Embedded FreeBSD: Custom Hardware

BY CHRISTOPHER R. BOWMAN

When I started this journey using FPGAs and FreeBSD, I chose the Digilent Arty Z7-20 because it was among the least expensive of the Zynq-based boards and it had expandability. It has both a set of pins in the physical form factor of the Arduino shield and a set of connectors conforming to the PMOD standard. In hindsight, I think I would choose a board that sacrificed the Arduino shield connector (which I've never used) for more PMOD interfaces. There are lots of PMOD devices and many of them are pretty cheap. For this column, I'm going to use the PMOD SSD.

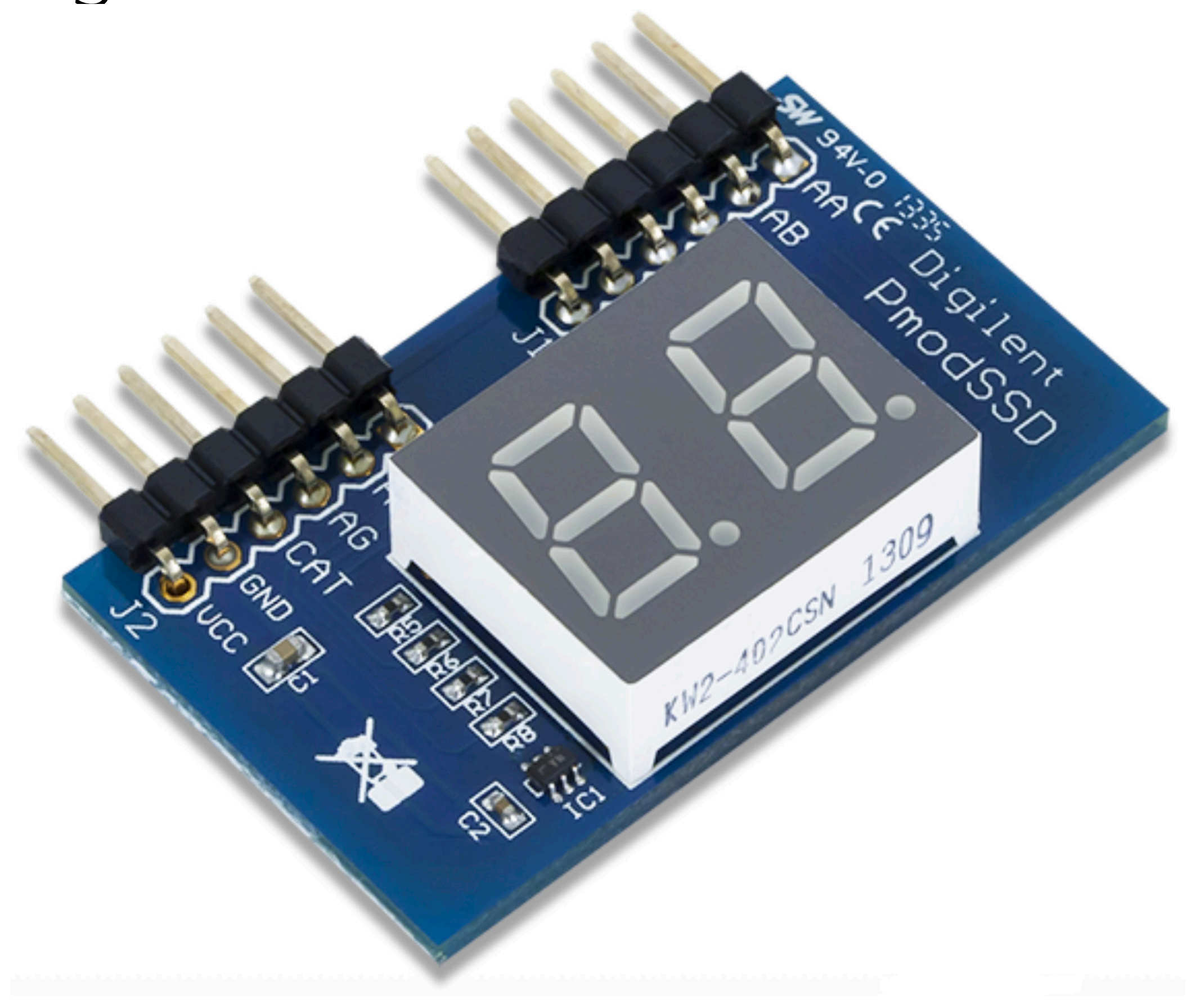

Figure 1 PMOD SSD

This is a dual Seven Segment Display (SSD, not to be confused with a Solid-State Disk) that you can plug into two of the PMOD connectors on the ArtyZ7 board. If we look at the schematic it's clear that simply setting a couple of pins high or low will configure the LEDs that are lit up, but then there is this line "Because only one digit can be lit at a particular time, users that want to use both digits to display a particular value will need to alternately light up the two digits at least every 20 milliseconds (50 Hz)." You can only have one digit or the other light up at a time. To display from both simultaneously, you need to change the AA-AF pins and then toggle the C pin every 20ms. I haven't tried it but I doubt I'd be able to call the GPIO system 50 times a second reliably. This looks like a good use case for hardware. It's not a difficult circuit. We'll have a pair of registers, and we will switch connecting each register to the pins at 50Hz: two registers, a counter, and some MUXes. Instead of registers, we could use 14 GPIO pins and just put the counter and MUXes in the hardware. That's certainly the quick and simple solution, but at some point, you're going to want to build hardware with a conventional register interface. So, let's build a simple memory-mapped register device and hang it off the AXI bus. It will be a simple use of the AXI bus but it will provide a powerful example for more complex uses. Because of this choice, we'll need to build our own driver, but that will be a good example too.

This project has both hardware and software components as well as documentation. You can get a copy using:

```
# git clone http://github.com/axi_mm_ssd
```

