CIRCUIT CELLAR®
THE MAGAZINE FOR COMPUTER APPLICATIONS

by Colin O'Flynn

Robust Bootloader for FPGAs

Colin built his LoonBoard Unified Bootloader (LUB) to program Xilinx FPGAs. The LUB, which takes only 207 words of program memory, can self-calibrate its internal RC oscillator.

My Atmel ATmega88-based LoonBoard Unified Bootloader (LUB) is a powerful bootloader capable of loading both an FPGA bitstream and an AVR file. What makes this bootloader better than any of the other bootloaders out there?

Well, in short, it's capable of programming Xilinx FPGAs, it has a robust communications protocol with CRC8, it has redundant bootloader support, and it can self-calibrate it's internal RC oscillator based on an external real-time clock to enable 115,200-bps communications without a crystal. Did I mention it takes only 207 words of program memory? Plus it has a convenient three-letter acronym for its name to make sure you can confuse people instead of speaking clear English!

DISAPPEARING ACT

The size of 207 words is a pretty small bootloader, especially considering that that includes both the FPGA loader and AVR loader. The key is that, when programming an FPGA, you need somewhere to store the programming file. I used an Atmel AT45DB041B DataFlash serial interface flash memory for this project. It's inexpensive, small (eight-pin SOIC), and easy to use. Compared to the 1.5 Mb that my FPGA uses, the AVR's 64-Kb programming file is barely noteworthy.

The magic comes in here. The bootloader resident in the ATmega88 microcontroller is less than 256 words; all it's able to do is load data from the DataFlash memory into the AVR and FPGA. During normal start-up, data is loaded from the AT45DB041B to the FPGA and program execution starts in the ATmega88's flash memory.

There are set bootloader sizes for the ATmega88, so the tinyloader fits in the 256-word space. ("Tinyloader" refers to the program native to the ATmega88 AVR. "LUB," "lub," and "bootloader" refer to the code loaded in the ATmega88 AVR for the purpose of bootloading. "Lubloader" refers to the computer program that is the user interface.) This means there is room for expansion yet! If you want to add some sort of encryption that would prevent an attacker from simply reading out the AT45DB041B, there's room for it.

When bootloading is requested, a bootloader program is loaded from the AT45DB041B memory into the ATmega88's flash memory. This bootloader isn't restricted to work only in the confines of the ATmega88's bootloader section. Now the bootloader can take up all the available application space for the program. Instead of downloading the file to the ATmega88's code memory, the code is downloaded to the AT45DB041B. The tinyloader takes care of loading the user program from the AT45DB041B to the ATmega88's application space.

DATAFLASH OVERVIEW

Atmel's small DataFlash devices are available in sizes ranging from 1 to 64 Mb. I used a 4-Mb AT45DB041B for this project. The interface for programming involves three pins: the master in slave out (MISO) pin, the master out slave in (MOSI) pin, and the serial clock (SCK) pin. Atmel AVR devices have the ability to talk to these devices via their SPI ports.

DataFlash devices are organized in a number of pages. Like any flash memory device, you can only erase or write entire pages at one time. Each page is a block of 264 bytes in the 4-Mb device. To make this easier, the device has two SRAM buffers. The advantage here is that you can write to one buffer and then tell it to transfer to flash memory. While that's writing, you can start writing data to the next SRAM buffer, which streamlines the transfer considerably. In fact, the LUB is faster than most other bootloaders. It takes only about 3 s to download an AVR file and 20 s to download an FPGA file.

Reading from the DataFlash is easy. You can set a start address, and then read the entire DataFlash device in one go. This is important because it enables the tinyloader to ignore the DataFlash's paging setup (it only reads).

Now if only I could get a big bag of cash from Atmel for writing all of that.

FPGA START-UP

Now that you've got an idea of how the LUB works, lets look at the technical details. At start-up, the ATmega88's program will normally remain stored in flash memory, so there is no need to reprogram it. However, the FPGA has its program stored in SRAM, which means you need to configure it every time you power up.

Kim Goldblatt outlines the programming algorithm for the Xilinx Spartan series of FPGAs in the application note, "The Low-Cost, Efficient Serial Configuration of Spartan FPGAs." Kim also describes the process of generating the programming files.

