Hacking the XBC Firmware:
Programming the XBC in Standard C++
Part 1
Jeremy Rand and Fahrzin Hemmati
Norman High School and La Jolla High School
jeremy@asofok.org, fahhem@berkeley.edu

Hacking the XBC Firmware: Programming the XBC in Standard C++ Part 1

1 Introduction

Have you ever gotten frustrated by the limitations of Interactive C [1]? Have you ever wished you could program the XBC [2] in standard C++? Well, it's possible, through hacking the firmware, and we're going to show you how. You don't even have to completely make the switch; you can mix IC and C++. C++ has the advantage of being *much* faster, and when writing code in C++, you will have access to more features.

DISCLAIMER: Using an unofficial firmware with an XBC is officially unsupported, meaning that if it doesn't work, KIPR [3] and Charmed Labs [4] are not obligated to help you. KIPR and Charmed Labs have both been very nice to me about this and have tried to be helpful, but there have been many times where they just said either, "We don't know why it's screwing up," or else, "We don't have time to investigate it, we have higher priorities." If this sounds like something you're not up for, firmware hacking may not be for you.

2 The XBC's Software

You've probably uploaded new firmware or bitstreams to your XBC many times, but you may not know what they actually do. The Xport [5] (the smaller top circuit board on the XBC) is essentially a glorified Flash Cartridge for the GBA [6], much like you would use if you wanted to load GBA homebrew or commercial ROMs on your GBA (a ROM is a program for a game console, for those of you unfamiliar with video game emulation). The XBC Firmware is, as a matter of fact, simply a GBA ROM. The Xport doesn't just have Flash, though. It also contains a Field-Programmable Gate Array (FPGA) [7], which is an interesting hardware-software hybrid which allows a computer file containing hardware schematics to be converted on-the-fly into actual hardware circuits. That hardware schematic is the XBC Bitstream. The distinction between what the firmware and bitstream do is important because the firmware is safely and easily hackable, while the bitstream is neither. The firmware has freely available source code, which, while not particularly well-commented, can be configured with some work to do lots of fun stuff. However, the bitstream is closed-source. `Charmed Labs has been known to give it away under a non-disclosure agreement if you have something specific that you want to do with it and if they determine that you know what you're doing, but since the source code was not intended for public release, it is very difficult to decipher. In addition, putting a bad bitstream on your XBC will both void your warranty and almost always force you to reflash it (we'll be telling you how to do this later). In addition, it is possible to actually damage the XBC hardware

with a bad bitstream, although this is very rare and has never happened to us. A bad firmware, on the other hand, won't cause much damage. Usually you'll just have to reload a good firmware through IC, and you're back in business. However, it is possible, though rare, to corrupt your bootloader with a bad firmware that writes to the Flash on the Xport. This has never happened to us, and if it does happen to you, you'll just need to reflash it yourself (following the instructions in Section 6) or just send it in to KIPR for a reflash; your warranty will not be affected.

Since the bitstream is closed-source and generally dangerous to mess with, even if Charmed Labs is nice enough to give you the source, we'll be covering the firmware only. So what is handled by the firmware, and what functions are in the bitstream? The bitstream mainly handles some hardware processing, which is generally unnecessary to mess with anyway. Examples of bitstream functionality include reading the BackEMF inputs, controlling the PWM output pulses to motors, controlling the pins of the serial port, and converting a camera image into "segments" of pixels based on whether they match a color model. The firmware's corresponding functions include running the BackEMF input through a closed-loop algorithm to decide what PWM output gets sent, deciding what bytes to send to the serial port and interpreting the bytes coming back from said serial port, and converting the pixel segments from the camera into full blobs and calculating the statistics of those blobs. As you can see, the firmware deals with much higher-level functions than the bitstream does, which is good for us, since those functions are much easier to deal with. The firmware is written in C++, so you should be at least somewhat familiar with C++ unless you want to be frustrated.

3 Downloading the firmware source code

First you'll need to download the Xport Devkit [8] (for either Windows or Linux, depending on your OS of preference) from Charmed Labs. (If you're on a Mac, you'll need to use a virtual machine such as Parallels [9] to emulate Windows or Linux. You're on your own here.) If you're using Windows, also download Cygwin from the same site (unless you already have Cygwin installed; you probably don't). Install Cygwin if you're on Windows, and then install the Xport Devkit that you downloaded. If all went well, you should now have a directory called xport sitting on your hard drive (for me on Windows, it was C:\xport) and a shortcut to the Xport Shell (on Windows, this was on my Start Menu and Desktop). If you browse through the xport directory, you'll see lots of code. Unfortunately, all that code is useless, because it's many years old (Charmed Labs hasn't updated the Xport Devkit download since August 13, 2004). In fact, it's so useless that you should delete all the subdirectories of xport *except for* devkitadv (which hasn't changed).

