AR.Drone Libraries for the CBC Botball Controller (Part 1)
Jeremy Rand, Marty Rand, Kevin Cotrone, Ali Hajimirza, Josh Maddux
Team SNARC (Sooners / Norman Advanced Robotics Coalition)
jeremy.rand@ou.edu, marsedge95@gmail.com, kevincotrone@gmail.com, a92hm@ou.edu, jdmaddux@gmail.com

AR.Drone Libraries for the CBC Botball Controller (Part 1)

1 Introduction

The Parrot AR.Drone is a quadcopter aimed at augmented reality gaming. It uses WiFi connectivity to be remotely controlled (typically from an iOS or Android device using tilt control), and claims to use military-grade flight stabilization (including accelerometers, gyrometers, and camera tracking). At the 2011 Global Conference on Educational Robotics, KIPR announced that the CBC Botball Controller's ability to use an external WiFi card (as had been used since 2009 by hackers) would be used to control an AR.Drone autonomously from a CBC.

The AR.Drone's addition to the lineup of Botball-related hardware added a new dimension (literally) in the 2011 KIPR Autonomous Aerial Vehicle (AAV) Contest. Unfortunately, integrating the Drone with the CBC (which teams were heavily encouraged to use) was no easy task. Out of the approximately 10 teams which originally registered, approximately 5 dropped out of the AAV Contest before the event, leaving only 5 teams left. The remaining teams utilized a variety of tricks to complete the tasks. Southwest Covenant School and SODA Zack entirely used dead-reckoning based on timing. St. John's used an awesome mechanical grappling hook to land on a moving target. SODA Chloe used a quite clever trick involving the Drone's flight stabilizer, making it follow a ground-based robot. All of these methods were reasonably effective, and were able to complete the first and second challenges with a good success rate.

However, all of these approaches were unable to complete the final challenge, involving navigating around an obstacle and landing on a colored target. This challenge required the ability to receive data back from the Drone, including XYZ position/velocity and color tracking data. And that's where we came in. Team SNARC was the only team who succeeded at this challenge. In this paper, we will explain how we won the AAV Contest and wrote AR.Drone Libraries for the CBC Botball Controller.

But first, the obligatory disclaimer:

DISCLAIMER: This paper describes certain techniques which we used for loading libraries onto the CBC (e.g. SSH) which could result in a brick if done improperly. We have never had a brick as a result of this, but we cannot guarantee that the process will go well for you. Damage caused by modifying the CBC's internal filesystem is not covered by KIPR's warranty, and don't expect us to fix your CBC for you if something happens. (But feel free to stop by the Botball Community [1] if something bad happens; someone there might be able to help you.)

2 Existing Libraries

Good programmers are lazy. They don't like writing things from scratch. In this spirit, we spent a few days Googling before we wrote any code ourselves. We found that there were a decent number of existing AR.Drone libraries which we could try to adapt.

The official AR.Drone SDK [2] appeared to be very complex given our timeframe (we had roughly 5-6 weeks from receiving the Drone to tournament day). In addition, the AR.Drone SDK was under a restrictive license; we were hoping for something under GPL, LGPL, MPL, BSD, or similarly permissive licensing.

We found a Java-based library, JavaDrone [3], which was under the BSD license. JavaDrone looked nice, except that we didn't want to use Java for a few reasons. Java on the CBC is somewhat less stable than C/C++ is, and none of us had any experience with Java on the CBC (which meant that the 5-6 week timeline wouldn't work in our favor). Also, we were hoping to implement some kind of vision, and the only vision code with which we had experience required C++. OpenCV [4] worked with Java, but we deemed it overkill for simple color tracking, and again, we had no experience with it, meaning that we couldn't be expected to get it working in 5-6 weeks. As a result of these problems with using Java-based libraries, we continued our search.

