Lab on HW-SW digital systems codesign
SEL B.Sc. Curriculum (Spring Semester)
Embedded Systems Laboratory
EPFL-STI-IEL

Exercises Session 6
Out: 2025-03-27

Device drivers for FPGA AXI peripherals in Linux

In this session, we will create a device driver in kernel space to control the peripherals, in contrast to the user-space classes we were using up to now to control them. We will also detect the completion of the peripheral work with an interrupt service routine (ISR) by setting up an interrupt request (IRQ). First, we will see how to interact with a timer peripheral (developed in VHDL). Then, we will see how to interact with the vector adder developed in Session 2 and how to capture the interrupt generated by the accelerator upon completion of its task.

1 Board setup for Linux

Before we can start developing our device drivers, we need to build the module system for the Xilinx image in Linux. In the Pynq (image version 2.6), connect by ssh and execute the following commands:

```
$ sudo ln -s /lib/modules/5.4.0-xilinx-v2020.1/build/include/asm-generic/ \
    /lib/modules/5.4.0-xilinx-v2020.1/build/include/asm
$ cd /lib/modules/5.4.0-xilinx-v2020.1/build
$ sudo make
```

The build process will not fully complete. This is normal:

```
  UPD     include/generated/compile.h
make[1]: *** No rule to make target 'init/main.o', needed by 'init/built-in.a'.  Stop.
Makefile:1652: recipe for target 'init' failed
make: *** [init] Error 2
```

2 Exercise 1: Timer with interrupts

In this example, we build an AXI4 slave peripheral that implements a timer of one second. Every time that one second passes, the peripheral generates one interrupt. As the frequency of the peripherals is by default 100 MHz in our system, the peripheral simply counts 100 000 000 cycles and generates the interrupt each time it reaches the maximum.

Build a Vivado project with the structure shown in Figure 1, observing how the interrupt signal is connected to the Zynq processors. Before connecting interrupt lines to the Zynq, we need to enable them as shown in Fig. 2.

Inspect the HDL file of the timer module (`HDL/TimerInterrupt.vhd`). The interrupt needs to be enabled and cleared (acknowledged) from the software side through the slave registers of the peripheral interface. The peripheral activates the interrupt line (i.e., `interruptOut` in Figure 1 becomes high) once the maximum count is reached. However, the peripheral does not deactivate the

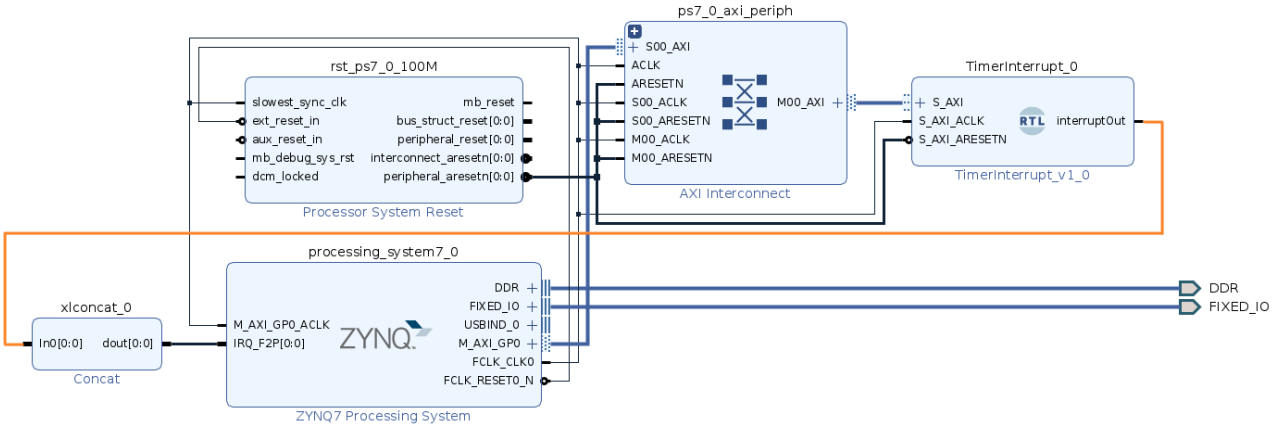

Figure 1: Block diagram of Example 1. The "concat" module converts a bit signal into a vector, e.g., from `std_logic` to `std_logic_vector(0:0)`. Do NOT remove the concat module: it is necessary to use the programming script in the Pynq board.

