Implementation of theV2BOT Control Drivers on the LPC1769

by Tony Li, Hien Nguyen, Will Zegers, and Yao Zhao
Computer Engineering Department, College of Engineering
San Jose State University, San Jose, CA 95112
will.zegers@gmail.com

***Abstract*—*V2BOT is an on-going project at San Jose State University that combines visual recognition software and processing algorithms with robotics, with the goal of designing a system that can autonomously identify objects, such as fruit, and mechanically manipulate the object it has identified. Underlying the robotic and mechanical functioning of V2BOT is the LPC1769, a 120MHz microcontroller unit that drives the movement of the robotic arm and platform. This paper discusses the design and implementation of the PWM and GPIO drivers that control the movement of the arm and base platform, respectively. Written to identify what has already been accomplished with the design of V2BOT robotic systems, this paper is also intended as future guidance in using the system as the project continues to develop.*

I. INTRODUCTION

V2BOT is a robotic system under development at San Jose State University that uses the Samsung S3C6410 ARM11, running on the FriendlyARM Tiny 6410, to process images fed to it through a USB camera and determine the movement of the V2BOT robotic assembly – namely, the movement of the arm and base. While the ARM11 handles much of the heavier processing needed for image recognition, algorithm processing, and statistical computation, actual movement of the arm and base is driven by the LPC1769. This lighter-weight processor, by NXP, offers ports for UART, PWM, GPIO, and other communication protocols to drive peripheral devices from a microcontroller unit (MCU). Moreover, development of the LPC1769 is done using he Eclipse-based XPresso development IDE, a feature-rich development interface that allows for code compilation, device debugging, and more.

The LPC1769 Xpresso board has several available I/O ports to communicate to external processors and devices, and this paper focuses on the uses of three of these – universal asynchronous transmitter receiver (UART), pulse-width modulation (PWM), and general purpose I/O (GPIO) – that drive the movement of V2BOT. The details of how these communications are designed and how the work internally will not be discussed here, but the reference at the end of this paper features some external references to learn more on these topics. Furthermore, these communication tasks are handled and scheduled with FreeRTOS, a real-time operating system for embedded devices, which runs between the LPC hardware and the communication software.

The remainder of this paper is organized as follows: section II discusses the overall architecture of the system, and examines some of the design choices in running the control software in FreeRTOS. Section III looks more in-depth at the individual drivers for each peripheral system, and discusses in the software itself and how each one is driving their respective components. Section IV discusses a sample program that uses a simple command-line interface written in python to manually control the system, or move it to pre-defined positions. In section V, we briefly introduces future goals and developments of the project to increase the functionality of the project. Finally, section VI concludes the paper and once again, gives a broad view of the project as a whole.

II. SYSTEM ARCHITECTURE

The control system centers around the LPC1769 and includes three subsystems: the microcontroller unit, the robotic arm (including the servos), and the movable base platform (including its drivers). Figure 1 show a diagram of how this system is organized in terms of its overall architecture, highlighting the relation and connections between the LPC and the other two systems.

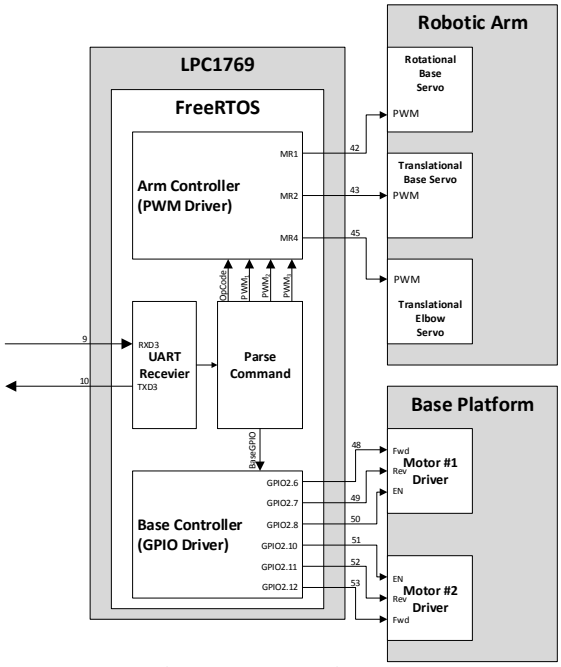

Fig 1. System architecture of the control system for the robotic arm and base platform.

