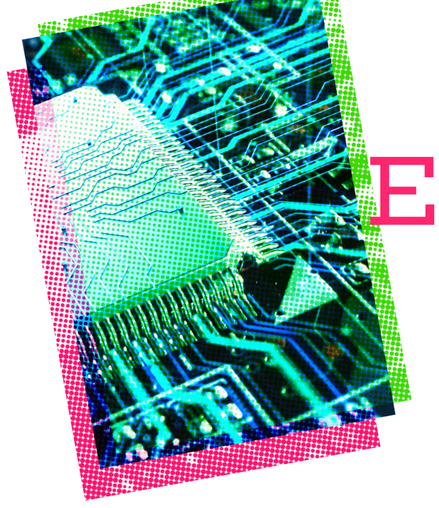

Embedded
FreeBSD

Embedded FreeBSD: Looking Back and Forward

BY CHRISTOPHER R. BOWMAN

We've covered quite a lot of ground over the last year. While I hazard to guess that most people run FreeBSD on conventional AMD64-based PCs, we examined one of the embedded boards that FreeBSD works on: the Digilent Arty Z7-20. While not inexpensive, the Arty Z7 provides an FPGA fabric connected to the CPUs, which differentiates it from less expensive boards like the Raspberry PI or Beagle Boards.

We began by discussing how to obtain a prebuilt image for the board and how to communicate with it over a serial port. In the following article, we discuss rolling our own images and how to use the FreeBSD cross build infrastructure for the ARMv7 system on the Arty board. This vastly speeds development time. We also discussed how to customize the FreeBSD build and then load it onto an SD card, allowing us to create our own custom images.

We examined one of the embedded boards that FreeBSD works on: the Digilent Arty Z7-20.

Having learned to build and customize our own images, we learned how to set up a bhyve instance to run the AMD/Xilinx FPGA software so that we could experiment with FPGA fabric circuits.

Once we had a Linux instance running, we looked at the basic process for building circuits and getting them into the FPGA fabric. There were a lot of details to look at here. We had to create our circuits in a completely new language for hardware design called Verilog. We had to learn how to use the AMD/Xilinx tools to connect our circuits to pins on our chip, which were then connected to LEDs on our board. There was a repo that pulled all this together so that we could use our Linux bhyve instance to build our circuit. Finally, we learned two ways to load the circuit into our chip: one before the system boots and the other from within FreeBSD. We then saw glorious blinking LEDs, reminiscent of lights on a Christmas tree.

Having gotten our first circuit to work, we started exploring more complicated hardware where the CPU and our fabric circuit could communicate. To do this, we used the FreeBSD GPIO system, which required us first to figure out why the GPIO system wasn't working in our initial image builds. We briefly examined the probing of the GPIO driver and discovered that it was absent from our system because the hardware wasn't described in our Device Tree Binary (DTB). This led us to a brief discussion of Flattened Device Tree (FDT) files and how they describe the hardware of many embedded boards. We learned how to modify our

FDT file and build a DTB from it using the Device Tree Compiler (DTC). We learned how to get the FreeBSD loader to load our customized DTB before booting the kernel. Finally, once we had gone through all that, we were able to call the GPIO system from userspace to toggle external pins and, again, light up our LEDs.

In the most recent article, things got interesting. We looked at one of the many available PMOD modules, a dual seven-segment display. We built hardware in the FPGA fabric that could display values on the two displays and presented a register interface over the AXI bus to the CPU. We wrote entries for our FDT, describing the register interface to our hardware, and developed a driver to control the values on the seven-segment displays. In the end, we used the Unix `sysctl` framework as an API for the user space to set the seven-segment displays.

We've now reached a point where we can design circuits in Verilog and put them in the FPGA fabric of our Zynq chips. We've learned to build register interfaces that communicate over the AXI bus so that our CPU can easily interface with our custom hardware. We've learned to describe that hardware to the kernel and to build drivers that allow the FreeBSD system to interact with our hardware. We also learned how to interface with that hardware from the user space. So, what's next?

We've now reached a point where we can design circuits in Verilog and put them in the FPGA fabric of our Zynq chips.

