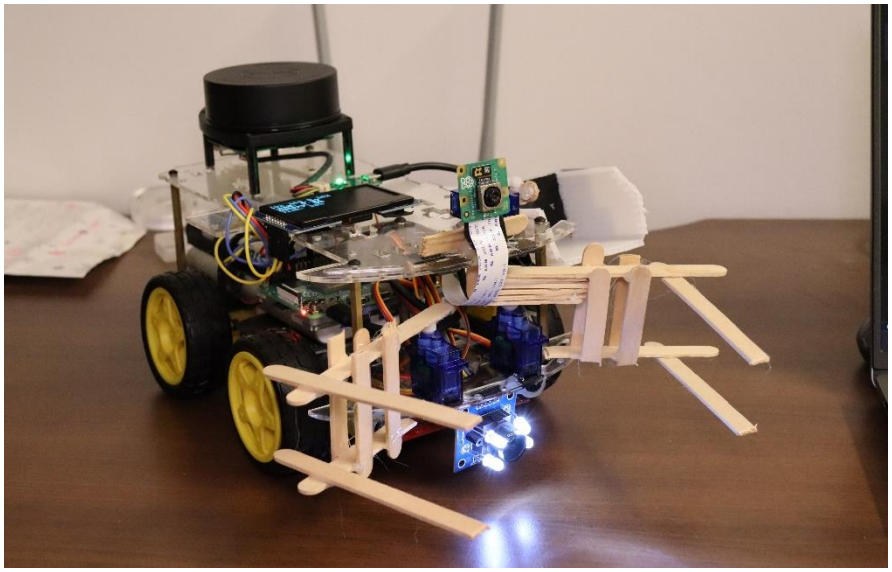
**CG2111A Engineering Principles and Practice II**

Semester 2 2024/2025



# "Alex to the Rescue"

# Final Report

# Team: B05-05



| Name | Student # | Main Role |
|---|---|---|
| Heng Teng Yi | A0308948M | Integration |
| Lim Zerui | A0300434X | Arduino Firmware |
| Wong Chuan Hui | A0282468R | RPi Programming |
| Juneja Akshat | A0288024A | Robot Assembly |

## Section 1 Introduction

We have split the search and rescue problem into 5 goals which our design aims to address:

1. The robot must explore an unknown environment via wireless teleoperation.
2. The robot is expected to provide the users (us) with a map of its environment, so that we can decide how to navigate the area to avoid running into walls and find important objects (astronauts). We also need this map to draw the unknown environment.
3. Upon seeing an astronaut on the map and moving towards it, the subsequent actions depend on the colour of the astronaut (red or green). Hence, reliable colour classification of the astronaut is required.
4. Before starting its run, Alex will be pre-loaded with an 8x5x5cm medpack. Upon encountering the green astronaut, it must deposit the medpack near the astronaut.
5. Upon encountering the red astronaut, Alex must be able to safely move it to the endpoint (parking zone).

Broadly speaking, Alex achieves this mission by processing its raw sensor data into useful information (e.g. SLAM maps, colours) that can be viewed with the team laptop. After we view the information, we use the team laptop to wirelessly control Alex's next action.

## Section 2 Review of State of the Art

**iScream by team iRAP ROBOT (Robocup Rescue 2023)**

This robot, developed for disaster response scenarios, is a tracked mobile platform equipped with advanced perception and manipulation capabilities. Its hardware comprises a caterpillar traction system for inclined surface traversal, a 6-DOF manipulator arm for precise object handling, and multiple sensors such as 3D-LiDARs, depth and thermal cameras for mapping and victim detection. Communication and control are enabled through a wireless network adapter.

The software stack is built on Ubuntu 20.04 with ROS Noetic, with LiDAR and camera-fused SLAM for accurate 3D mapping, MoveIt! for robot arm control, and OpenCV for detecting victims.

This robot's main strengths lie in its autonomous navigation capabilities, achieved through probabilistic localization and 3D SLAM, which reduce operator burden. Its compact chassis also allows for easier manoeuvrability in confined environments.

However, the robot's 77kg weight (100kg with its transport case) limits its portability, making it less suitable for rapid deployment in certain field conditions.

Before building Alex, iScream showed us the importance of **mobility and compactness** in rescue scenarios for navigating tight spaces. While Alex adopts a simpler wheeled design for ease of control and lower mechanical complexity, we shared the same goal of **reducing the robot's footprint** as much as possible.

iScream's design also highlighted the need to **reduce the operator's burden**. While we did not add autonomous navigation, we ensured that our map visualisation also contained a **size-accurate visualisation** of the robot, making it easier to avoid collisions.

