

How to configure Linux to receive IRQs from the FPGA? A step-by-step guide for Altera Cyclone V SoC

Philémon Favrod
Supervisor: René Beuchat

March 14, 2016

This guide describes how to write a Linux device driver that handles *interrupt requests* (IRQs) generated by a *soft IP* synthesized on the FPGA portion of Altera Cyclone V SoC devices. This basic set up is outlined in Figure 1.

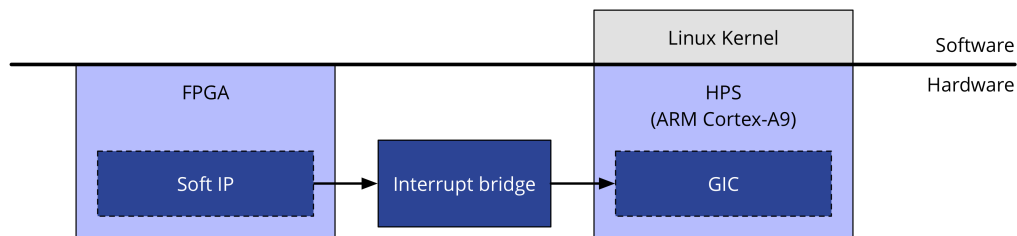


Figure 1: The setup in question.

For educational purpose, this guide is not only a hands-on tutorial but it also tries to detail every steps along the way for the reader to understand the essentials of what is happening under the hood. Therefore, in section 1, this guide starts with a quick overview of the *Generic Interrupt Controller* (GIC) featured by ARM Cortex-A9 MCUs, the *hard processor system* (HPS) integrated in Altera Cyclone V SoC devices. Then, in section 2, it quickly discusses how Linux handles interrupts and lets device drivers register handlers for IRQs. Finally, as a practical example, it provides the reader with a step-by-step guide using a dummy driver in section 3.

1 Overview of the Generic Interrupt Controller

ARM Cortex-A9 MCUs include an interrupt controller referred to as the *Generic Interrupt Controller* (GIC). Its integration in Altera Cyclone V devices is shown in Figure 2.

The GIC basically handles two main tasks as outlined in Figure 3. First, it acts as a *Distributor* by properly distributing the IRQs to the different cores of the system. In Cyclone V SoC devices, the HPS features two cores. It therefore allows *Private Peripheral Interrupts* (PPIs) and *Software-Generated Interrupts* (SGIs) to reach only their CPU of interest while it broadcasts *Shared Peripheral Interrupts* (SGIs) to both cores.

Subsequently, it provides each core of the system with a programming interface referred to as *CPU Interface*. Among other things, those interfaces allow the CPUs to query for interrupt

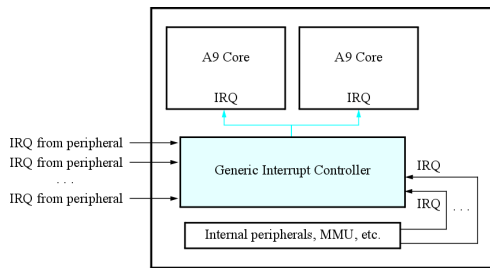


Figure 2: GIC overview. Source: [6].

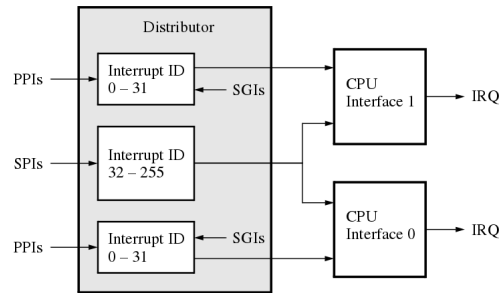


Figure 3: GIC architecture. Source: [6].

sources. The GIC also features interrupt prioritization and masking features. However, as well as the specifics of the *CPU Interface*, they are out of the scope of this document.

The GIC featured by the HPS of Cyclone V SoC devices can handle up to 180 interrupt sources among which 64 are dedicated for soft IP synthesized on the FPGA portion of the SoC [5]. The interrupt numbers allocated for those IRQs ranges from 72 to 135 [5].

2 Interrupt handling in Linux on ARM processors

This sections aims at giving an overview of how interrupts are handled by Linux in the ARM architecture. It is by no mean a complete description and just intends to give the reader a big picture.

2.1 The `irq_chip` structure

An interrupt controller driver — this one is architecture dependent — registers an `irq_chip` structure to the kernel. This structure contains a bunch of pointers to the basic routines that are needed to manage IRQs. For instance, this includes routines to enable and disable interrupts at the interrupt controller level, as well as interrupt acknowledgment stub. Listing 1 depicts how the data structure looks like.

```

1 struct irq_chip {
2     ...
3     void (*irq_enable)(struct irq_data *data);
4     void (*irq_disable)(struct irq_data *data);
5     ...
6     void (*irq_ack)(struct irq_data *data);
7     ...
8 }

```

Listing 1: `irq_chip` in Linux 4.4.

