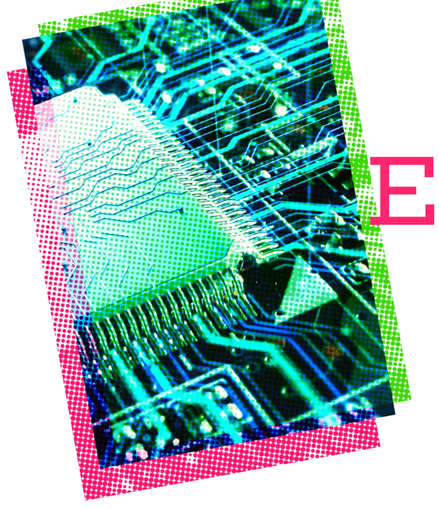

Embedded FreeBSD

Embedded FreeBSD: Learning to Walk–Interfacing to the GPIO System

BY CHRISTOPHER R. BOWMAN

In the last column, we created a simple circuit that blinked the LEDs on the board, and we learned two different ways to load this circuit into the FPGA. Sadly, when we loaded our circuit, the CPU stopped running. Furthermore, while this is mildly interesting, there is no interaction with the CPUs on the chip. In this column, we'll take a little more complex step into Vivado, learn how to keep our CPU running when we load circuits, and explore the GPIO system in FreeBSD.

Previously when we used either U-boot or `xbit2bin` and `/dev/devcfg` under FreeBSD we saw that FreeBSD halted. What I think happens is the processor's system stops running. Turns out the circuit `FPGA.bit` file we used didn't include configuration information for the processor system. In this installment, we'll fix this.

I vacillated over how to present the information in this episode. From a learning standpoint, the most natural way to do this is probably using Vivado's GUI. On the other hand, GUIs do not lend themselves well to automation for the obvious reason that they require a human to run the GUI. Further, it is difficult and tedious to describe the GUI steps. Fortunately, Vivado has two features that make it relatively easy to work around this. When working with the GUI, the Vivado tool produces a `.jou` file which is a journal of all the TCL commands that the GUI is executing under the hood. Vivado also provides the TCL command `write_project_tcl` which can be used to recreate the project file that Vivado creates when you use the GUI. I generally prefer using the `.jou` file as I find the scripts more compact and understandable and if I run the scripts I can then either start the GUI or use `write_project_tcl` to write a project script. Scripts also seem a more natural fit for a revision control system like git.

> In this column, we'll take a little more complex step into Vivado.

If we look at "Figure 1-1: Zynq-7000 SoC Overview Figure 1-1: Zynq-7000 SoC Overview" from "UG585: Zynq-7000 SoC Technical Reference Manual" we can see that there are a

variety of peripheral blocks (UART, I2C, SPI, etc.) which can be connected via a multiplexor to external pins. Section "1.2.3 I/O Peripherals" of the same manual details a bit more of the capabilities. For our purpose, now, we are just going to take note that we can route signals from the gpio device out to pins on the chip with a fair amount of flexibility. If we also look at the Arty Z7 Reference Manual section 12 "Basic I/O," we can see that the green LEDs on the board are connected to chip pins which sink to ground via current setting resistors. If we can set these pins high, the LEDs will light up, and conversely, if the pins are set low, the LEDs turn off.

To toggle these pins, we'll use the Vivado software to route the GPIO device outputs to the LED pins which will allow the GPIO device to control them. I didn't know it when I started this journey, but FreeBSD has a GPIO subsystem, and somebody has even kindly written a driver to make this all usable from user space.

FreeBSD has a GPIO subsystem, and somebody has even kindly written a driver to make this all usable from user space.

To get started, clone the git repo onto a Linux host with the Vivado tools (I showed how to set up a bhyve in a previous installment) and type **`make`** at the top of the repo. If you have the Vivado tools in your path and everything works right, it should run Vivado and pull in a script that will instantiate the processor subsystem and connect the first four EMIO pins of the GPIO device to the LED pins. The inclusion of the processor subsystem will fix the problem we previously had with the processors stopping when we programmed the device with a bit stream.

Look for the **`zynq_gpio_leds.bit`** file built by running **`make`** in the top level of the git repo. Program this into the chip as we did last time using the **`xbit2bin`** program:

```
# xbit2bin zynq_gpio_leds.bit
```

You should see exactly nothing happen. Not very exciting, but at least the processor should still be running.

Now, we need to use FreeBSD's GPIO subsystem. Typing **`man gpioctl`** gives a nice summary of what is possible.

