CBC Hacking 2011: Vision Enhancements and Sensor Speedups (Part 1)
Jeremy Rand
Norman Advanced Robotics
jeremy.rand@ou.edu

# CBC Hacking 2011: Vision Enhancements and Sensor Speedups (Part 1)

## 1 Introduction

Since the CBC Botball Controller was introduced to Botball in January 2009, hackers have tinkered with many of its components. But other components were less lucky. Some unhacked components, e.g. the CBOB, were ignored because of the bricking risk. But there is one component which has received virtually no modification even though it carries minimal risk of bricking the CBC: the vision system. Matthew Thompson did discover a framerate enhancement in 2009 [1], but no other vision hacking was attempted. Finally, that is changing. This paper covers various vision hacks which can make the CBC vision system more flexible, more useful, or faster. In addition, while the CBOB firmware itself is not getting hacked anytime soon, its Chumby driver has had some useful hacks developed which heavily speed up sensor access. 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.

## 2 Installing the Hacks

To install the hacks discussed in this paper, download the latest userhook0 from the NHS Patchset [2]. Install it on your CBC as you would an official update. The NHS Patchset userhook0 will preserve the USER partition, so your programs, camera models, and mods will still be intact. (This feature was copied from the NHS Patchset by KIPR early this year.)

## 3 Multiple Cameras

When the CBC firmware boots, `cbcui` sets up a file pointer to `/dev/video0` (the first camera to be plugged in). The vision system then reads images from this pointer. Any additional cameras to be plugged in are assigned to `/dev/video1`, `/dev/video2`, etc., and are not touched by the default CBC firmware.

By changing where this file pointer points to, other video devices can be accessed. The first challenge was to allow the user program to interact with the vision system so that a program can request a camera change. This was solved with a Linux feature called a FIFO. A FIFO ("first-in, first-out") appears to programs to be a file in the file system, but is actually a queue of data handled by the operating system which allows different programs to communicate in real-time. The vision system sets up a FIFO in `/tmp/switch_cam`, and attempts to read a byte from it every frame. When the user program wants to switch to the 2nd camera (starting with 1, not 0), for example, it writes the byte 2 to `/tmp/switch_cam`. The next time the vision system checks the FIFO, it discovers that a byte is available to be read, which yields the byte 2, and the vision system then knows to switch to the 2nd camera.

Initially, switching cameras was implemented by closing the current camera device and opening the requested one. This worked, but initializing a camera device causes a delay of about 1-2 seconds, during which the camera is inoperable. A faster solution was found, in which an array of file pointers was created, and the switch camera function simply copied the requested file pointer into the main file pointer used by the vision system. This means that once a camera is opened, it stays open, even if the vision system switches to a different camera later. As a result, switching cameras takes effect by the next frame. Up to 255 cameras can be multiplexed in this manner, making the CBC's USB and power hardware the effective limiting factor. (There's a possibility that leaving multiple cameras open can eat battery life faster, but so long as the CBC is charged between game rounds, this shouldn't be an issue.)

A benefit of switching cameras rather than tracking multiple cameras simultaneously is that since only one camera is processed per frame, the tracking speed is independent of the number of connected cameras. Very few users require multiple cameras per frame, and the lag introduced by the resulting increased CPU load would probably cripple the CBC's ability to do anything else.

Example code to switch cameras:

```
unsigned char cam_id = 2; // Open 2nd camera; default is 1
int g_switch_cam = open("/tmp/switch_cam", O_WRONLY); // Open the FIFO
write(g_switch_cam, &cam_id, 1); // Write 1 byte to the FIFO
close(g_switch_cam); // Close the FIFO
```

### 3.1 Driver Settings for Additional Cameras

The framerate hack and associated driver settings which Matthew Thompson discovered in 2009 [1] will also work with cameras beyond the first. Just replace `video0` with the appropriate device number (e.g. `video1`). These settings will allow customizing the framerate, exposure, white balance, brightness, contrast, and sharpness from within a KISS-C program.

## 4 Excluding the Rainbow

Both the XBC and CBC vision systems were designed to track colors which fall on the rainbow, meaning that they do not track grayscale colors by default. This limitation is implemented by blocking the user from setting the maximum saturation and maximum value to anything other than 255. However, the XBC offered an undocumented library function which permitted those two parameters to be set to arbitrary values, effectively allowing tracking of grayscale colors. In the CBC, there is no such library function, so some hacking was necessary. A FIFO was setup to receive requests to set the color models' HSV parameters (including max saturation and max value), and this FIFO is checked once per frame in the color tracker. The hue parameters can also be set to arbitrary values, and do not have to be close together like the vision GUI requires (which is important for tracking grayscale, since grayscale colors can have any hue).