**Kamuy2 by Team NITRo (RoboCup Rescue 2023)**

Kamuy2 is a quadrupedal search-and-rescue robot designed to handle challenging terrain and perform victim detection in disaster zones. It features four-legged locomotion powered by 12 DYNAMIXEL servo motors, a 6-DOF manipulator for object handling, and an array of sensors like 3D-LiDARs and depth and thermal cameras.

The robot is supported by high-performance onboard processing (Intel NUC11 and NVIDIA Xavier NX), while its software stack is built upon Ubuntu 20.04 with ROS Noetic. It uses RTAB-Map for precise SLAM, MoveIt! for robot arm motion planning and computer vision tools like YOLO and OpenCV for hazard and victim detection. The robot is teleoperated wirelessly using a DualSense controller.

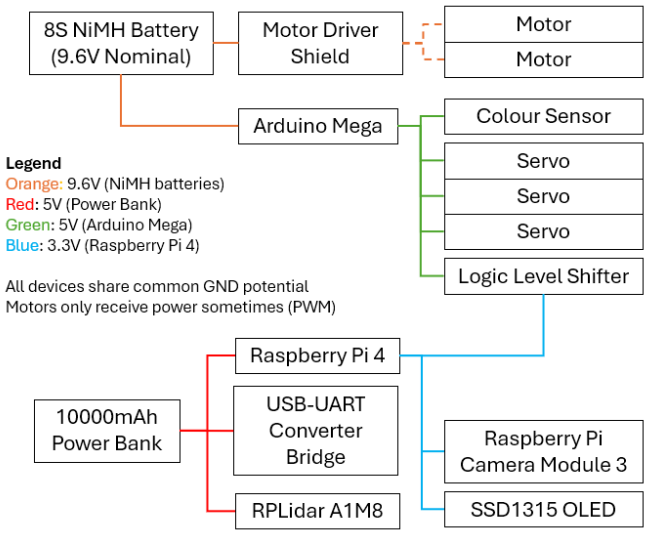
Kamuy2's main strength is a quadrupedal design with suspension, allowing it to traverse uneven terrain effectively. Its lightweight 15kg build also enhances its portability.

However, its design also exhibited several limitations. The robot had limited payload capacity (1kg at full arm extension), which restricts its ability to handle debris. It also lacks autonomous navigation features, relying more on the operator's dexterity.

For Alex, our initial design of the arm had similar weaknesses to Kamuy2, where the astronaut was **too heavy** to be moved by the arm. Thus, we had to ideate and improvise. Eventually, we **improved the payload capacity** by strengthening the mechanical connection between the arm and servo shaft (replacing cable ties with screws and hot glue).

# Section 3 System Architecture

## Section 3.1 Power Architecture

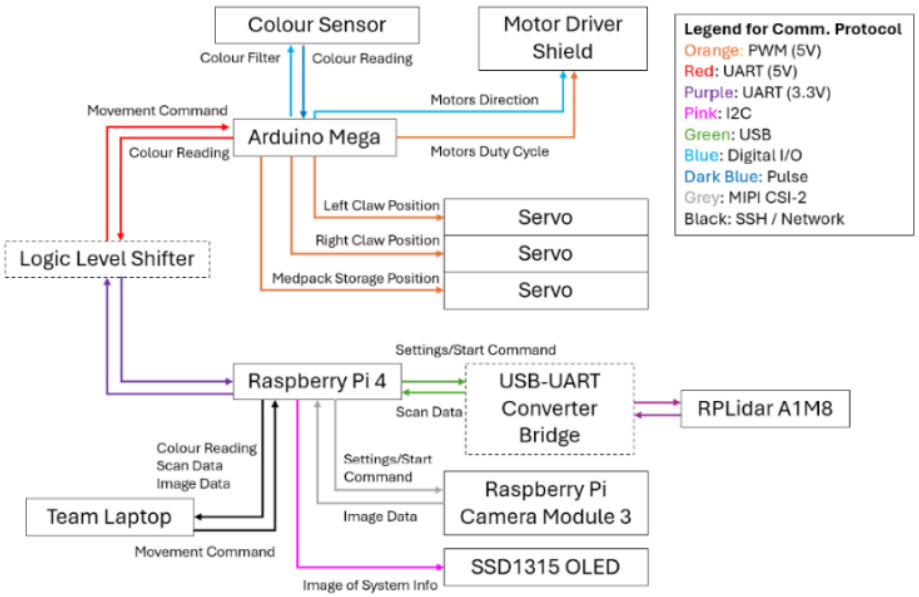



The power architecture diagram above depicts the voltages and/or power sources for each component of the robot, which all share a common ground potential (0V).

The Mega and Pi 4 have their own built-in voltage regulators which regulate their input voltages down to their logic level voltage. Some devices are powered by these regulated power lines.

Additionally, the motors are shown in the diagram receiving 9.6V, but it should be noted that it is a 9.6V PWM signal; it is not continuously powered.