As root, we can run the **`gpioctl`** program to list the available pins:

```
# gpioctl -f /dev/gpioc0 -l
```

Didn't work, did it? Yep, I was a little surprised by this, too. Looking at the GPIO source in **`/usr/src/sys/arm/xilinx/zy7_gpio.c`** I see there are probe and attach functions in the driver, but looking at my ARTYZ7 system **`dmesg`** output, I don't see anything indicating the device was found. Looking more closely at the probe function:

```
static int
zy7_gpio_probe(device_t dev)
{

        if (!ofw_bus_status_okay(dev))
                return (ENXIO);
```

```
        if (!ofw_bus_is_compatible(dev, "xlnx,zy7_gpio"))
                return (ENXIO);

        device_set_desc(dev, "Zynq-7000 GPIO driver");
        return (0);
}
```

I can see that just about the only thing required to find the device is having the function **`ofw_bus_is_compatible(dev, "xlnx,zy7_gpio")`** return true.

In embedded systems, like most ARM systems, the hardware generally isn't self-identifying like a modern PCIe bus. The software can't auto-identify what hardware is present and where its control registers are in the memory address space. For this reason, many operating systems use FDTs (Flattened Device Trees) to describe their device memory map. FDTs are text files that describe information about an embedded system including, among other things, what devices are present and where they are in memory. This allows the software to work with a variety of devices without needing to hard code information. The same kernel can often work with slightly different devices just by using a different FDT. FDTs are translated from DTS files (Device Tree Source) into DTB (Device Tree Binary) files via a tool called **`dtc`**, the device tree compiler. **`dtc`** has options that allow you to compile a DTS or decompile a DTB. The latter comes in very handy. For instance, you can ask the kernel for the DTB it's using and have **`dtc`** turn it into text with:

```
# sysctl -b hw.fdt.dtb | dtc -I dtb -O dts
```

If we look for the gpio section, we see (among other things) this:

```
        gpio@e000a000 {
            compatible = "xlnx,zy7_gpio";
        };
```

The compatible string in the DTB isn't what the driver is expecting, so it assumes the device isn't present. If you look at the FreeBSD boot output, you'll see that the kernel is using the DTB from the EFI firmware emulation that U-boot is providing. We could change this by fixing what U-boot is providing, but that would require patching and recompiling u-boot. Instead, FreeBSD provides the capability to load a DTB file at the kernel loader prompt or using a **`loader.conf`** variable. You can load a DTB file from the loader using the following **`/boot/loader.conf`** variables:

```
# fdt_name="/boot/dtb/zynq-artyz7.dtb"
# fdt_type="dtb"
# fdt_load="YES"
```

This works but there is also a third way. Turns out you can create FDT/DTS overlays which are patches to an FDT/DTS/DTB. We just need a DTS overlay that adds the correct compatible string and we should be good to go. Here is the heart of the DTS overlay I've included in the **`dts`** directory with a **`Makefile`** from the project repo:

```
&{/axi/gpio@e000a000} { compatible = "xlnx,zy7_gpio"; };
```

We may look at FDT files in more detail in a future column, so I won't explain it here. Instead, I'll just give you a flavor of the file. Once you've built the DTB overlay, take the generat-

ed DTB and put it in **/boot/dtb/overlays**, and add the following to **/boot/loader.conf**:

```
fdt_overlays="artyz7_gpio_overlay.dtb"
```

Reboot and note the new **dmesg** output:

```
gpio0: <Zynq-7000 GPIO driver> mem 0xe000a000-0xe000afff irq 5 on simplebus0
```

Now let's try that **gpioctl** command again and you should see a line like this among others:

```
pin 64:   0 EMIO_0<IN>
```

We need to tell the GPIO subsystem to configure the pin as output, and then we can try toggling it:

```
gpioctl -f /dev/gpioc0 -c EMIO_0 OUT
gpioctl -f /dev/gpioc0 -t EMIO_0
```

I'll wait. Try not to stand up and dance when the light comes on. See if you can figure out how to turn on the other LEDs and then run the script in **scripts/blink.sh** with an argument of 2. You should see the LEDs blink as they count in binary.

If you've got questions, comments, feedback, or flames on any of this I'd love to hear from you. You can contact me at articles@ChrisBowman.com.

CHRISTOPHER R. BOWMAN first used BSD back in 1989 on a VAX 11/785 while working 2 floors below ground level at the Johns Hopkins University Applied Physics Laboratory. He later used FreeBSD in the mid 90's to design his first 2 Micron CMOS chip at the University of Maryland. He's been a FreeBSD user ever since and is interested in hardware design and the software that drives it. He has worked in the semiconductor design automation industry for the last 20 years.