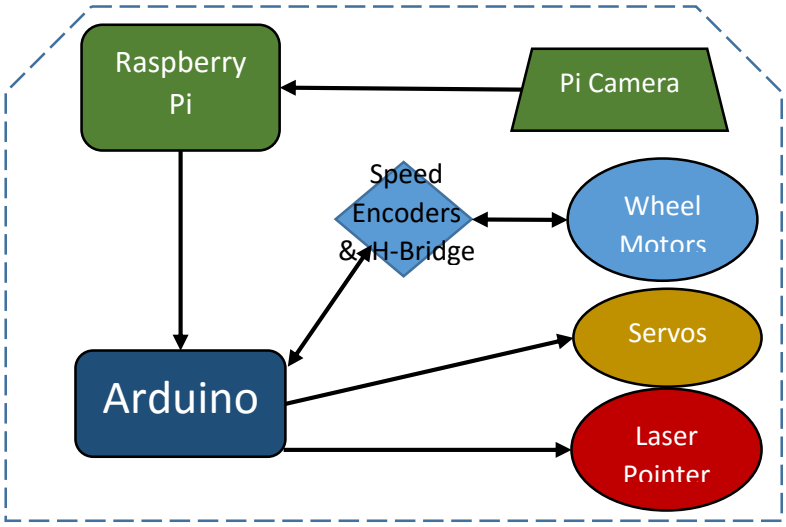
Quadrotor Drone Protocol for the Raspberry Pi and Arduino

For this implementation of how the drone operates, the Raspberry Pi does some calculations and image processing to determine what course of action the drone should take in order to navigate the room and locate the drones target. Once the Pi has determined what action is needed, it sends a message to the Arduino located on the drone over a serial USB connection. The Arduino then processes the message the Pi sent, and executes the action specified within the received message. A similar setup for the ground robot is illustrated in the diagram below. The following report describes in detail the protocol of these messages.

Ground Robot Components



Protocol Design

Each message that the Pi sends to the Arduino is 6 bytes in length, and will consist of characters and numbers only. The first byte of each message will always be a character which specifies the hardware on the drone that is to be moved. This idea is the foundation in which the rest of the protocol is built around. The hardware components on the drone that the Arduino can operate are the motors/propellers, the servos that the camera and laser are mounted on, and the laser itself. Therefore, the first byte of each message is restricted a small set of characters, shown with the piece(s) of hardware that the character represents, and a short description of that command is shown in Table 1 below.

Character Of The 1st Byte In The Message	Drone Hardware Represented By This Character	Description Of What Action The Drone Will Carry Out When Receiving This Character
m	Motors/Propellers	**MOVEMENTS** Move the drone forward, backward, strafe to the left, strafe to the right, move up, and move down.
s	Servos	**SERVOS** Move the horizontal/vertical servos left, right, up, or down.
l	Laser	**LASER** Toggle the laser on or off.
t	Motors/Propellers	**TURNS** Turns the drone left or right.
i	*	**INITIALIZATION** *This additional 'category' was added to be used for tasks on the drone that did not fit into the above 'categories' as well as for testing new implementations.

Table 1

The second byte of each message will also always be a character. The one exception to this is the toggle laser action which has no parameters since it can only turn on and off, this will be describe in greater detail later on. The purpose of this character can be described as a parameter to the action specified by the first character, in the sense that it provides a direction to the action. The possible characters that can be used in the second byte of the message are represented in the tables below, along with what the remaining 4 bytes of the message will represent. The remaining four bytes are read together as one string, and then are parsed into an integer to be passed along into the various methods used within the Arduino code.

MOVEMENTS

(Table 2)

Given that the first byte of the message is an *m*, which specifies drone movement, the second byte of the message is limited to the characters shown in the table below. In this situation, the remaining 4 bytes of the message will represent the target speed that the drone's motors/propellers will need to be set to in order to carry out the indicated movement. **(Target Speeds can be variable. How the speeds shown below were calculated is described further down.)*

Character Of The 2nd Byte In The Message	Direction Represented	Target Speed	Correlated Plane
f	Forward	1600	Pitch
b	Backward	1400	Pitch
l	Left	1400	Roll
r	Right	1600	Roll
u	Up (Increase Altitude)	1621	Pitch, Roll, and Yaw
d	Down (Decrease Altitude)	1421	Pitch, Roll, and Yaw

Table 2

Example protocol message for the move action (Moves the drone forward):

1st Byte	2nd Byte	3rd Byte	4th Byte	*5th Byte*	*6th Byte*
m	*f*	1	6	0	0

TURNS

(Table 3)

With this protocol, the turns work the exact same way as the movement commands. The reasoning behind the turns having their own category was to avoid conflicts with drones strafe left and right movements.

Character Of The 2nd Byte In The Message	Direction Represented	Target Speed	Correlated Plane
l	Left	1424	Yaw
r	Right	1624	Yaw

Table 3

Example protocol message for the turn action (Turns the drone right):

1st Byte	2nd Byte	3rd Byte	4th Byte	*5th Byte*	*6th Byte*
t	*r*	1	6	2	4

TARGET SPEEDS

(Table 4)

The initial target speeds shown in this documentation were gathered manually. In order to gather these values, the drone's Arduicopter was plugged into a computer running the Mission Planner software to monitor each of the drone's motor speeds while operating the drone with its remote control. By changing these speeds of the drone's motors, the drone is able to move in various directions. The values shown in Table 4 are the recorded values.

