InterruptsÂ 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 itâ€™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 â€œintâ€ 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 youâ€™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
```

MSB of int A	LSB of int A	char B	Unnamed, junk data	MSB of int C	LSB of int C

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 â€œunnamed, junk dataâ€ 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 doesnâ€™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 â€œusingâ€ 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 â€œusingâ€ 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.***