Now, you're probably wondering, "So if the download isn't updated, where am I supposed to get the code?" Luckily for us, Charmed Labs has a CVS server hosted by SourceForge, which has all the latest code on it. This is literally up-to-the-minute; every time a KIPR programmer updates the firmware source code and submits it to Charmed Labs, it's available on CVS instantly. If you've never used CVS, don't be alarmed! It's very easy to get the code from the CVS server. If you're on Windows, all you need to do is download and install TortoiseCVS[10]. (If you're on Linux, they expect you to be able to do this on your own... sorry.) Once you've installed TortoiseCVS, right-click in any location in Windows Explorer, and click CVS Checkout. Enter the settings from Figure 1, and start the checkout.

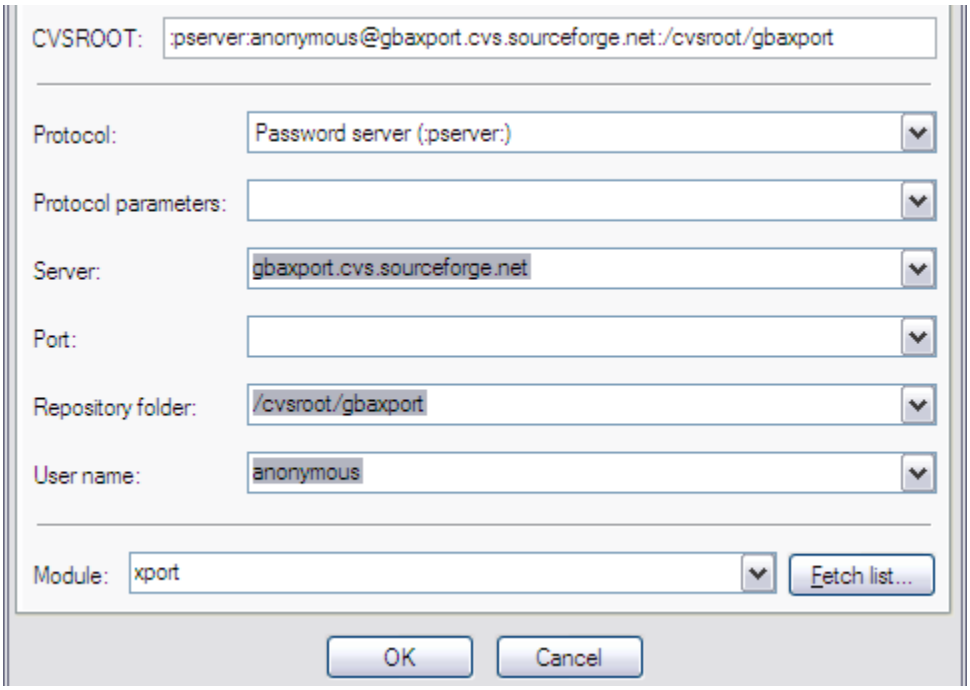

Figure 1

All of the code from Charmed Labs will be downloaded to the directory in which you clicked CVS Checkout. This will take a while, so feel free to go get a snack and come back in 5-10 minutes.

Once it completes, you'll notice that you have a new xport directory where you checked out the code from CVS. Take a look; it has up-to-the-minute versions of every directory you deleted! (As mentioned previously, it doesn't have devkitadv; if you didn't follow instructions and you deleted that, you're going to have to reinstall Xport Devkit.) Simply move these directories to the xport directory that the Xport Devkit installed. You now have the XBC firmware source code on your computer!

4 Compiling the firmware

Now that you have the source code on your computer, it's time to compile it. Start up the Xport Shell that the Xport Devkit installed. You will now be at a Bash prompt. For you Windows users, this is relatively similar to the DOS prompt, however, the commands are Linux commands instead of DOS commands. The dir and cd commands are all you really need to navigate, and they're identical to DOS, but if you need additional help with the Bash prompt, search Google for a list of Linux or Unix commands. If you've never used either a DOS prompt or a Bash prompt, then search Google for how to use the cd and dir commands in DOS.