In our case, the driver of interest is the ARM GIC driver and can be found in `/drivers/irqchip/irq-gic.c`.

2.2 Multiple irq_domain

The kernel internals use a single number space to represent IRQ numbers, i.e. there are no two IRQ having the same numbers. This is also true from the point of view of the hardware when a system has a single interrupt controller (IC). However, a mapping is needed as soon as two ICs, i.e. two `irq_chip`, are available (e.g. GIC and GPIO IC).

To solve this problem, the Linux kernel came up with the notion of an IRQ domain which is a well-defined translation interface between hardware IRQ numbers and the one used internally in the kernel.

Any user of a modern Linux system can experience this mapping by displaying the content of the `/proc/interrupts` pseudo-file that shows the currently registered IRQ handlers. One might see that the IRQ number on the left aren't the same as the one shown after the interrupt controller name.

The GIC driver therefore creates its IRQ domain and its translation on `init`¹.

2.3 The irq_desc structure

In the Linux kernel, all configured IRQs are associated with an `irq_desc` data structure. It contains all the data that is useful to handle the IRQs. For instance, it stores from which interrupt controller the IRQ comes from by storing a pointer to its driver (`irq_chip`). As depicted in Listings 2 and 3, the `irq_desc` structure also contains a pointer to the `irq_domain`.

```
1 struct irq_desc {
2     ...
3     struct irq_data irq_data;
4     ...
5 }
```

Listing 2: `irq_chip` in Linux 4.4.

```
1 struct irq_data {
2     ...
3     unsigned int irq;
4     ...
5     struct irq_chip *chip;
6     struct irq_domain *domain;
7     ...
8 };
```

Listing 3: `irq_data` in Linux 4.4.

Device drivers then traditionally use `request_irq` to install a custom interrupt service routine. Let's trace a call to this function:

- (1) `request_irq` calls `request_threaded_irq`²;
- (2) `request_threaded_irq` passes the IRQ number to `irq_to_desc` that looks up a radix tree to return a pointer to previously allocated `irq_desc` structure³;
- (3) `request_threaded_irq` then calls `__setup_irq`, an internal stubs whose role is to register the new handler for that IRQ⁴.

¹lxr.free-electrons.com/source/drivers/irqchip/irq-gic.c?v=4.4#L1092

²<http://lxr.free-electrons.com/source/include/linux/interrupt.h?v=4.4#L134>

³<http://lxr.free-electrons.com/source/kernel/irq/irqdesc.c?v=4.4#L111>

⁴<http://lxr.free-electrons.com/source/kernel/irq/manage.c?v=4.4#L1103>

The question at this point is who allocated the `irq_desc` returned by `irq_to_desc`. The answer to this question is in the device tree handling.

A device node for a device that generates interrupts looks like the following:

```
1 mydevicenode {
2     compatible = "my-compatible-string-used-to-know-what-driver-to-use";
3     interrupts = <GIC_SPI MY_HW_IRQ_NUMBER IRQ_TYPE_EDGE_RISING>;
4     interrupt-parent = <&intc>; /* where intc is the appropriate interrupt
5     controller node in the device tree */
6 };
```

Note that the `interrupt-parent` property is generally inherited from parent nodes and might not be mentioned explicitly.

An error one can easily make concerning the device tree is to think of it as way to avoid architecture-dependent code. The device tree is just a binary file that is parsed by the kernel upon initialization and that can be query through a well-defined API by different drivers. Typically, in the case of `mydevicenode`, this would be a platform driver, a driver for a device that is directly available to the CPU, e.g. a device that does not require an intermediate driver to handle a bus like USB.

If, while writing the platform driver for `mydevicenode`, one wants to call `request_irq` to register a custom ISR, then one would have to call `platform_get_irq`⁵ before. After a lot of nested calls, this function ends up indirectly calling `irq_create_mapping`⁶ which allocates the `irq_desc`.

3 The step-by-step guide

For the remaining of this section, we will assume that you have your cross-compiler set up on your host. We are using the one provided by Altera which can be downloaded here: ftp://ftp.altera.com/outgoing/SoC_FPGA/ethernet_3.7/gcc-linaro-arm.tar.bz2.

We are going to use the latest version of the Linux kernel sources, i.e. Linux 4.4.4. Note that this might no more be the latest version when you will be reading these lines. You might find it at <ftp://ftp.free.fr/mirrors/ftp.kernel.org/linux/kernel/v4.x/linux-4.4.4.tar.gz>.

This guide also supposes that you have a working SD card image. If you do not have your own and, as us, you are using a Terasic board, you might download the images provided by Terasic.