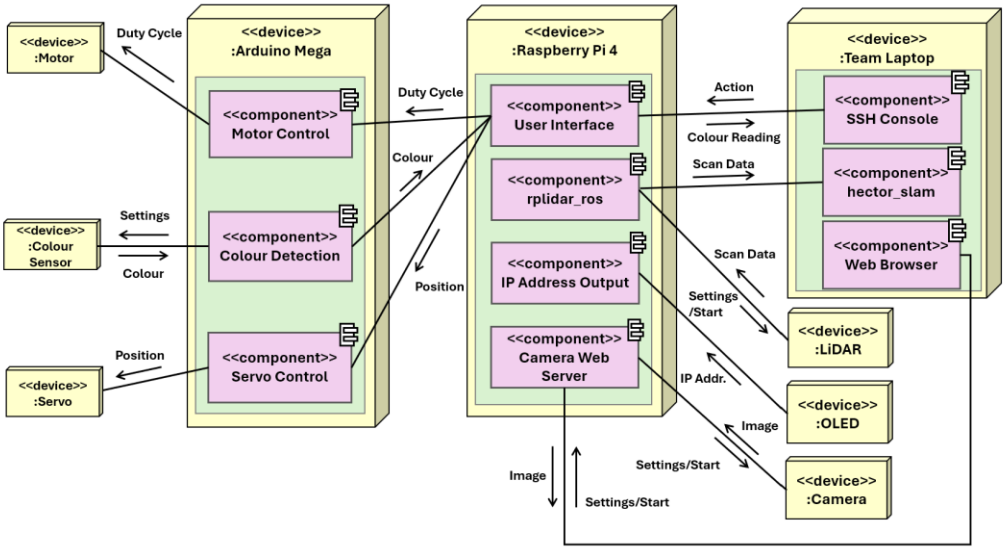
## Section 3.2 Communications Architecture




The communications architecture diagram above shows how all the devices communicate with each other and the protocols used.

Converters/shifters are used to convert the same data between different protocols or logic levels.

## Section 3.3 Program Flow

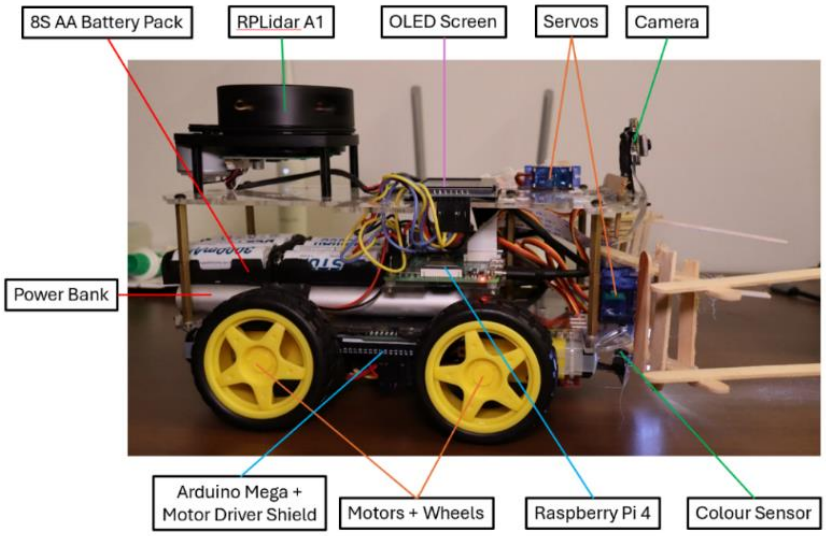



The program flow (UML-like) diagram above depicts how the software components running on each of our computers interact with each other & other hardware, specifically with regards to the data flow and contents.

Since the Pi 4 & team laptop have multiple processor cores and schedulers, they can run multiple processes concurrently or in parallel. Hence, most of the software components shown are separate programs.

Conversely, the Mega can only execute 1 instruction at a time, but is able to contain all components within a single program using timers and interrupts; functionality is retained as long as all interrupts are serviced within an acceptable amount of time.

# Section 4 Hardware Design




The diagram above shows the layout of Alex's main components (omitting minor hardware), wired according to the architecture diagrams in Section 3.

In the spirit of conciseness, we will omit obvious hardware applications (LiDAR used for mapping, motor driver controls motors, etc.) but go into more detail on areas we put more consideration into, or noteworthy hardware features.

**Section 4.1 Minimizing Vibration from Movement**

Because our mapping relies solely on finding relations between consecutive LiDAR scans, we aimed to reduce noise in our scan data by reducing vibration from the robot's movement.

If Alex's weight is evenly distributed amongst the 4 wheels, each wheel must slip perpendicular to its axis of rotation when the robot turns, creating friction and vibration.