The algorithm used in the LUB was designed to work with the Atmel DataFlash parts. First, it sets the

DataFlash to start reading from the page that stores the FPGA bitstream. Second, it pulses the PROGRAM pin low for several clock cycles. Third, it waits for the INIT line to go high. Fourth, it clears the CCLK pin. Fifth, it delays about 100 µs. Sixth, it pulses SCK to the DataFlash to get 1 bit. Seventh, it puts that bit on the DI line of the FPGA. Eighth, it pulses CCLK to the FPGA. And finally, if the FPGA's DONE pin is low, it jumps to the sixth step. Otherwise it exits.

The application note doesn't tell you how to use the convenient bit file, which is generated by Xilinx ISE to program your device. The document does include information about using the rawbits file, but it isn't generated by default. The bit file is a direct bitstream that you can download to the FPGA without much processing. Well, almost. There is some text at the beginning of the file that needs to be separated from the actual raw binary to download. Luckily, it has been reverse engineered. Some detective work on your part will make it easy to see.

Listing 1 is what the bit file looks like in hex format with about 80 bytes in it. The exact location isn't constant, but the format is. The `0x65` byte is followed by 4 bytes that indicate the file size in bytes. In this example, it's 0x0002C01C, which is 180252 in decimal format.

For my Spartan-IIE with 200,000 gates, that is the exact size for the configuration file specified in the datasheet. Because the configuration file size will never use more than 3 bytes in this system, the `0x65 0x00` combination can be used as the synchronization byte. Following this, 3 bytes are skipped, and data loads in starting with the `0xFF` byte and ending with the end of the file. At the end of the file, several `0xFF` bytes need to be shifted in.

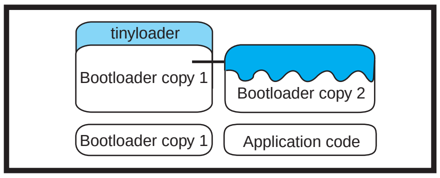

Figure 1—*The upper left corner of the lubloader's layout is the current executable loaded in the ATmega88 microcontroller. The outside files are in the AT45DB041 DataFlash. I'm in the process of updating bootloader copy 2.*

For the LUB system, the bitstream stored in the AT45DB041B is padded with 0xFF. This means that the tinyloader doesn't have to keep track of a 24- or 32-bit counter. It doesn't have to figure out how many bytes have been put in the system, how many more need to go, or when to shift in extra 0xFF. Instead, it simply shifts bytes until the DONE pin goes high. If the system must start when the FPGA is incorrectly loaded, the INIT pin can be checked as well. If the INIT pin goes to a low state from a high state while loading configuration memory, the CRC embedded in the bitstream is incorrect and the FPGA won't start.

AVR BOOTLOADING

Loading data from the ATmega88 is fairly straightforward. In fact, the ATmega88 datasheet includes an example of how to write to the application flash memory from the bootloader section, which is copied almost exactly in the tinyloader. The ATmega88 has a flash memory setup arranged in pages, just like the AT45DB041B. Each page in the ATmega88, however, is only 64 bytes instead of 264 bytes. This means it's a simple procedure of copying a byte from the AT45DB041B to the ATmega88's flash page buffer until the page is full, and then writing it.

SAFETY FIRST

Eventually, it comes time to update the bootloader, which is ideally done via the bootloader. This is normally a risky process. If the bootloader update fails, there's no way to get back in the bootloader to fix it. However, the LUB solves this easily. The ATmega88's resident tinyloader is never updated because it's so simple. Instead, the more complex bootloader is stored in the DataFlash memory, where there is room for a backup. Now there are two bootloaders present, so you always have a functional bootloader. The process is shown in Figure 1.

At start-up, the tinyloader will ask the computer which bootloader to enter. If the first one doesn't work, it's easy to ask the tinyloader to load the backup bootloader, known as the "safe mode" bootloader. The PC software stops you from updating both bootloaders simultaneously, which should stop you from causing any trouble.

The AT45DB041B DataFlash has a hardware write protection feature. So, you can be sure that faulty software won't have a big window in which it could overwrite the AT45DB041B. To ensure that data was reliably written to the AT45DB041B, a checksum of the AT45DB041B's DataFlash content is performed locally. This value is sent back to the computer. This is considerably faster than most bootloader verification, which sends every byte back to the computer. However, the checksum doesn't ensure that the proper data is written from the DataFlash to the AVR or FPGA at start-up, which is a possible point of failure.