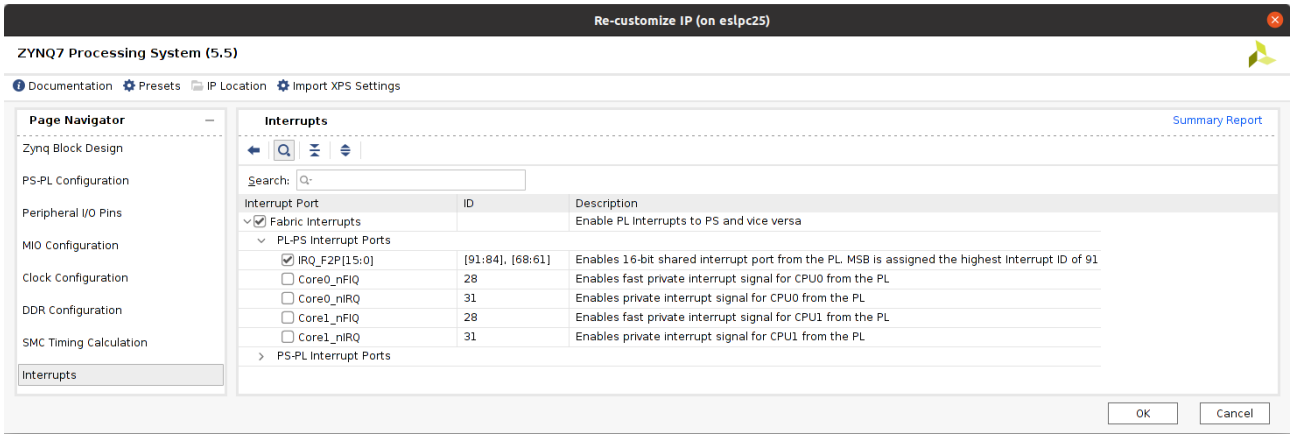

Figure 2: Enabling interrupts in the Zynq block.

Table 1: Registers presented by the slave interface of the peripheral used in Example 1.

Offset	Direction	Name
0x00	R	Count
0x04	W	Enable
0x08	W	Enable interrupt
0x0C	W	Clear interrupt

interrupt line until the software writes into the `ClearInterrupt` register. If the software does not clear the interrupt on time, the processor will not detect the next interrupt as there will be no transition in the interrupt signal.

The peripheral presents a slave interface with the register map described in Table 1:

- The `Count` register allows the software to read the current count value.
- The `Enable` register starts the count of the timer peripheral.
- The `Enable interrupt` register enables activation of the `InterruptOut` signal when the condition is met in the timer.
- The `Clear interrupt` which allows the software the acknowledge the interrupt and lower the interrupt signal, so that the next event can trigger a new transition.

Generate the bitstream, extract the files and copy them to the Pynq board.[1] Copy also the software files available in Moodle to the board (from the `SW/` folder). The `Makefile` will generate two programs, `testTimer` and `testTimerInterrupt`.

2.1 testTimer: Counting seconds in software

Inspect the source code of the application `testTimer.cpp` . In this example, we try to detect when the peripheral counter passes by zero in software. Every time the peripheral counter passes by zero, we know that one second has elapsed.

The application code can be compiled normally with make. In this first experiment, we are going to use only the `testTimer` application. To test the design, program the FPGA and execute the program. As usual, both operations require sudo:

```
$ sudo ../programOverlay.py Example1_TimerInterrupt.bit
...
$ sudo ./testTimer
```

▶ **Question:** *What do you expect the behavior of the application to be?*

▶ **Question:** *What is the actual behavior of the software?*

▶ **Task:** *Explain what you observe and propose an explanation. Use `htop` or `top` to find the CPU utilization of the application. Is this method efficient? Does it work properly?*

To verify your conclusions, modify the main condition of the application (line 55) as follows, and explain the newly observed behavior:

```
if ( (lastCounter > 100) && (newCounter < 100) ) {
```

[1] The bitstream (files `.bit` and `.hwh`) are also available in Moodle in case you encounter issues regenerating them.

2.2 testTimerInterrupt: Using a device driver with interrupts

Instead of having the CPU poll continuously the values in the peripheral registers, we can ask the peripheral to generate an interrupt every time it reaches its maximum value. We can then create a device driver that automatically manages the interrupt notifications and clears the interrupt line. The `driver/` folder contains an example of the device driver that implements this behavior. Read the driver code carefully to understand all its operations. Finally, on the user application side (`testTimerInterrupt.cpp`), we can simply call the device driver to obtain the number of times that the interrupt has happened —check the variable `count` in the device driver.

The device driver exports the operations `init(), exit(), open(), read(), release()`, which correspond to the loading and unloading of the kernel module, and the system calls `open(), read() and close()`. These functions can be called from user space:

- `init()` is called by the kernel when loading the driver. This process can be started with this command line: `sudo ./load` in the `driver` folder.
- `exit()` is called by the kernel when unloading the driver. This process can be started with this command line: `sudo ./unload` in the `driver` folder.
- `open()` corresponds to the system call `open`. From C, it is called with the standard `open()` function: `int open (const char *__path, int __oflag, ...)`
- `read()` corresponds to the system call `read`. From C, it is called with the standard `read()` function: `ssize_t read (int __fd, void *__buf, size_t __nbytes)`
- `release()` corresponds to the system call `close`. From C, it is called with the standard `close()` function: `int close (int __fd)`

The driver associates one ISR with the IRQ line connected to the peripheral. Our peripheral is connected to the global interrupt controller (GIC) of the ARM processors through its IRQ line 61. However, the Linux kernel remaps the IRQ to the virtual IRQ number 48 —the determination of the correct number is beyond the scope of this exercise.

