Interrupts short and simple: Part 1 - Good programming practices

October 03, 2012

Editor's note: *In this first part in a series on the appropriate use of interrupts in embedded systems design , Priyadeep Kaur of Cypress Semiconductor starts with general guidelines and good practices that should be followed.*

Any embedded application generally involves a number of functions. Even a simple temperature control application, for instance, includes a number of tasks like reading the user input, displaying the data on an LCD, reading the temperature sensor/ADC output, and controlling the fan/heater output. The controller's bandwidth is to be divided among all these tasks in such a way that, to an end user, the functions seem to be executed in parallel. Designing this involves deciding a background process - i.e., the main process for the controller - and interrupting the controller at regular intervals for all other tasks. Note that there may be asynchronous interrupts as well, such as a master trying to communicate with the slave controller on an as-needed basis. Proper interrupt handling thus becomes a critical task.

Interrupts must be carefully and cautiously handled, mainly because carelessly written interrupts can lead to some mysterious run-time errors. These errors are difficult to uncover and understand since the controller might enter into an undefined state, report invalid data, halt, reset, or otherwise behave in an incomprehensible manner. Being cognizant of some simple interrupt handling practices can help us prevent such events.

In this series of articles, we discuss, with relevant examples, the following simple yet important interrupt-handling nuggets that help prevent such errors, including:

- Decide a background/main process
- Prioritize interrupts properly
- Keep them short :use flags
- Keep it simple :use state machines
- Global variables :know when it's modified
- Local variables :know your compiler
- Using data buffers :be heedful of overflows
- Shared memory :read complete at once
- A little more on buffers
- Multi-Byte Buffers :know the Endianness
- Structured buffers :understand the structure padding
- Calling functions in an ISR :be cautious
- Time critical tasks :understand the latency
- LVD Interrupt- make it blocking
- Decide a background/main process

Although this sounds easy, it's nevertheless important. When a controller has a number of tasks to handle, it is important to understand that the â€œmain()â€ function is a just background process. It also has the least priority, in the sense that any interrupt can disrupt the execution flow, breaking in to cause the CPU to run the interrupt routine rather than the main process.

For example, you may scan a matrix keyboard in the background, where the delays caused by

all other interrupts put together are less than the estimated key holding time of a user, whereas an emergency STOP switch has to be an interrupt.

Prioritize interrupts properly
Interrupt prioritization is important in determining the order of execution when two or more interrupts occur simultaneously. Here it's the importance, urgency, and frequency of tasks that decide the priority.

Consider, for example, a system with a controller using a digital-to-analog converter (DAC) and an I2C slave. There are two situations possible:

- The controller uses the DAC to output a fixed frequency waveform while an I2C master communicates independently with the controller. In this case, the DAC has to be updated at fixed intervals so that the output waveform's shape remains the same, irrespective of whether the controller is communicating to the I2C master or not. Thus, the DAC has to be prioritized over the I2C interrupt.
- The controller uses the DAC to output a waveform with a variable frequency, with the frequency being decided by the I2C master. In this case, the I2C interrupt can take priority since the DAC output timings themselves are controlled by the I2C commands.

Frequently occurring interrupts should be assigned higher priority so that all interrupt requests are serviced. Otherwise, there is a possibility that multiple interrupt requests result in servicing multiple requests only once. This might occur if the second, third, or a number of interrupts occur before the first request is serviced.

Keep them short :use flags
A well understood and frequently discussed practice is that interrupt code should be as short as possible. This assures that the CPU can return to the main task in a timely manner.

