

**CG2111A Engineering Principle and Practice**

Semester 2 2021/2022

**“Alex to the Rescue”**

**Final Report**

**Team: B02-1B**

|  |  |  |
| --- | --- | --- |
| Name | Student # | Main Role |
| Hao Zeyu | A0240287E | Hardware Advisor |
| Woo Wen Jun | A0239038B | Software Advisor |
| Amit Rahman | A0244348B | Assistant Team Lead |
| Gau Kiat Lok Jerald | A0234058H | Team Lead |

**Section 1 Introduction**

WOOOOOOOOO…… The emergency sirens resonated in the air as the city went into a state of emergency after the earthquakes. Buildings collapsed and survivors trapped under the rubble. We are only given 72 hours (also known as the golden hour) to locate as many survivors as possible. The longer the rescuers delay, the lower the chances of the survivors' survival. Rescuers would have to traverse through treacherous terrain with risk of collapsed buildings in order to search and reach the survivor. Fortunately, with the advancement in technology, we are able to carry out search-and-rescue operations easily with our robot Alex. We are able to send Alex to go deep into the rubble to search for survivors, reducing the risks that the rescuers have to take when carrying out rescue missions. Through the use of Lidar, we are able to map out the surroundings, giving us a full view of the terrain. Now, we are ready to save these survivors.

**Section 2 Review of State of the Art**

## 2.1. MSRBots

a) The MSRBOTS search-and-rescue system is developed by researchers from the Beijing Institute of Technology. It is a system to enable robots to carry out rescue in a coal mine environment. (Zhao et al., 2017) [1]

The robots can be used as mobile sensors and send out warnings to the rescuers when they detect danger. The robots are equipped with gas and infrared sensors, which enabled the robot to detect flammable gas while being able to detect obstacles while navigating. A fibre-optic cable is being used to communicate with the robot up to 2km underground. A software system is installed in an operating control unit (OCU) to control the robot. A microcontroller (STM32) without the operating system is responsible for the motion control of the robot. The communication between the OCU and electric apparatus in the robot electric box is based on the TCP/IP protocol.

b) The advantage of this robot is that it is waterproof and explosion proof. However, in a coal mine environment, an electronic compass, gyro and two code plates of travelling motors are used to deduce the robot movement trajectory. This is due to GPS signals not being available underground. This means that the localisation of the robot is not accurate.

