Hacking the XBC Firmware:
Programming the XBC in Standard C++
Part 2
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 2

1 Introduction

Welcome to Hacking the XBC Firmware, Part 2! Before proceeding, you should already have read Part 1, and followed all of its instructions. If not, you're likely to be confused as heck before the end of Part 2. Now that you've set up the firmware, learned how to upload it to your XBC, learned how CallMLs work, and possibly messed with a Cport cable, you're ready to start modifying the firmware source. We're going to give you some examples on how to hack the firmware to change the colors on the GBA screen, detect when a motor has stalled, run a C++ function of your design when an event happens, and run your entire program in C++ without using IC at all. Remember, this is just the tip of the iceberg when it comes to firmware hacking – we're still finding new things you can do with it. So, after the obligatory disclaimer, let's jump right in!

DISCLAIMER: Using an unofficial firmware with an XBC is officially unsupported, meaning that if it doesn't work, KIPR and Charmed Labs 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 Adding a CallML: Setting the Background Color

Let's add a couple of CallMLs to XBCRobot.cxx. The first will take 3 arguments, so it will be of the form 3xxx. Let's make it 3700. Have you ever wished you could change the colors of the text on the GBA screen? Well, if you look through libgba, in palette.h and palette.cxx, you'll see some functions that might help us. (Get used to looking through libgba and libxrc to find interesting functions -- Charmed Labs doesn't have any documentation for most of these, so you'll have to find stuff out yourself.) You should be eyeing the SetColor() and WriteToGBA() functions. Read those functions now if you haven't already; the comments on SetColor() are especially useful, as they tell us that the RGB range is 0-31. Looking through ICRobot.cxx (ICRobot constructor function), you can see that m_palette is the name of the palette object used by the firmware, and you can see the numbering of the two text colors (foreground = 253, background = 254) as well. Let's set the background color to whatever the IC program sends to the CallML. Putting this all together, you get the following added to the XBCRobot::CallML3Translator() function, at the end (but before the "default" label):

```
case 3700:
      m_palette->SetColor(254, argument1, argument2, argument3);
      m_palette->WriteToGBA();
      break;
```

argument1, argument2, and argument3 will be the RGB values of the background color we wish to use.

3 Adding a CallML: Reading the Motor Outputs

Next we're going to add an XBC function rather than a GBA function. If you're familiar with the Lego NXT controller, you may have tried to read the PWM output of a motor so that you could detect if it had stalled. This works because if the motor has additional load on it, it has to work harder to maintain the same velocity, so the PWM output increases. On the NXT, this is directly available to programmers. Well, we're going to do the same thing with the XBC. Check out the libxrc directory, and the file within it called axeso.h. Do you see a function there that might be useful? (Again, looking through libgba and libxrc to find interesting stuff can be very productive.) You should be eyeing GetPWM(). This CallML will only take 1 argument (the motor), but will return a value as well. Let's add a new CallML with ID 1700 to the XBCRobot::CallML1Translator() function:

```
case 1700:
      return(GetPWM(argument1));
```

After we've added these CallMLs, we'll need to run "make clean" on the firmware (so that it removes old versions), and then make and upload it as usual (with the "USE_BOOTLOADER=1" if you're using the serial uploader, without if you're using Cport). The “make clean” command deletes old versions of the binary files that the compiler generated. Always remember that whenever you change a file after you’ve already compiled the firmware, you need to run “make clean”. If you don’t, the compiler will see that the source files you changed have already been compiled, and it won’t recompile them. Similarly, if you change a file in libgba or libxrc, you’ll need to navigate (using the cd command) to that directory, run make clean, and then make it as normal.

