
106    Read  A, ADCoutL
107    Store A, Data1Low
108    Read  A, ADCoutH
109    Store A, Data1High
110    Load  A, #0x01
111    Store A, Data1Ready
112  SkipISR1:
113  <clear Int1 flag>
114  Pop A
115  Return
```

If an **Int1** interrupt is received while the main program is still processing the last ADC sample then **ISR1** returns without reading a new ADC sample.

Although these methods prevent the conflicts between program modules they cannot provide complete solutions always. Some applications require sampling and processing of inputs at regular time intervals with precise timing specifications. The processor should be fast enough to complete all periodic processing tasks. Delaying or skipping sampling of inputs is not an acceptable solution in these applications. Process control equipment used in industrial applications and transportation vehicles are the common examples of the systems that have strict timing requirements.

Circular queues and other types of data buffers are the flexible tools in controlling data flow when immediate processing of inputs is not required. For example, ADC samples acquired for data logging or display purposes can be stored in a circular queue before they are transmitted to another computer. In this type of applications a counter variable tracks the number of buffered data elements, and it serves as a semaphore between the program modules. One of the modules increment the counter after it stores a new data element in the buffer. The receiver module checks the counter to find out if there is any data in the buffer available for processing.

7.5 Interrupt Response Time

Although the interrupt mechanism provides better timing accuracy compared to polling, a processor cannot respond to an interrupt request immediately. **Interrupt response time** is the time delay from the activation of an interrupt input to the service of the interrupt request. We should take a closer look at the series of operations that take place during an interrupt call to be able to estimate the interrupt response time.

1. An interrupt source (keyboard, timer, USART, etc.) needs service and activates its interrupt request signal.

2. Interrupt controller receives the request at one of the interrupt inputs (name it **Int1**) and sets the corresponding flag according to the specified interrupt sensitivity (edge-sensitive or level-sensitive).

3. Interrupt controller makes the following checks before it initiates the interrupt call:

- **a)** Check if global interrupts are enabled. If they are disabled, then **Int1** interrupt call will wait until the interrupts are enabled.
- **b)** Check if the **Int1** interrupt is enabled. If it is disabled or masked then **Int1** interrupt call will wait until it is enabled.
- **c)** If there are other active interrupts at the moment, then check if **Int1** has the highest priority among them. If there are no other active flags, or **Int1** has the highest priority, then initiate the interrupt call for **Int1**.

4. Processor completes the instruction currently being executed before it starts the interrupt call. Note that the interrupt controller performs the checks listed above continuously. If the last instruction executed by the processor disables interrupts, then the interrupt call will be canceled.

5. Processor saves the return address in the program counter in the stack memory and loads the interrupt call address for **Int1** into the program counter. If redirection of the interrupt call is required then the processor executes a **JUMP** instruction to the starting address of the ISR.

6. Processor starts executing the ISR. Any status information and the register contents are saved in the memory (usually on the stack) at the beginning of the ISR code.

7. The main part of the ISR code performs the service task.

8. The interrupt controller automatically clears **Int1** flag right after the interrupt call, or the processor manually clears the flag by writing to the related SFR. If the **Int1** is level-sensitive, then the interrupt source must clear its interrupt signal after it receives service. Otherwise, interrupt flag will be set again right after it is cleared.

9. Processor loads back the saved status information and the registers contents before the ISR returns. The processor should be ready to resume normal operation after the ISR returns. Therefore, all processor settings should be exactly same as they were before the interrupt call.

10. Processor executes the **RETURN** instruction at the end of the ISR. It restores the saved return address into the program counter, and resumes normal program execution.

After an ISR returns, the interrupt controller waits for one more instruction to be executed before it initiates another call. This is necessary to allow the processor return back to regular program execution before starting another interrupt call.

The processor can start the actual service task after Step-6. All of the steps listed above except Step-4 take a fixed amount of time that remains the same for all ISR calls. On the other hand, the service delay due to Step-4 changes randomly from one interrupt call to another, because an interrupt request may arrive any time during the execution of an instruction. The minimum delay time due to Step-4 occurs when an interrupt request arrives right at the end of an instruction, and it is practically zero. An interrupt request received at the beginning of an instruction will be served after a longer delay. The maximum delay time is given by the execution time of the longest instruction. As an example, consider the following delay time specifications:

Step-3:	Interrupt controller decision time:	20 ns
Step-4.	Execution of the current instruction:	0 ns-200 ns
Step-5.	Execution of the interrupt call:	160 ns
Step-6.	Saving register contents in the ISR:	240 ns
Total interrupt response time:		**420 ns-620 ns**

The longest time to complete an instruction is **200 ns** in this example. A programmer can make the necessary corrections on the related timing data applying an average offset of **520 ns**. The **520 ns** offset corrects the timing errors due to the fixed delay times, but the **+/-100 ns** random variation due to **Step-4** cannot be corrected.

So far we have only considered the delay times as a result of the required operations in an ISR call. There may be other reasons that delay the execution of the required interrupt service task:

- If **interrupts are disabled or the received interrupt is masked**, then the ISR call will be delayed until interrupts are enabled or the interrupt mask is removed.
- If the **processor is serving another interrupt**, and the received interrupt does not have a higher priority, then it will wait until the current service task is completed.
- If **another interrupt request with a higher priority is received** during the ISR call, then it will delay the service task.

All of these possibilities depend on the management of interrupts. Disabling interrupts to secure the execution of several instructions may cause additional delays much longer than the execution time of a single instruction. Furthermore, such long delays cannot be compensated by adding an offset to the timing parameters since they randomly occur only once in a while. Interrupts should not be disabled for long periods of time just to obtain safe and stable system operation. Taking such radical safety measures is likely to render the entire interrupt mechanism useless. In the following code, **ISR1** call is delayed for a long time if an **Int1** interrupt is received right after it is disabled in the main program.

Main program:

```
- - -
<disable Int1 interrupt>
Sum += Count;
<more statements using Count>
- - -
<enable Int1 interrupt>
- - -
```

ISR for Int1:

```
- - -
- - -
Count ++;
- - -
- - -
- - -
- - -
```

A copy of the shared variable can be used to minimize time period where the interrupt is disabled.

Main program:

```
- - -
<disable Int1 interrupt>
CountCopy = Count;
<enable Int1 interrupt>
Sum += CountCopy;
<more statements using CountCopy>
- - -
```

ISR for Int1:

```
- - -
- - -
Count ++;
- - -
- - -
- - -
- - -
```

The main program copies **Count** into a different memory location and uses that copied value for the rest of the operations. The ISR is allowed to change **Count** right after the main program has a safe copy in **CountCopy**.

The timing precision that can be achieved by using efficient programming methods is limited. Solution to strict timing requirements lies in the microcontroller hardware. If a timer has buffered counter outputs and if it starts/stops automatically with external triggers then timing of the related ISR will not be critical when it reads timer results. Similarly, immediate response to I/O requests is not required, if buffers are provided at the I/O ports, and I/O synchronization is done by dedicated circuits.

7.5.1 Timing Errors

Random variations in the interrupt response time may cause different amounts of timing error. As an example, consider the typical period measurement application shown in the timing diagrams given below. A microcontroller measures the speed of a motor by detecting pulses generated once every revolution. A timer is used to measure the time between the rising edges of successive pulses. Every rising edge marks the beginning and end of a revolution period. Timing diagrams are given for three different cases.

Case-1: The timer used in this case is ideal for this application. At every rising edge of the input signal the timer stops counting, stores the count result in a buffer, starts a new counting period automatically, and sends an interrupt request to the processor. The processor can respond to the interrupt and read the buffered count result at any time before the next input pulse. Interrupt response time has no effect on the accuracy of the time measurements. The measurement error is limited by the accuracy of the timer clock and the quantization error depending on the clock period. If we assume that the timer has an ideal clock input, then the measurement error is given by

$$\textbf{Quantization error} = +/-T_{clk}$$

where, T_{clk} is the timer clock period.

