Debugging simulated hardware on Linux, Part 2: Create an environment for virtual device driver development

Testing the Interrupt Service Routine (ISR)

Arun Prasad Velu

November 02, 2005

This two-part series is geared toward easing device driver development. This second part describes the various strategies and implementation details that you can apply to interrupt simulation, including the prerequisites, hardware, software setup, and test cases for testing the Interrupt Service Routine (ISR).

Why would anybody want to simulate hardware when developing a device driver? This article lays out the problem and proposes an approach to solve it. Part 1 of this series provides a broader understanding of the issues and implementation details.

You can apply these methods and strategies to many operating systems and hardware architectures. These strategies work for Linux®, VxWorks, and Windows® NT/2000 operating systems on IBM PowerPC® 405GP, Intel x86®, MIPS, and Motorola PPC architectures. This article series focuses on Linux on x86.

This article describes how to debug interrupts and Interrupt Service Routine (ISR) in a systematic manner, and gives detailed explanations and algorithms that let you step through the source on all possible paths/flow of the ISR. These techniques are helpful in all possible worlds, including combinations of interrupts and ISR, such as slow interrupt, fast interrupt, tasklet, bottom-half, and so on. Finally, this article discusses the hardware and software environment you need to achieve these objectives and run the test cases.

A scenario for simulation

This article helps device driver developers test the interrupt service routine as much as possible by simulating the various interrupts. Following a successful implementation of this simulation technique, you can also perform a Functional Verification Test (FVT) that may involve the device driver, application programming interface (API), and the application.
Consider a hypothetical device driver. Suppose your device driver must be written from scratch, and you do not have the actual hardware while developing this device driver. Your driver is complex: it could be used by multi-threaded applications. The driver will perform hardware register accesses and advanced programming by making use of `mmap()`, for example.

Your device is going to generate different types of interrupts -- multiple and nested interrupts at the same time -- and that leads to the design and implementation of a complex Interrupt Service Routine (ISR). Your driver will perform some data manipulation based on a sequence of interrupts. This particular device driver is meant for an embedded system where you may not have much sophisticated debugging environment. This device driver should perform some diagnostics of the device itself. And finally, the driver is tightly coupled with various APIs and applications, and you may have to debug a device driver where interrupt losses and out-of-sequence interrupts happen.

This is a bit more complex than regular porting work.

The requirements

You will need two different systems for the strategies described here.

The first setup: Development/host machine

The first setup requires any Linux distribution and the device for which you are writing the device driver. You will also be applying a patch (extra routines) to your device driver. These extra routines are required only for this interrupt simulation.

You will also have to write a kernel thread that will generate the various interrupts. As part of implementation, we may break this thread into a few support routines. I explain this in detail below.

The second setup: Test machine

The second setup requires the kernel debugger-enabled kernel.

To enable the kernel debugger, you’ll need the kernel debugger patches. There are two well-known kernel debuggers available. I have chosen kgdb instead of kdb, because kgdb lets you view the C source code.

Source-level debugging of ISR is the main objective here. You will need the device for which you are writing the driver and a null-modem (serial) cable for remote debugging. You need to apply the kgdb patch on the kernel, build an image with kernel debugging support, then run that kernel on the test machine.

You will control the test machine from the development/host machine through a serial port. Once you are in debug mode, the target machine’s kernel stops. Even the jiffies (small packets of kernel time for timing interrupts) are not altered and this lets you debug the Interrupt Service Routine.

More setup caveats

Tip: Take extra care for drivers that include features like TTL (Time to Live), such as connection-oriented networking device drivers.
You will have complete control over raising interrupts (simulated) in the example described throughout this article.

When you need to debug any particular interrupt (one interrupt at a time), you will use the debugger-enabled setup. When you run the rigorous test (sequence of interrupts), you will use the first setup that does not require kernel debugger support. A combination of both these setups will give you the best result.

Before proceeding further, I will describe an ioctl interface that you will be using in the two approaches.

The ioctl interface

A new ioctl command should be added to the device driver to control the interrupt simulation from the test application. This ioctl can be used in the FVT test application code. This ioctl interface is meant for our hypothetical driver. Actual implementation would depend on the device and the driver.

In our example, part of the interrupt handling is carried out at the application level and part is in the driver. To achieve this you need application threads and kernel threads. The kernel thread and the application thread will handshake with each other.

This special purpose ioctl interface will be able to control the interrupt generation sequence and the number of interrupts to be generated through the test application.

I will discuss two different sets of interrupts, normal interrupts and error interrupts.