The interrupt service routine should only execute the critical code; the rest of the task can be relegated to the main process by setting a flag variable. Note that since flags generally take binary values (0 or 1), these should be declared in bitwise memory wherever possible (like in 8051). This reduces the push/pop overhead and the execution time. Example:

```
bit flag;

#pragma interrupt_handler ISR

void ISR(void)
{
   flag=1;
}

void main()
{
   --
   --
   while(1)
   {
      --
      --
      /* Wait for the ISR to set the
      * flag; reset it before
      * taking any action. */
      if (flag)
      {
         flag = 0;
```

```
         /* Perform the required action here */
      }
   }
}
```

Keep it simple :Use State Machines

State machines help make seemingly large service routines short and simple to execute. There are a number of cases when decisions have to be made inside an interrupt service routine (ISR) and the function to be performed by the ISR depends on the state of the application prior to the interrupt being triggered. Consider the following hypothetical case, for example, where a timer ISR is to be implemented in such a way that it generates timings in the order 10ms, 14ms, 19ms, and the cycle should continue. A simple way to change the period inside an ISR could be as follows:

```
#define PERIOD_10ms 0x01
#define PERIOD_14ms 0x02
#define PERIOD_19ms 0x03

void Timer_ISR(void)
{
   static char State = PERIOD_10ms;

   switch(State)
   {
      case PERIOD_10ms:
      {
         // Toggle pin;
         // Timer Stop;
         // Change period to 14ms;
         // Timer Start;
         break;
      }
      case PERIOD_14ms:
      {
         // Toggle pin;
         // Timer Stop;
         // Change period to 19ms;
         // Timer Start;
         break;
      }
      case PERIOD_19ms:
      {
         // Toggle pin;
         // Timer Stop;
         // Change period to 10ms;
         // Timer Start;
         break;
      }
      default:
      {
         /* Timer_ISR entered undefined state */
         // Make default period 10ms
         break;
      }
   }
}
```

Note here that it's a good practice in C coding to have a default statement in any switch case.

This helps to easily recover the CPU from any undefined state.
In certain cases involving very few states (2 to 3 states), if-else constructs may generate shorter code. However, for larger state machines, if-else constructs generate a much larger assembly listing than the jump table implementation generated by the switch statement. Therefore, general rule for code of any complexity is to use the switch statement.

So far, we've discussed some general interrupt handling practices to help us in structuring ISRs in the right way. In the next part of this article, we'll go through memory-related aspects of ISRs. We will discuss the implications of the liberal use of global/local variables, data buffers, shared memory, etc. We will also talk about interrupt timing/latencies, the implications of calling a C function inside an ISR, and LVD (low voltage detect) interrupts.

Interrupts short & simple: Part 2 - Variables, buffers & latencies

Priyadeep Kaur, Cypress Semiconductor

OCTOBER 11, 2012

Editor's note: *In this second part in an on-going series on the appropriate use of interrupts in embedded systems design, Priyadeep Kaur discusses ISRs, global/local variables, data buffers, shared memory and the interrupt timing latencies.*

In the first part of this series on interrupts, we discussed the importance of careful interrupt handling and some general interrupt handling practices related to the robust structuring of ISRs. Now we will discuss the implications of the liberal use of global/local variables in an ISR.

Global variables – Know when it's modified

Global variables need to be carefully handled while used with ISRs, because interrupts are generally asynchronous and if a global variable is being written into by an ISR, it can get modified at any time. We need to be careful of the following aspects:

Reading/Writing Global Variables at multiple places Global variables should be modified at only a few necessary places inside a program. If a global variable is being modified by multiple threads (i.e., the main process, ISRs, and other hardware functions like DMA inside an MCU), there is a chance of the variable getting corrupted.

Reading a global variable at multiple instances inside a main process is as worrisome as its modification in different threads. Consider the following example:

```
unsigned char Command;

void main()
{
   while(1)
   {
      if (Command == 1)
      {
         /* Send data 1 */
      }
      else if(Command == 2)
      {
         /* Send data 2 */
      }
      else
      {
         /* Send data 3 */
      }
   }
}



void I2C_ISR(void)
{
      Command = I2C_BUF;
}
```

Case1: Incorrect

```
unsigned char Command;

void main()
{
   unsigned char LocalCmd;
   --
   while(1)
   {
      LocalCmd = Command;
      if (LocalCmd == 1)
      {
         /* Send data 1 */
      }
      else if(LocalCmd == 2)
      {
         /* Send data 2 */
      }
      else
      {
         /* Send data 3 */
      }

   }
}

void I2C_ISR(void)
{
      Command = I2C_BUF;
}
```

Case2: Correct

Table1: Reading global variable at multiple places

Click on image to enlarge.