Things are a little more complicated now, some components are meant to be built on a Linux machine or VM (the FPGA bitstream), others are meant to be built under FreeBSD (everything else). I'm using my **`bhyve`** setup from an earlier column for the Linux side so I can use a single machine and don't have to continuously reboot each time I make a hardware change requiring a rebuild of the FPGA. The hardware should be built under Linux (unless you've gotten the Xilinx/AMD Vivado tool to run native under FreeBSD using Linux emulation, in which case, could you tell me how you did that?) As GNU **`make`** is the standard **`make`** under Linux, there is a GNU-style **`make`** file. You'll also see that the hardware build script is much more complicated.

I constantly struggle to find a good way to script my hardware builds for Vivado. This time I worked out much of what I needed to do in the GUI and used the Vivado command **`write_project_tcl`** from within Vivado to write out a project creation script which I've modified and augmented. I'm not super happy with it, but it seems to work reliably for me.

Using Vivado's GUI and IP creation I created a simple AXI slave example which helpfully generated Verilog for a complete AX4I-lite memory-mapped slave interface. I took this example and modified it a bit for my purposes adding the counter and MUXes and routing the pins. Then I used the GUI to connect that AXI4-lite slave to the AXI bus and assign it an address using **`assign_bd_address`**. You can just call **`make`** which will do all this via a script.

Let's start by looking at the hardware that's going to be built. If you go into the documentation directory in the repo and build the documentation using **`make`**, you'll see a section describing the hardware interface. Briefly, we're going to build a memory-mapped device with 3 registers. The first will have a constant read-only value that will positively identify the device. I find this useful when you're starting from zero and you're not really sure what you're doing from a hardware or software perspective. If nothing works the way you expect it but you see that magic number in memory at least you know you're on the right track. I'm also looking at ways to use the **`git`** hash and the build date to tag my designs so that when a circuit loads in the fabric I know exactly which version of my design loaded. Not so much a problem when you're dealing with a fully debugged board from a commercial vendor, but a huge help when you're doing it all by yourself, adding bugs in the hardware and the software both of which you're just learning how to write. Unlike building a chip, we don't have to create a $5 million set of masks for each hardware spin, so we can afford this overhead and take it out if it's an issue when everything is working. The other two registers will each control the display of a digit, with each having a bit for each segment of its digit. Each digit can have all the segments set or cleared simply by writing a value to the memory address of the register controlling that digit.