> ▶ **Task:** *Complete the code of the interrupt handler to count the number of interrupts and clear the interrupt in the device.*
> *Tip: Read the code of `testtimer_open()` to know how to write to the peripheral registers from kernel space.*

The driver is loaded and unloaded from the command line (in the `driver` folder):

```
$ make
...
$ ./load
... Use the driver ...
$ ./unload
```

The driver prints informative and error messages to the kernel log. Once the driver is loaded, the command `dmesg` can be used to read these messages. It is also possible to see the association of the driver to the IRQ line:

```
$ cat /proc/interrupts
CPU0       CPU1
16:          0          0     GIC-0  27 Edge      gt
17:     515255     658262     GIC-0  29 Edge      twd
...
47:          0          0     GIC-0  41 Edge      f8005000.watchdog
48:        139          0     GIC-0  61 Edge      testtimer_driver
IPI1:          0          0  Timer broadcast interrupts
IPI2:     193965     536737  Rescheduling interrupts
IPI3:       2516       2489  Function call interrupts
IPI4:          0          0  CPU stop interrupts
IPI5:          1          0  IRQ work interrupts
IPI6:          0          0  completion interrupts
Err:           0
```

Table 2: Registers presented by the slave interface of the peripheral used in Example 2, including the registers for interrupt control. "Toggle on write" (TOW) bits toggle their value when a "1" is written to that specific bit in the register.

Offset	Direction	Name
0x00	R/W	Control and status
0x04	R/W	Global Interrupt Enable (bit 0)
0x08	R/W	Interrupt Enable Register (IER)
	R/W	bit 0: enable done interrupt
	R/W	bit 1: enable ready interrupt
0x0C	R/W	Interrupt Status Register (ISR)
	R/W	bit 0: done (toggle on write)
	R/W	bit 1: ready (toggle on write)
0x10	R/W	input 1 address
0x1C	R/W	input 2 address
0x28	R/W	output address
0x34	R/W	length
0x3C	R/W	bias

Columns "CPU0" and "CPU1" correspond to the number of interrupts of each IRQ line that have been attended by each CPU in the system. Use this information to verify later that the driver is working correctly.

Once the driver is loaded, inspect the source code of the file `testTimerInterrupt.cpp` . What do you expect that will be the behavior of the application? Execute it and verify your hypothesis. Use again `htop` or `top` to determine the CPU utilization of the application.

Applications interact with device drivers using file operations. Typically, a driver gets one node under `/dev/` for each instance (peripheral) that it controls. In our case, the application opens `/dev/testtimer`. The application can use all the file operations exported by the driver; in our case, they are `open()`, `close()` and `read()`. The `read()` operation of our driver returns the count of interrupts that the peripheral has produced, e.g., the number of seconds since it was activated. The operation returns exactly 4 bytes (one `uint32_t`) every time it is called, and requires that the user-supplied buffer is at least as large as this.

The driver is executed in kernel space, whereas the application resides in user space. That means that none of them can directly access the variables of the other. Therefore, the driver uses the functions `raw_copy_to_user() and raw_copy_from_user()` to copy to/from user buffers.

The interrupt handler can be executed at any time, preempting the user application or other functions of the driver itself. Therefore, any operations it performs on data must be atomic.

The application does not need to read the counter of interrupts with any specific frequency. The peripheral and the driver continue counting the interrupts in the background, and the application can check the value whenever it needs. The example shows this independence by *sleeping a random time* every time it reads the counter.

3 Exercise 2: Implementing a device driver and interrupt handling for the vector adder

In this example, we see how to build a device driver for a master peripheral that takes advantage of the interrupt mechanism to allow the calling thread to go to sleep until the peripheral completes its operation. Use the files in the folder `Exer02_VectorAdder` for this example. The peripheral is the same vector adder from Session 2. Therefore, we can reuse the same files for the hardware generation used in that session. While generating the block diagram in Vivado, remember to connect

the interrupt of the peripheral to the PS block as done with the timer in the previous example.

The map of peripheral registers should look like Table 2. In particular, in this example we will use the interrupt control registers of the peripheral.

3.1 The device driver

First, program the bitstream into the FPGA. Then, enter the folder `driver`, make the driver, and load it. Use `dmesg` to verify that the driver is correctly loaded (the name of this driver is "adder").