Here, in case1, if the ISR occurred while the CPU was executing the statement

else if(Command == 2)

and if the command received was 1, the CPU will still send data 3, while actually data 1 should have been sent. This problem would not occur in case2 mentioned above.

Accessing multibyte global variables in a 8-bit system Using multibyte global variables in an 8-bit system requires careful attention because multibyte variables are read byte-by-byte. Care needs to be taken that the ISR does not occur and hence modify the variable when one or more bytes of the multi-byte variable have already been read but the read has not been completed. This would lead to data corruption. The following example illustrates this scenario:

```
unsigned int Data;

void main()
{
      unsigned int LocalData;
      --
      while(1)
      {
            LocalData = Data;
            --
      }
}

void I2C_ISR(void)
{
      Data = I2C_BUF[0];
      Data =
      (Data<<8)|(I2C_BUF[1]);
}
```

Case1: Incorrect

```
unsigned int Data;

void main()
{
      unsigned int LocalData;
      --
      while(1)
      {
          Disable_interrupts;
          LocalData = Data;
          Enable_interrupts;
            --
      }
}

void I2C_ISR(void)
{
      Data = I2C_BUF[0];
      Data =
      (Data<<8)|(I2C_BUF[1]);
}
```

Case2: Correct

Table2: Accessing multi-byte variables modified by ISR

Click on image to enlarge.

Note here that the above two cases assume that the I2C_ISR is serviced as soon as it occurs; i.e. the ISR is serviced and data is read from the buffer before the I2C starts updating I2C_BUF again (i.e., before the next 8 bits are received).

The assumption will hold true considering the I2C runs at a much slower speed (like 100 kHz) compared to the CPU (generally MHz) and there are no other interrupts in the system or the total delay caused by all the interrupts is less than the time taken to receive 8 bits (=8/100kHz i.e. 80us).

If this not the case, there may be data corruption.

Accessing Multi-Byte Peripheral Registers in an 8-bit system Some microcontrollers have an 8-bit CPU but their peripherals may have registers which are > 8 bit in size. For example, there may be a 12-bit ADC with an 8-bit CPU in a microcontroller. In such cases, it's important to be careful while reading the multi-byte peripheral registers in an ISR.

Data may be corrupted when an ISR tries to access a multi-byte register in an 8-bit MCU. Consider the following ISR which tries to read the High and Low bytes of an ADC conversion:

Case1: Incorrect

```
unsigned int Data;

void main()
{
   unsigned int LocalData;
   --
   while(1)
   {
      Disable_interrupts;
      LocalData = Data;
      Enable_interrupts;
      --
   }
}

void ISR1(void)
{
      --
}

void ISR2(void)
{
      --
}

void ADC_ISR(void)
{
   Data = ADC_MSB;
   Data = (Data<<8)|ADC_LSB;
}
```

Case2: Correct

```
unsigned int Data;

void main()
{
   unsigned int LocalData;
   --
   while(1)
   {
      Disable_interrupts;
      LocalData = Data;
      Enable_interrupts;
      --
   }
}

void ISR1(void)
{
      --
}

void ISR2(void)
{
      --
}

void ADC_ISR(void)
{
   do
   {
      Clear_ADC_Interrupt;
      Data = ADC_MSB;
      Data =
      (Data<<8)|ADC_LSB;
   }
   while(ADC_Interrupt_Reg);
}
```

Table3: Reading multi-byte, hardware modified registers

Click on image to enlarge.