The first thing you might notice is an error, e.g. "Failed to initialize" or "Xport is not detected" at the top of the screen. Not all users will see this; it depends on your setup. This error can be ignored for now (and will be explained later). The first thing you should do is use the cd command to enter the directory xport/src/libgba. This contains all the libraries for compiling programs for the Game Boy Advance. Type "make" and press Enter. If everything is set up correctly, it will compile, copy some files, and dump you back at the Bash prompt. If you got

any major errors, then you did something wrong. Back up, reread everything above this point, and make sure you followed the instructions. If you think you did it properly, you may want to ask at the Charmed Labs forums [11] for assistance. (But remember, since this is unsupported, they are not obligated to help you! Be polite!) Once you've compiled libgba, repeat the process for xport/src/xpcomm, xport/src/waveutil, xport/examples/xrc/libxrc, and xport/examples/xrc/librpc. xpcomm and waveutil are not 100% necessary to mess with the firmware, but for the sake of completeness, just compile them now.

Once all the libraries are compiled, you can compile a Hello World application to make sure that the libraries work properly. Navigate to xport/examples/helloworld_cxx, and make that. It should generate a GBA ROM called helloworld_cxx.bin, which you can test in your favorite GBA emulator (VisualBoyAdvance is included in the xport/bin directory, and is recommended). If VisualBoyAdvance runs the ROM correctly (it should display "Hello World!" at the top of the GBA screen), then libgba is set up properly.

Now you have to test the XBC libraries. This is harder because obviously VisualBoyAdvance isn't going to have a clue how to deal with that. The temporary solution is just to make sure it compiles – you'll be testing it on a real XBC later. Navigate to xport/examples/xrc/botball1/icfirmware. As you can guess by the name of the directory, this is the IC firmware that your XBC runs. Try to make it. If everything is set up properly, it will generate another GBA ROM (called icxbc.bin). If you get an error, don't panic! I got errors the first time I ran it, and the culprit is usually missing empty directories. Make sure that you have the following directories in the icfirmware directory: bin, icsrc, include, lib, src. You may have others, but if any of those are missing (bin, include, and lib were missing for me), just create them. If there is already a file named what the directory should be (e.g. a file named include), you can delete that file. Try again, and it should compile.

5 Uploading to your XBC

Okay, now you're ready to upload the firmware you compiled to your XBC. But first, there's a catch. As you've noticed, the compilers generate a standard GBA ROM which will run all by itself with no additional software present on the GBA. This is fine normally, but you may have noticed a Bootloader on your XBC (when you turn it on, it flashes the words "XBC Bootloader"). This Bootloader is responsible for the factory test mode (nicknamed the "Christmas Light mode") when you hold L or R when powering on the GBA, and also is responsible for letting you upload a firmware or bitstream through the serial port. Unfortunately the Bootloader is the "main" GBA ROM running on your XBC, not the firmware. If you were to upload the standard firmware GBA ROM that you got in the last step, it would expect to be at the start of the GBA cartridge on the Xport's Flash memory. But the Bootloader would put it 128KiB past that point, since the Bootloader needs its own space at the start of the GBA cartridge. The result would be that all of the memory locations would be off by 128KiB, causing your firmware to crash the GBA as soon as the bootloader runs. You would get the cryptic text "Running at 0x8020000", which is the address where the firmware resides. Luckily, there's a very easy way to make the firmware expect to be 128KiB past the start of the GBA cartridge. Just type "make USE_BOOTLOADER=1" instead of "make" when you're compiling the firmware.

Now that you've compiled the firmware so that it knows it's being used with the Bootloader, you can take the icxbc.bin file that it generated, and rename it libxbc.frm. Copy this to your Interactive C library directory, backing up the existing firmware file if you're so inclined. Start up IC, and tell it you want to upload the firmware to the XBC. Follow the instructions just like normal. When the firmware upload is finished, your XBC should still look like normal, and be running what looks like the same firmware as previously, but it's actually the firmware that you just compiled! If you were to change anything in the firmware, you'd see those changes on your XBC now.

6 Cport Cable and Reflashing

A quick aside here. You probably noticed that uploading the firmware takes a long time. How'd you like to upload the firmware and bitstream, together, in less than 15 seconds? Well, you can with a Cport cable. The Cport cable is how KIPR unbricks the XBC's that have a corrupted bootloader or bitstream, and they transfer at about 300kbit/s, compared to 33kbit/s for the serial port. You can buy one from the Botball Store [12]. There is a very nice tutorial on using the Cport cable in the Xport User Guide in xport/docs, so we won't waste your time here with details. You will need a parallel port to be able to use the Cport cable. If you don't have a real parallel port, you'll need to buy a Quatech SPP-100 Cardbus card [13]. A USB adapter or a cheap Cardbus card just plain will not work. If you don't have a Cardbus slot, or don't want to spend $100 on the Quatech, then you will have to live without the Cport cable. Sorry.