COMPUTER COMMUNICATIONS

I needed a way of getting data from the computer to the AT45DB041B DataFlash, so I used a specialized protocol. The main feature is that it's optimized for the DataFlash because each data packet is 264 bytes. This made loading the AT45DB041B easy because I could send data quickly to fill up one page buffer, write the page, and start sending data for the next buffer. At 115,200 bps, my FPGA configuration file takes about 15 to 20 s to send over the serial port.

Having the ability to use all of the ATmega88's bootloader space is important. It's easy to create a communications protocol using CRC-8 and other such features for reliable data transmission. Normally, the added overhead in code size wouldn't be worth it, but

Listing 1—*The bit file generated by Xilinx ISE is about 80 bytes. The 0x00 0x02 0xC0 0x1C is the file size and can be used as synchronization. You can use 0xAA 0x99 0x55… if you want, but you'll need to backtrack to get the file size.*

```
0x65 0x00 0x02 0xC0 0x1C
0xFF 0xFF 0xFF 0xFF 0xAA
0x99 0x55 0x66 0x30 0x00
```

```
→ [SYNC] [1] [LUB_SOT] [CRC8]
← [SYNC] [1] [LUB_ACK] [CRC8]
← [SYNC] [2] [LUB_SIGN_ON] [Minor Ver.] [CRC8]
→ [SYNC] [2] [LUB_ACK] [CRC8]
→ [SYNC] [3] [LUB_SEND_AVR] [CRC8]
← [SYNC] [3] [LUB_ACK] [CRC8]
→ [SYNC] [4] [LUB_264_DATA] [D1] [D2] ... [D263] [D264] [CRC8]
← [SYNC] [4] [LUB_ACK] [CRC8]
  262 more sets of data packets
← [SYNC] [266] [LUB_264_DATA] [D1] [D2] ... [D263] [D264] [CRC8]
← [SYNC] [266] [LUB_NACK] [CRC8]
→ [SYNC] [266] [LUB_264_DATA] [D1] [D2] ... [D263] [D264] [CRC8]
← [SYNC] [266] [LUB_ACK] [CRC8]
→ [SYNC] [267] [LUB_EOT] [CRC8]
← [SYNC] [267] [LUB_ACK] [CRC8]
```

Figure 2—*Typical LUB communication is simple. The right arrows signify communication from the computer. The left arrows signify communication from the LUB.*

there's no real penalty here. The following LUB packet isn't anything too special. It shows just the normal verification features such as packet numbers and CRC:

[SYNC] [Packet Number] [Packet Type]
[Data LSB] ... [Data MSB] [CRC8]

The synchronization byte is used as a simple start of packet. The packet number starts at 1 for the first packet and is 1 byte long. It rolls over to 0 for more than 255 packets. The packet type defines how many data bytes should be expected. It also issues commands. After every packet, an ACK or NACK returns and tells the sender if it needs to resend something.

This process of ACKing every packet is useful for controlling the data rate. The sender will send the next packet of 264 bytes only after it receives an ACK for the previous packet. This means the receiver can wait until the 264 bytes have been processed and stored before requesting the next packet by ACKing the previous packet (see Figure 2).

There is a common code base for both the PC end and the AVR end. There is a slight trade-off in speed, but the code is much easier to maintain this way.

HARDWARE

I/O pins were scarce, so I used the dedicated ADC channels in the ATmega88's TQFP package. I can easily change to normal I/O pins if I need to save some code size.

Figure 3 shows only the most important section of the design. If you use an external crystal, you don't need to worry about the real-time clock, which is used for a precision source to calibrate the internal oscillator.

SELF-CALIBRATION

Yet another feature of the LUB system is that space-hogging calibrations that only need to be run occasionally can be kept in the bootloader. In this example, there is no external crystal. Instead, the internal RC oscillator is calibrated against a real-time clock that provides a 4,096-Hz output signal. The target frequency is 7.37 MHz, which enables error-free data rate communication at 115,200 bps. After calibration, the value is stored in the DataFlash, where it can be read by the tinyloader and stored in the OSCCAL register at every start-up.