The MCU accepts commands via UART, passes these commands into a parsing functioning that extracts the specific control parameters for the servos and base platform, and then passes these parameters to the appropriate drivers to drive movement of the arm and base platform. The remainder of this section discusses the architectural details of each subsystem.

A. LPC1769 and FreeRTOS

The user-facing side of the control system is the UART receiver, which accepts a specially formatted command string and proceeds to parse the string to drive movement of the V2BOT system. All tasks, including the receiver, parser, and drivers, run on top of the FreeRTOS real-time operating system in order to handle task scheduling and communication (i.e., passing the control parameters).

The UART receiver accepts strings from device that can transmit serial data, such as either a command terminal running on a client machine (mainly for debugging purpose) or an external processor (namely, the ARM11 for this project). Once the receiver recognizes a properly terminated command string, it passes a pointer to this string into a parser function that extracts the bytes corresponding to the op-code, three server positions (as PWM pulses), and a command to the base platform. These byte are then put into three queues: an op-code queue, a pointer to an array containing the three PWM pulses, and a queue with movement directions for the platform. Once the parser has sent to the three queues, the function returns and the UART receiver is ready for the next command.

B. Arm Controller

The driver for the arm controller blocks until command parser sends data to the op-code and PWM queues. Once the controller receives data from the queues, it processes these bytes to drive movement of the arm.

Current implementation has three possible op-codes: two which control the type of positioning and one which simply returns the current position of a desired servo based on its value in the corresponding match register. Positioning of the arm is specified by either issuing an absolute or relative position for the servos; an absolute position will move the servos to a position corresponding to the position specified by the byte received for that servo, whereas a relative position will move the servo to a relative position based on its current position.

Section III discusses how these positions are computed by the driver; however, to give a high-level example of these instructions work suppose the drivers receives an "absolute" op-code and a servo position corresponding to 1337 µs. In this case, the servo rotates however much it may need and in the proper direction until it is in the 1337 µs position. Alternatively, if the drivers receives a "relative" command and the servo position corresponding to 337 µs, then the driver will driver will set the corresponding match register so that the servo rotates in the *clockwise* direction by 337 µs. Were the parameter -337µs, this would cause the servo to rotate in the *counter-clockwise* position.

Positioning is controlled by setting the match registers for the for each PWM output on the LPC, and then setting the latch enable register to latch these value into the register and send the values as PWM pulses over the PWM lines to their respective servos.

C. Base Platform Control

Similar to the arm controller, the base controller receives the commands to drive via a queue loaded by the command parser. In this case, the byte loaded into indicates a direction of movement for the platform (forward, reverse, left, right, stop, or continue). The command is a single byte that is mapped to a specific output of the six GPIO pins that drive that platform. For example, if the platform driver receives the byte 0x46 (ASCII 'F' for forward), the GPIO outputs will cause the motor drivers to drive both motors in the "forward" direction; the byte 0x4C (ASCII 'L,' or left) will drive one of the motors forward and the other in reverse causing the platform itself to turn in the leftward direction.

Two of the three GPIO outputs correspond to motor direction, where setting the output to "1" of one or the other line will cause the motor to move in the corresponding direction. The third pin is the enable line that essentially acts as the on-off switch for each motor. Note that only one or zero direction pins should be asserted, but not both.

III. EMBEDDED SOFTWARE DESIGN

We used the LPC XPresso IDE, based on Eclipse, to develop the drivers and software running on the LPC1769. Using a pre-made FreeRTOS+IO demo project as a base, our team developed the device drivers on top of FreeRTOS to take advantage of operating system functionality such as task scheduling and the ability to communicate between running tasks. The overall OS design of the system assigned a task to each subsystem, while using queues to handle communication and data forwarding between tasks. The following sections describe in greater details the inner-working of the driver tasks and design choices made in developing.

A. main.c

By default, the file main.c is the first file accessed by the LPC microcontroller to begin running a user program. The body of this file handles calling the functions that will start all necessary tasks – initializing stack sizes, priority, and task handle – and then adds these tasks to the scheduler. Once all calls to xTaskCreate are called for each driver, the scheduler is started and this begins concurrent execution of each task.

The first function called by main.c is the NVICSetPriorityGrouping function which simply sets the priority grouping field using the required unlock sequence. This particular function is, in fact, the only code from the original FreeRTOS+IO demo project that included in our custom code, so as to ensure underlying interrupt and priority issues were not changed and altered the operation of the underlying operating system.

