InterruptsÂ 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.*

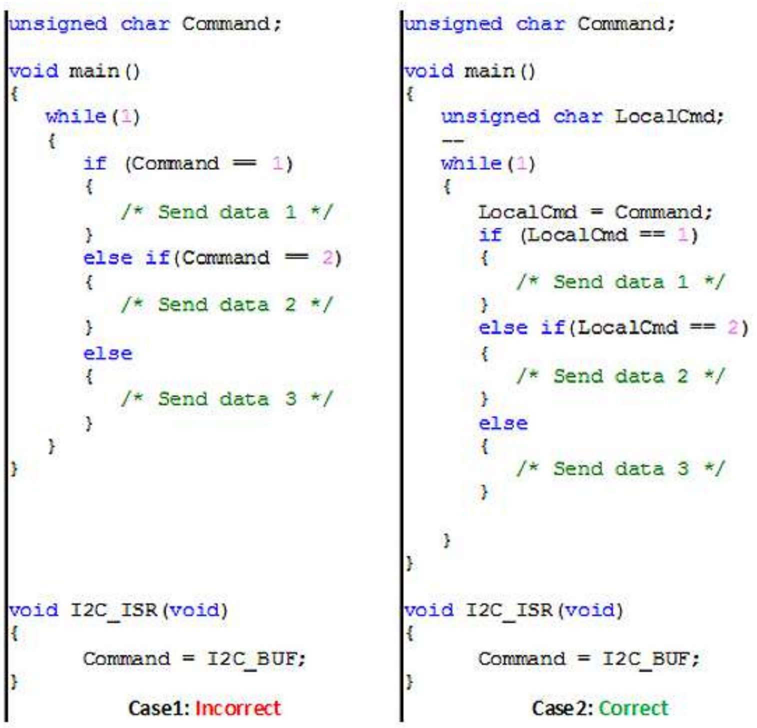
In the first part of this series on interrupts, we discussed the importance of careful interrupt handling and some general interrupt handling practices related to the robust structuring of ISRs. Now we will discuss the implications of the liberal use of global/local variables in an ISR.

Global variables â€“ Know when itâ€™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

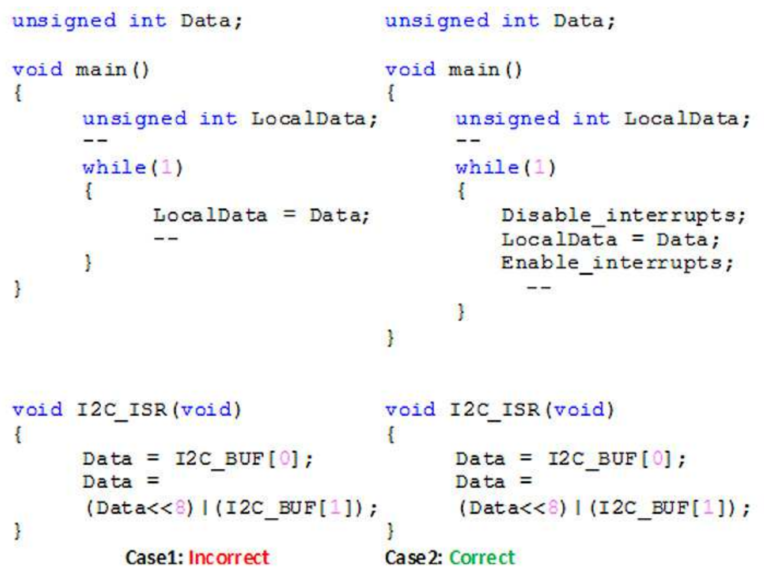
Click on image to enlarge.

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.

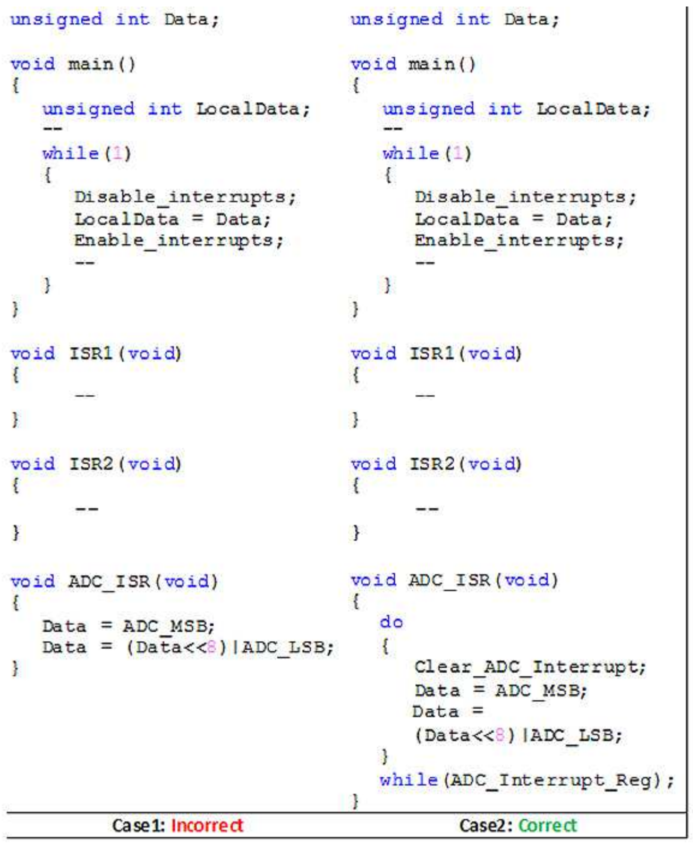
Note here that the above two cases assume that the I2C_ISR is serviced as soon as it occurs; i.e. the ISR is serviced and data is read from the buffer before the I2C starts updating I2C_BUF again (i.e., before the next 8 bits are received).

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, itâ€™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:



```
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;
}
```

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

Case2: Correct

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. â€œDataâ€ 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 â€œInterrupt_Clearâ€ 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? â€“ 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 donâ€˜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, letâ€™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 â€œnormalâ€ and â€œISRâ€ 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, weâ€™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.

Part 1: Good programming practices
Part 3: Using buffers and ISRs

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.***