The sharp-eyed among you are probably wondering if you'll be left with a dead system that can't communicate if power fails while you're writing this value. No! If the power fails, the ATmega88 AVR will still have the bootloader code loaded into its flash memory. At the next start-up, the tinyloader won't see valid data from the computer. It will proceed to start the code in the ATmega88. This code, which is the bootloader, will calibrate the oscillator and store it. Now you'll have a system that works. At that point you can finish working with the bootloader. Whether or not emulating a certain operating system—an operating system for which troubleshooting involves rebooting—is advantageous is up to you.

GETTING THE BOOTLOADER

The entire LUB project is released under the GNU General Public License. If you distribute the binary code, you must distribute the source code. Remember, this just means the bootloader sources and changes to the PC program. The application you load isn't connected to the bootloader in any way. So, there is no worry about using it in a commercial application for which you wouldn't want to release the source code.

When you download the source code, you'll find it in two folders: PC and AVR. The latter folder is separated into the LUB and tinyloader folders. The AVR targets will compile with any AVR-GCC package such as WinAVR for Windows. To change target AVRs, change both the LUB and tinyloader make files. The rest should be automatic. If you change the make files, be sure to perform a "make clean" and then "make" of the project. Otherwise, the changes might not take.

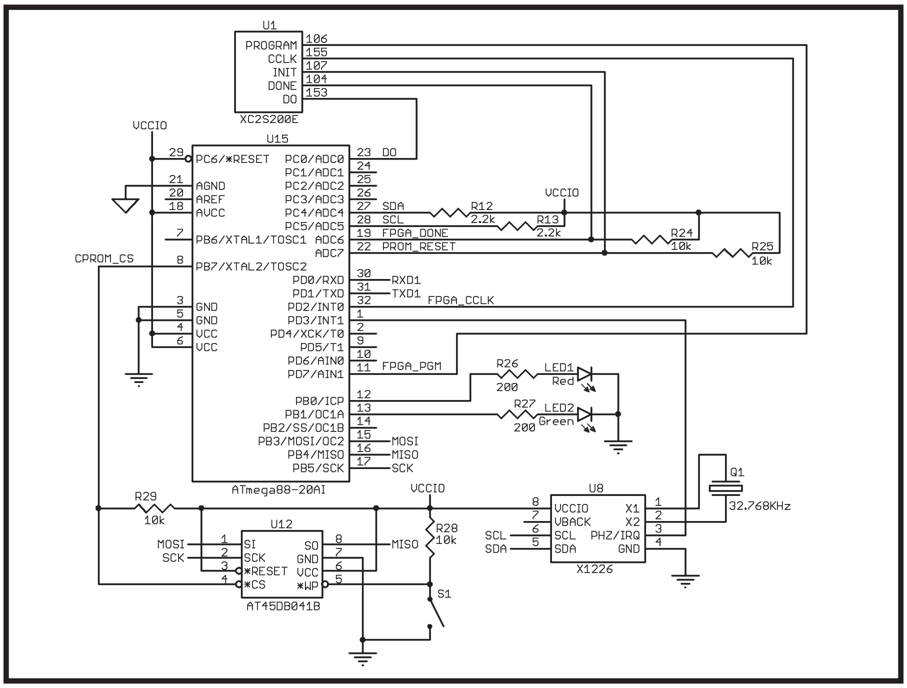

Figure 3—*The LUB code runs from this schematic. I've included only the FPGA's programming interface pins. The complete schematic for my LoonBoard video development system is posted on my web site (www.newae.com).*

Lubloader options	Function
-P <port>	Serial port device
-s	Use Safe mode bootloader
-f <filename>	Bit file to download.d to FPGA
-a <filename>	Hex file to download to AVR
-q <filename>	Hex file for new bootloader
-w <filename>	Hex file for new Safe mode bootloader
-v	Verbose output, use -vv for more
-b <baudrate>	Use data rate specified
-d	Enters Debug mode in bootloader
-h	Display this usage

Table 1—*There are several possible program options to pass to the lubloader program. Of course, if you don't like what you see, you can add your own!*

The PC target should compile with any GCC package in either Windows or Linux. To change the target, you'll have to change the code slightly. There's a hard-coded warning that will come on if the AVR code is bigger than the expected target size. This is likely to change in the future.