Following this, the next three calls are to the functions that start the UART receiver, the arm controller (PWM driver), and the platform controller (GPIO driver), respectively. As was previously mentioned, each of these functions is simply a call to xTaskCreate for each driver function and assigns all necessary operating system parameters and structures. Additionally, the function to create the UART receiver task also initializes the three queues that serve as communication structures between the UART receiver and the other two device drivers.

Finally, main.c makes a call to vTaskStartScheduler which starts all tasks added to the scheduler and allows the scheduler to manage their execution.

B. UART Recevier

Once the task scheduler has start the UART receiver task, appropriately named prvUARTReceiver, the lines 66-91 initialize the microcontroller to use UART3. Line 70 opens the UART device for reading and writing, and asserts in the following line that this operation was successful. Lines 75 and 83 configure the transmitter and receiver lines of the UART device; the first call assign the Tx line to use zero copy when sending bytes, meaning the UART device use interrupts to transmit characters directly from the buffer to the FreeRTOS write function, instead of using polling. We designed the UART to parse the input character by character as the bytes come in, so line 83 specifies that Rx should use a single character queue instead of polling.

Finally, the last call in UART initialization on line 90 raises the interrupt priority level to allow for greater real-time processing of received commands.

The rest of the UART receiver consists of an infinite while loop, waiting on a character to appear on the Rx line. The current implementation of the structure of user-issued commands are terminated with a newline (0x20) character, and the first thing the receiver does is check whether the byte is a newline character or not on line 104. For a non-newline character, the execution enter the *else* section in lines 119-121 in which the receiver simply adds this new character to the next index of the input buffer, cInputString, and increments to the next index to await the next received character.

Upon receiving the terminating character, the program executes lines 113-115 which begins by passing the input buffer to the parse function. Returning from the parseCommand function, the receiver then clears and resets the input buffer to prepare it for the next command.

C. Command parsing

Figure 2 shows a diagram of command formatting. After sending received bytes to its input buffer terminated with a newline character, the UART receiver makes a call to prvParseCommand which handles the job of extracting bytes from the input buffer and sending them to the appropriate queues. The first byte extracted is the op-code for the movement of the servos. This byte can be either 0x61 (‘a’ for “absolute”), 0x72 (‘r’ for “relative”) or 0x70 (‘p’ for “position”) and control how the 4 parameters will move the robotic arm and end-effector. This single byte gets placed into the xOpCodeQueue, but note that the op-code plays no role in determining the operation of the platform.

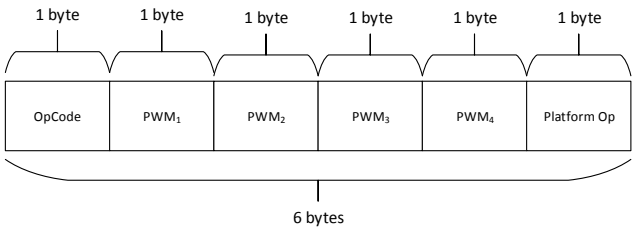

Fig 2. Format of the input command

The next four bytes extracted are *signed* bytes that correspond to positioning the servos and end effector. Details of how these bytes are processed are discussed in the next section. These four bytes get stored in an array, sPWM, and a pointer to the base of this array is sent to the xPWMQueue.

The final byte read is a movement command for the platform, which have six possible values that determine the movement of the treads on the platform. These bytes are 0x00, 0x46, 0x4C, 0x52, 0x53, and 0x56, correspond to pneumonic ASCII values and are discussed in section E. This byte is sent to the xPlatformGPIOQueue.

D. PWM Initialization

Lines 258-306 contain the code to initialize the PWM pins on the LPC 1769 and prepare the send micropulses to control the movement of the arm. Lines 265, 268, and 271 power the PWM port, resets all PWM counters, and clears the interrupts respectively. Line 274 specifies a pulse resolution of 1 μs, which will be used when assign match register values and when determining the duty cycle. The code then sets pins 42, 43, 44, and 45, corresponding to PWM ports 2.0, 2.1, 2.2, and 2.3, to operate as PWM with pull-up resistors disable (lines 277-281). At lines 284 and 285, match registers (MR) 0 and 5 are assigned values of 20000 and 19000 to serve as the timer period and interrupt time, respectively, which is set on line 299.