Note that Case1 (Incorrect) in Table3 is similar to Case2 (Correct) in Table2. How can the same method be correct in one case and incorrect in other? The answer to this question goes as below:

Case1 in Table3 or Case2 in Table2 above would work fine as long as it is ensured that the ADC_ISR or the I2C_ISR will be served completely before it is triggered again.

Now, as seen in Table3, there are two more ISRs in the system. In this case consider that the ADC is continuously operating and that the ADC_ISR is triggered whenever an ADC conversion completes. Consider the ADC_MSB and ADC_LSB to be hardware synchronous and so the ADC conversion result registers gets updated by a hardware latch operation as soon as the ADC conversion completes.

Now consider that ADC_ISR was triggered for the first time when ISR1 was already executing. This increases the delay in servicing the ADC_ISR. Now consider that another ADC conversion got completed when just the MSB of the previous conversion had been read in the ISR. In this case, both the ADC_MSB and the ADC_LSB register would be updated with the new value, but the previous ISR is still executing and the he ADC_MSB of previous conversion has already been read. The ADC_LSB read now will be from the current conversion. “Data” is now corrupted.

```
void ADC_ISR(void)
 {
   Data = ADC_MSB; ->Previous Conversion, conversion result was 0x01FF, ADC_ MSB and hence Data is
```

```
0x01
Another ADC conversion completed, current data = 0x200.
            (Data<<8)|ADC_LSB; -> Current ADC_LSB = 0x00, Data now becomes 0x0100.
        }
Valid Data is either 0x1FF or 0x200, however, the Data reported by the ISR is 0x0100 which is incorrect.
```

There are two ways to handle this problem: ensure the ADC_ISR completes before another conversion is expected to be completed; read the Data as illustrated in Case2 in Table3 above.

Note that every interrupt generally has something similar to an "Interrupt_Clear" register associated with it. Writing into this register prevents the Interrupt from posting to the controller and reading the register returns the current status of the interrupt i.e. whether a corresponding interrupt has been registered by the device.

The Interrupt_Clear register may be used to understand if another ADC conversion has been completed while the MSB and LSB of the previous conversion were being read. Re-read the conversion result if the same had occurred.

Local variables? – Know your compiler

Compilers can allocate memory for local variables in two different ways. In order to understand the problems that could occur due to the use of local variables in ISRs and the methods that can be used to prevent the same, we should first understand how local variables are stored in memory. The two different ways for storing local variables in memory are described below:

Using stack for local variables

In this case, the local variables are created in stack whenever a function is called. This is the most common way of storing local variables since the stack and hence the memory used by the called function would be freed once the control is returned to the calling function. With such compilers, it is important to be aware of the total stack usage if the program involves large number of function calls in the form of a tree -> one calling second, second calling third and so on.

With respect to ISRs, it becomes even more important to analyze stack usage since ISRs can be triggered at any time during a program flow. The total stack usage and hence the RAM required for the stack can be calculated as follows:

Case I: Only one ISR is served at a time

If interrupts are not nested, only one ISR is serviced at a time. In this case, the maximum stack usage at any time is calculated as the total stack required for the largest calling tree (including space for local variables, Program Counter, function arguments, etc.) plus that of the ISR which uses the largest amount of stack.

Case II: Nested interrupts

If interrupts are nested, an ISR can preempt another ISR and even itself. In this case, total stack usage can be excessive; hence, nested interrupts should be avoided even if there are no local variables. This is because when an ISR is triggered, all (or nearly all, depending on the compiler) the special function and general-purpose registers used by the ISR are also stored on the stack.

If an ISR is allowed to preempt itself, the stack will keep on filling (the case where interrupt keeps on occurring and preempting the ISR) and will easily overflow the stack. Avoid nesting interrupts unless absolutely necessary and, in the latter case, ensure that the ISR can never preempt itself. This can be ensured by analyzing the total time required to execute the ISR and the minimum possible

delay between two occurrences of the interrupt.

If the maximum stack usage exceeds the available RAM for the stack, memory could get corrupted when the MCU is running the program. Such errors may be fatal and are hard to detect, hence, should be avoided.