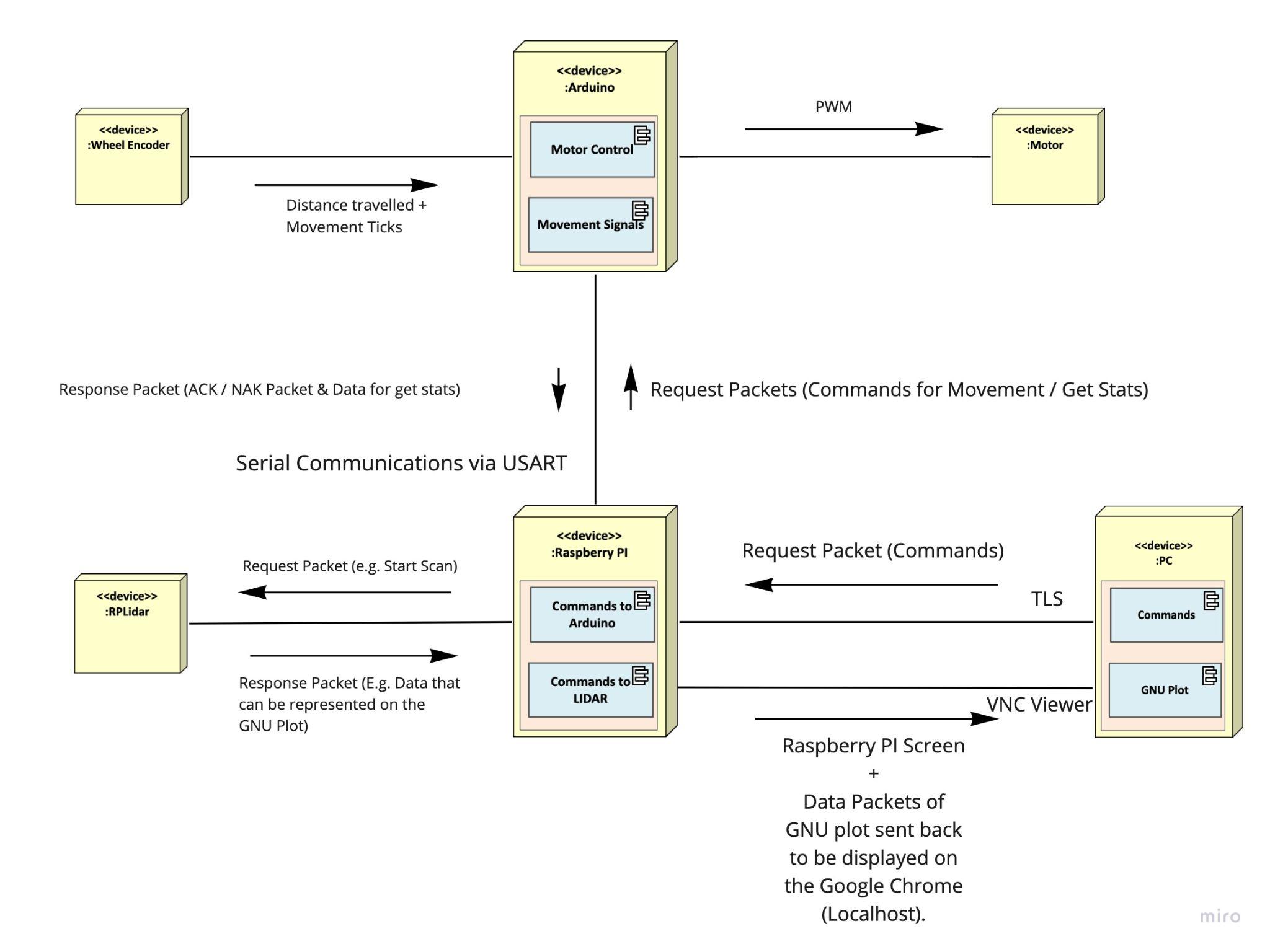
## 2.2. Boston Dynamics’ Atlas

a) Atlas the humanoid robot was engineered to perform basic but potentially life-saving tasks in dangerous conditions: flipping switches, shutting off valves, opening doors, running power equipment. (Boston Dynamics, 2019) [2]