Hence, we concentrated most of Alex's weight (power bank, batteries and LiDAR) right above the back 2 wheels (connected to the motors), such that the robot's movement behaves more like a 2-wheeled robot with differential steering. When turning, the back 2 wheels will not slip (producing consistent, predictable movement), whereas the 2 front wheels will still slip but with low friction due to their lower load weight.

Additionally, Alex's heaviest items (batteries, power bank) are placed as low as possible above the back wheels, lowering its centre of gravity for smoother, more stable movement.

We did consider removing the front 2 motors so the gearboxes could spin with less friction; however, the robot had trouble moving the astronaut with 2 motors. Since the batteries, motors and drivers could not be replaced, we retained our 4-motor drivebase.

**Section 4.2 Minimizing Robot Footprint**

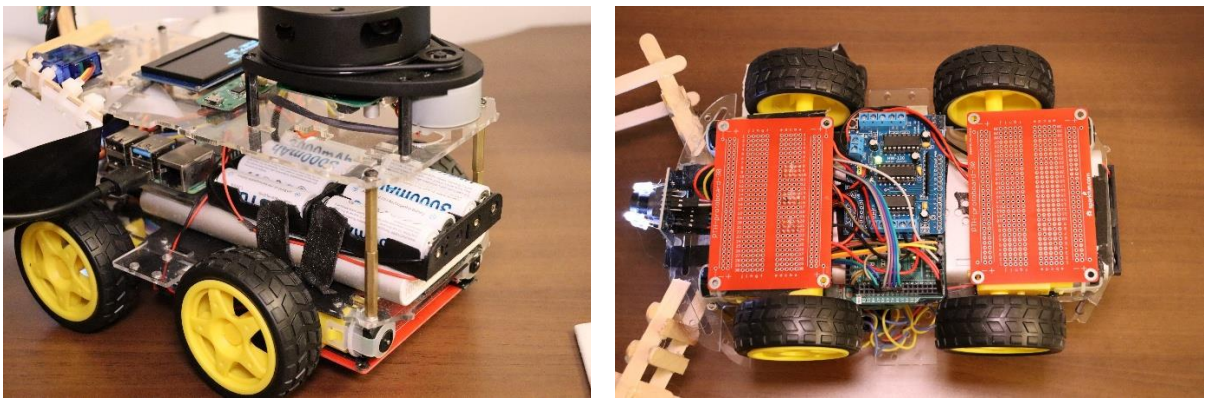
As we learned from our trial run, reducing the 2D footprint of Alex is crucial to give the operator more leeway when navigating tight spaces.

However, our previous decision to mount the power bank and batteries above 2 wheels for smoother movement left little space for the Arduino Mega (amongst other components).

To achieve our desired mounting without compromising on footprint, we flipped the 4 gearboxes to make space for the Mega on the robot's underside. For wiring, we then used solderable breadboards for various benefits:

- Structure: Added rigidity, preventing chassis from flexing under load

- Wiring: Built-in traces allowed for simple and secure power/signal rails for servos
- Cable Management: Prevented long jumper wires from touching the floor



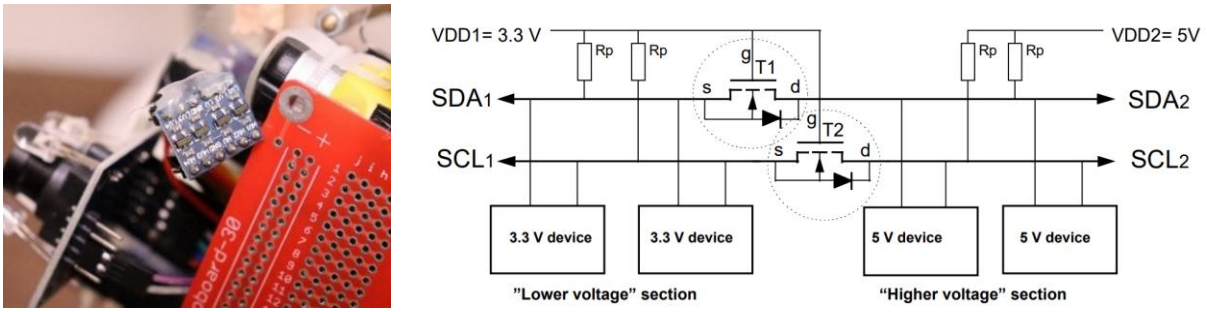
*Back (left) and underside (right) of robot*

**Section 4.3 Pi-Mega UART Communication with Logic Level Translation**

As explored in the studios, bi-directional communication between the Pi 4 and Mega is necessary to both control the robot and view colour.

However, our USB cable was bulky and when connected via USB, the Pi does not always attach the Mega as the same TTY interface. Moreover, our team preferred to upload programs from our laptops, so leaving the Mega's USB port open was desirable.