Once we had the basic capability to build circuits and put them in the FPGA fabric, we started learning ways to communicate between our hardware and the CPU subsystem of our Zynq chip. This opens a vast space for exploration and implementation, but there are some limitations. One of those limitations is bandwidth and concurrency. While extremely powerful and flexible, a register interface to hardware is bandwidth-limited. The CPU can only write to the registers so fast, especially when it needs to perform other tasks. Currently, our hardware is bandwidth-limited. It was great for the seven-segment displays, but if we wanted something more bandwidth-intensive, it wouldn't suffice.

Think about a video display. Our Arty board contains an HDMI output port. While a register interface might be viable for a character display, it wouldn't cut it for bit-mapped graphics. A 24-bit color depth 1280x720x60Hz display requires about 166 MB of data per second. We don't want to try to provide that via a register interface. For bit-mapped graphics, the conventional approach is to dedicate a chunk of memory into which the CPU can write and from which the display hardware can read. We need to explore how to build hardware to fetch (or store) data in main memory without using the CPU to move the data. We can utilize our register interface knowledge to enable a processor to configure parameters, such as the base address. However, we prefer our hardware handle the specifics of fetching the display buffer from main memory 60 times a second without CPU intervention.

Adding the capability for the CPU to describe objects in memory to which the hardware should read and write opens a whole new set of possibilities for our Zynq system designs. It also makes me wonder what the impact of that kind of bandwidth competition will be on our dual-processor Arm Cortex A9 system. Digilent makes another Zynq-based board that is like our Arty Z7-20. The Digilent Zybo Z7 is more expensive than the Arty Z7, priced

at $399 for the dual processor version compared to $249 for the Arty. However, the Zybo's memory bus is twice as wide as the Arty's, operating at nearly the same frequency. Further, the Zybo offers 6 PMOD interfaces in comparison to the Arty's two. However, you'll lose the Arduino shell pinout. I think I'm more interested in the PMOD ports. Otherwise, both boards are based on the same chip. There shouldn't be any new drivers that need to be written. The FDT should remain essentially unchanged; it would be interesting to investigate the necessary changes to run this board.

Other things that might be interesting to investigate would be new PMOD modules. You can find a whole slew for sale on the Digilent site. We used the PMOD SSD: Seven-Segment Display in an earlier article. Digilent has retired the PMOD GPS, but I bought one before they did. It uses a UART interface, which, conveniently, is an onboard peripheral in our Zynq chip that can be connected to external pins via the fabric. It should be straightforward to connect this to the system, and I suspect there's open-source software that can communicate with this device via the UART link, enabling various GPS functions such as position and time. What I find interesting about this device is that it also provides a Pulse Per Second (PPS) output. I know that Poul-Henning Kamp has done some work with FPGAs and time-keeping in the past, and I would like to see how that is applicable here.

We haven't done any work with interrupts so far. Still, it should be relatively easy for a fabric circuit to generate an interrupt to the processor and then keep a register with the number of clock cycles between the PPS and the current time. When interrupt servicing software is scheduled, it could read this register and account for the latency between the interrupt and when the driver or time software runs. This might be useful to NTP software, I really have no idea, but it's something I'm curious about. It might be nice to have a local GPS syncronized stratum 1 time server.

Other things that might be interesting to investigate would be new PMOD modules.

I've got a variety of PMOD modules, including accelerometers, OLED displays, and LCDs. It might be interesting to interface with some of them. For example, if you ran an NTP server on your board (perhaps using hardware described above to improve accuracy), you could use an LCD to display atomic time and location continuously.

It almost slipped my mind, but the Zynq boards have Analog to Digital Converters (ADCs) built in. This would undoubtedly be an interesting area for exploration. Still, it may require some external analog circuitry for signal conditioning or buffering before passing on to the FPGA, and this might be outside the scope of an article in the *FreeBSD Journal*. It might be interesting to look at what is required to interface to these on-chip peripherals.

Of course, you can build your own hardware and easily interface it to the pins of the FPGA. I'm curious to hear what you would build if you could.

One obvious place I was going to explore that I hadn't mentioned is running Vivado under FreeBSD. If you've looked at some of my repos, you may have noticed that there is some experimental support for running Vivado under FreeBSD. However, it seems like someone has beaten me to the punch. Michał Kruszewski has written a detailed blog post on the topic. For most of what I do, this is perfect; I can build and simulate my circuits.

Things that aren't quite there yet are loading bitstreams from my FreeBSD host system and using the Vivado Logic Analyzer. The latter two don't work in my behyve Linux instance either, but perhaps I'll experiment with pass-through when FreeBSD 15.0 is released.

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.