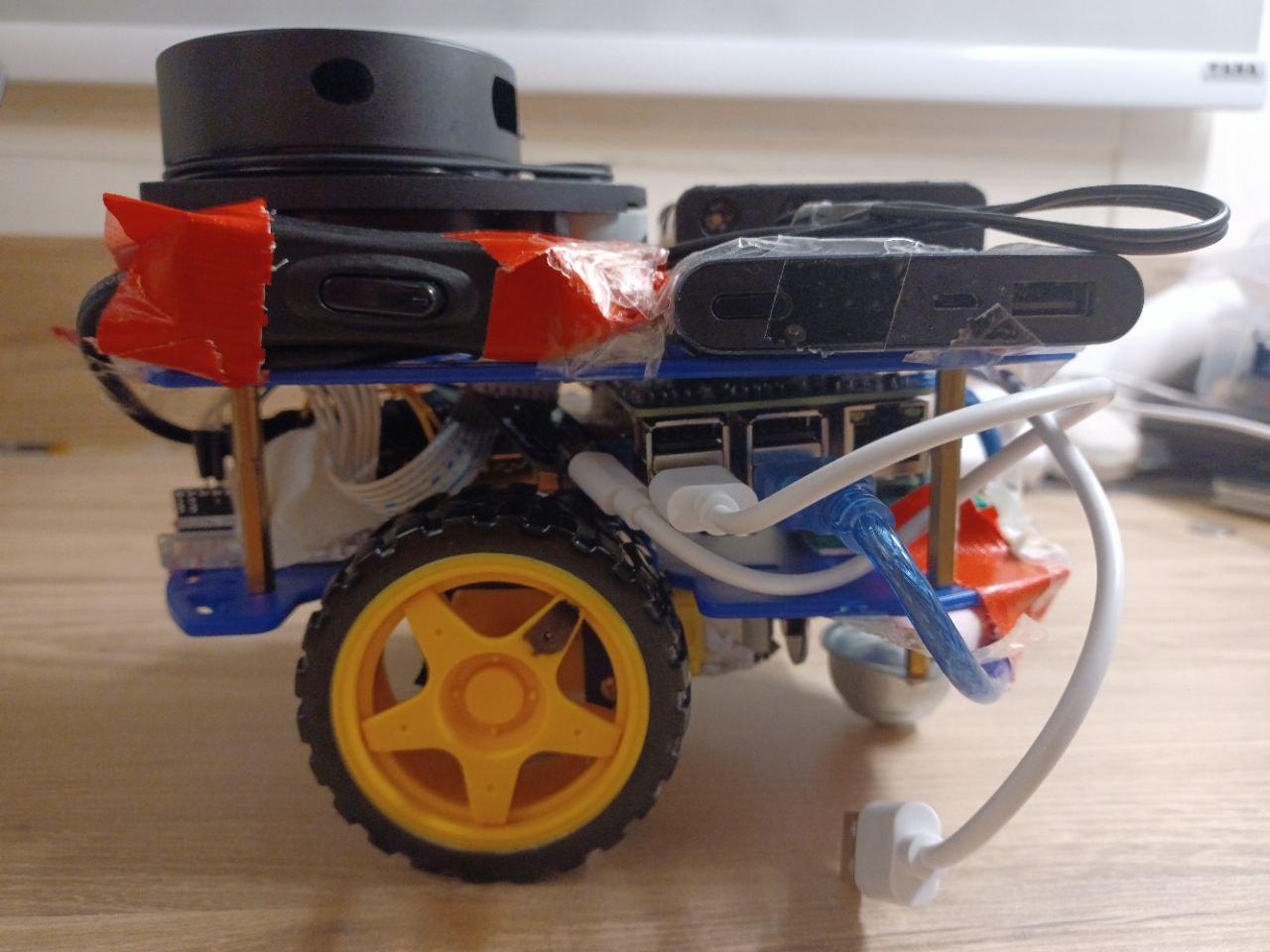
It is electrically powered and hydraulically actuated. It uses sensors in its body and legs to balance, and it uses LIDAR and stereo sensors in its head to avoid obstacles, assess the terrain, help with navigation, and manipulate objects. Valves, and a compact hydraulic power unit enable Atlas to deliver high power to any of its 28 hydraulic joints for impressive feats of mobility. Algorithms implemented and sensors equipped enabled Atlas to generate point clouds of the environment and detect its surroundings. ROS and C++ are used for most things in programming the robot and python is used for data analysis. (*Gazebo : Tutorial : Atlas Control over ROS with Python*, n.d.) [3]

b) The advantage of the robot is that it is very agile and can perform tasks on behalf of human rescuers, which means that humans do not need to be exposed to dangerous conditions while rescuing. The disadvantage is that its battery life is only about one hour, which might be insufficient for long rescue efforts. (*ATLAS DRC Robot Is 75 Percent New, Completely Unplugged*, n.d.) [4]

**Section 3 System Architecture**



**Section 4 Hardware Design**

1.

Rpi

Batteries

LiDAR

Breadboard

Arduino

Chip

Powerbank

2. We used BluTack to stick the magnetic sensors to the bodies of the motors. This is to ensure that the sensors do not shift easily. We found that BluTack is a better adhesive than Scotch Tape after experimenting with both, as the scotch tape would peel off after a while.

3. We found out that when the metal part of the “serial to USB bridge” chip touches the pins of the arduino, the LIDAR will be shorted. To solve this problem, we covered the metal part of the chip with Scotch Tape and added a layer of cardboard on top of the arduino pins to prevent possible contacts. In addition, we cut the chassis so that wires can pass through.

4. We mounted the Arduino Uno to the top of the first layer of the chassis at the front, and used jumper cables to tie and secure it into place. We then mounted the Pi to the bottom of the second layer of the chassis at the back and secured it into place with more jumper cables. This arrangement of the Arduino Uno and the Pi makes it such that when the first and second layer of the chassis is put together, they will be placed side by side with no overlap, minimising the space taken up by them. The mini breadboard is placed at the back of the first layer in the gap below the Pi. We used screw extensions to increase the height between both layers of the chassis so that there would be sufficient space for all the hardware components inside.

5. We also ensured that with the arrangements in point 4, the ports of the Pi are easily accessible so that we could connect and remove the necessary cables easily (such as the power cable and HDMI cable to hook up to an external monitor) without dismantling Alex everytime.