Fixed memory locations for local variables and memory overlaying
Certain compilers do not save local variables on the stack as is generally done in C. Instead, they use fixed memory locations to store local variables and function arguments, and share those locations among local variables of functions that don‘t call each other. Such a form of memory overlaying is done in order to prevent stack overflows and enable efficient memory utilization. Use of local variables inside ISRs with such compilers requires careful attention. In order to understand why, let’s first look at how memory overlaying works.

Overlaying of fixed memory locations to store local variables and function arguments of different functions at the same address is achieved using a well-defined procedure. First, the linker builds a call tree of the program, with the help of which it can figure out which data segments for which functions are mutually exclusive and thus overlay them.

For example, suppose that the Main() function calls function A, function B, and function C. Function A uses 10 bytes, function B uses 20 bytes and, function C uses 16 bytes of local variables. Assuming that functions A, B, and C do not call each other, the memory they use may be overlaid. Thus, rather than taking 46 bytes of data memory (10 for A + 20 for B + 16 for C) only 20 bytes of data memory is consumed.

Problems with local variables in ISR when memory is overlaid
If the local variables of an ISR occupy the same memory space as one of the functions in the main flow, and if the interrupt gets triggered when one of those functions using same memory space is executing, the data in those local variables could get corrupted by the ISR.

The following techniques are helpful in preventing corruption of data space by interrupts:
* Use static keyword on the local variables used in ISRs. This allocates a separate data space for these variables that is not overlaid with any other function.
* Use a lower level of optimization settings for the compiler that will disable variable overlaying. Optimization level details can be found in compiler/linker reference manuals.
* Use a different data space for “normal” and “ISR” locals. For example, use XDATA for "normal" locals and DATA for interrupt locals. This can be done either by using memory specifiers in the variable declarations or using different memory models for the different files.
* Depending on the optimization settings, the compiler may be using register banks instead of RAM for the ISR locals. In this case, make sure the memory usage of the ISR is not greater than the register-bank it is using when it "fires".

We have just covered a small aspect of memory management for ISRs. In the next part of this series, we’ll go a little further and discuss data buffers and shared memory, where we need to be heedful, and what are the right and wrong practices while using buffers with ISRs.

Interrupts short and simple: Part 3 - More interrupt handling tips

Priyadeep Kaur, Cypress Semiconductor

OCTOBER 16, 2012

Editor's note: *In this third part in an on-going series on the appropriate use of interrupts in embedded systems design, Priyadeep Kaur discusses right and wrong practices while using buffers with ISRs.*

So far, we have covered the importance of careful interrupt handling, ways of robustly structuring ISRs, and the attention required to global and local variables. In this part, we dive a little deeper into additional interrupt handling practices.

Be Heedful of data buffer overflows

We generally use software data buffers for communication interfaces. For example, the microcontroller may provide an I2C serial communication slave interface with a 1-byte I2C data buffer. Consider that the I2C interface is such that an interrupt is generated (a) when a complete byte is received, and (b) when a stop condition is received.

In these cases, we would like to have a software buffer declared such that whenever a byte is received, the ISR automatically transfers the data to the data buffer. Such buffers are generally implemented in the form of arrays. A very common mistake is to increment the array index beyond the array size, so we need to prevent any such overflows. The following table shows a correct and incorrect implementation.

```
unsigned char Buffer[BUF_SIZE];
unsigned char cBufIndex = 0;

void main()
{
   while(1)
   {
      /*Main code*/
   }
}

void I2C_StopReceived_ISR(void)
{
   cBufIndex = 0; //Re-initialize buffer
   pointer
}

void I2C_ByteReceived_ISR(void)
{
   Buffer[cBufIndex] = I2C_BUF;
   cBufIndex++;
}
```