A sophisticated way to have more control over raising interrupts and testing the ISR is to follow a two-tier architecture, having a special ioctl function that would give the user application the freedom to raise a particular interrupt or sequence of interrupts at specified timings and the ioctl implementation in the kernel land. In this approach, you could have more control over interrupt generation. You have to set the appropriate fields and then pass the same to the special ioctl that would in turn either raise the interrupts or signal the kernel thread to raise the interrupts.

The ioctl structure

Listing 1 demonstrates the structure of ioctl.

Listing 1. The ioctl structure

```c
struct simulation_struct
{
    struct interrupt_type EventsArray [MAX_INTR_TYPE];
    unsigned iteration_count;
    unsigned num_events;
};
```
The explanation of this code goes like so:

- **EventsArray [MAX_INTR_TYPE]** is an array of `struct interrupt_type` as defined based on the device and the different types of interrupts.
- **iteration_count** is the count to control the number of iteration of interrupt simulation.
  - If it is 0, raise all MAX_INTR_TYPE interrupts one by one (preprogrammed in the interrupt simulation module).
  - If it is 1, normal interrupts in sequence only once.
  - If it is greater than 1 and less than MAX_COUNT, normal interrupts in sequence iteration_count times.
  - If it is MAGIC_NUMBER, get the data/interrupt register values from the structure passed and generate the interrupt as per the structure values. In this case, num_events will give the number of interrupts to be generated.

MAX_INTR_TYPE, MAX_COUNT, and MAGIC_NUMBER should be defined by you based on your needs and the actual hardware. As a rule of thumb, MAX_COUNT should always be less than MAGIC_NUMBER.

- **num_events** is the number of valid entries in EventsArray. The minimum value is 1 and the maximum value is MAX_INTR_TYPE. num_events interrupts will be generated as per the input passed to EventsArray.

**The ioctl command**

**Listing 2. The ioctl command**

<table>
<thead>
<tr>
<th>ioctl name:</th>
<th>INTR_SIMULATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input:</td>
<td>Pointer to struct simulation_struct</td>
</tr>
<tr>
<td>Function Type:</td>
<td>New feature</td>
</tr>
</tbody>
</table>

The ioctl command works like so:

- If the interrupt simulation status flag is set, return EBUSY. This scenario arises when interrupt simulation is already in progress.
- Check the initialization status of the driver. If the status is not good, then return the appropriate error code.
- Copy the contents of **arg** (pointer to `struct simulation_struct`) to the global structure. Make sure this copying happens inside the critical section by holding a spinlock. **Note:** The kernel thread will read the global structure, and interrupts will be generated based on the elements in the global structure. The spinlock is needed here, because the kernel thread will be running independently and will access the global structure.
- Set the interrupt simulation status flag to indicate that interrupt simulation is in progress.
- Wait until the interrupts are generated.
- Once the interrupts are generated, reset the interrupt simulation status flag and return success. This gives control back to the application.
The `ioctl` command returns the following codes:

- On successful execution, it returns 0.
- Otherwise, it returns the appropriate error status flag.

Keventd should run to create a new kernel thread.

**The strategies**

The three main strategies I would use for interrupt and hardware simulation are:

- Software-generated IRQ
- Using the kernel debugger
- Using the polling thread

Each strategy requires that a kernel thread is run.

**Strategy 1: Software-generated IRQ**

The primary aim of this approach is to simulate interrupts and test whether the ISR handles all possible interrupts. You can automate this activity and simulate the conditions so that the ISR would be invoked as in an actual runtime environment.

The kernel thread that we implement will raise the interrupt (the software-generated IRQ) for our device driver (and not for our card) by making use of the `INT` assembly instruction.

Before the interrupt is raised, all other pre-requisites (setting up any address, data, etc.) for that interrupt should be handled in the kernel thread. Once the interrupt has been raised, the driver's ISR will be called. In the ISR you will not read the actual device's registers; instead you will have to read values from the local variables that are assigned by the kernel thread (simulation module). You should actually duplicate the device's registers. Before you raise the interrupt in the kernel thread, you will have to set the bit/mask values on these local registers.

Depending on your device driver, you may need to have a copy of the buffers, if the device has any. This depends on the implementation and the device. Since you have already set the (simulated) register values appropriately, the ISR will process normally as if it was a real interrupt. You might notice that this is more of a *hardware simulation* and not just interrupt simulation.

This approach requires some changes in the ISR. The code that accesses the actual card registers should now be changed to access the local variables that mimic the device's registers.