6. We placed the Lidar unit on top of the second layer at the front of the Alex, and ensured that there were no wires or other hardware components blocking its surroundings to prevent the Lidar readings from being affected. We secured the Lidar unit by mounting it with screws to the frame.

7. We placed the power bank on top of the second layer behind the Lidar unit, and placed the battery holder on top of the power bank. Both of them were secured into place with double sided tape. We wanted the battery holder to be on the upper most layer to easily be able to replace the batteries without having to dismantle Alex.

8. For cable management, we secured the switch for the motors to the side of the Lidar unit with electrical tape, taking care not to block the Lidar sensor. For the Pi power cable, Arduino Uno power cable and Lidar cable, we used more electrical tape and scotch tape to secure them onto the frame of Alex so that they do not stick out too much, which can cause them to hit the walls of the maze while moving Alex.

**Section 5 Firmware Design**

**High Level Algorithm on the Arduino UNO:**

1. Receives a command from the Raspberry PI (Refer to Section 5.2.).

**Communication Protocol (RX):**

1. Create a Struct that holds a local copy of the data. The Struct has a total size of 140 bytes. We will also create a char Receive Buffer of size 140 to accommodate the bytes that we receive. Do note that even though packet.h only contains 100 bytes, the Struct in serialize.cpp contains 140 bytes and our group chose to proceed with 140 bytes since both the RPI and Arduino uses the same serialise and deserialize.

|  |  |
| --- | --- |
| typedef struct comms  {  uint32\_t magic;  uint32\_t dataSize;  char buffer[MAX\_DATA\_SIZE];  unsigned char checksum;  char dummy[3];  } TComms;  Serialize.cpp (140 bytes) | typedef struct  {  char packetType;  char command;  char dummy[2]; // Padding to make up 4 bytes  char data[MAX\_STR\_LEN]; // String data  uint32\_t params[16];  } TPacket;  Packet.h (100 bytes) |

**Fig. 5.1. Comparing 2 packets**

1. When there is data to be received, the USART\_RX\_ISR will be triggered. We are only able to read in 1 byte at a time during each execution of the ISR. Therefore, we will allocate the byte in the UDR0 into the first element of the Receive Buffer. Increase the counter by 1. Repeat Step 2 until the counter reaches 140.
2. Execute ReadPacket which will deserialize the Receive Buffer and convert the bytes into data. We will first check if there is a Bad Magic Number and Bad Checksum. If there is any of the either, we will transmit a NAK packet back to the sender. Otherwise, we will copy the data onto the local copy of the Struct which will then be used to execute the corresponding command.

**High Level Algorithm (Continued)**

1. The type of command to be executed will be based on the command that we have copied over into the char command of the Struct created in packet.h.

|  |
| --- |
| switch (command->command)  case COMMAND\_FORWARD:  sendOK();  forward((float) command->params[0], (float) command->params[1]);  break; |

**Fig. 5.2. Code Snippet used to determine command to run**

1. If the command is to move in a certain direction, calculate the required ticks and generate the necessary PWM to drive the motors. When the motors are being driven, execute an **EXTERNAL INTERRUPT** that will increase the number of ticks based on the number of times that the interrupt was executed. Use the ticks to calculate the distance travelled or angle turned by the robot. The tick value is being calibrated during the lab session and being stored in the global variable.
2. If the command received is to getStats, we will store the data in the local Struct and serialise it.

|  |
| --- |
| void sendStatus()  {  TPacket statusPacket;  statusPacket.packetType = PACKET\_TYPE\_RESPONSE;  statusPacket.command = RESP\_STATUS;  statusPacket.params[0] = leftForwardTicks;  statusPacket.params[1] = rightForwardTicks;  statusPacket.params[2] = leftReverseTicks;  statusPacket.params[3] = rightReverseTicks;  statusPacket.params[4] = leftForwardTicksTurns;  statusPacket.params[5] = rightForwardTicksTurns;  statusPacket.params[6] = leftReverseTicksTurns;  statusPacket.params[7] = rightReverseTicksTurns;  statusPacket.params[8] = forwardDist;  statusPacket.params[9] = reverseDist;  TXLen = serialize(TXBuffer, &statusPacket, sizeof(TPacket));  UCSR0B |= 0b00100000;  } |

**Fig. 5.3. Code Snippet of the Struct being serialised**

1. We would convert the Struct into a series of bytes which will be placed inside of a char Transmit Buffer of size 140. At the same time, we would calculate the checksum using an XOR operation and place the magic number and checksum inside of the Buffer as well. We will activate the UIRE0 interrupt.
2. We have a counter that starts from 0. We load the first element of the transmit buffer into the UDR0 which will then send the data over. We will increment the buffer by 1. The ISR will continuously be triggered.
3. When the counter reaches 140, we know that we have sent all of the bytes inside of the Transmit Buffer. We can reset the counter and turn off the interrupt.