Case1: Incorrect

```
unsigned char Buffer[BUF_SIZE];
unsigned char cBufIndex = 0;

void main()
{
   while(1)
   {
      /*Main code*/
   }
}

void I2C_StopReceived_ISR(void)
{
   cBufIndex = 0; //Re-initialize buffer
   pointer
}

void I2C_ByteReceived_ISR(void)
{
   if(cBufIndex < BUF_SIZE)
   {
      Buffer[cBufIndex] = I2C_BUF;
      cBufIndex++;
      if(cBufIndex == BUF_SIZE)
      {
            I2C_ENABLE_NACK;//Do not
      accept more data, let I2C not
      acknowledge the received bytes
      }
   }
}
```

Case2: Correct

Table1: Buffer overflow example

Click on image to enlarge.

Note that the above point is illustrated with I2C as an example, which requires the NACK of data as a requirement of the protocol itself. This mistake is more common while using UART communication where a NACK is not required as a part of the protocol. In this case, the protocol should be consciously defined such that buffer overflow conditions do not occur. This can be done either by transmitting a byte indicating that an overflow has occurred or by simply ignoring the received data after the buffer is full, depending on the application.

Read shared memory complete at once

Here, the principle is the same as reading multi-byte variables that could be modified by an ISR. If a shared memory/buffer is implemented between an ISR and your main routine, the complete buffer should be read together at one place. If such is not the case, you may read half of the previous data and half of current data, which may lead to some unexpected condition. Following is an example:

```
unsigned char Buffer[2];
unsigned char cBufIndex = 0;
unsigned char FlagRxComplete = 0;

void main()
{
   unsigned char Command = DEFAULT;
   while(1)
   {
      if(FlagRxComplete)
      {
         FlagRxComplete = 0;
         Command = Buffer[0];
         switch(Command)
         {
            case A:
            {
               PerformFunctionA();
               ExecuteCommand(Buffer[1]);
            }
            case B:
            {
               /*---*/
            }
            default:
            {
               /*---*/
            }
         }
      }
   }
}
void I2C_StopReceived_ISR(void)
{
   cBufIndex = 0; //Re-initialize buffer
   pointer
   FlagRxComplete = 1;
}
void I2C_ByteReceived_ISR(void)
{
   if(cBufIndex < BUF_SIZE)
   {
      Buffer[cBufIndex] = I2C_BUF;
      cBufIndex++;
      if(cBufIndex == BUF_SIZE)
      {
         I2C_ENABLE_NACK;//Do not accept
         more data, let I2C not
         acknowlede the received bytes
      }
   }
```

Case1: Incorrect

```
unsigned char Buffer[2];
unsigned char cBufIndex = 0;
unsigned char FlagRxComplete = 0;

void main()
{
   unsigned char Command[2] = {0,0};
   while(1)
   {
      if(FlagRxComplete)
      {
         FlagRxComplete = 0;
         DISABLE_INTERRUPTS;
         Command[0] = Buffer[0];
         Command[1] = Buffer[1];
         ENABLE_INTERRUPTS;
         switch(Command[0])
         {
            case A:
            {
               PerformFunctionA();
               ExecuteCommand(Command[1]);
            }
            case B:
            {
               /*---*/
            }
            default:
            {
               /*---*/
            }
         }
      }
   }
}
void I2C_StopReceived_ISR(void)
{
   cBufIndex = 0; //Re-initialize buffer
   pointer
   FlagRxComplete = 1;
}
void I2C_ByteReceived_ISR(void)
{
   if(cBufIndex < BUF_SIZE)
   {
      Buffer[cBufIndex] = I2C_BUF;
      cBufIndex++;
      if(cBufIndex == BUF_SIZE)
      {
         I2C_ENABLE_NACK;//Do not accept
         more data, let I2C not acknowlede
         the received bytes
```

Case2: Correct

Table2: Reading memory shared with an ISR

Click on image to enlarge.

More on buffers

The following two points apply to buffer implementations in general. The use of buffers with communication-related ISRs is very common and these small mistakes either lead to an exchange of corrupted data or misinterpretation of data.