To achieve this mapping, you may include conditional compilation `#ifdef` in places where you access the registers. To limit the number of `#ifdef`s in the ISR, you should `#define` all the registers and keep all these `#define`s in a separate header file. On top of these register `#define` macros, you should also define another macro that dictates whether the ISR will run in simulation mode or in the original interrupt context. For example:
Listing 3. Example conditional compilation

```c
#ifndef INTR_SIMULATION
    // Only for interrupt simulation
#define PCIINTRSTATUS   local_pciintrstatus   // Access the local variable
#else
    // Actual Interrupt
#define PCIINTRSTATUS   Dev-> DataStruct.ulIpcIntrStatus
#endif
```

In the implementation of the driver, wherever you access the device's registers you have to use these `#define` macros instead of using the structure variables directly. This also provides more clarity, because you have avoided the multiple indirections of union or structure variables.

You may extend this `#ifdef` technique to the API and to the applications so that you can link this interrupt-simulated driver to those modules to perform levels of FVT testing. For the unit testing of this interrupt-simulated driver (unit testing of ISR), you can have your own test application that will simulate some functionality of the APIs and applications that are part of the overall system.

**Code changes for Strategy 1**

The following code changes are necessary to use software-generated IRQs:

1. All the registers being accessed need to be defined (`#define`).
2. A separate `ioctl` command should be introduced to have control over interrupt simulation (see the section on the `ioctl` interface).
3. A separate kernel thread should be written to raise the interrupt. This kernel thread will get registered during `open` in the device driver on successful registration (`request_irq`) of the ISR.

Use the kernel API `kernel_thread` to register this kernel thread in this pseudo code:

**Listing 4. Starting the kernel thread**

```c
#ifndef INTR_SIMULATION
    //
    // Start the Kernel Thread
    //
    start_kthread( raise_intr_thread, &raise_intr );
#endif // end of INTR_SIMULATION
```

The function `start_kthread` launches the thread by calling kernel API `kernel_thread`.

This kernel thread should be destroyed in the `close` of the device driver in this pseudo code:
Listing 5. Stopping the kernel thread

```c
#ifdef INTR_SIMULATION

    //
    // Stop the Kernel Thread
    //
    Stop_kthread( raise_intr_thread);

#Endif // end of INTR_SIMULATION
```

These parts of the code (kernel thread registration and destruction) should again be within the `#define INTR_SIMULATION` conditional compilation block.

4. A test application should be written to handle these interrupts. This test application should simulate some part of the functionality of the APIs and applications to handle the raised interrupts. In our example, the test application will spawn threads and wait (blocked) for interrupts to release the threads. This blocking functionality is achieved by making use of `sleep_on_interruptible` with a mutually exclusive lock inside the driver's `ioctl` function. Whenever an interrupt occurs, one thread will be woken up (`wake_up_interruptible`) and resume execution based on the interrupt.

5. The special `ioctl` function `INTR_SIMULATE` needs to be called to simulate the interrupts.

**Strategy 2: Using the kernel debugger**

The primary aim of this approach is to step through the source code of the tasklet and/or the bottom half which services the interrupts. Since you step through the kernel in this approach, you will not have the exact timing sequence. As mentioned earlier, extra care needs to be taken for device drivers like this that involve features like TTL (in this case one that uses connection-oriented networking device drivers).

This strategy lets you examine the device driver's complete code flow on a per-interrupt basis. This approach could be used along with the first strategy and you can use this approach to test the driver with the actual target setup.

This strategy requires the kernel thread to raise the interrupt so that the device's ISR will get called. You will have to place a `break point` in the tasklet or bottom half. The kernel will stop at this point when the ISR schedules the tasklet/bottom half. Once the break point is hit, you can step through the source and view or modify the variables.

In this strategy, you'll access the device's register the same way as in Strategy 1 -- using local register variables. If the device and the target architecture permit, you could access the device's register through the debugger.

By effectively making use of the kernel debugger, you can reduce the work of the kernel thread that was described earlier. With this approach, you could simulate the various conditions,
sequence, and variables. While you are in the tasklet, you will be able to modify the (local) register values at debug time and be able to step through all the paths and flow of the source code.

**Required code changes for Strategy 2**

All the code changes that are required for Strategy 1 also apply to Strategy 2. However, some of the initialization and prerequisite code in the kernel thread will not be required, because you will be able to achieve those initializations during the debugging session itself.

You can decide whether to implement everything in the source code or to change the parameters during runtime using the debugger. You will not need a larger number of threads, since this approach runs on a per-interrupt basis.