Hence, we used the Pi's GPIO UART pins to communicate with the Mega's USART2 pins instead. Since the Pi's GPIO uses 3.3V logic while the Mega uses 5V logic, we used a bi-directional logic level shifter (MOSFETs and pull-ups) to translate between the 2 voltages.




*Logic Level Shifter Board (left) and Circuit (right, but replace SDA/SCL with TX/RX)*

**Section 4.4 OLED Screen**

When the Pi is powered on, it is not known whether it has connected to the network, which can make setup unpredictable. Hence, we added an SSD1315 OLED screen connected to the Pi's GPIO I2C pins.

When the Pi boots, it automatically starts a Python script which refreshes the screen with system information and (if connected to a network) the Pi's IP address. This addition was invaluable in speeding up our setup and debugging while providing some peace of mind.

**Section 4.5 Colour Sensor Lens**

Like most teams, we mounted a front-facing TCS3200 colour sensor to sense astronaut colour.

However, reflected light sensors face the issue of wide angular sensitivity; they receive and measure light from a wide range of angles (even perpendicular to the sensor!) even though our area of interest only lies within a narrow angular range. This makes them susceptible to changes to background lighting, resulting in poor accuracy.

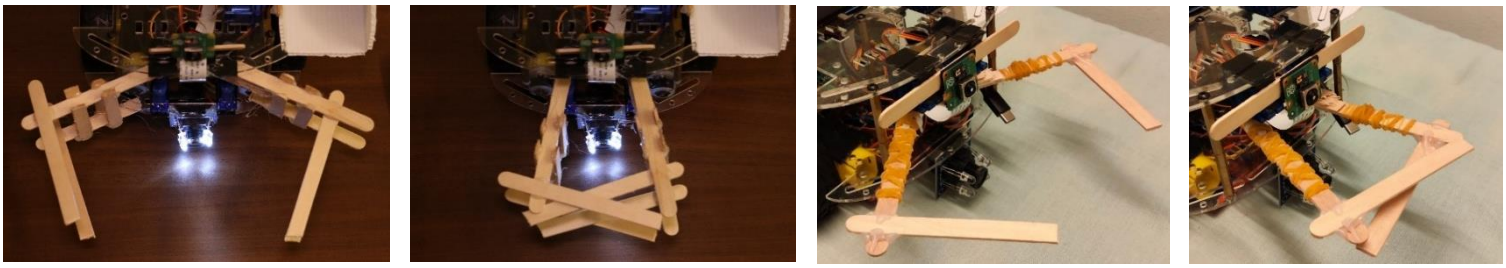
To rectify this issue, we attached a cheap lens to the sensor board's built-in M12 lens mount, which focuses the reflected light from a narrow field of view onto the sensor, providing better background light immunity and effective range.


*OLED Screen (left) and Colour Sensor with M12 Lens (right)*

**Section 4.6 Claw**

In our trial run, our 2-pronged claw failed to stop the astronaut from toppling, and one arm even broke off in the process. We redesigned the claw to have 2 interlocking arms, each with 2 prongs, which provided advantages in these areas:

- Positioning Tolerance: Since the claw was wider when opening, the operator could position the astronaut between the arms with less difficulty.
- Reach: As the arms close and the prongs interlock, the astronaut is naturally centred and nudged towards the robot, allowing us to grab astronauts from a greater distance.
- Grip: When closed, the arms fully enclose the astronaut (preventing it from toppling) and can interlock more if more grip is needed

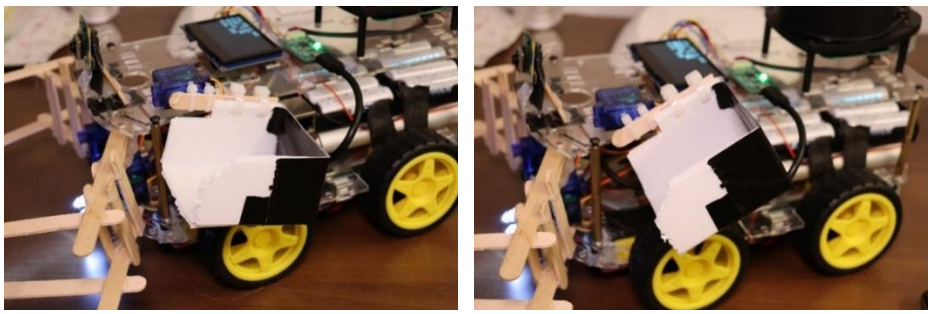


*From left to right: Final claw open and closed, old claw open and closed*

**Section 4.7 Medpack Deposit**

Our medpack deposit mechanism is placed on the side of the robot, facing the same direction as our claw.

This means we can deposit even when the claw is holding onto an object (e.g. red astronaut) and do not require additional rotation to deposit after finding the green astronaut, saving us some time and affording extra flexibility in our movement decisions.