Know the Endianness of multibyte buffers

Endian format refers to how multibyte variables are stored in a byte-wide memory. In 'big endian' format, the most significant byte is stored in the first byte (lowest address). In 'little endian' format, the least significant byte is stored in the lowest address.

To understand why it's important to be aware of the endian format of your compiler/MCU, consider the below example of an integer array being transferred from one 8-bit MCU to another 8-bit MCU using UART.
Consider that "int" is a 16-bit variable and that transmit and receive C code used in the two controllers is as follows:

```
#define BUF_SIZE 3

unsigned int RxBuf[BUF_SIZE]= {0};

unsigned char cBufIndex = 0;

void main()
{
   while(1)
   {
      /*---*/
   }
}

void UART_ByteReceived_ISR(void)
{
      if(cBufIndex == (BUF_SIZE*2))
      {
            cBufIndex = 0;
      }
      *(RxBuf+cBufIndex) =
UART_RX_BUF;
      cBufIndex++;
}
```

RX method implemented on MCU 1

```
#define BUF_SIZE 3

unsigned int TxBuf[BUF_SIZE]={0x1122,
                              0x3344,
                              0x5566};
unsigned char cBufIndex = 0;

void main()
{
   while(1)
   {
      /*---*/
   }
}

void UART_ByteTransmit_ISR(void)
{
      if(cBufIndex == (BUF_SIZE*2))
      {
            cBufIndex = 0;
      }
      UART_TX_BUF =
*(TxBuf+cBufIndex);
      cBufIndex++;
}
```

TX Method implemented on MCU 2

Table3: UART Buffer Tx and Rx functions implemented on two different controllers: Incorrect method

Click on image to enlarge.

The above implementation would be correct if the compilers for MCU 1 and MCU 2 use the same endian format for storing multi-byte variables in memory. However, if in the case of MCU1, the endian format is little endian (least significant byte stored in the lower address) and for MCU2 it is big endian (most significant byte stored in lower memory address), the RxBuf would contain {0x2211, 0x4433, 0x6655} instead of {0x1122, 0x3344, 0x5566}.
Note that the data bytes got swapped, even though MCU2 was sending correctly; if you probe the UART line, you would find correct data being transferred, but the data in the RxBuf would still be inverted!

If you're aware of the endian format, you would change either the Tx or Rx code to take care of the swapping.

Understand the padding of structured buffers

If your MCU has a multibyte CPU architecture but the memory is still accessible in byte-sized chunks (which is generally the case), be careful about structure padding and alignment performed by your compiler.

Microcontrollers normally require that data be aligned on natural boundaries for efficient access. For instance, 32-bit data type should be aligned on 32-bit (word) boundaries and 16-bit data type should be aligned on 16-bit (word) boundaries. Although the compiler normally allocates individual data items on aligned boundaries, data structures often have members with different alignment requirements. To maintain proper alignment, the compiler normally inserts additional unnamed data members so that each member is properly aligned.
For example, if the following structure is declared for a 16-bit architecture CPU, the compiler would store the data as shown below:

```
struct TxBuf
{
   int A;
   char B;
   int C;
} TxBuf
```

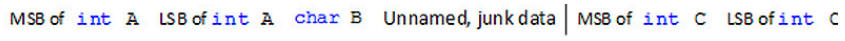

Table 4: TxBuf memory alignment for 16 bit CPU

Click on image to enlarge.

Sending this TxBuf using a transmit function as below would lead to transfer of unnecessary "unnamed, junk data" as shown in Table4 above.

```
void UART_ByteTransmit_ISR(void)
{
   if(cBufIndex == (sizeof(TxBuf))
   {
      cBufIndex = 0;
   }
   UART_TX_BUF = *(TxBuf+cBufIndex);
   cBufIndex++;
}
```

By changing the ordering of members in a structure, it is possible to eliminate or change the amount of padding required to maintain alignment, as in:

```
struct TxBuf
{
  int A;
  nt C;
  char B ;
} TxBuf
```

It is also possible to tell most C compilers to "pack" the members of a structure on CPU Cores that support unaligned accesses. Refer to your compiler manual for the keyword used to pack the structured members such that no unnamed data members are added for alignment.