Finally, we found a C++ library by Pongsak Suvanpong [5]. The library implemented most features which JavaDrone implemented (we are under the impression that Pongsak based the library on JavaDrone), and looked relatively simple in terms of prerequisites, build process, and learning curve (it was a single .cpp file, with a couple of .h files). The license wasn't clearly stated by Pongsak, so we checked with him, and he confirmed that it was under the BSD license, the same as JavaDrone. We were, however, nervous that we couldn't find anyone other than Pongsak who had reported using his library. (Of course, that's par for the course in Botball.)

After Google failed to provide any additional options, we decided that Pongsak's C++ library was our best bet.

3 Build Environment

Technically, the beta versions of the CBC firmware at this time supported C++. However, we had doubts as to the stability of this beta feature, so we went with a tried-and-true method: cross-compiling. Information about cross-compiling for the CBC can be found in *CBC Hacking 2010*

by Jeremy Rand, Matthew Thompson, and Braden McDorman [6].

4 Building libcommonc++

Pongsak's library required libcommonc++ [7], a library which appears to be designed to facilitate porting Java programs to C++ in a cross-platform manner. libcommonc++ is a quite complex library, but it turned out to be reasonably easy to build for the CBC, with a few notes.

We had to build it on Linux due to the use of various Linux tools in its build process. It is plausible that building it on Cygwin would have been doable, but we did not investigate it (doing so would have been more trouble than it was worth). The instructions in *CBC Hacking 2010* [6] for using Linux to build the CBC firmware are also sufficient for setting up a build environment for libcommonc++.

The following terminal commands will build libcommonc++ for the CBC:

```
./configure arm-linux --host=arm-linux --target=arm-linux --build=i386-linux
make
```

If you're following along with us, you may do this step and then wonder why there's no compiled library file produced. For some reason which we don't understand (but which Linux experts may understand), the `.a` and `.so` files are placed in a hidden directory: `libcommonc++-0.6.5/lib/.libs` . Inspect this directory by typing its name manually into your folder viewer or `ls` command to verify that the `libcommonc++.so` file is present.

Assuming all looks good, navigate to the directory (using cd) where libcommonc++-0.6.5 is located, and type the following into a terminal:

```
ssh root@192.168.1.1
```

Where `192.168.1.1` is the IP address of your CBC. You will be shown the Chumby ASCII logo, and logged into your CBC. Type the following lines in the CBC's SSH prompt:

```
mkdir /mnt/user/code/TestDrone
mkdir /mnt/user/code/TestDrone/libARDrone
sync
exit
```

You will be returned to your Linux terminal. Now type (all on one line):

```
scp ./libcommonc++-0.6.5/lib/.libs/* root@192.168.1.1:/mnt/user/code/TestDrone/
libARDrone/
```

Wait a few minutes while libcommonc++ is loaded onto your CBC.

SSH back into your CBC, and type:

```
sync
```

Syncing is very important. Not syncing after loading libcommonc++ onto your CBC has a good chance of corrupting your filesystem, which is only fixable by reflashing the CBC's firmware.

After syncing, you can type the following to be returned to your own computer's terminal:

```
exit
```

At this point, libcommonc++ is installed on your CBC.

5 Building Pongsak's Library

Once you've built libcommonc++, you can copy the `.libs` directory back to your Windows computer. You can then create a new CBC project in Code::Blocks called TestDrone in a new folder (instructions are in *CBC Hacking 2010*).

Once you've created the new project, create a new folder in it called `ARDrone`. Copy the `libcommonc++-0.6.5` directory as a subdirectory of `ARDrone`. Then move the include folder in `libcommonc++-0.6.5` into the `ARDrone` folder.

Now go to Build Options in Code::Blocks. Under "Search Directories → Compiler", add the following directory: "`ARDrone\include`". Under "Search Directories → Linker", add the following directory: "`ARDrone\libcommonc++-0.6.5\lib\.libs`". Under "Linker Settings", add the following library: "`ARDrone\libcommonc++-0.6.5\lib\.libs\libcommonc++.so.10.0.0`". Do all of this for both the Debug and Release targets. (The simulator targets discussed in *CBC Hacking 2010* should not be used; they are not compatible with the Drone libraries at this time.)