Now that we've talked about the hardware and how to build it and since this is the *FreeBSD Journal* let's take a closer look at the software side. In s**`oftware/driver/freebsd/kld/ssd.c`** you'll find a fairly straightforward KLD that adds a few **`sysctl`** entries that end up writing the memory-mapped registers which drive the seven-segment displays. This KLD is a fairly straightforward example as you might find in Joseph Kong's excellent "*FreeBSD Device Drivers: A Guide for the Intrepid*." There are a couple of interesting things that differentiate this driver from those you might find on ubiquitous AMD64 PC workstations with their standard PCIe bus. Here is the probe function:

```
static int
ssd_probe(device_t dev)
{


//   device_printf(dev, "probe of ssd\n");
     if (!ofw_bus_status_okay(dev))
          return (ENXIO);

     if (!ofw_bus_is_compatible(dev, "crb,ssd-1.0")){
          return (ENXIO);
   }

     //device_printf(dev, "matched ssd\n");
     device_set_desc(dev, "AXI MM seven segment display");
     return (BUS_PROBE_DEFAULT);
}
```

Note the `ofw_*` functions, we talked about these in the last column and discussed how they're used to read properties from the FDT/DTS files. This time around we'll create our own entries in our FDT overlay to describe our device type, address, and register set. Our overlay source is in `software/driver/freebsd/kld/arrtyz7_ssd_overlay.dts` and the interesting part looks like this:

```
&{/axi} {
     axissd: ssd@043c00000 {
               compatible = "crb,ssd-1.0";
               reg = <0x43c00000 0x0004>;
          };
};
```

As we described in the last column, the `compatible` line identifies our device, via the vendor and version separated by a comma. In this case, I built the device, so I used my initials as the vendor. This project is pretty much done so I'm unlikely to modify it, but if I did modify it and produced a V2 or subsequent version, I could allow the driver to differentiate those versions in case the register layout or programming paradigm was different. This one driver could handle multiple different versions of a similar device and would know which was present based on the version in the FDT.

The `reg` line tells the kernel where the device registers are located in physical address space and how much address space is used for device registers. In our case, the device was placed at physical address `0x43c00000` and there are four registers. While the interface only requires three registers and the documentation describes three, the Verilog actually implements four registers.

If we turn our attention to the attach function it looks as follows:

```
static int
ssd_attach(device_t dev)
{
     struct ssd_softc *sc;

     device_printf(dev, "attaching ssd\n");
     sc = device_get_softc(dev);
     sc->dev = dev;

     int rid;

 AXI_MM_SSD_LOCK_INIT(sc);

     /* Allocate memory. */
     rid = 0;
     sc->mem_res = bus_alloc_resource_any(dev,
               SYS_RES_MEMORY, &rid, RF_ACTIVE);
     if (sc->mem_res == NULL) {
device_printf(dev, "Can't allocate memory for \
device\n");
          ssd_detach(dev);
          return (ENOMEM);
     }
#define MAGIC_SIGNATURE 0xFEEDFACE
#ifdef CHECKMAGIC
int32_t value = RD4(sc, AXI_MM_SSD_SGN);
     if (value != MAGIC_SIGNATURE) {
     device_printf(dev, "MAGIC_SIGNATURE 0xFEEDFACE \
not found! value = %x\n", value);
          ssd_detach(dev);
          return (ENXIO);
     }
#endif
   axi_mm_ssd_sysctl_init(sc);
   device_printf(dev, "ssd attached\n");

   return (0);

}
```

This code first initializes a lock. I'll come back and speak about this more later. Next, the driver allocates bus memory. I believe that by setting `rid=0` we're asking the bus DMA system to allocate the first bus resource associated with this device. I haven't actually traced this code, but I think since we have `reg = <0x43c00000 0x0004>;` in our FDT overlay this block of memory is the resource that gets allocated. I'm guessing if we have multiple such lines, they would be successive resource IDs. If you know whether I got this right, I'd be happy to hear.

Since this is the very first device I've built and the first driver I've created for it, I've added some `ifdef'd` code to read the first register and look for my magic number: `0xFEEDFACE`. The RD4 macro is defined in the file like so:

```
#define RD4(sc, off) bus_read_4((sc)->mem_res, (off))
```

This passes the bus resource we received from `bus_alloc_resource_any()` to `bus_read_4()` and returns the value stored in our register at the offset from the resource base.

By using the bus resource system, we avoid having to hard code our register addresses in our driver. This makes it very easy to change the FDT/DTS if I move the hardware address in my design. It also means I don't have to translate the physical address to a kernel virtual address. As I learned the hard way, kernel addresses don't map directly to physical addresses (and the magic number in the first register helped me figure this out.)