At this point, initialization executes lines 290-292 which loads the reset values for each servo into MR1, MR2, MR3, and MR4, which are then latched by the latch enable register (LER) on line 296. Enabling the outputs of PWM 2.0, 2.1, 2.2, and 2.3 on line 302 sends the micropulses stored in the match registers to causes the servos to move the arm into its default, reset position shown in figure 3.

Need pic of V2 in reset position
Fig 3. V2BOT in its default, reset position

Initialization finally ends by enabling the timer and prescale register that will now detect changes in the values of the match registers once the counter reaches interrupt time period, set to 1000μs before the end of the cycle.

E. Arm Controller

Once the arm controller task has returned from the initialization function InitPWM, it enters into an infinite loop at line 175. At this point, the task will block until it receives data from both the xOpCodeQueue and xPWMQueue, which are populated by the previously discussed prvParseCommand function. Once data from the queues are passed to variable cCommand (for the op-code) and sParameters (the PWM positions), the task checks the op-code to see it’s either an ‘a’ (byte 0x61), ‘r’ (byte 0x72), or ‘p’ (byte 0x59). Any other op-codes received will be ignored and the queues flushed.

The positional parameters will be converted to PWM positions in one of two ways, depending on the op-code. Since the parameters are signed integers of size 8 bits, the range for these parameters is between -128 and 127, meaning there are this many respective negative and positive positions to which the servos can be moved.

For a relative movement (op-code 0x72), the task will multiply each parameters by 8 and then add this value to the corresponding match register. This creates movement range between -1024 and 1016, with 8μs steps, that the servo can be moved from its current position. Boundary checking in lines 218 – 228 ensure that this movement cannot go under the 1000μs or over the 2000μs pulse limits specified in the Hitec HB-785 datasheet, so a relative movement under or above these limits will stop the servo at the lower and upper limits, respectively.

For an absolute position (op-code x61), each parameters is multiplied by 8, as in the relative movement calculation, and then added to 1000 (the base limit of the servos positions) to calculate the servos new position. While the absolute positioning could, in theory, give a range between -24 and 2016, boundary checking in lines 218 and 228 once again ensures that the servos do no rotate based their lower and upper limits.

The limits for each servo were established experimentally and based on the requirements of the arm. The base is free to rotate between the full 1000 and 2000μs pulses, while the other two servos are more tightly constrained to prevent damage to the physical arm itself, and to eliminate any

unnecessary range of motion. For example, the arm should not need to reach backward since it will be moving based purely what is in front of it.

Once the match register values have been set, the task sets the bits in the LER to latch the values and output the pulse to the PWM servos, moving them to their new position.

The positional command (0x59) is included in the software design in the event that it may be needed for future functionality of the project. This returns the value to the user, via UART, of the current position of the desired servo. To specify which servo, all parameters should be null (0x00) except for the servo for which the position is desired. For example, the command 0x5900ff0000000a will return value in MR2, corresponding to the translational base servo. Presently, this command does not have an application in the current implementation of V2BOT and is included only for future extensibility.

Finally, regardless of the command, the two queues are cleared and prepared for the next command to come in.

F. Platform Controller

Lines 324 and 325 constitute the entirety of the GPIO initialization used to drive the platform on which the arm travels. GPIOs 2.6, 2.7, 2.8, 2.10, 2.11, and 2.12 (pins 48 through 53 on the LPC Xpresso board) are set as GPIO outputs, and will generate the signals that select the direction and operation of the platform's motors. GPIOs 2.6 and 2.10, when asserted high along with the enable signals at GPIOs 2.8 and 2.12, will cause the platform to move forward by driving the left and right motors, respectively. Conversely, GPIOs 2.7 and 2.11 will drive the motors in the opposite direction that, when asserted with the enable signals, will drive the platform in reverse. Note that only one, but not both, of the directional signals should be asserted with the enable at a given time.

The sixth byte in the parsed command string corresponds to the op-code used determine the operation of the motors and, hence, the movement of the platform. Lines 331 through 364 assert or de-assert GPIO signals based on the op-code, assigned using mnemonic characters, such as 'F' (byte 0x46) for forward. An unrecognized op-code (the result of line 368 being true) is interpreted as an error, and the enable signals to the motors are de-asserted as a fail-sage mechanism. Table 1 gives the states of the GPIOs and their resulting direction, along with the corresponding.