Some example code:

```
unsigned char model_op = 5; // Operation ID
unsigned char model_out = 2; // Output color model
unsigned char model_in_A = 1; // Input parameter A
unsigned char model_in_B = 300 >> 8; // Input parameter B
unsigned char model_in_C = 300 & 0xFF; // Input parameter C

int g_model_cmd = open("/tmp/model_cmd", O_WRONLY); // Open the FIFO
write(g_model_cmd, &model_op, 1); // Write 1 byte to the FIFO
write(g_model_cmd, &model_out, 1); // Write 1 byte to the FIFO
write(g_model_cmd, &model_in_A, 1); // Write 1 byte to the FIFO
write(g_model_cmd, &model_in_B, 1); // Write 1 byte to the FIFO
write(g_model_cmd, &model_in_C, 1); // Write 1 byte to the FIFO
close(g_model_cmd); // Close the FIFO
```

Available operation ID's are:

- 5: Read model model_in_A, change minimum hue to ( (model_in_B-1) << 8) | model_in_C, write the result to model model_out.
- 6: Read model model_in_A, change maximum hue to ( (model_in_B-1) << 8) | model_in_C, write the result to model model_out.
- 7: Read model model_in_A, change minimum saturation to model_in_B, write the result to model model_out.
- 8: Read model model_in_A, change maximum saturation to model_in_B, write the result to model model_out.
- 9: Read model model_in_A, change minimum value to model_in_B, write the result to model model_out.
- 10: Read model model_in_A, change maximum value to model_in_B, write the result to model model_out.

The above example code reads model 1, changes the minimum hue to 300, and saves the result to model 2. Note that in this system, models range from 1-4, not 0-3. If a command does not use model_in_B or model_in_C, leave those values at 1 (do not use 0).

## 5 Boolean Transformations

Simple HSV ranges, even with the ability to exclude the rainbow, do not always offer sufficient control to detect some desirable color models. A new FIFO-based command was setup to perform Boolean operations on color models. This is easy, because of the way color models are implemented. In the CBC firmware, each color model is stored as an array of bits, one for each possible color (1 indicates that the color fits the model; 0 indicates that it does not). Since all of these calculations are done when setting the color models initially, this lookup-table-based method allows faster lookups than trying to do calculations on the fly. It also makes it possible to do Boolean operations on those arrays of bits to combine or invert color models as we wish.

The Boolean operations AND, OR, and NOT are all available to be applied to color models, as well as a COPY command. How is this useful? Let's say you want to track a stack of pink and green poms. Make a color model for pink and a color model for green, then OR them to get a color model that will track the entire stack as one blob. Or maybe you want to track arbitrarily colored objects, but you know the background will be the white game board. Make a color model for white, then NOT it to get a color model for everything else. You can make any combination of AND, OR, and NOT, so you could, for example, implement "NOT (white OR black OR pink OR green OR orange)" to find anything on the game board that isn't a board marking or game piece (i.e. probably a team-placed structure). By De Morgan's Law [3], you could also implement the same Boolean logic by using "NOT white AND NOT black AND NOT pink AND NOT green AND NOT orange."

The same FIFO (`/tmp/model_cmd`) and code are used for these commands as for the functions to exclude the rainbow, but with different operation ID’s. These ID’s are:

- 1 (Copy): model model_out = model model_in_A.
- 2 (Not): model model_out = NOT model model_in_A.
- 3 (And): model model_out = model model_in_A AND model model_in_B.
- 4 (Or): model model_out = model model_in_A OR model model_in_B.

Please note that changes to color models made through these FIFO methods will not survive a reboot, and they will also be reset every time you enter the Vision screen on the CBC GUI. If you want to use the Vision screen with modified color models, run your program which changes the models via SSH [5] while already on the Vision screen.

## 6 Clipping the Image

It is common in Botball to only want a subset of the environment to be visible to the camera, e.g. to avoid the camera seeing a bright green shirt on someone in the audience. KIPR’s standard suggestion is to angle your camera down. This is nonoptimal, since the degree to which you want to angle your camera may vary, which would require adding a motor and lots of unnecessary mechanical complexity. As a more flexible alternative, I added additional FIFO operation ID’s which clip the camera image to a user-specified minimum and maximum X and Y coordinates on the image:

- 11 (MinX): Minimum X value = model_in_A.
- 12 (MaxX): Maximum X value = model_in_A.
- 13 (MinY): Minimum Y value = model_in_A.
- 14 (MaxY): Maximum Y value = model_in_A.

These commands are only reset when the CBC is rebooted. Due to some quirks of the CBC firmware’s blob assembler, these commands also cause the Vision screen’s display of matched pixels to glitch (the blob bounding box rectangles do display correctly). I haven’t measured the resulting effect on the camera’s framerate, but this hack does reduce the necessary processing, so it’s plausible that framerate could be helped by this hack (although probably not by much).

## 7 End of Part 1

That’s it for Part 1.  Part 2 will continue where we leave off here, covering additional hacks which can speed up camera vision and sensor access.  See you there!