4 Using the CallMLs From IC

Now we're going to write a simple IC program that uses these CallMLs. Read through the following code:

```
void main() {
      int getpwm;
      int color;

      mav(0, 25); // Tell the motor to move slowly.
      while(! b_button())
      {
            getpwm = callml(1700, 0); // GetPWM, -255 to 255
            color = (getpwm+255) / 16; // Scale it to 0 to 31
            callml(3700, color, 31-color, 0); // Set the background color so
that Red and Green are inverses of each other, while Blue is 0
```

```
            msleep(20L); // Wait a bit
      }
      mav(0, 0); // We're done, stop the motor.
}
```

Now load this IC program onto an XBC with your modified firmware. Plug a motor into port 0. Run the program. You should notice that the background color of the text changes to a greenish red and hovers around a certain value. Grab the motor, and try to stop it turning. The color should jump to bright red. Now push the motor in the other direction, so that it's going faster than it should. The color should immediately jump to green. Congratulations, you have just made your first useful program using a hacked firmware! You could use this code to detect when you've hit a wall, without using a touch sensor. How cool is that?

5 Entire Programs in C++

You don't have to only use C++ for simple CallMLs, you can also run the entire program in C++ in the firmware. The main advantage to this is that the firmware runs at a much higher speed, due to it being run natively and not interpreted. However that is not the only reason, and in your own trials you will realize how much a real programming language (with C99 features) makes your day easier and how much more efficient your programs will be.

First we will be creating a function at the end of XBCRobot.cxx, which will be our program. We could have used ICRobot.cxx if we were so inclined. Within this function you'll be able to do anything you could in IC (since the IC library isn't made for you, you'll have to recreate it) but with a much greater speed and with access to cooler features. Let's recreate the IC program we showed you in Section 4.

```
void XBCRobot::GetPWMTest() {
      int current_getpwm;
      int color;

      HappyBeep(); // ICRobot.cxx: send out a beep to say we're alive and in
the firmware

      IcMoveVelocity(0, 25, m_motionAccel[0]); // ICRobot.cxx: Tell the motor
to move slowly.

      while( CallML1Translator(113, 0) & GBA_KEY_B ) // ICRobot.cxx and
libxbc.ic: we're combining the IC function check_button, the CallML, and the
#define for the B button from libgba/gba.h.  So this is while(! b_button()).
      {
            current_getpwm = GetPWM(0); // GetPWM, -255 to 255
            color = (current_getpwm+255) / 16; // Scale it to 0 to 31
            CallML3Translator(3700, color, 31-color, 0); // Set the color so
that Red and Green are inverses of each other, while Blue is 0

            // C++ version of sleep (from HappyBeep()), times are in
microseconds
            CSimpTimer timer;
            long long unsigned startTime, currentTime;
            timer.GetCount(&startTime);
            timer.GetCount(&currentTime);
```

```
            while(currentTime - startTime < 20000L) // 20000 microseconds =
20 milliseconds
                  timer.GetCount(&currentTime);
      }
      IcMoveVelocity(0, 0, m_motionAccel[0]); // ICRobot.cxx: We're done,
stop the motor.
}
```

Remember that in C++, you need to prototype your functions. So add the following line to the XBCRobot class declaration in XBCRobot.h:

```
void GetPWMTest();
```

The whole thing looks unholy and messy, due to the fact that the C++ libraries are somewhat more complex than their IC counterparts, but it's nothing that a few nice additional wrapper functions or #defines couldn't fix. Functions you might want to recreate in C++ are msleep() and the GBA button functions (you could use the IC library versions as a guide).

6 IC Bootloader for C++ Programs

Okay, so we have a C++ function in XBCRobot.cxx, which composes our main program. How do we make it run? You could mess with the internals of the firmware, so that it runs your function on boot, but there's a simpler way (always remember the KISS Principle!). Just make a CallML in XBCRobot.cxx that calls your function, and make a simple "Bootloader" IC program which calls that CallML.

The CallML:

```
case 1800:
      GetPWMTest();
      break;
```

And the IC program:

```
void main() {
      callml(1800, 0);
}
```

That was easy, wasn't it? Just run that IC program on an XBC running a firmware that contains your GetPWMTest() function and CallML, and your C++ code will run instead of an IC program!