Case-2: In this case, the timer stops counting and sends an interrupt request after a rising edge at the input, but the processor is required to restart the timer after the count result is read. We can apply an average offset to the timer result to compensate for the fraction of the period where the timer is idle waiting to be restarted:

$$\textbf{Correction offset} = \frac{T_{ISR2} + (T_{Rmin} + T_{Rmax})/2}{T_{clk}}$$

where, T_{ISR2} is the time it takes until the timer is restarted (roughly the ISR2 run time), T_{Rmin} and T_{Rmax} are the minimum and maximum interrupt response times.

The additional timing error due to the random variations in the interrupt response time is given by,

Additional random error = +/- (T_{Rmin} - T_{Rmax})/2

If the interrupt response time varies between 420ns and 620ns, then there is an additional measurement error of +/-100ns as a result of this variation.

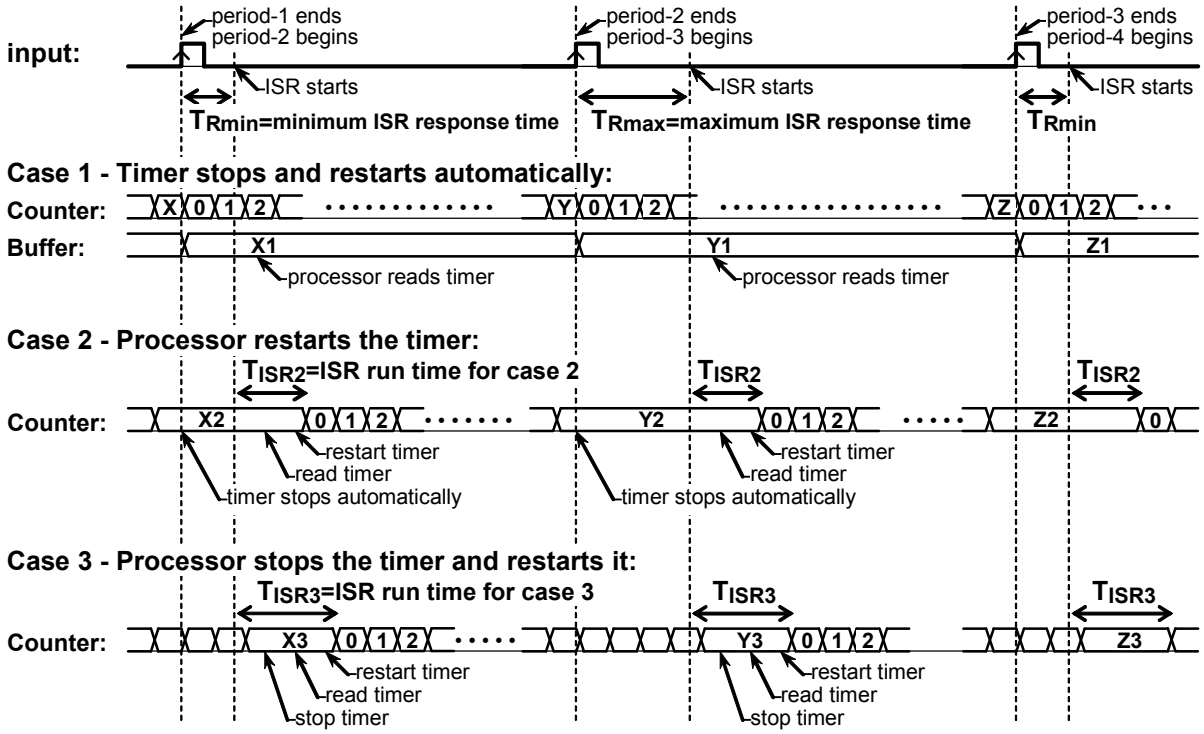
Case-3: The input signal is used directly as an interrupt input to the processor. The processor stops the timer, reads the count result, and restarts the timer in the ISR. The timer keeps counting until the ISR is called and the correction offset in this case is independent of the interrupt response time:

$$\textbf{Correction offset} = \frac{T_{ISR3}}{T_{clk}}$$

where, T_{ISR3} is the time it takes until the timer is restarted (roughly the ISR3 run time). The additional time measurement error can be twice as much compared to the case 2, since the response time variations affect the beginning and the end of each counting period.

Additional random error = +/- (T_{Rmin} - T_{Rmax})

If a count period is started **100 ns** earlier and **100 ns** later than the average response time, then the time measurement is **200 ns** longer than the actual period. Conversely, if a count period starts early and ends late, then the measurement error is **-200 ns**.



7.5.2 Register Banks

Register contents that exist at the beginning of an interrupt call should be restored when the ISR returns. The ISR should save the contents of the registers that will be used for any purpose and restore them at the end. Register banks provide a fast alternative to saving register contents in the stack or other memory locations. Each register bank contains a duplicate set of the processor registers used in data operations. A single machine instruction can select one of the register banks. As an example consider the following program modules:

Main program:

```
001  <disable interrupts>
002  UseBank0;
...  <initialize peripherals>
...  ---
030  UseBank1;
031  Load  C, #100;
032  UseBank0;
...<initialize other tasks>
...  ---
050  <enable interrupts>
051  MainLoop:
...  ---
...  ---
...  ---
192  Jump MainLoop
```

ISR for Int1:

```
100  UseBank1;
...  ---
...  ---
110  Sub  C, #1;
111  JumpIfZero Restart;
...  ---
...  ---