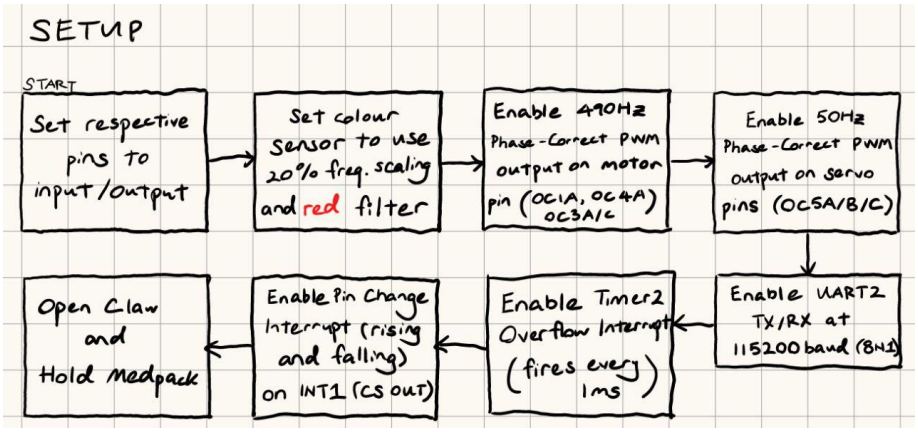
*Medpack holder in hold (left) and deposit (right) states*

## Section 5 Firmware Design

The Arduino Mega is responsible for motor control, servo actuation and colour classification. Additionally, it must control the robot based on commands received from the Pi and transmit the classified colour to the Pi.

While the firmware (Appendix A) is quite simple, we prioritized reliable behaviour over flashy but ultimately unnecessary features and aimed to showcase our learning by using bare-metal programming as much as possible.

Step 1. Setup




The diagram above shows all the high-level steps needed to setup the Mega before entering the main program loop. Interrupts are disabled until all setup is complete.

The colour sensor uses 20% frequency scaling, as we found that 100% frequency pulses were too fast for the Mega to time microseconds properly. We only use the red and green filters as they are sufficient for distinguishing red from green.

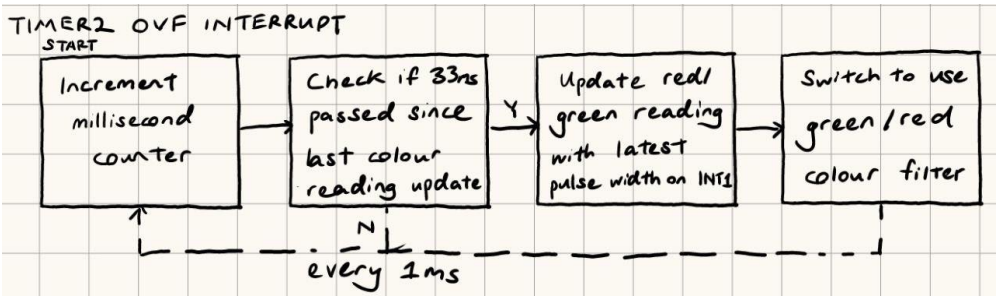
The servos' 50Hz PWM outputs are all controlled by Timer5. Timer5's clock is pre-scaled to 2MHz with TOP=20000, so that we can conveniently control servo positions by writing the pulse width (in us) directly to the corresponding OCR5x register.

For the motors' PWM outputs, their timers (1,3 and 4) use a pre-scaled 2MHz clock with TOP=255, so that we can control the duty cycle by writing an 8-bit duty cycle value to the corresponding OCRxx register.

For communications, we enable full-duplex UART on the USART2 pins with a baud rate of 115200, 8 message bits, no parity bit and 1 stop bit (8N1).

Finally, we initialize a recurring timer interrupt and a pin change interrupt, which will be explored in more detail below.

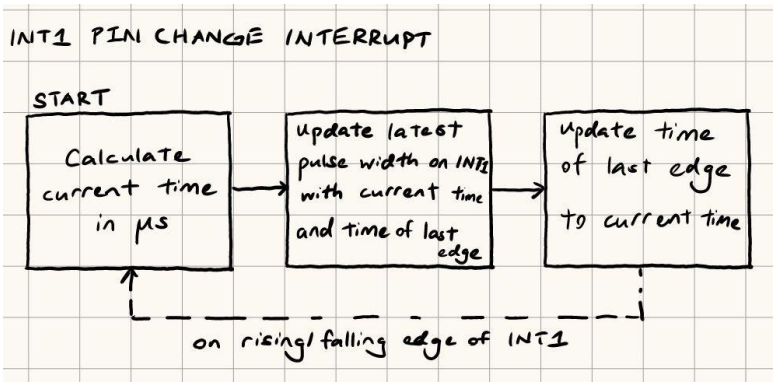
Step 2a. Timer2 Overflow Interrupt




The diagram above shows the flow of the timer interrupt, which interrupts the flow of the main program loop every 1ms.