7 Interrupts

Now, we will do something that IC can't – interrupts. This, we believe, is among the best features hidden-yet-available to the Botball programmer. An interrupt, briefly, can be used to run a function of your design automatically when a certain event happens, without requiring any change in the code that was running at the time the event occurred. If you're navigating through a course but want to stop as soon as a touch sensor gets hit, an interrupt can, for example, be triggered by the touch sensor and cause the robot to stop its motors, then returning you to your

running code seamlessly. Unlike multitasking in IC, an interrupt allows you to wait for an event while using *no* processing time whatsoever. When the event is triggered, the program is 'interrupted' and the Interrupt function is executed immediately; you don't have to wait for a process to switch after 5ms like you would with IC processes. As an example, we're going to modify our above C++ program's CallML so that when a digital port changes state, the GBA beeps.

```
case 1800:
      // m_pIntCont is the Interrupt Controller of XBCRobot's ancestor class,
CAxesOpen (libxrc/axeso.h)
      // 21 is an interrupt "vector" which indicates the digital port
interrupt

      // Store a pointer to the existing Interrupt() routine; otherwise we'll
overwrite the existing digital port interrupt functionality
      m_orig_dig_int = m_pIntCont->m_vectors[21];

      // Set up Interrupt
      m_pIntCont->Register(this, 21); m_pIntCont->Unmask(21);
      *(m_gpio.m_intMask) = 0x01; // Interrupt for first digital port; see
libxrc/gpioint.h for details

      GetPWMTest();

      // Put Interrupt back the way we found it
      m_pIntCont->Register(m_orig_dig_int, 21);

      break;
```

And, we need an Interrupt() function; this goes at the bottom of XBCRobot.cxx

```
void XBCRobot::Interrupt(unsigned char vector) {
      if(vector == 21) { // Check if it's the digital port
            m_orig_dig_int->Interrupt(vector); // do what normally happens
with a digital interrupt
            HappyBeep();
      }
      else
            // XBCRobot is descended from CAxesClosed, which has its own
Interrupt; we don't want to interfere with it, so pass through to CAxesClosed
if the Interrupt vector isn't the digital port.
            CAxesClosed::Interrupt(vector);
}
```

And don't forget the prototype and variable declaration in XBCRobot.h:

```
IInterrupt * m_orig_dig_int;
virtual void Interrupt(unsigned char vector);
```

When you set up an interrupt, you are putting the address of the Interrupt() function in a place in memory so that when an interrupt is triggered and you've designated an Interrupt() function, it will jump to the point in memory and start executing. Encoders were implemented using interrupts, but their function simply increments their counter. Interrupts allow a non-linear

program based on events as opposed to the original parallel program system and can be quite useful if used properly.

8 Testing it Out

Okay, so we've rewritten our IC program in C++, and configured it using interrupts to beep when a digital port is activated. Compile, upload to your XBC, and upload your Bootloader IC program using the IC interface. Run the program. You should see roughly the same behavior as the IC version. However, try out a touch sensor plugged into the first digital port (port 8). Every time you toggle it, the GBA will beep! That particular interrupt was kind of useless, but you can probably think of some good uses.

9 Additional Notes on Interrupts

There's a bug in our code above – during the beep, our code stops running because the Interrupt() function hasn't returned. A notable property of interrupts is that they don't run in parallel or automatically return after 5ms like with IC processes. An Interrupt() function will keep control of the CPU until it returns. In fact, even other interrupts won't run while your Interrupt() function is active – this means that, for example, motor speeds will stop being dynamically controlled, and your sonar sensor won't return an accurate reading. This means that you should engineer your Interrupt() functions such that they don't take too long to execute, because your main program won't continue until the Interrupt() function returns. In fact, if your Interrupt() function has to *wait* for anything, it's probably poorly engineered. In general, rather than waiting for something else to occur, your Interrupt() function should set a variable and/or set up another interrupt to trigger later. Interrupts are available for just about everything that you might want to wait for (see Section 11), and by using them instead of blocking, your main code won't have to halt as well. For example, rather than sleeping for 500ms within an interrupt, you might set a GBA timer to generate another interrupt after 500ms (again, see Section 11).