You will also need to enable Exceptions. Chumby Industries recommends against this, but Pongsak's libraries require them, so in Code::Blocks, go to "Settings → Compiler and Debugger", choose the GNU ARM-Linux GCC Compiler, click "Compiler Settings", and replace "`-fno-rtti -fno-exceptions`" with "`-fno-rtti`". Be sure to change the setting back if you're developing non-Drone programs in C++ with Code::Blocks; we're not sure why Chumby discourages Exceptions, but they probably know something we don't.

At this point, your TestDrone project is ready to add Pongsak's library. Copy ARDrone.cpp and ARDrone.h to the ARDrone folder of your project. You'll also need to download the MemoryLibrary.h file, and put it in a new folder in the root of your project called Common. (Pongsak had forgotten to post this file until we asked where to find it; that explained why no one else had reported success with the library.)

Make a new file to hold your main program called TestDrone.cpp, and put this code in the main() function:

```
printf("Creating Drone...\n");
```

```
myDrone = new Drone();
printf("Drone created\n");
myDrone->start();
printf("Drone started\n");
```

Build the project, and you should have a TestDrone.bin file. Next step: loading this onto the CBC!

6 Booting TestDrone.bin

CBC Hacking 2010 provides an easy way of loading .bin files which access shared libraries such as libcommonc++. Unfortunately, this requires the NHS Patchset firmware, which isn't compatible with recent changes by KIPR. So, we have to resort to a less pretty hack.

Make a file called "bootloader.c", and put it in the same folder as TestDrone.bin. Fill it with the following:

```
int main()
{
    chdir("/mnt/user/code/TestDrone/");
    system("./RunDrone.sh");
}
```

Then make a file called "RunDrone.sh" in the same folder, with the following contents:

```
#!/bin/sh

export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/mnt/user/code/TestDrone/libARDrone
./TestDrone.bin
```

Open up a Cygwin prompt, navigate to the directory where these files and TestDrone.bin are, and type the following:

```
scp ./* root@192.168.1.1:/mnt/user/code/TestDrone/
```

After doing this, make sure to sync the filesystem as described above; you don't want a corrupted filesystem.

What we've done is create a shell script which sets up the shared library and runs the .bin file, and create a bootloader C program which executes the shell script when run on the CBC. Next time you modify the TestDrone.bin program, you can simply rerun the `scp` and `sync` commands to load it onto your CBC.

You can now go to your CBC, and compile bootloader.c. Connect to the Drone's WiFi network, and run the bootloader.c program. Your program will connect to the Drone and run! Of course, all we did was connect and then end the program. Time to add something useful.

7 Firmware Compatibility

The first thing you should do now is make sure that your Drone is running the latest firmware. We had disconnect and timeout issues with the Drone's default firmware; updating to the latest version fixed these issues. Refer to the Parrot website [8] for instructions on this. Once you've done this, read on.

8 Moving and XYZ Tracking

Your code should call the following function periodically:

```
myDrone->controller().sendWatchDogReset();
```

We call it every 50ms; we're not certain exactly what frequency is required, but this seems like it works. The watchdog is a safety feature which prevents your Drone from continuing to fly if the connection is lost.