Inspect the user application in `VectorAdder.cpp` . The `Open()` function and the destructor `CAccelDriver.cpp` have changed: They are not aware anymore of the register mapping of the peripheral. Instead, this structure has been moved to `adder.c` in the device driver code. Additionally, in `CVectorAdderDriver.cpp` the call to the accelerator has also changed: It is not configuring the slave registers of the accelerator anymore, since this configuration is now done by the device driver in `adder.c`.

The driver and the application communicate through the `read()` system call. We have defined a structure (`user_message`) with the same fields both in the driver and in the application:

```
struct user_message {
  uint32_t input1;
  uint32_t input2;
  uint32_t output;
  uint32_t length;
  uint32_t accum; // => "bias"
};
```

The application passes a buffer of this type to the driver with the parameters for the accelerator. In this way, the code of the driver, which interacts directly with the peripheral, is isolated from the user code. This can improve the stability of the system and can also make changes in the peripheral implementation transparent for the applications —as long as the driver interface remains unaltered.

Once the application calls the driver, the execution passes to the driver until it returns. Now, inspect the code of the driver in `driver/adder.c` . In particular, go to the function `adder_read()` and observe how the device registers are programmed. The driver uses the function `raw_copy_from_user()` to read the user-space buffer into a kernel-space one. Then, it programs the peripheral registers in the same way that we were doing previously from the user space.

Finally, the driver instructs the peripheral to start and waits until the "done" flag is signaled:

```
do {
status = ioread32(adder_mem.baseAddr + REG_STATUS);
} while ( ( (status & 2) != 2)  );
```

► **Question:** *What do you think will happen while the driver is waiting? Will the CPU be busy? Use `htop` or `top` to observe the system behavior.*[2]

► **Question:** *Do you observe any difference between the system occupation report of the previous examples and this one? Use `top` and look into the summary at the beginning of the report to see the user, system, and idle times.*

The execution is mostly focused inside the driver function, yet the time is accounted to the application. This is because system time is accounted for the process that asked for the kernel services.

[2]The application has been modified to repeat the computation ten times so that there is enough time to observe the system behavior. The total execution time should be around 9 s. If the system cannot allocate enough direct memory access (DMA) memory, try to reduce the length of the vectors and increase the number of iterations proportionally.

3.2 Manage the interrupt and sleeping

We can use a driver not only to isolate user applications from the peripheral interface and increase the system robustness, but we can also use the interrupt mechanism to free the processors. While the task is performed by the accelerator, the processors are free to run other tasks, or they can be put to sleep to save energy and reduce their temperature.

The peripheral can generate an interrupt signal when it finishes its work. Therefore, we can program it to start processing a set of vectors, and wait until the interrupt is signaled. One mechanism to allow the process to sleep is to use a "wait queue." Check the function `adder_read()` in `driver/adder.c` to see how it works.

The calling thread (which is running in kernel space, inside the device driver, and on behalf of the application) gives back the use of the processor to the scheduler until it is signaled by an external source to execute again. When the peripheral interrupt arrives, the interrupt handler signals the driver task to be activated again, which makes it resume its original execution. Once the wait ends, the driver knows that the peripheral has completed its execution and can return the results to the application (check the additional tests in the source code to verify that the thread is not awakened by other signals).

To execute this example, first generate the bitstream of the vector adder (with the interrupt line connected to the PS) and program it into the FPGA. Then, enter into the `driver` folder, run `make` and, if there are no errors, `./load`. Finally, run `make` to build the applications and execute `sudo ./VectorAdder`. When the experiment is finished, do not forget to unload the driver with `./unload`. `dmesg` can be used at any time to inspect the messages sent by the driver to the kernel log.

As we are still using the Xilinx-provided interface to allocate DMA buffers and obtain their physical addresses, rather than kernel functions inside the driver, the application has to be executed with `sudo` in this case.

> ► **Question:** After running the application, what do you observe in the timings?

> ► **Question:** Check the CPU utilization with `htop`. What do you see compared to the version of Session 2? *Consider that, while the peripheral is processing the vectors, the processor is free to execute other tasks; in a sense, it is as if our system now had three processors, two general-purpose CPUs, and one specific-purpose CPU.*

IMPORTANT NOTE: Never allow an application to pass a physical address to a peripheral in a production environment. The reason is that the application can fabricate a non-valid address and crash the complete system, or use it to inspect/modify the memory contents with malicious intentions. Instead, the driver must perform the memory allocation and pass a virtual address to the application and the physical address to the peripheral as necesary.

References

[1] The kernel development community, "The Linux Kernel 5.12.0," Apr. 2021. [Online]. Available: https://www.kernel.org/doc/html/latest/admin-guide/index.html

[2] S. Venkateswaran, *Essential Linux Device Drivers*, 3rd ed. Prentice Hall, Dec. 2008.