**Flaw in this Setup:**

We acknowledge that using an array is a poor implementation as if there are 2 ISRs being triggered side by side, the original Receive Buffer will be overwritten and the data would be corrupted. Hence, a better implementation would be to use a Circular Buffer. However, considering that most of our teammates have yet to take CS2040C, an array would be the next best alternative instead of a Circular Buffer which capitalises on Linked Lists.

**Section 6 Software Design**

1) RPi will first start the communication using startSerial, and it specifies the same length and parity bits as the arduino. It will then spawn a receiver thread. This is to enable RPi itself to receive the response packets from the arduino. RPi will then initialise the communication by sending a hello packet to the arduino. This is done by creating a packet using the type Tpacket before serialising it. The arduino will respond to it by sending RPi a response packet. How RPi handles the response packet will be explained soon.

|  |
| --- |
| TPacket helloPacket;  helloPacket.packetType = PACKET\_TYPE\_HELLO;  sendPacket(&helloPacket); |

**Fig 5.4 Code Snippet of sending a helloPacket**

2) User will start a teleoperation command by typing the directions, represented by f (forward), b (reverse), r (right turn), and l (left turn). The controller will then enter the distance (angle) travelled as well as the motor power. The information will be serialised before sent to the arduino. The function below changes the command in the struct variable commandPacket as well as the params[0] and [1] in commandPacket through the “getParams” function.

|  |
| --- |
| void sendCommand(char command)  {  TPacket commandPacket;  commandPacket.packetType = PACKET\_TYPE\_COMMAND;  switch(command)  {  case 'f':  case 'F':  getParams(&commandPacket);  commandPacket.command = COMMAND\_FORWARD;  sendPacket(&commandPacket);  Break;  }  } |

**Fig 5.5 Code Snippet of sending a command**

This shows a situation where the command is forward. The actual command will be sent in the commandPacket. “getParams” will prompt the controller to enter the distance (angle) and power. Serialisation will be done in the function “serialise” as specified in the header file serialise.h. Other remaining cases implement similar patterns of code, with forward and f being changed to other directions and their corresponding letters.

3) The arduino will send back to the RPi a packet storing the response. RPi will first deserialize the packet using the function defined in serialise.h. The variable “result” is returned from the deserialize function and indicates whether there is corruption in the data transmitted. If the status of the packet is “Packet Ok”, RPi calls the handlePacket function. Otherwise, when the message sent from arduino is erroneous, the handleError function will be called and the error message printed could either be “bad magic number” or “bad checksum”.

|  |
| --- |
| if(len > 0)  {  result = deserialize(buffer, len, &packet);  if(result == PACKET\_OK)  {  counter=0;  handlePacket(&packet);  }  else  if(result != PACKET\_INCOMPLETE)  {  printf("PACKET ERROR\n");  handleError(result);  }  } |

**Fig 5.6 Code Snippet of handling response from arduino**

In the handlePacket function, RPi will check whether the arduino has received an error message previously from RPi itself or is simply sending a response to acknowledge that it received the command. If there is no error and no request to get the arduino’s status, “command ok” will be printed. Otherwise, depending on the type of errors, error messages will be printed as specified in the handleErrorResponse function. The RPi might also only print messages from the arduino if the packetType is Message. Shown below is the function of handlePacket.

|  |
| --- |
| void handlePacket(TPacket \*packet)  {  switch(packet->packetType)  {  case PACKET\_TYPE\_COMMAND:  break;  case PACKET\_TYPE\_RESPONSE:  handleResponse(packet);  break;  case PACKET\_TYPE\_ERROR:  handleErrorResponse(packet);  break;  case PACKET\_TYPE\_MESSAGE:  handleMessage(packet);  break;  }  } |

