CBC Hacking 2010 (Part 2)
Jeremy Rand, Matthew Thompson, Braden McDorman
Norman Advanced Botball, Nease High School, Norman Advanced Botball
biolizard89@gmail.com, matthewbot@gmail.com, braden@betabot.org

CBC Hacking 2010 (Part 2)

15 Welcome to Part 2!

Welcome back to CBC Hacking 2010. In Part 2, we'll finish discussing our hacks, and tell you how you can make your own. Let's jump in! But first, the obligatory disclaimer:

DISCLAIMER: Installing unofficial firmware on your CBC carries an inherent risk of bricking your CBC. Usually, reflashing an official firmware will cure such a brick, but in rare cases this will not work. KIPR's warranty does not cover damage caused by an unofficial firmware, and we are unable to offer a warranty ourselves (but if something does happen, please do notify us so that we can attempt to help you fix it). If this concerns you, that should be a hint that CBC hacking is not for you.

16 Separating the Compile and Link Stages

KISS-C typically has a single `.c` file which contains the `main()` function, and various `.h` files which contain the code for libraries. When the program is compiled, the preprocessor causes the `.h` files' content to be loaded into the `.c` file., and the `.c` file is then passed to GCC. This means that all of the code in every library must be recompiled every time the `main()` function needs a recompile.

Professional C/C++ programs use a different method [10]. Instead of libraries being in `.h` files which are #included into the main `.c` file, each library is a `.c` file, which is only compiled when it is updated. The `.c` files, instead of being compiled into `.exe` or `.bin` files, are compiled into `.o` files (object files), and a linker merges the `.o` files into a `.exe` or `.bin` file. As a result, unmodified libraries do not need a recompile, even if the main program or other libraries change. The resulting speed increase quickly grows as the program gets larger and uses more libraries. Code::Blocks does this by default.

To use linking in Code::Blocks, make sure that each library consists of a `.c` file (containing the functions and variables) and a `.h` file (containing function prototypes and extern declarations for global variables). Add both the `.c` and `.h` files to your project, but only `#include` the `.h` file from your main program. Code::Blocks will take care of the rest, and you will see much faster build times.

17 Compile+Download Performance Tests

The combination of cross-compiling, rsync downloads, and separate compile and link stages produces some drastic speed increases, as the following comparison of compile+download times demonstrates:

	Simple C program (1 file, 34 lines)	Complex C program (15 files, 20,282 lines)
KISS-C 2.1.1 (Windows)	12.0 seconds	14 minutes, 8 seconds
Norman/Nease Wi-Fi 1.2 [11]	9.1 seconds	43.1 seconds
Code::Blocks with CodeSourcery G++ Lite Cross-Compiler and `rsync` Downloads	2.2 seconds	10.6 seconds

18 Compiler Optimization

Compiling with CodeSourcery G++ Lite also allows compiler optimizations [12] to be used. A simple but effective optimization which CodeSourcery G++ Lite is able to perform on CBC programs is constant folding. This optimization calculates as much of the mathematical expressions in your program as possible at compile time, before the program is run. This means the compiler will replace something like `2 * PI * WHEEL_RADIUS` with `37.699` (for wheels of radius 6) in the program binary, instead of generating code to perform that calculation on the CBC each time the program is run. Many other optimizations, often involving much more complex rewriting of the programmer's code, are also performed.

These optimization features are disabled in KISS-C, because the Chumby simply does not have the necessary CPU power to quickly optimize your code every time you compile (remember, despite our release of binary caching last year, the official CBC firmware does a complete recompile every time you switch programs, so optimizations would slow down the process of switching programs and become exceedingly annoying). However, a PC can easily perform these optimizations at compile time, and still compile far faster than the Chumby does. We have configured Code::Blocks to enable optimizations, so CBC programs created in Code::Blocks will not only compile and download faster, but will also run faster on the CBC. Although the average Botball programmer does not need the added execution speed (as demonstrated by the proliferation of CBCJVM [13] and CBCLua [14], which are slower than C), power users who need every CPU cycle they can get will likely benefit from compiler optimization.