With this code, I can be pretty sure that if I read the first register and get back the correct hard-coded value I'm actually talking to the device I designed. When this is your first time with a whole toolchain for building devices and attaching them to the AXI bus and you don't really understand it all, and you didn't know that kernel physical addresses aren't virtual addresses, this can be *incredibly* reassuring. If I was a real software guy, I might even try adding a custom resource in my FDT/DTS overlay that specified the expected value that would read back instead of hardcoding `0xFEEDFACE`. You can see how that could be very useful. A different constant could signify a different version of the device with a different register layout or semantics and this would allow me to verify that the FDT/DTS matched the device the driver is expecting. For now, I'll leave this as an exercise for the reader (patches welcome).

We can now probe and attach to our device and be fairly certain the device is actually there. Let us turn our attention to how we use it. It's not clear to me what the software interface to such a device should look like from user space. The Unix philosophy seems to be that everything is a file, but this device just doesn't seem like a file. I could create a device file in `/dev` that I could open and then I could define `ioctl` functions that would allow me to set the values of the two registers controlling the seven-segment displays. This would allow me to use file permissions to control access to the displays. I chose a different but similar approach. I chose not to use a filesystem device node and instead of `ioctl`s I used `sysctl`s.

Just before `ssd_attach()` returns it creates two `sysctl`s. One gets and sets the value of one of the seven-segment digits and the other `sysctl` sets the value of the other digit. The `sysctl`s should show up in the `sysctl` tree as `dev.ssd.0.tens` and `dev.ssd.0.ones`. Depending on your orientation to the board, these may seem reversed to you.

`axi_mm_ssd_sysctl_init()` registers the two `sysctl`s, and the code for one of these is as follows:

```
static int
axi_mm_ssd_proc0(SYSCTL_HANDLER_ARGS)
{
     int error;
     static int32_t value0 = 1;
     struct ssd_softc *sc;
     sc = (struct ssd_softc *)arg1;

     AXI_MM_SSD_LOCK(sc);

     value0 = RD4(sc, AXI_MM_SSD_SR2);

     AXI_MM_SSD_UNLOCK(sc);

     error = sysctl_handle_int(oidp, &value0,
                  sizeof(value0), req);
     if (error != 0 || req->newptr == NULL)
          return (error);
```

```
    WR4(sc, AXI_MM_SSD_SR2, value0);

    return (0);
}
```

This code uses the lock created in our attach function and reads from the device register for the first digit using the RD4 macro discussed above. Given I'm only doing a single read or write I'm not sure I need it, but I used the lock to ensure that competing processes trying to set the register will not interfere. Perhaps I should have had the lock wrap the whole procedure, but I'm new to kernel programming and still not sure what functions are safe to call with a lock (If you know I'd love to hear from you). **`sysctl_handle_int`** will take the old value, publish it back to userland, and return the new value to be written into our register using WR4. John Baldwin has written an excellent article on the **`sysctl`** system if you're looking for more information on how it works.

Now that we have a driver that can probe, attach, and set our two hardware registers we just need to build our kld that contains our driver, build our overlay, install it in **`/boot/dtb/overlays`**, and stitch it into the **`loader.rc`** script as we did in the last article. Reboot the system, and after it boots, load the bitstream and then the kld, **`dmesg`** should show you the successful probe messages.

That's all the important pieces of the driver. Let's see how it works in practice. In the git repo, you'll find a small C shell script in **`software/app/test.csh`**. It looks like this:

```
#!/bin/tcsh
set digits = (126 48 109 121 51 91 31 112 127 115)
set delay = $argv[1]
foreach tens ($digits)
  sysctl dev.ssd.0.tens=$tens
  foreach ones ($digits)
    sysctl dev.ssd.0.ones=$ones
    sleep $delay
  end
end
```

There is an array with the values that correspond to the bits that need to be set in one of our registers to encode the numerals 0-9. The script takes a delay value and counts from zero to 99 pausing after each digit changes.

I spent a ridiculous amount of time watching that display count. While in and of itself it's not a hugely useful device, it is a complete soup-to-nuts example of how to communicate to hardware via memory-mapped registers and have that impact on the world outside the chip.

I hope you've found these columns useful. I'd appreciate your comments or feedback. 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.