Remember the error that shows up when you start an Xport Shell ("Failed to initialize" or "Xport is not detected")? That's your computer complaining that it can't find a Cport cable. Once you have a Cport cable installed, that error will stop showing up. You'll be able to compile the firmware, and upload it through the Cport cable, just by typing "make" (*not* "make USE_BOOTLOADER=1"). It will prompt you to power-cycle your GBA, and the firmware and bitstream will be uploaded in less than 15 seconds. Note that this will overwrite your bootloader, and you will therefore be unable to use the serial port to upload a firmware or bitstream in the future. To put the bootloader back on, make the xport/examples/xrc/botball1/bootloader program, and let it upload to the XBC with the Cport cable. This will of course overwrite any firmware which you uploaded through the Cport cable, so you have a catch-22. The bottom line is that if you know that someone without a Cport cable will need to upload firmware to your XBC, be nice to them, and, after you're done testing your firmware mods with the Cport cable, reload the bootloader through Cport and upload the firmware through serial. If, after you finish reading this paper, you decide that this is too much trouble, try hacking something together using the --pad-to 0x8020000 flag on the objcopy command in the bootloader makefile combined with the cat command to stick the bootloader and firmware into one file to be Cported to the XBC. We did it, so we know it’s possible, but if we did everything for you, you wouldn’t learn, would you?

7 CallMLs

Okay, now that you know how to compile and upload the firmware, let's start messing with it! You've probably noticed a function in the IC libraries called callml. You probably have no idea what it does. Well, you're about to find out. CallML is used for an IC program to call certain functions in the firmware. By adding CallMLs, you can add new features to IC, implemented in C++!

CallMLs are stored in two files within the icfirmware directory: src/libicxbc/XBCRobot.cxx and src/libicxportcommon/ICRobot.cxx. The reason for them being in two files is historical, and it doesn't really matter which file you edit. If you're looking for an existing CallML, look through both. For both files, the CallMLs are handled by the CallMLTranslator functions. There are three of these functions per file, based on whether the CallML takes 1, 2, or 3 arguments. The "ID" of a CallML is a number starting with the number of arguments it takes, and having 3 digits for ICRobot.cxx CallMLs and 4 digits for XBCRobot.cxx CallMLs. For example, a CallML taking 2 arguments and residing in ICRobot.cxx might have an ID of 218 (starts with 2, has 3 digits).

Looking through the CallML lists, you can see that they are a simple C++ switch statement, based on the ID of the CallML. You can learn a lot about the workings of the firmware by looking at CallML implementations and seeing what firmware functions they call. Look through the IC libraries that IC installed (they're in the lib/xbc subdirectory of wherever you have IC installed). Wherever a library function uses callml in its implementation, note the first argument (the CallML ID), and look it up in XBCRobot.cxx or ICRobot.cxx. You'll see the full C++ implementation, which is highly useful if you want to see how something works. Once you've taken a look, read Part 2 of this paper, which will walk you through adding some CallMLs, and coding for the XBC in C++.

We'll see you in Part 2!

8 References

[1] KIPR. Interactive C. http://www.botball.org/educational-resources/ic.php, March 2008.
[2] R. LeGrand et al. The XBC: a Modern Low-Cost Mobile Robot Controller. http://www.kipr.org/papers/xbc-iros05.pdf, July 2005.
[3] KISS Institute for Practical Robotics. http://www.kipr.org, 2008.
[4] Charmed Labs. http://www.charmedlabs.com, April 2008.
[5] Charmed Labs. The Xport 2.0 for the Game Boy Advance. http://www.charmedlabs.com/index.php?option=content&task=view&id=25, September 2003.
[6] Nintendo. Game Boy Advance. http://www.gameboyadvance.com, 2006.
[7] Wikipedia. Field-Programmable Gate Array. http://en.wikipedia.org/wiki/Field-programmable_gate_array, April 2008.
[8] Charmed Labs. Downloads – Xport – Software. http://www.charmedlabs.com/index.php?option=com_docman&task=cat_view&gid=30&Itemid=44, August 2004.
[9] Parallels. Desktop Virtualization. http://www.parallels.com/en/products/vm/, May 2008.
[10] TortoiseCVS. http://www.tortoisecvs.org/, April 2008.
[11] Charmed Labs Forum. http://www.charmedlabs.com/index.php?option=com_smf&Itemid=36, May 2008.
[12] KIPR. Botball Store: Cport Cable. https://botballstore.org/catalog/product_info.php?products_id=165, 2008.
[13] Quatech. EPP Parallel PCMCIA Cards by Quatech. http://www.quatech.com/catalog/parallel_pcmcia.php, 2008.