19 C++ Programming

Code::Blocks and CodeSourcery G++ Lite support C++ as well as C. Simply click File → New → Empty File, and give it the file extension `.cpp` rather than `.c`. Code::Blocks will automatically compile with `g++` (the C++ compiler) rather than `gcc` (the C compiler). The entire GNU standard C++ library [15] is available to you, so object-oriented fans who want low-level hardware access no long have to choose between C's low-level access and Java's object-orientation.

Make certain that if you are linking C++ code with C code, your C code's `.h` files are C++-compatible. This usually means putting the following at the top of your `.h` file:

```
#ifdef __cplusplus /* If this is a C++ compiler, use C linkage */
extern "C" {
#endif
```

And this at the bottom of your .h file:

```
#ifdef __cplusplus /* If this is a C++ compiler, use C linkage */
}
#endif
```

Wikipedia has an explanation of why this is necessary [16].

20 Shared Libraries

When KISS-C and the official CBC firmware compile and download your program, all of the program's libraries are merged into the final binary file. This can be annoying, because if you identify a bug in a library and fix it, you need to track down, re-download, and re-compile every program that uses the library. It also increases the binary size of your programs, which eats unnecessary storage and slows your downloads.

We have configured Code::Blocks and the NHS Patchset CBC Firmware to allow shared libraries (also called dynamically linked libraries) [17]. This is the system used by most modern operating systems. The library binaries are stored in different files on the CBC from the main program binaries. When you run the main program binary, the library routines in the library binary are dynamically linked into your main program. As a result, you can update your library once, and the enhancements or bug fixes will automatically propagate to all your programs on the CBC, after just one download. This also speeds up downloads, since only a library or a main program file must be downloaded, instead of both.

To create a shared library in Code::Blocks, follow these steps:

1. Create a new CBC project, but base it on `libCBCLibrary` instead of `CBCProject`. Make sure that the name of your project starts with `lib`.
2. Write your library in the new project.
3. When you're done creating the library, build the project.

4. Assuming you got no errors, the library and its include headers will be copied to the appropriate directories on both your PC and CBC.
5. Now, open a main program project which you want to utilize to the library.
6. Right-click on the program project on the left-hand side and choose Build Options.
7. For each target on the left-hand list, click on Linker Settings → Add, and type the name of the library WITHOUT the `lib` at the beginning. For example, for the example library, you would type `CBCLibrary`. (We're trying to find a good way to keep you from having to do this 4 times.)
8. Click OK.
9. Add code to the program project which utilizes the library.
10. Now, when you build the program, it will automatically link with the shared library at run-time. Building the library will update it for all programs on the connected CBC which utilize that library.

Note that shared libraries probably don't work properly in the simulator. We're planning to fix this, but it's not a high priority.

21 KISS-Sim from Code::Blocks

To compile a program in Code::Blocks for KISS-Sim rather than targeting the CBC, simply click SimulatorDebug (debuggable without optimizations) or SimulatorRelease (optimized, no debugging) in the Target list. The Build button will compile and link the program, and the Run button will launch KISS-Sim with your program. This target switching works much the same way that KISS-C does. Debug and Release use the CodeSourcery G++ Lite compiler with the CBC `include` and `lib` directories (copied from the CBC firmware with minor modifications) and automatically `#include` the `autoinclude_cbc.h` file (which loads the same header files that the official CBC firmware's compile script uses. SimulatorDebug and SimulatorRelease, in contrast, use the PC GNU compiler with the KISS-Sim `include` and `lib` directories (copied from the KISS-C install folder) and automatically `#include` the `autoinclude_sim.h` file (which loads the same header files that KISS-C uses).