	YAW	ROLL	PITCH	THROTTLE
Maximum Speed Recorded	1035	1000	1000	1029
Minimum Speed Recorded	2012	2000	2000	2012
Median of Recorded Speeds	1524	1500	1500	1521
40 % Speed	1424	1400	1400	1421
60 % Speed	1624	1600	1600	1621
Related Movement	Turn Left/Right	Strafe Left/Right	Move Forward/Backward	Arm, Disarm, and Move Up/Down

Table 4

SERVOS

(Table 5)

Similar to how the movement message work, the second byte of the message specifies a direction for the servos to move. The exception to this is when the second byte of the message is an *x*, which tells the Arduino to reset the servos to their default positions. The third byte of a servo message isn't used since the range of degrees the servos may move only requires 3 digits, which are specified by the remaining 3 bytes as shown in Table 5.

Character Of The 2nd Byte In The Message	Direction Represented	3rd Byte	Allowed Range Of Remaining 3 Bytes	Correlated Servo
u	Up	0	001 - 180	Vertical
d	Backward	0	001 - 180	Vertical
l	Left	0	001 - 180	Horizontal
r	Right	0	001 - 180	Horizontal
x	Resets Servos to Their Default Positions	0	0	Both

Table 5

LASER

Because turning on and off is the only possible action for the laser, only one byte is needed to execute this action. However since all messages are required to be 6 bytes in length, the remaining 5 bytes are simply used as space fillers and can just be all zeros.

Example protocol message for the laser action:

1st Byte	2nd Byte	3rd Byte	4th Byte	*5th Byte*	*6th Byte*
l	0	0	0	0	0

INITIALIZATION

The initialization messages are used for any remaining commands that did not fall in the above categories, as well as any additional messages we may need to implement in order to test out other things. Currently, there are four supported initialization messages which are described below in Table 6. Each message below has a corresponding method currently coded into the Arduino code.

Character Of The 2nd Byte In The Message	Method the Arduino Calls	What the Method Does	Remaining Bytes Are Currently Unused
a	arm()	Arms the drone	0000
k	disarm()	Disarms the drone	0000
s	setAltitude()	Holds the drones altitude	0000
t	firstTest()	Runs a series of test	0000

Table 6

Autonomously Creating Protocol Commands

In order for the Pi to create a protocol command to send the Arduino on its own after processing some image and making some calculations, we created the following method in Python. First, to make things easier we implemented a switch similar to that of C++ since Python does not support switch statements on its own. The switch in Python is shown below.

```
class switch(object):
    def __init__(self, value):
        self.value = value
        self.fall = False

    def __iter__(self):
        """Return the match method once, then stop"""
        yield self.match
        raise StopIteration

    def match(self, *args):
        """Indicate whether or not to enter a case suite"""
        if self.fall or not args:
            return True
        elif self.value in args:
            self.fall = True
            return True
        else:
            return False
```

With the above switch statement in place, the following method was then created to allow the Pi to dynamically create protocol commands to send the Arduino.

```
#*******************************************************************************
# This method creates a command string to be sent to the Arduino. This method ensures
that any
# commands sent to the Arduino are always 6 characters long by adding the needed
digits.
#
# Note* this method should not be used for the laser toggle command on the robot..
# Laser toggle cmds should just be hard-coded as 'l00000'
# Parameters:
#             first - the first string of the desired command (1 char in length)
#             second - the second string of the desired command (1 char in length)
#             mod - the value of the modifer for the command
# return:  ret - string representation of the command
# Example: createCommand("m", "f", "5")  -> returns "mf0005" (Moves forward 5 ft)
# Example: createCommand("s", "r", "45") -> returns "sr0045" (Turns servo right 45
degrees)
def createCommand(first, second, mod):
       modValue = int(round(mod))        # distance, degree, or motor speed
       includes = [first, second, str(modValue)]
       cmd = ["*","*","0","0","0","0"]
       # for every string included in the call..
       for s in includes:
              # check to see where it belongs int the command
              for case in switch(s):
                     if case("m", "s"):
                            cmd[0] = s[0]
                            break
                     if case("f", "b", "r", "l", "u", "d", "x"):
                            cmd[1] = s[0]
                            break
                     if case():
                            number = int(s)
                            numDigits = numLen(number)
                            nums = [c for c in s]
                            if (numDigits == 1):
                                   cmd[5] = nums[0]
                            if (numDigits == 2):
                                   cmd[5] = nums[1]
                                   cmd[4] = nums[0]
                            if (numDigits == 3):
                                   cmd[5] = nums[2]
                                   cmd[4] = nums[1]
                                   cmd[3] = nums[0]
                            if (numDigits == 4):
                                   cmd[5] = nums[3]
                                   cmd[4] = nums[2]
                                   cmd[3] = nums[1]
                                   cmd[2] = nums[0]
                            break
       # Return the command
       ret = ''.join(cmd)
       return ret
```