Op-code	FWD_L	REV_L	EN_L	FWD_L	REV_L	EN_L	Movement
0x00	FWD_L	REV_L	EN_L	FWD_L	REV_L	EN_L	Continue
0x46	1	0	1	1	0	1	Forward
0x4C	0	1	1	1	0	1	Left
0x52	1	0	1	0	1	1	Right
0x53	X	X	0	X	X	0	Stop
0x56	0	1	1	0	1	1	Reverse

Table 1. Op-code bytes, their resulting GPIO outputs, and the direction the drive the platform to move

An important point to note for implementation is that the GPIO signals to the motors remain until they are explicitly changed by a new op-code. So between commands, a platform will continue to move in its current direction. Moreover, to issue a command without changing the operation of the platform – for example, if the user simply wanted to change the position of the arm slightly while keeping the platform moving in the forward direction – sending a null byte for the op-code will cause the current GPIO signals to remain as they are.

The final command of the platform control loop is to flush its queue and begin blocking once again in xQueueReceive.

IV. IMPLEMENTATION TESTING

Appendix A features a small program capable of controlling the system via a UART connection to test functionality, and demonstrate the basic capabilities of V2BOT. After being run, this simple Python script polls for user-entered keystrokes and sends appropriate commands to the system over UART. Due to its dependence on the PySerial library for Python, the program can only be run on UNIX and Windows 32-bit systems; however, extending the portability of the testing script could easily be accomplished by using a language, such as C, with native serial communications libraries. For quick and dirty demonstration purposes, we used Python to design a quick testing script.

The user executes the script by typing `sudo python V2ControlCLI.py` at the command line. The program uses the PySerial library to open a serial communication over a specified purpose (in this case, /dev/ttyUSB0 on a Linux Ubuntu system) and, therefore, must be run with root or administrative privileges to access the serial port. V2ControlCLI opens with an exception handler, returning a "No serial ports found" message if either the connection does not exist or if the program is not run with sufficient privileges.

Keystroke recognition is done using an instantiation of the _Getch class, which simply returns the bytes associated with a particular keystroke. For example, typing an "A" returns an `A`, while hitting the up arrow key returns the three bytes `^[[A` (where `^[` is the escape byte, 0x1b).

The bulk of the script is handled by the manualMove function, which creates an instantiation of the _Getch class to get keystroke as the users presses keys. Using multiple *if-elif-else* to respond to these keystrokes, in some cases resorting to nested *if-elif-else* structures to detect multi-byte keystrokes, the function writes a command string to the serial output line which is received by the V2BOT system. Figure 4 shows the keyboard layout for the movement of V2BOT, with key descriptions in table 2.

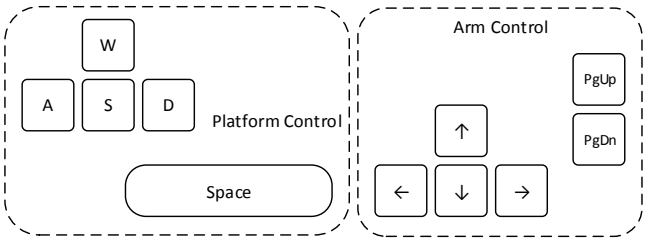

Fig 4. Keyboard keys used in manually controlling V2BOT

Key	Movement	Command String
Platform controls		
A	Platform forward	720000004c0a
S	Platform reverse	72000000560a
D	Platform right	72000000520a
W	Platform left	72000000460a
Space	Platform stop	72000000530a
Arm controls		
Up arrow	Arm up	72000004000a
Down arrow	Arm Down	720000fb000a
Right arrow	Arm rotate right	72fb0000000a
Left arrow	Arm rotate left	72040000000a
PgUp	Arm pitch forward	72000400000a

PgDn	Arm pitch back	7200fc00000a
Preset positions		
R	Reset	613f2b06530a
M	Maximum reach	613f513f000a

Table 2. Key mappings and their resulting commands string, sent to the V2BOT system via UART.

V2ControlCLI will continue to run until the user presses the 'q' key, which will terminate the program.

V. FUTURE WORK

A. End-effector

In its current implementation, no controls exist for the movement of the end-effector of V2BOT, since the actual component itself has not yet been integrated into the system. Since it is likely to use another servo driven by PWM, adding the embedded code and testing should be simply a matter of establishing operating limits of the end-effector itself and adding an additional parameter to the command string. (In fact, this parameter is already prepared but is currently commented out in the source code.