3.1 Compiling the kernel

In any terminal you will be using in this guide, we supposed that you defined the global variable `ARCH` and `CROSS_COMPILE`. The former tells the Linux build system which architecture your

⁵<http://lxr.free-electrons.com/source/drivers/base/platform.c?v=4.4#L86>

⁶<http://lxr.free-electrons.com/source/kernel/irq/irqdomain.c?v=4.4#L459>

Linux build is targetting. The later tells the Linux build system where to find your cross-compiler toolchain. Assuming your cross-compiler is in your PATH, this might be done as follows:

```
1 export ARCH=arm
2 export CROSS_COMPILE=arm-linux-gnueabihf-
```

Then, the build should be configured to target a SoC FPGA, one of the available default configuration in the ARM architecture:

```
1 make socfpga_defconfig
```

Finally, the kernel can be build as follows:

```
1 cd <WHEREVER-THE-KERNEL-SOURCES-ARE>
2 make -j <P> zImage
```

where <P> is a placeholder for the number of cores of your host you want to involve in the compilation. After completion, the file *arch/arm/boot/zImage* has been generated and can be placed on the primary partition of your SD card.

3.2 Updating the device tree

It is now time to update the device tree with the description of our soft IP component. Since we are using a Terasic DE0-SoC board, we will base our updated device tree on *socfpga_cyclone5_de0_sockit.dts*, the default device tree of the board. However, the strategy should be very similar for other boards.

Let's create a file named *socfpga_test_int.dts* in *arch/arm/boot/dts* with the following content:

```
1 #include <dt-bindings/interrupt-controller/irq.h>
2 #include <dt-bindings/interrupt-controller/arm-gic.h>
3 #include "socfpga_cyclone5_de0_sockit.dts"
4
5 /* Extends the SoC with a soft component. */
6 / {
7     soc {
8         mysoftip {
9             compatible = "altr,socfpga-mysoftip";
10            interrupts = <GIC_SPI 40 IRQ_TYPE_EDGE_RISING>;
11        };
12    };
13 };
```

All children of the *soc* node inherit its *interrupt-parent* (see *interrupt-parent*) property which points to the GIC device node. We specified that we want the GIC to trigger an IRQ on the rising edge of the 40th SPI, namely for the 72th interrupt line of the GIC as shown on Figure 3. This corresponds to the first IRQ dedicated for the FPGA [5].

The device tree might then be compiled by issuing the following command in the kernel source directory:

```
1 make socfpga_test_int.dtb
```

After completion, you might replace your current device tree on the primary partition of your SD card by the one you just generated, i.e. `arch/arm/boot/dts/socfpga_test_int.dtb`.

At this point, you are strongly encouraged to test that your board is able to boot from the modified SD card.

3.3 A simple platform driver

The dummy driver code is shown in Listing 4. Its associated Makefile is shown in Listing 5. Make sure to fill in the placeholder in the Makefile with the correct path to the kernel sources.

```
1 #include <linux/module.h> /* Needed by all modules */
2 #include <linux/kernel.h> /* Needed for KERN_INFO */
3 #include <linux/init.h> /* Needed for the macros */
4 #include <linux/interrupt.h>
5 #include <linux/sched.h>
6 #include <linux/platform_device.h>
7 #include <linux/io.h>
8 #include <linux/of.h>
9
10 #define DEVNAME "test_int"
11
12 static irq_handler_t __test_isr(int irq, void *dev_id, struct pt_regs *
    regs)
13 {
14     printk(KERN_INFO DEVNAME ": ISR\n");
15
16     return (irq_handler_t) IRQ_HANDLED;
17 }
18
19 static int __test_int_driver_probe(struct platform_device* pdev)
20 {
21     int irq_num;
22
23     irq_num = platform_get_irq(pdev, 0);
24
25     printk(KERN_INFO DEVNAME ": IRQ %d about to be registered!\n", irq_num
    );
26
27     return request_irq(irq_num, (irq_handler_t) __test_isr, 0, DEVNAME,
    NULL);
28 }
29
30 static int __test_int_driver_remove(struct platform_device *pdev)
31 {
32     int irq_num;
33
34     irq_num = platform_get_irq(pdev, 0);
35
36     printk(KERN_INFO "test_int: IRQ %d about to be freed!\n", irq_num);
37
38     free_irq(irq_num, NULL);
39
40     return 0;
```

```

41 }
42
43 static const struct of_device_id __test_int_driver_id[] = {
44     {.compatible = "altr,socfpga-mysoftip"},
45     {}
46 };
47
48 static struct platform_driver __test_int_driver = {
49     .driver = {
50         .name = DEVNAME,
51         .owner = THIS_MODULE,
52         .of_match_table = of_match_ptr(__test_int_driver_id),
53     },
54     .probe = __test_int_driver_probe,
55     .remove = __test_int_driver_remove
56 };
57
58 module_platform_driver(__test_int_driver);
59
60 MODULE_LICENSE("GPL");