Be cautious when calling functions in an ISR
Stack usage Having multiple function calls inside an ISR can lead to excessive stack usage, either because of storing of SFRs or local variables (in cases where local variables are created on stack). Ensure that the stack usage doesn't exceed the available limits.

ISR execution time If the ISR execution time is a concern, use macros instead of function calls. This saves on CPU time (for push/pop etc) and conserves stack usage.

Reentrancy warnings Reentrancy problems are common with compilers that use fixed memory locations (as explained in Part 2 of this series) instead of stack for local variables and function arguments.

Neglecting any reentrancy warnings indicate that the linker has found a function that may be called from both main code and an ISR (or functions called by an ISR) or from multiple ISRs at the same time. One problem is that the function is not reentrant and it may get invoked (by an ISR) while the function is already executing. The result will be variable and probably involve argument corruption. Another problem is that memory used for local variables and arguments may be overlaid with the memory of other functions. If the function is invoked by an interrupt, that memory will be used. This may cause memory corruption of other functions.

Understand the latency of time critical tasks

Higher priority interrupts getting serviced before the lower priority interrupt is serviced. The maximum latency here is the sum of the execution time of all the higher priority interrupts + the largest lower priority interrupt. Note that the largest lower priority interrupt is considered because this largest interrupt may be triggered just before the triggering of the interrupt in consideration. In this case, the current ISR would not be serviced unless control is returned from the lower priority interrupt.

The push/pop operation done at interrupt entry/exit increases the latency to service the ISR.

In order to keep interrupt latency as low as possible and within requirements, ensure the following:

- Assign proper priority to interrupts and minimize the time spent in servicing interrupts in general, as explained in Part 1 of this series.
- Reduce push/pop overhead before the interrupt code execution starts/ends. This can be done by using higher optimization levels with your compiler. In this case, the compiler will push/pop only those registers affected in the Interrupt service routine. It also optimizes the code inside the ISR to reduce interrupt code execution time.

Some CPUs, like the 8051, have multiple register banks, each bank having multiple general-purpose registers. Only one of the multiple banks can be active at a given time. Some compilers provide special attributes, like the "using" attribute supported by the Keil compiler, which can be applied in the ISR function definition to specify the register bank to be used by the ISR. These attributes facilitate the management of different register banks for both interrupt and non-interrupt code, and hence result in decreased latency in interrupt execution by reducing the number of push/pop operations to be done on interrupt entry/exit. When attributes like "using" are applied, the push/pop operations are usually performed only for SFRs and not for general-purpose registers.

Make LVD Interrupts blocking ISRs

Many modern MCUs provide an interrupt for low voltage detection (LVD) that is triggered whenever the Vdd falls below a particular voltage level. This interrupt should be the highest priority interrupt and can be used to perform any emergency operations before the MCU is powered down; for example, saving important data in EEPROM/Flash.

The LVD interrupt should be made a blocking ISR, unlike other general ISRs, to ensure that the MCU remains in the LVD ISR and does not return to the normal program flow unless the voltage is returned to normal; this can be done by constantly monitoring the low voltage detect comparator output inside the ISR. This is done because it is preferable NOT to perform any MCU operation below the recommended Vdd voltage. Making the LVD a blocking ISR ensures that any brown-out condition does not cause the MCU to run in an incoherent state.

Part 1: Good programming practices
Part 2: Variables, buffers, and latencies

Priyadeep Kaur *has completed her BE in Electronics and Electrical Communication Engineering from PEC University of Technology, Chandigarh and is currently working with Cypress Semiconductor India Pvt. Ltd. as an Application Engineer. Her interests are embedded systems, analog circuits, and DSP. She can be reached at* ***pria@cypress.com.***