Apart from updating colour readings every 33ms, this interrupt increments a millisecond counter; this counter is used throughout the firmware to time events and add delays.

Step 2b. INT1 Pin Change Interrupt

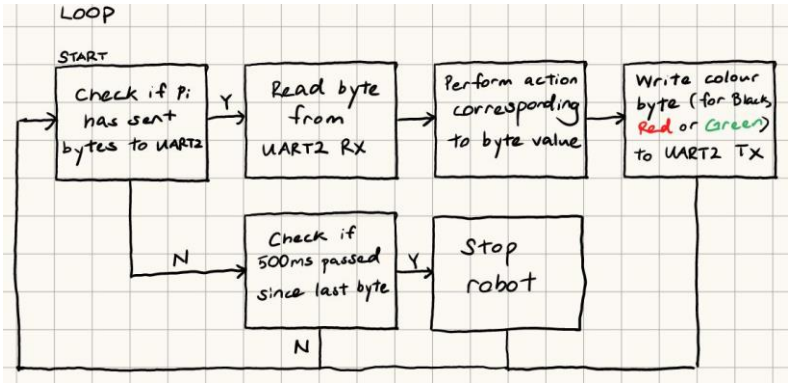



The diagram above shows the flow of the pin change interrupt, which interrupts the flow of the main program loop upon any falling or rising edge on the INT1 pin.

Measuring the width (duration) of the latest pulse on INT1 requires keeping track of the previous edge's time and current time; since TCNT2 ticks every 4us and we have a millisecond counter, we use the formula below to calculate current time (in us):

*(millisecond counter) * 1000 + TCNT2 * 4*

Step 2c. Main Loop




The diagram above shows our main program loop.

In our communication model, the Pi periodically polls the Mega by sending a 4-bit movement command. After performing the action for the received command, the Mega sends a 2-bit number in response, which represents the classified colour read by the colour sensor. The formats for movement commands and colour responses can be found in the table below.

This model ensures that the Pi's serial buffer will not overflow, as it will not receive incoming data at a rate it cannot handle. When not polled quickly enough, some data may be lost, but this is not important for our mission.

Most of Alex's actions involve setting the relevant PWM outputs, but setting motor direction involves sending serial data to the motor driver shield's shift register; this is handled by the AFMotor library. More details on control over movement duration are in Section 6.

For colour classification, Black/Unknown is returned when both red and green readings are lower in frequency (intensity) than a preset threshold. Otherwise, the colour with the higher intensity (Red or Green) is returned instead. This logic is simplistic but robust enough to distinguish between the 2 astronauts.

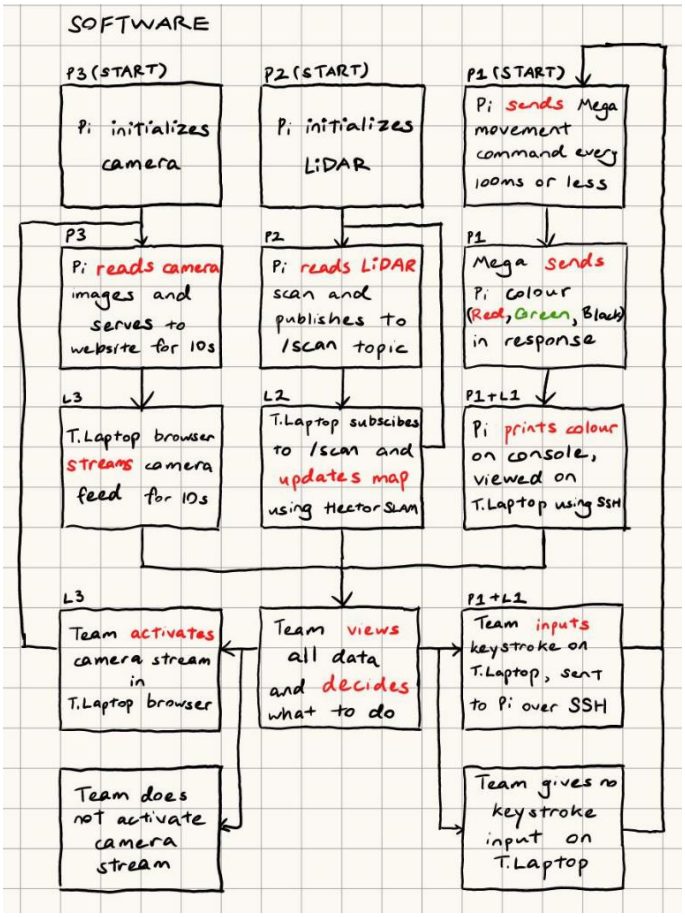
Finally, as a safety feature, the command byte also acts as the Pi's heartbeat; if more than 500ms passes with no byte received, it is taken that the Pi's control program is not running, and the robot stops.