Once you've implemented this, you can move the Drone by calling the following functions:

```
myDrone->controller().takeOff();
myDrone->controller().land();
myDrone->controller().sendEmergencyShutdown();
myDrone->controller().hover();
myDrone->controller().sendControlParameters(int enable, float pitch, float roll,
float yaw, float gaz);
```

These should be fairly self-explanatory. The `enable` parameter of the last function should be either 0 or 1. 0 forces the Drone to hover in place; 1 allows the other parameters to control how the Drone moves. `pitch` and `roll` tilt the Drone (resulting in horizontal movement), `yaw` turns the Drone horizontally, and `gaz` moves the Drone vertically. The range for the floats is [-1.0,1.0]. Be very careful; usually only very small numbers are necessary. For our first test, we tilted the Drone's pitch at 0.2 for 5 seconds, figuring that would be safe for us to observe. Instead, within about 3-4 seconds, the Drone blasted all the way along the long dimension of the Norman North High School cafeteria into a wall with a loud crash. (The Drone was completely unharmed by the collision; the thing is insanely well-built.) During the AAV Contest, we never had to go above 0.05 pitch or roll.

To find out how far the Drone has traveled, use the following code:

```
NavigationData latest_data;
myDrone->navigationDataReceiver().copyDataTo(latest_data);
```

`latest_data` now contains a `NavigationData` struct with various data, as defined in `ARDrone.h` (look through that struct before continuing here).

The struct has velocity fields, but no position fields (other than altitude). To find XY position, integration is necessary. For readers unfamiliar with calculus, we're essentially following this formula each time we receive a velocity update, where Δt is the time interval between velocity updates.

$$\Delta x = vx \bullet \Delta t$$
$$x \mathrel{+}= \Delta x$$

The Drone's firmware does significant smoothing of the velocity data using accelerometers, gyrometers, and the downward camera, which makes this integration surprisingly accurate. Instructing the Drone to fly 40ft resulted in less than 2ft of error, and the error was consistent (everyone likes consistent error, because fudge factors [9] can fix them).

We weren't satisfied with controlling the Drone via its tilt; we were hoping for something more like the `move_relative_position` command on the CBC. We quickly threw together a proportional position controller, which would apply less tilt as the Drone approached the target position, but it was completely unstable due to the Drone's high rate of acceleration. We instead made a proportional velocity controller which attempted to maintain a constant velocity until the target position was reached. This worked and was mostly stable, but had a tendency to drift over time. Despite the drift, it served us well at the AAV Contest. Unfortunately, we won't be releasing that velocity controller code because it's extremely ugly and finicky, and in most cases doesn't converge anywhere near the target velocity. We're not even sure if it's better than a constant tilt. Readers are probably better off making something themselves.

We imagine that a PID position controller using a trapezoid-trajectory generator would be much better than our approaches. (Or, in laymen's terms, implemented the same as the `move_relative_position` command on the XBC.). Unfortunately, the XBC motor controller's license (MPL) is not compatible with most CBC software, and we haven't investigated developing our own trapezoid-trajectory generator. (This would be an interesting project.)

9 References

[1] Botball Youth Advisory Council. Botball Community. http://community.botball.org , June 2012.
[2] Parrot S.A. AR.Drone Open API Platform. https://projects.ardrone.org/ , July 2011.
[3] Codeminders. Javadrone. https://code.google.com/p/javadrone/ , May 2012.
[4] OpenCV. http://opencv.willowgarage.com/wiki/ , May 2012.
[5] Pongsak Suvanpong. ARDrone control in C++ cross platform no SDK needed. https://projects.ardrone.org/boards/1/topics/show/3397 , September 2011.
[6] Jeremy Rand, Matt Thompson, Braden McDorman. CBC Hacking 2010 (Parts 1 and 2). Proceedings of the 2010 Global Conference on Educational Robotics. July 2010.
[7] Mark Lindner. commonc++. http://www.hyperrealm.com/main.php?s=commoncpp , Retrieved June 2012.
[8] Parrot S.A. Firmware Update Procedure. http://ardrone.parrot.com/parrot-ar-drone/usa/support/update , Retrieved June 2012.
[9] Wikipedia Contributors. Fudge factor. Wikipedia, The Free Encyclopedia, https://

en.wikipedia.org/wiki/Fudge_factor . January 2012.

10 See You in Part 2!

That's all we could fit into Part 1. See you in Part 2!