USING THE BOOTLOADER

Using the LUB is fairly easy. The PC software has been tested on Linux and Windows, and it might work on other platforms too. It's loosely based on the AVRDUDE open-source programming utility.

The PC software is a command-line-based software called lubloader. The options you can pass to it are shown in Table 1. When the software is running, it will ask you to power cycle the attached device. This forces a reset of the ATmega88, which will make it execute the tinyloader. An example of the FPGA update process is shown in Figure 4.

One of the most useful options is the `-v`, or verbose, option to tell you what it is doing. Keep adding `v`s until you see what you want. For example, `-v` adds some debugging information such as file names. `-vv` adds information about each packet sent. `-vvvv` prints each byte sent and received.

LOADING THE LUB

Before you start using the LUB, you need to get all the code into the AT45DB041B. The process is straightforward.

First, load the main LUB file into the AVR directly and program the fuses to use the internal oscillator. When the AVR resets, it will calibrate the internal oscillator and attempt to communicate with the PC.

Next, use the lubloader software to load the main LUB file into the main and backup spaces in the AT45DB041B. The proper data is now in the DataFlash, but there's no way to load it from the AT45DB041B into the ATmega88.

Now, program the tinyloader file into the AVR directly. Program the fuses to jump to the bootloader at reset and the proper bootloader size. Run the lubloader program with no arguments to check if the system works and force the ATmega88 to load the code from the AT45DB041B.

You should now have a working bootloader system. The lubloader program's verbose outputs are extremely useful when things aren't working right.

FUTURE DIRECTION

I built the LUB because I wanted to become independent of the expensive configuration memory for FPGAs. There are other advantages too. Now one interface can program all the devices in your system. This greatly simplifies field upgrades.

I hope to add new features and support devices to the LUB system. Remember that the code is released under the GNU GPL. Commercial use is not only allowed, it's encouraged (provided you release any changes you make to the core LUB system). Because the LUB binary will be completely separate from your application binary, this shouldn't be a problem.

The project is hosted on Sourceforge, so you have all the standard features at your disposal (e.g., CVS access and a mailing list). Of course, it's quiet right now, but I don't think the software is so perfect that it will stay that way too long.

```
lubloader -P/dev/ttyUS0 -f hardware_interface.bit

lubloader: Attempting to communicate with LoonBoard unified bootloader
lubloader: Power cycle device now to initiate LUB

lubloader: Sending FPGA bit file

Writing | ################################################## | 100% 21.75 s

lubloader: Instructing LUB to calculate CRC
lubloader: Program verify OK
lubloader: Resetting AVR and forcing load of new code
```

Figure 4—*This example sends a new FPGA file to the AT45DB041 and forces its load. The number of pound signs slowly increases and the time counts up during programming.*

JMP 0X0000

Hopefully, you've found this bootloading approach to be as useful as I have. Just one simple interface to load your entire system. By using a low-cost microcontroller and memory chip that may already be on your system, you can drive down the project's cost and keep a rich feature set. ■

Colin O'Flynn is an electrical engineering student at Dalhousie University in Halifax, Nova Scotia. He has been interested in electronics for years. In November 2005, he launched his company NewAE in an effort to reverse the flow of money that he usually pours into his projects and never sees again. You may contact him at coflynn@newae.com.

PROJECT FILES

To download the code, go to ftp://ftp.circuitcellar.com/pub/Circuit_Cellar/2006/187.

RESOURCES

K. Goldblatt, "The Low-Cost, Efficient Serial Configuration of Spartan FPGAs," Xilinx, Inc., XAPP098, November 1998, www.xilinx.com/bvdocs/appnotes/xapp098.pdf.

LUB Sourceforge project, http://lubloader.sourceforge.net.

Xilinx, Inc., "Configuration and Readback of the Spartan-II and Spartan-IIE Families," XAPP176, March 2002, www.xilinx.com/bvdocs/appnotes/xapp176.pdf.

SOURCES

AT45DB041 DataFlash and ATmega88 microcontroller
Atmel Corp.
www.atmel.com

X1226 Real-time clock
Intersil Corp.
www.intersil.com

LoonBoard video development system
NewAE
www.newae.com

XC2S200E FPGA
Xilinx, Inc.
www.xilinx.com