**Strategy 3: Using the polling thread**

This approach is designed to rigorously test the tasklet/bottom-half code. In this approach you will not raise the interrupt. You can test all the interrupt sequences (out of sequence) by using a polling technique. This approach may also be used in conjunction with the kernel debugger (Strategy 2).

You will need two kernel threads for the implementation of this strategy. The first one is similar to the kernel thread mentioned in the previous strategies except that it will not raise the interrupt. However, you will change the local register variables, and once you finish the initialization/prerequisites for a particular interrupt, you will indicate that fact to the second kernel thread (the polling thread).

The polling thread waits for the signal from the first thread. It could keep polling for the signal (change) to occur or it could just sleep. Once it gets the signal, it schedules the tasklet/bottom half (software interrupt). The tasklet/bottom half executes in the same context as when an interrupt occurs.

It is important to note that these tasklets/bottom halves will run close to the interrupt context (software interrupt). However, the polling thread will run in a normal process context.

**Required code changes for Strategy 3**

The following code changes are necessary to use the polling strategy:

1. All the registers being accessed need to be defined (**#define**).
2. You will need two kernel threads in this approach.
   - The first thread is similar to the one defined for Strategy 1, but it will not raise the interrupt. All other initializations for the interrupts should be carried out in this thread.
   - You will need another separate polling thread, which will get notified by the first thread when to schedule a tasklet.
   - You will need to use an interprocess communication (IPC) mechanism between these two threads.
3. These kernel threads should be destroyed in the close of the device driver. These portions of the code (kernel thread registration and destroy) will again be in the **#define INTR_SIMULATION** conditional compilation.
Note: If you do not enable this conditional compilation flag, you will get the release version of the driver object that will be used in the target environment.

4. The test application will not require much change. It will simulate some parts of the functionalities of the APIs and applications to handle the raised interrupts. This test application will spawn threads and will keep waiting (blocked). The blocking functionality is achieved by making use of `sleep_on_interruptible` inside the driver's `ioctl` function. Whenever any interrupt occurs, these threads will be woken up (`wake_up_interruptible`) and resume execution based on the interrupt.

Note: Whenever we schedule the tasklets, blocked threads will start waking up and continue processing. Extra care must be taken not to infinitely block the kernel.

Designing kernel threads and test applications

The kernel thread(s) will be initialized in the `open` entry point of the driver, provided `request_irq` succeeds on successful registration of the interrupt service routine, for instance.

These threads will be destroyed in `close`. The code to initialize and destroy the threads will be under `#ifdef INTR_SIMULATION`, so that under normal compilation this code will not affect the release version of the driver object.

In this section, I'll examine:

- two threads (an interrupt and polling thread),
- a test application, and
- test cases.

**Interrupt thread**

This thread keeps generating all possible interrupts based on the following algorithm:

**Listing 6. Interrupt thread algorithm**

1. Read the global interrupt simulation structure.

   ```
   If iteration_count is 0

   1.1. For each and every iteration,

   1.1.1. Set the particular interrupt status bit
   1.2.1. Do any other preparation, if required.
   1.3.1. If compiler option is polling mode (#ifdef POLLING)
       Intimate interrupt status register change to the polling thread.
   1.4.1. Else
       Raise the card's interrupt by calling INT mnemonic
   ```
1.5.1. Delay the thread.

1.2. Continue until MAX_INTR_TYPE iterations (all MAX_INTR_TYPE possible interrupts once)

Tip: You may use or instead of using the INT mnemonic to make it portable between platforms, but make sure you are taking care of enabling and disabling interrupts in the proper place and sequence.

2. If iteration_count is 1, raise the normal interrupts (not the error interrupts) in their sequence.

2.1. For each and every iteration,

2.1.1. Set the particular interrupt status bit in the normal interrupts sequence.
2.2.1. Do any other preparation, if required.
2.3.1. If compiler option is polling mode (#ifdef POLLING)

   Intimate interrupt status register change to the polling thread.
2.4.1. Else

   Raise the device's interrupt by calling INT mnemonic
2.5.1. Delay the thread.