**Fig 5.7 Code Snippet of handling a Packet**

**Section 7 Lessons Learnt - Conclusion**

First Lesson Learnt and Mistake Made as a Group

The first most important lesson we learnt was the importance of proper hardware planning. When we were first given the components to assemble Alex with, our group simply had a rough idea of what we wanted Alex to look like and where certain components should go. For us, these were the requirements that we wanted for the layout of Alex:

1. The battery pack has to be on the uppermost layer so that we could place and remove the batteries easily without having to dismantle Alex everytime we needed to charge the batteries.
2. The Lidar unit also has to be on the uppermost layer at the front, and that the surroundings of the lidar unit cannot be obstructed by any objects such as wires or other hardware components, otherwise the lidar readings would be affected.
3. The ports of the Pi have to be easily accessible so that we can connect the necessary wires such as the power cable and HDMI cable when needed easily without having to dismantle Alex everytime.

However, other than the above requirements, we did not have much more of a solid plan for where to place the components, such as the specific locations for the Arduino Uno, the Pi, or the breadboard. This would end up tying in to the first greatest mistake we made as a group. After throwing together these components wherever they could fit and securing them with zip ties, we realised that the zip ties and certain parts of the Pi unit got in the way of installing the Lidar unit, which is to be located directly on top of the Pi on the frame of Alex. In the end after much frustration dealing with the semi-permanent zip ties and trying to create more space on the frame, we ended up having to completely dismantle all the components and redoing the layout, this time with proper planning, and thus we were finally able to fit everything onto the frame of Alex and at the same time satisfying our requirements above. However, this process had cost us to spend the whole lab session on it without having any other productive results to show for it.

Therefore, this taught us that we really cannot skimp on the planning stage, and just by spending some time to make a sound plan on the layout of the components, we can save ourselves a whole lot of time and agony in trying to fix problems that could have been mitigated from the start.

Second Lesson Learnt and Mistake Made as a Group

The second most important lesson we learnt was to never doubt Murphy’s Law, in that anything that can go wrong will go wrong. That is to say that during the testing phase, when an issue crops up once or twice, never assume that these are one-off issues that can be waived off as coincidences, and instead make sure to investigate deeply into their root cause and fix them to eliminate the chances of such an issue popping up again in the future. This lesson is derived from the next greatest mistake that we made as a group, which is arguably the most fatal mistake we made. This mistake was to use an unstable mobile hotspot connection for the communication between our laptops and Alex through VNC. During our development and testing process for Alex, when we used an Android phone for its mobile hotspot to connect Alex and our laptops, there were no issues and a stable connection was maintained. However when we used an iPhone for its mobile hotspot, we noticed that the connection was spotty at times and outright had problems even connecting Alex to the hotspot once. We simply assumed that this was a trivial issue and after managing to get Alex to connect to the hotspot, we carried on as per normal and paid no more heed to this issue.

Unfortunately, this issue would come back to bite us in the foot at the worst possible time. During our final demonstration, we again went with an iPhone for its mobile hotspot. When we were setting up Alex before being put in the maze, it suddenly dropped connection to the hotspot and the VNC connection was lost. We were in a panic as we only had a minute left before having to send Alex to be put into the maze, and we had to quickly try to re-establish the connection. Luckily, connection was restored in the nick of time and we managed to send off Alex on time. However, this was where our luck ended. With around two minutes left on the timer to complete the navigation of the maze, Alex once again dropped its connection to the mobile hotspot and without being able to physically handle Alex, we were unable to do anything about this problem and had to simply do our best to draw out the map with whatever data we had. This goes without saying that we were unable to park Alex and thus lost out on those marks as well.

This served as a very painful lesson to us to never just assume a problem will magically fix itself, and we have to make sure that each problem that crops up during development has to be seriously looked into and ensure that it will not happen again (as well as that iPhone hotspot connections are very unreliable and we should never use them in projects ever again).

**References**

1. <https://www.mdpi.com/1424-8220/17/10/2426/htm>l
2. <https://www.bostondynamics.com/atlas>
3. <http://gazebosim.org/tutorials?tut=drcsim_ros_python&cat=>
4. <https://spectrum.ieee.org/amp/atlas-drc-robot-is-75-percent-new-completely-unplugged-2650272089>