*Independent Study*

# Exploration: Parallax Propeller Platform

**Kenneth Beck**

*Advisor:* Zack Butler

Department of Computer Science
B. Thomas Golisano College of Computing and Information Sciences
Rochester Institute of Technology
Rochester, New York

August 15, 2011

## Abstract

The goal of this independent study was to determine the feasibility of using a Parallax Propeller processor for conducting various operations related to controlling the Pioneer and iRobot Create robots. The following paper outlines the results of this independent study, showing what I've accomplished and what I feel the Propeller platform is capable of.

# 1 Introduction

## 1.1 Chip Specifications

The Parallax Propeller processor runs on a 32-bit architecture and contains eight "cogs" (Parallax's name for each of the cores). Additionally, the chip has 32 I/O pins that are accessed by the eight cogs in a round-robin fashion, allowing a theoretical I/O speed of approximately 10MHz. Each cog also has 2KB of RAM (512 longs) and the processor as a whole possesses 32KB of RAM and 32KB of ROM.

## 1.2 Device Iterations

### 1.2.1 Ybox2

The Ybox2[1] was the device that got me initially interested in conducting this independent study. Having just taken an embedded systems course, I wanted to explore some more embedded systems, and this was the perfect tool. The user was required to build the device by hand, which took a lot of time, patience, and soldering. Upon completion, the user had in his hand a do-it-yourself PC complete with a piezo speaker, an ethernet port, a video out port, an infrared receiver and a multi-colored LED. Onboard, it was running the Parallax Propeller chip.

Using the Ybox2, I familiarized myself with both the recommended IDE ("Propeller Tool") and the languages used for developing on the Propeller platform (Spin and the low-level Propeller Assembly). It was at this point that I realized I wanted to work with this Propeller chip for my independent study. At first, I wanted to develop an operating system for it, as I had previous experience in this area from Systems Programming I and II. However, I didn't have a firm grasp on what I would want to put into this operating system, so I took a different route. I decided I wanted to put my Artificial Intelligence knowledge to use and program the Propeller chip to control robots.

It was at this point that I realized the limitations of the Ybox2. Its only means of power was an AC adapter, which proves challenging when you want the device to run on top of a robot; since it had a significant number of unnecessary add-ons, the number of free cogs and I/O pins was reduced; and the layout of the device itself did not allow for easy expandability. In particular, I knew I would have to add on an external serial port (for connecting to the robot), and given the layout of the device, there was no place to attach it. Thus, I decided to purchase another Propeller product, one that was designed for prototyping and came pre-pre-assembled.

### 1.2.2 Propeller Platform USB

Gadget Gangster's Propeller Platform USB[2] offers much more flexibility than the Ybox2. It can be powered by either AC or DC, provides both 3.3V and 5V rails, comes with an SD card reader, and is designed with expandability in mind. This platform is what I worked with throughout my independent study

[1] http://www.ladyada.net/make/ybox2/

[2] http://gadgetgangster.com/find-a-project/56.html?projectnum=257

# 2 First Steps

Before I began development, I needed to get a DB9 serial port, with on-board level shifting (which translates between the serial standard's +/-12V inputs and outputs to the Propeller chip's 0-3.3V values). It took two iterations of such serial ports before I got a working version. This serial port is attached to the top of the case that came with the Propeller Platform, on a small breadboard.

After getting a working serial port, I needed to determine if the Propeller was capable of communicating with a robot. I found a library someone developed for the iRobot Create written for the Propeller platform. I decided to work with this library to try and control the Create. After some initial failure, I realized I needed a null-modem adapter to connect my platform to the Create. When this was done, I was able to communicate with the Create and perform some basic operations (playing some songs, initiating a demo, driving around). With the knowledge that my serial port was working as it should be, I decided to move on.

# 3 Evolution of Goals

My goals for this independent study changes a fair amount from my original plans. I originally wanted to work with the CMUcam to grant the Pioneer robot I worked with the ability to follow blobs of color (for instance, a red ball rolling down the hall). After working with the CMUcam for several days, I gave up as I kept running into difficulty (packets were arriving corrupt and the entire communication protocol was a little messy).