2.2. Continue until iteration MAX_NORMAL_INTR_TYPE (all MAX_NORMAL_INTR_TYPE normal interrupts once)

3. If iteration_count is greater than 1 and less than MAX_COUNT, raise the normal interrupts in sequence iteration_count times

3.1. For each and every iteration,

3.1.1. Set the particular interrupt status bit in the normal interrupts
3.1.2. Do any other preparation, if required.

3.1.3. If compiler option is polling mode (#ifdef POLLING)
   
   Intimate interrupt status register change to the polling thread for all the MAX_NORMAL_INTR_TYPE normal sequence interrupts (loop of MAX_NORMAL_INTR_TYPE iteration, 1 per interrupt)

3.1.4. Else
   
   Raise the card's interrupt by calling INT mnemonic for all the MAX_NORMAL_INTR_TYPE normal sequence interrupts (loop of MAX_NORMAL_INTR_TYPE iteration, 1 per interrupt)
   
3.1.5. Delay the thread.

3.2. Continue until iteration equals iteration_count

4. If the iteration_count is MAGIC_NUMBER,
   
   Get the interrupt register values from structure passed and generate the interrupt as per the structure values. In this case num_events will give the number of interrupts to be generated

4.1. For each and every iteration,
   
4.1.1. Set the particular interrupt status bit in the normal interrupts sequence.
4.1.2. Do any other preparation, if required.
4.1.3. If compiler option is polling mode (#ifdef POLLING)
   
   Intimate interrupt status register change to the polling thread as per the input passed.

4.1.4. Else
   
   Raise the card's interrupt by calling the INT mnemonic as per the input passed.

4.1.5. Delay the thread.

4.2. Continue until iteration equals num_events
**Note:** To start with, you may go for a one-second delay. Then you can tune the loop so that you will be generating as many interrupts as in the case of the original system.

### Polling thread

A few things to remember about a polling thread:

- This thread will keep polling whether or not any change happens in the local interrupt status register in a loop.
- If there is any change in the status register, it means an interrupt has occurred.
- If there is an interrupt, schedule the tasklet using `schedule_tasklet`.
- Continue the earlier-mentioned tasks.

### A test application

This test application will inherit some part of code from the APIs and applications that make use of the driver. Here are seven steps to enabling the test application:

1. In the main program, spawn the required number of threads.
2. Issue `ioctl` that would be blocked (`sleep_on_interruptible`) inside the driver/kernel.
3. Fill the input structure for `ioctl INTR_SIMULATE`.
4. Issue `ioctl INTR_SIMULATE`.
5. Whenever an interrupt wakes up the thread, process the interrupt the same way the actual API and application process it.
6. Register the sequence number, interrupt nature, and thread attributes to the main program.
7. The main program keeps track of the information provided in step 6 and monitors whether any out of sequence or interrupt loss happens.

This is one of the crucial tests that you could carry out using this hardware simulation technique.

### Enabling the test cases

The following steps illustrate how to enable the test cases.

#### Listing 7. Enabling the test cases

1. Raise all possible (MAX_INTR_TYPE) interrupts and check whether they are getting handled appropriately in the driver.

   1.1. Use printk statements to check whether the appropriate interrupt handling steps are getting executed.

   1.2. User `/proc` entry registered for our device driver.

   1.3. Use kernel debugger kgdb and check whether the appropriate interrupt handling steps are getting executed.

Debugging simulated hardware on Linux, Part 2: Create an environment for virtual device driver development
2. Raise all normal sequence interrupts (MAX_NORMAL_INTR_TYPE interrupts) and check whether they are getting handled appropriately in the driver. Some device drivers need to handle a series of interrupts before they collectively perform some task.

3. Check to see if any interrupt is getting lost.

   3.1. In the test application, when the thread gets woken up, check for the Interrupt ID (sequence number)

   3.2. Check whether the interrupt that we have simulated is getting captured in the test application. This is a test for thread wake up.
Related topics

- In Part 1 of this two-part series, "Debugging simulated hardware on Linux, Part 1: Interrupts and Interrupt Service Routine" (developerWorks, November 2005), learn about strategies and implementation details that you can apply to interrupt simulation, including the prerequisites, hardware, software setup, and test cases for testing the Interrupt Service Routine (ISR).
- "Smashing performance with OProfile" (developerWorks, October 2003) introduces a tool that can help you identify issues such as loop unrolling, poor cache utilization, inefficient type conversion and redundant operations, and branch mispredictions.
- *Understanding the Linux Kernel*, Third Edition (O'Reilly, November 2005) provides a guided tour of the code that forms the core of all Linux operating systems.
- *Linux Device Drivers*, Third Edition (O'Reilly, February 2005) includes full-featured examples that programmers can compile and run without special hardware.
- Build your next development project on Linux with **IBM trial software**, available for download directly from developerWorks.

© Copyright IBM Corporation 2005

Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)