10 Additional Notes on C++ VS IC

When should a program be written solely in C++, and when should it be written in IC with a few C++ CallMLs? The key advantages to pure C++ are speed (C++ is 85 times faster than IC in our informal benchmarks) and interrupts. If you have a program that, in IC, would be written without processes, it is almost always preferable to use C++ by itself, since C++ is native, while IC uses p-code, which makes C++ vastly faster. Similarly, interrupts are more efficient than processes, so if you would ordinarily use IC processes, but you can accomplish the same thing with interrupts, go for it – your code will run much faster. In the rare case that you need fully-fledged processes to run in parallel, rather than your code being interrupted periodically, IC is the way to go (just stick all the C++ functionality you need into CallMLs). However, there is a middle ground. If, for part of your program's execution, you don't need multiple processes, you can make that portion a C++ program, and access it as a CallML from the middle of your IC program. Once the C++ program finishes executing, your IC program will resume. Interrupts will also function while an IC program is executing, and you can easily add CallMLs to manipulate the interrupts. This kind of "hybrid" C++/IC programming can be very effective, as it takes advantage of the best features of both languages.

11 List of Available Interrupt Vectors

If you'd like to use interrupts, looking at the existing implementations can be very helpful. Here's a list of the available vectors, their functions, and (for the XBC vectors) their files. GBA vectors (0-12) are (we think) unused by the XBC firmware. (List of GBA vectors taken from libgba/gba.h.)

Vector	Function
0, 1, 2	GBA Video Hardware
3, 4, 5, 6	GBA Timers (useful for interrupting after a specified time period)
7	GBA Serial Port (NOT the XBC serial port)
8, 9, 10, 11	GBA DMA
12	GBA Keypad
16	XBC Serial Port / Bluetooth Receive (icfirmware/src/libicxportcommon/CBluetoothDevice.cxx, icfirmware/src/libicxbc/CUartDevice.cxx)
17, 18, 19	Vision Controller (libxrc/vision.cxx)
20	BackEMF/Analog Input Update (libxrc/axeso.h and libxrc/axesc.cxx)
21	Digital Inputs (icfirmware/src/libicxbc/CXBCGpio.cxx)

12 Conclusion

So, now we're finished with this guide. Where to now? We recommend reading through libxrc and libgba to get more ideas and learn more about how the XBC and GBA hardware works. You'll learn loads, and probably find some useful prewritten code you didn't know about, just waiting to be hooked into the IC firmware. You also might learn from GBA programming guides on the Internet (homebrew GBA development is very popular; just search Google). Keep in mind that very few, if any, of the GBA homebrewers on the Internet use the Charmed Labs libraries, so you'll probably need to modify whatever you find to work well with the Charmed Labs libgba. (Charmed Labs' libgba is *not* the same as the libgba used by most homebrewers). If you're looking for a technical reference on programming the GBA, check out GBATEK [1], which details all of the GBA hardware (including all of the GBA interrupt vectors listed above) and how to access it. It's pretty technical, so you probably wouldn't want to solely use it for programming something, but it's helpful if you're looking through libgba and can't figure out what something is doing.

We'd love to hear what you do with the firmware. If you do something awesome with it, or if you have simple questions, please don't hesitate to e-mail us. (Please note that we cannot provide full-fledged technical support for your modifications or for the firmware and/or libraries themselves.) Thanks for reading!

13 References

[1] M. Korth. GBATEK. http://nocash.emubase.de/gbatek.htm, 2007.