Additionally, I wanted to grant the Pioneer wall-following capabilities as I had done previously in an Artificial Intelligence course. I put together some code to do so, but was unsatisfied with the performance. Instead of spending more time on this task, I decided to focus on using the robot's sonar to perform some basic mapping.

It would not be possible to test out the mapping logic without having the robot be capable of driving around. To this end, I came across this remote-controlled helicopter at a store a few weeks into the quarter and realized that this helicopter had an infrared remote. I bought the helicopter, and purchased an infrared receiver, and reverse engineered the protocol the remote used. After tying in the remote to the Pioneer driver that I wrote earlier, I was able to drive the robot around.

# 4 Breakdown of Components

## 4.1 p2os_packet.spin

The first major component that I had to develop for the platform was the driver for the Pioneer robot. The p2os_packet object provides the means of sending and receiving packets to and from the Pioneer robot. When sending messages, you first build the packet using the BuildPacket* methods and then call the SendPacket routine to send the packet to the Pioneer.

## 4.2 p2os.spin

This object contains two instances of the p2os_packet object, one for transmitting and one for receiving. Receiving packets is performed in its own cog, in the ReceiveLoop method. The packets received are parsed depending on their type and certain values are stored in local variables that are made available to other objects utilizing this object.

Packet transmission is abstracted away so higher levels do not need to know the specifics of how to structure the p2os packets. Currently, the driver is capable of sending Drive and Turn commands to the robot, enabling and disabling the sonar on the robot, requesting a configuration packet (which tells the robot to send additional configuration information that the normal packet does not contain, like the current and max translational and rotational acceleration values), and sending the Kill command which stops the Pioneer robot as quickly as possible.

Before any command can be sent to the robot, it is necessary to send a series of synchronization packets. This is handled when the p2os object is initialized, through the Sync routine. When synchronization is complete, the motors are enabled on the robot and it is ready to receive commands.

## 4.3 Propel_IR_Remote.spin

IR communication with the RC remote that I found is conducted in this object. The getPulse method performs pulse-width modulation on the infrared receiver to break it down into individual bits. Calling getCode returns a 24-bit representation of the stream sent from the remote. This 3-byte value is then parsed to determine the values sent from the remote. The horizontal position of the right joystick on the remote assumes values between 0 and 215, with 0 being the right-most position and the initial position (in the center) being 115.

The vertical position is described by values between 0 and 7, with 7 being the position all the way down. The left joystick's initial position is all the way down and it is only capable of moving up and down. The highest value this joystick can assume is 63.

A button at the top-left of the remote is used to control a light on the helicopter. When depressed, it turns on a bit in the stream sent. When released, this bit is turned off. I use this value to toggle the mapping capabilities of the robot on and off, which also enables / disables the robot's sonar.

## 4.4 bitmap_writer.spin

In order to perform map-drawing, I needed an output mechanism for the map. While my device had video out capabilities, this requires a significant amount of on-board RAM to maintain, which I had trouble keeping available given the amount of code the rest of the project required. Thus, I resorted to exporting the map to a bitmap file using the SD card reader that came with the device. As I could not find any existing Spin libraries for writing bitmaps, I developed my own and made use of another library that provided an engine for reading and writing from/to the SD card (SD-MMC_FATEngine.spin).

The bitmap writer is broken into two major sections: the CreateBitmap method and the WritePixel method. Creating the bitmap generates a bitmap file on disk with the specified file name, width

and height. Depending on the width and height, creating the bitmap can take a fair amount of time. With some of my testing, writing an 800x600 pixel image to disk sometimes took up to a minute. The bitmap follows the BITMAPINFOHEADER header format and currently supports a grayscale set of colors. By default, the bitmap is written as a 4-bit bitmap, though with the mapping code, only three colors are used. (black, white and gray). By changing some constants at the top of the code, it's possible to adjust the number of colors in the color table, and the code should adapt appropriately (as long as standard color depths are used).