150  Restart:
151  Load  C, #100
...  ---
180  <clear Int1 flag>
181  UseBank0;
182  Return;
```

The main program uses register bank-0, and the ISR for Int1 uses register bank-1 in this example. The main program enables interrupts at line-**50** and starts the main loop after the initialization. All operations in ISR1 are performed on the bank-1 registers while the register contents in bank-0 are preserved. Note that the main program temporarily selects the register bank-1 in line-**30** to initialize the C register in bank-1 that is used as a counter in ISR1.

Using register banks has two advantages. First, the ISR can start executing its own task immediately, since it does not need to save and restore the register contents. A single "**UseBankN**" instruction replaces a bunch of **Push** and **Pop** instructions. Second, a subprogram or an ISR can have its dedicated set of registers. Every time ISR1 is executed, the bank-1 register contents set in the previous interrupt call are readily available in the example given above. Dedicated register banks should be reserved for the highest priority interrupts that demand efficiency and speed.

The register banks are device-specific, and their availability may vary from one manufacturer to another. Two microcontrollers that have the same instruction set architecture may have different number of register banks. A microcontroller code that relies on register banks to satisfy the speed requirements may not be ported (assembled or compiled for another microcontroller) easily.

7.6 How Fast Processor, How Deep Stack?

Calculation of the required processor speed depends on how much time is spent in every ISR in addition to the process time required for regular program execution. It is difficult to estimate the time spent for each task without writing the actual code unless someone has in-depth experience in developing similar programs on similar platforms. Assuming that the time required for each task is known, a worst-case estimate can be obtained for the processor speed. One can calculate the total process time required in unit time (i.e. in one second) using the maximum time spend for each task and the maximum frequency of the task activity:

Task Description	Time per single run	Maximum frequency	Process time for the task in 1s
Activity in main loop	500 µs	200 /s	100000 µs
ISR1	100 µs	5000 /s	500000 µs
ISR2	200 µs	1000 /s	200000 µs
Total process time required for all tasks in 1 s:			**800000 µs**

In this example, the processor will be busy running the necessary tasks in 80 % of the time. The remaining 20 % will be spent running the main loop waiting for interrupt calls or other activities.

The stack is used for storing data during subprogram calls for these purposes:
1. Saving return address to restore the program counter when the subprogram returns (i.e. 2 bytes for 16-bit address).
2. Saving register contents to be restored before the subprogram returns. A subprogram should only save the registers it uses.
3. Passing parameters to a subprogram (if using stack-based calling).

ISR calls does not involve any parameter transfers, but they require stack storage for saving return addresses and register contents just like normal subprogram calls. The total stack size required for call operations depends on the maximum number of nested calls (calls from within subprograms). Remember that the stack memory drawn with every PUSH instruction is returned back to the stack after a POP instruction. Similarly, all stack memory used during a subprogram call is returned back to the stack when the subprogram returns. There are three nested subprogram calls at maximum in the example given below:

```
MainProg:       Sub1:           Sub2:           Sub3:
----            ----            ----            ----
Loop:           Call Sub2;      ----            ----
----            ----            Call Sub3;      ----
Call Sub1;      ----            ----            ----
----            ----            ----            ----
Call Sub2;      Return;         Return;         Return;
----
GOTO Loop;
```

The longest call chain is **MainProg->Sub1->Sub2->Sub3**. If each subprogram call requires **15** bytes of stack storage then the total stack requirement is **45** bytes. The second call operation in the main program, "**Call Sub2**" does not require any

additional stack memory. All stack memory used for the previous call operations is returned back before **MainProg** calls **Sub2**.

Stack requirement for interrupt calls can be calculated in a similar way. Now consider the above example with addition of two ISRs where each ISR call requires **10** bytes of stack storage. In the worst case, an interrupt is received while **Sub3** is being executed after **45** bytes of stack memory has already been used for normal subprogram calls. If we block the interrupt calls in ISRs, then a second interrupt waits until the current ISR returns, and the total stack requirement is **55=45+10** bytes. If we allow other interrupt calls in ISRs, then the two ISR calls may be nested, and the total stack requirement is **65=45+2x10** bytes.

Nested ISR calls may occur when one of the interrupts has a higher priority, or the interrupt flag is cleared in the middle of an ISR allowing other interrupt calls regardless of their priority. Having short ISR run times can be critical to minimize the stack memory requirement as well as to avoid several interrupt request being queued in a real-time system.