B. Remote debugging

To connect to the LPC and issue command via UART, a wired serial connection is necessary to connect the client machine to the LPC microcontroller. While it will ultimately fully autonomous and should not need user input to be driven since, instead, it will be relying on visual recognition and control algorithms, an unwired connection would be desirable for testing and debugging. This can be accomplished using a simple wireless module, such as the XBee by Digi to connect the UART communications wirelessly.

VI. CONCLUSION

Using the LPC1769 microcontroller, our development team has successfully built functional drivers for the components of the V2BOT system. A user or external hardware is able to issue commands serial communications, which are parsed on the LPC side and used to drive the PWM servos and GPIO platform of the system.

REFERENCES

[1] Hitec, "General Specification of HS-785HB Winch Servo" HS-785HB datasheet, Feb. 20, 2003.

[2] NXP, *LPC176x/5x User Manual*, NXP Semiconductors, Rev. 3.1, Apr. 2, 2014.

[3] NXP, *LPCXpresso 1769 Schematic*, NXP Semicondutors, Feb. 11, 2009.

[4] *FreeRTOS.org*, [Online]. Available: http://www.freertos.org.

[5] "Coding Standard and Style Guide", *FreeRTOS.org*, [Online]. Available: http://www.freertos.org/FreeRTOS-Coding-Standard-and-Style-Guide.html

[6] "Servo PWM on the LPC17xx," *OpenLPC*, [Online]. Available: http://openlpc.com/4e26f1/examples/pwm.lpc17xx

APPENDIX

A. Testing Python script

Appendix A

```
import sys, serial, termios, tty, time

SERIAL_PORT = "/dev/ttyUSB0"
BAUD = 115200

try:
   from serial.tools.list_ports import comports
   control = serial.Serial(SERIAL_PORT, BAUD)
except:
   comports = None
   control = None

# Thanks to Newb for the Getch code - http://stackoverflow.com/questions/22397289/finding-the-values-of-the-
arrow-keys-in-python-why-are-they-triples
class _Getch:
   def __call__(self):
      fd = sys.stdin.fileno()
      old_settings = termios.tcgetattr(fd)
      try:
         tty.setraw(sys.stdin.fileno())
         ch = sys.stdin.read(1)
      finally:
         termios.tcsetattr(fd, termios.TCSADRAIN, old_settings)
         return ch

def sendCommand( cmd ):
   control.write( cmd.decode( "hex" ) )
   return

def manualMove( ):
   inkey = _Getch()

   while(1):
      i = inkey()

      if i == '\x1b':
         j = inkey()

         if j == '[':
            k = inkey()

            if k == 'A':
               sendCommand( '72000004000a' ) # Move arm up
            elif k == 'B':
               sendCommand( '720000fb000a' ) # Move arm down
            elif k == 'C':
               sendCommand( '72fb0000000a' ) # Rotate right
            elif k == 'D':
               sendCommand( '72040000000a' ) # Rotate left
            elif k == '5':
               if inkey() == '~':
                  sendCommand( '72000400000a' ) # Arm forward
            elif k == '6':
               if inkey() == '~':
                  sendCommand( '7200fc00000a' ) # Arm back

      elif i == 'a':
         sendCommand( '720000004c0a' ) # platfrom forward
      elif i == 's':
         sendCommand( '72000000560a' ) # platform back
      elif i == 'd':
         sendCommand( '72000000520a' ) # platform turn right
      elif i == 'w':
         sendCommand( '72000000460a' ) # platform turn left
      elif i == ' ':
         sendCommand( '72000000530a' ) # stop platform
      elif i == 'r':
         reset()
      elif i == 'm':
         maxReach()
      elif i == 'q':
         print
         break
      else:
         print("Key not a control")

      time.sleep(0.100)
```

```
def reset():
   reset = '613f2b06530a' # reset to default position
   sendCommand( reset )
   return

def maxReach():
   maxReach = '613f513f000a'
   sendCommand( maxReach )
   return

def main():
   if ( control == None ):
      print( "No serial ports found" )
      exit()
   else:
      print( "Opening serial on ttyUSB0. Dis gon b gud." )
   reset()
   manualMove()
   control.close()

if __name__ == "__main__":
   main()
```