```

Listing 4: *test_int.c*

```

1 obj-m += test_int.o
2 KERNELPATH=<WHEREVER-THE-KERNEL-SOURCES-ARE>
3
4 all:
5     make -C $(KERNELPATH) M=$(PWD) modules
6
7 clean:
8     make -C $(KERNELPATH) M=$(PWD) clean

```

Listing 5: *Makefile*

Then, assuming you have the correct environment variables set for ARCH and CROSS_COMPILE (see section 3.1), you can just issue the following command:

```

1 cd <WHEREVER-YOU-PUT-YOUR-DRIVER>
2 make

```

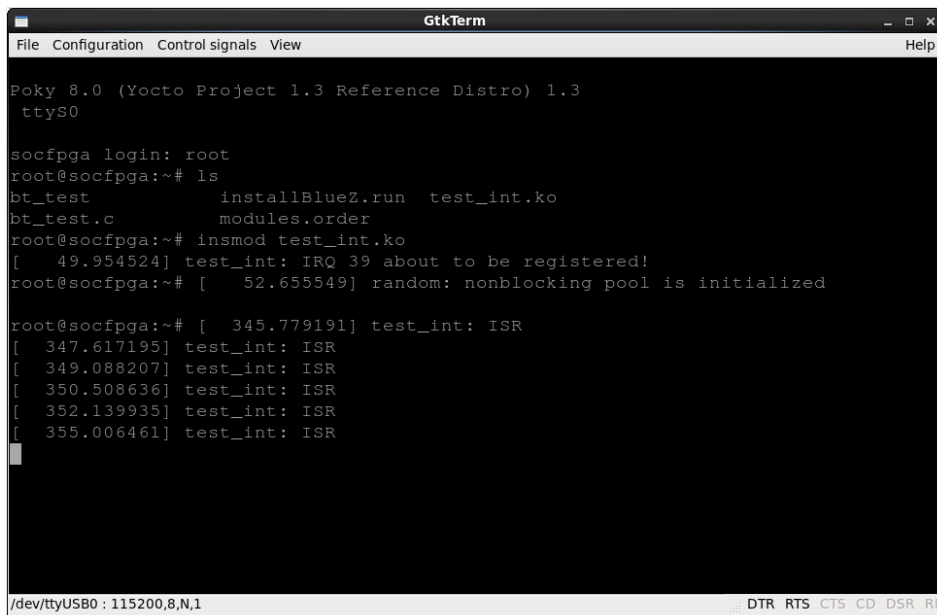
Finally, copy the resulting file *test_int.ko* on the secondary partition of your SD card, i.e. your root filesystem. Then, on the target, issue the following command:

```

1 insmod test_int.ko

```

You might now toggle your IRQ signal and observe your target's console printing a pretty message on each IRQ rising edge as in Figure 4.



```
Poky 8.0 (Yocto Project 1.3 Reference Distro) 1.3
ttyS0

socfpga login: root
root@socfpga:~# ls
bt_test          installBlueZ.run  test_int.ko
bt_test.c        modules.order
root@socfpga:~# insmod test_int.ko
[  49.954524] test_int: IRQ 39 about to be registered!
root@socfpga:~# [  52.655549] random: nonblocking pool is initialized

root@socfpga:~# [  345.779191] test_int: ISR
[  347.617195] test_int: ISR
[  349.088207] test_int: ISR
[  350.508636] test_int: ISR
[  352.139935] test_int: ISR
[  355.006461] test_int: ISR
```

/dev/ttyUSB0 : 115200,8,N,1 DTR RTS CTS CD D5R RI

Figure 4: GtkTerm showing our target output.

References

- [1] A Tutorial on the Device Tree (Zynq). available at <http://xillybus.com/tutorials/device-tree-zynq-1>.
- [2] Interrupt definitions in DTS. available at <http://billauer.co.il/blog/2012/08/irq-zynq-dts-cortex-a9/>.
- [3] irq_domain interrupt number mapping library. available at <https://www.kernel.org/doc/Documentation/IRQ-domain.txt>.
- [4] Linux Kernel IRQ Domain. available at <http://invo-tronics.com/linux-kernel-irq-domain/>.
- [5] Altera Corporation. *Cyclone V Device Handbook - Volume 3: Hard Processor System Technical Reference Manual*, November 2012.
- [6] Altera Corporation. *Using the ARM Generic Interrupt Controller*, April 2014.
- [7] Grant Likely. Linux and the Device Tree. available at <https://www.kernel.org/doc/Documentation/devicetree/usage-model.txt>.