The file system library that I came across provides the ability to seek to specific locations within the file. I used this to my advantage when it came time to write individual pixels discovered by the mapper to the bitmap. Based on the information used for the construction of the bitmap, I determine the offset within the file for the specific pixel being written and write out the new color value. For color depths of less than 8 bits per pixel, this has an additional burden of reading in the current value so that any existing color information for an adjacent pixel is not overwritten.

## 4.5 mapper.spin

The mapper object originally had more responsibility than just forwarding events to the bitmap_writer object. My original plan was to provide a circular queue for pixels being written because I thought that writing out to disk would prove to be a bottleneck. The mapper object would run a write loop that waits for items to be added to the queue and calls the bitmap_writer's WritePixel method whenever something new comes up. I ran into some problems with this approach, though, as I was initializing the bitmap object on one cog and calling methods from it on another. Apparently this is undefined behavior, and the results I achieved were not what I expected. Thus, I set up the mapper class to simply forward calls to the bitmap. It also provides some higher-level logic that allows the main program to determine whether or not a file is currently open (used when toggling the mapping functionality on and off).

## 4.6 bootloader.spin

This object provides the entry point into the program. It starts off by initializing all required objects: a Math library which provides sine and cosine functionality for transforming sonar co-ordinates into world space; the robot driver; the mapping functionality; and the controller. When this is all complete, it enters an infinite loop where it reads the controller configuration and sets the driving and turning speed dpeending on the values it reads. If the controller has not been updated for a second or more, then the robot is stopped (a safety precaution) until communication with the controller is regained.

All eight cogs are used for controlling the robot: 1 cog for the math library, since most routines are run through Propeller Assembly, which must be run on its own cog; 3 cogs for the robot driver, 1 each for the send and receive p2os_packet objects (as the FullDuplexSerial object that is used is started on its own cog) and another for the loop that parses incoming packets; 1 for the mapping functionality; 1 for reading from the controller and 1 for the main program.

# 5 Conclusions

Starting this project off at the beginning of the summer, I did not know what I would be capable of achieving. The biggest hurdle that I thought I would run into was the extremely small amount of on-board EEPROM that I had to work with. It turns out that this was not as big a deal as I expected. In the end, my code only takes up about 18KB out of the available 32KB. Within these 18KB, I've developed a program capable of: parsing data packets sent to it from a Pioneer robot through a serial interface; sending data packets to this Pioneer robot to control it; parsing infrared signals from a remote control; creating and editing a bitmap image on a micro-SD card; and performing basic mapping logic using the eight sonars present on the Pioneer robot.

In all areas but one, the program works very well. The one area that didn't turn out as I wished it to was the mapping. Finding walls using the distance readings acquired from the Pioneer sonars requires first locating the coordinates of the potential wall relative to the robot, then transforming these coordinates to world-space, so they're relative to the robot's position in the world and then further transforming the coordinates to the (x,y) position on the bitmap. I had sample code written in C from my AI class that I tried porting over, but I had to make some sacrifices to accuracy because I was worried the mapping would prove too slow.

These sacrifices have apparently thrown off the transformations so much that the results are nowhere near ideal. The three images included alongside the source code (test-map-1.bmp, test-map-2.bmp and test-map-3.bmp) show the results of a few mapping runs. Images 1 and 3 take place around the Computer Science floor while the second image was drawn from within the AI lab. The gray pixels represent the path the Pioneer robot thought it followed (through dead reckoning), while the black pixels are my transformation values for the sonar readings. As can be seen, these are far from accurate. However, the images do show the capabilities of the bitmap library that I wrote.

Overall, I believe this independent study was a success. While I didn't achieve all the goals I originally set out for, I did develop a system for controlling a Pioneer robot using only the Propeller platform. Given more time, I would like to fix the mapping code to make it as efficient as possible and as accurate as sonar readings can be. It would also be nice to provide autonomous wall-following or pathfinding, which I believe the Propeller platform is perfectly capable of.