22 VNC Remote Desktop Enhancements

Last year, we explained how to use VNC [18] to enable remote desktop access on the CBC, so that the CBC could be operated without using the touch-screen. Unfortunately, the code we released then required the Chumby screen to be disabled while the VNC user was connected, and the Chumby needed to be rebooted to switch back to the Chumby screen. As a result, VNC was less usable than we would have liked, and we do not know of any other Botball teams who used VNC over the past year (there was certainly no discussion of the feature on the Botball Community). This year, in addition to recompiling the VNC plugin for the Chumby 1.7.1 firmware, we have modified the VNC code to permit simultaneous touch-screen and remote access. Simply open Code::Blocks and click Tools → Remote Desktop Via VNC. A VNC session will open on your computer, and the CBC will remain uninterrupted.

23 Fixing GDB Debugging

Last year, we explained how GDB [19] can bring back the interaction features that IC had but KISS-C didn't have. Unfortunately, the CBCv2 broke GDB debugging; KIPR apparently forgot to recompile GDB for Chumby firmware 1.7.1, so the GDB binary refuses to run. As such, we had to either recompile GDB or find a new method for debugging. We weren't up to the task of compiling GDB, but luckily we found another solution: `gdbserver`. `gdbserver` is included in the Chumby firmware, and allows the CodeSourcery PC-side GDB binary to debug programs on the Chumby via Wi-Fi. The setup is somewhat less user-friendly than standard GDB, because the user program's output and the GDB output are in different windows. To fix this, we used some bash script trickery to merge both output streams into one window. The result feels roughly the same to the user as our CBCv1 GDB method, except that the user program's standard input stream cannot be accessed from the GDB window.

To debug a program (to achieve IC-like functionality), switch to the Debug (not Release) target which enabled debugging support and disables optimizations. Make sure that the Debug target has been built and downloaded to the CBC, then click Tools → Interact/Debug Via GDBServer. Once you are presented with a GDB prompt, use the following controls:

- `cont`
  - Resumes your program (it starts out paused; you do not need to use the run command).
- `Ctrl+C`
  - Pauses your program and displays a GDB prompt.
- `call fd(0)`
  - Executes the command `fd(0)` the way IC would; you can use any command (including variable names or math expressions). This only works when your program is paused.
- `quit`
  - Ends your program and your interaction/debugging session.

For more details, see the GDB manual which came with CodeSourcery G++ Lite.

For cases in which you need to enter text into the user program's standard input stream, we have included an alternate method which keeps the user program and GDB in separate windows. Click Tools → Start GDBServer, and wait for `Listening on port 5000`. Then click Tools → Connect to GDBServer, and wait for a GDB prompt. Type GDB commands and the Ctrl+C into the second window, and use the first to access the program's standard input and output streams. This is more annoying than the single-window method described above; only use this method when you need access to the user program's standard input stream.

24 Setting Up a Firmware Development Environment

Firmware development makes extensive use of the GNU toolchain, Qt Embedded [20], and other Unix tools like `make` and `wget`, and as such is easiest to set up on a Linux environment (Cygwin [21] is not supported). If you already use Linux, you're in luck, but if you're a Windows user the

easiest way we've found to get started is simply to get Linux in a virtual machine such as VMWare Player [22].

Once you've got a Linux install up, you'll want to follow the directions on the ChumbyWiki, `http://wiki.chumby.com/mediawiki/index.php/GNU_Toolchain`, to set up the GNU toolchain. This is the native Linux equivalent to the CodeSourcery toolchain introduced above as part of Code::Blocks.
KIPR uses a version control system known as Git [23] to manage their source code. You'll need to install Git through your Linux distribution's package manager. If you use the ever-popular Ubuntu or any other Debian based distribution, simply run `sudo apt-get install git` in a terminal session, and Git will be automatically downloaded and installed.

The last thing needed in a firmware development environment is an installation of Qt Embedded [20], the framework with which the CBC user interface is written. We've created a simple build script that automates the download and installation of Qt Embedded which can be obtained through Git. Just run the following commands to use it:

```
1. git clone git://github.com/matthewbot/CBCBuildScripts.git
2. cd CBCBuildScripts
3. sudo make
```

This script unfortunately takes a while to run. Expect it to take 30 minutes on a decently fast computer, and possibly several hours on a slower computer or laptop. When this command completes, you're all set to build the source code.

