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

Although this sounds easy, itâ€™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 itâ€™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 waveformâ€™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 â€“ 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 â€“ 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 itâ€™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, weâ€™ve discussed some general interrupt handling practices to help us in structuring ISRs in the right way. In the next part of this article, weâ€™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.

**Part 2: ISRs, variables, data buffers, and shared memory**

**Part 3: Using buffers and ISRs**

***Priyadeep Kaur*** *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.***