25 Downloading and Building the Source Code

The CBC and CBOB firmware source code are hosted on GitHub, a free Git repository hosting service. KIPR's official repository can be found at `http://github.com/kipr/cbc`, while the NHS Patchset repository can be viewed at `http://github.com/matthewbot/cbc`. To build the NHS Patchset, just use these commands: (simply change the clone URL to refer to the KIPR repository if you'd like to build the official code).

1. `git clone git://github.com/matthewbot/cbc.git`
2. `cd cbc`
3. `make`

This command could also take some time to run, although generally it will take less time than the CBCBuildScripts. When it finishes, you'll find your own custom made `userhook0` at `filesystem/upgrade/userhook0`. Unfortunately, making changes to various parts of the firmware is beyond the length and scope of this paper, but if you've gotten this far, you won't find it difficult. Just stop by the Botballer's Chat and ask someone where to begin to make your desired change. If you begin to make a lot of changes, you can even set up your own GitHub repository to publish them, allowing them to easily be merged into the main NHS Patchset!

26 Conclusion

We hope you enjoyed reading about the CBC hacks from the past year. Now it's your turn! Build a hack, and post about it on the Botball Community [24]. And please give us feedback on our hacks as well -- most of our ideas come from teammates' or other Botball friends' comments. We'd also like to thank our one-person beta-testing team, Marty Rand from Norman Advanced Botball, for putting up with hours of repeated failed tests. (His complaints are why our hacks are as stable as they are.) We can be found at the Botball Forums and the Botballer's Chat at the Botball Community.

Happy Hacking!

References

[10] tepples. Frequently Asked Questions: C and C++: How do I put multiple files in a program? http://forum.gbadev.org/viewtopic.php?p=6772#6772, January 2008.
[11] J. Rand, M. Thompson, B. McDorman. Norman/Nease CBC Mod Installer v1.2. http://community.botball.org/forum/technical/programming/normannease-cbc-mod-installer-v12-supports-cbcv2, February 2010.
[12] Wikipedia contributors. Compiler optimization. http://en.wikipedia.org/wiki/Compiler_optimization, May 2010.
[13] B. McDorman, B. Woodruff, A. Joshi, J. Frias. CBCJVM - Applications of the Java Virtual Machine with Robotics. Proceedings of the 2010 Global Conference on Educational Robotics, July 2010.
[14] M. Thompson. CBCLua: Bringing Lua Scripting to Competitive Robotics. Proceedings of the 2009 Global Conference on Educational Robotics, July 2009.
[15] Wikipedia contributors. C++ Standard Library. http://en.wikipedia.org/wiki/C%2B%2B_Standard_Library, April 2010.
[16] Wikipedia contributors. Compatibility of C and C++: Linking C and C++ code. http://en.wikipedia.org/wiki/Compatibility_of_C_and_C%2B%2B#Linking_C_and_C.2B.2B_code, January 2010.
[17] Wikipedia contributors. Dynamic linking. http://en.wikipedia.org/wiki/Library_%28computer_science%29#Dynamic_linking, June 2010.
[18] Wikipedia contributors. Virtual Network Computing. http://en.wikipedia.org/wiki/Virtual_Network_Computing, June 2010.
[19] Wikipedia contributors. GNU Debugger. http://en.wikipedia.org/wiki/GNU_Debugger, June 2010.
[20] Nokia Corporation. Embedded Linux - Qt. http://qt.nokia.com/products/platform/qt-for-embedded-linux/, 2010.
[21] Red Hat, Inc. Cygwin. http://www.cygwin.com/, April 2010.
[22] VMWare, Inc. VMWare Player. http://www.vmware.com/products/player/, May 2010.
[23] S. Chacon. Git - Fast Version Control System. http://git-scm.com/, April 2010.
[24] Botball Youth Advisory Council. Botball Community. http://community.botball.org, May 2010.