| Command Received | Action |
|---|---|
| 0 | Hold Medpack |
| 1 | Move Forward (100ms) |
| 2 | Stop |
| 3 | Turn Left (100ms) |
| 4 | Turn Right (100ms) |
| 5 | Move Backward (100ms) |
| 6 | Open Claw |
| 7 | Close Claw |
| 8 | Deposit Medpack |
| 9 | Turn Left (25ms) |
| 10 | Turn Right (25ms) |
| 11 | Move Forward (25ms) |
| 12 | Move Backward (25ms) |

| Response Sent | Colour |
|---|---|
| 0 | Red |
| 1 | Green |
| 2 | Black/Unknown |

## Section 6 Software Design

While the firmware on the Mega handles the low-level actuator control and colour sensor processing, we need higher-level software to handle the reading and processing of more complex data (LiDAR scans, camera images) and wirelessly control the robot.




The software diagram above shows the high-level flow of events when going through the mission, but it consists of several programs running concurrently. The program(s) related to each event are labelled at the upper left of each block.

There are 3 groups of programs, each presenting one form of useful data to the team. The team views all available data and chooses what to do next.

However, the programs themselves run on different platforms (Pi or team laptop), depending on the nature of the load.

The onboard Raspberry Pi 4 has direct access to the Mega, RPLidar A1 and Camera Module 3, making it suitable for data collection. However, it is computationally constrained, making it less suited for running onboard SLAM algorithms and generating complex graphics.

Therefore, our software design was largely shaped by these considerations; we used the ROS Noetic framework to offload the SLAM processing and map visualisation to the team laptop. We also used RPi OS Lite (which omits a graphical user interface) to minimize the Pi's load.

Next, we will describe the programs individually, adding relevant technical details.

**Section 6.1 Raspberry Pi**

P1. Control (Teleoperation and Colour Output)

A C++ program (source code in Appendix B) which handles both keystroke input for robot control (teleoperation) and colour output.

After initializing Pi GPIO UART (8N1) at 115200 baud (using the WiringPi library), the program starts a loop where it waits for keystroke input from the user (using the ncurses library); this function times out after 100ms.

After receiving a keystroke (or timing out), the program maps the keystroke to a given action following the table below and sends the corresponding command byte to the Mega. For ease of use, our mapping uses the WASD and IJKL key clusters to control movement, mimicking keyboard controls popular in video games.

This protocol means that tapping a movement key once will send a "move" command to the Mega, followed by a "stop" command 100ms later, thus achieving robot movement for 100ms. Finer movement control (e.g. move for 25ms) is achieved on the Mega by stopping the robot early, before the next command is received.

Lastly, the program reads incoming response bytes from the Mega (format in Section 5) and prints the decoded colour to the console before restarting the loop.


*Screenshot of Control Program Interface*

| Category | Action | Key Pressed | Command Sent |
|---|---|---|---|
| Movement | Stop | None | 2 |
| Coarse Movement | Move Forward (100ms) | W | 1 |
| | Turn Left (100ms) | A | 3 |
| | Move Backward (100ms) | S | 5 |
| | Turn Right (100ms) | D | 4 |
| Fine Movement | Move Forward (25ms) | I | 11 |
| | Turn Left (25ms) | J | 9 |
| | Move Backward (25ms) | K | 12 |
| | Turn Right (25ms) | L | 10 |
| Medpack Control | Hold Medpack | Z | 0 |
| | Deposit Medpack | C | 8 |
| Claw Control | Open Claw | Q | 6 |
| | Close Claw | E | 7 |

P2. RPLidar ROS Node

Started using the command `rosrun rplidar_ros rplidarNode`. Handles the initialization and reading of scan data from the LiDAR over USB, before publishing the scan data to the `/scan` ROS topic.

```
pi@cg2111a:~ $ rosrun rplidar_ros rplidarNode
[ INFO] [1745273715.242800294]: RPLIDAR running on ROS package rplidar_ros, SDK
Version:2.1.0
[ INFO] [1745273715.267920171]: RPLIDAR MODE:A1M8
[ INFO] [1745273715.268010578]: RPLIDAR S/N: DD97ED95C4E493CAA5E69EF04C474B6E
[ INFO] [1745273715.268070059]: Firmware Ver: 1.29
[ INFO] [1745273715.268115319]: Hardware Rev: 7
[ INFO] [1745273715.269582575]: RPLidar health status : OK.
[ INFO] [1745273715.513006650]: current scan mode: Sensitivity, sample rate: 8 K
hz, max_distance: 12.0 m, scan frequency:10.0 Hz,
```

*Program Output*

To achieve this, we needed ROS Noetic and the rplidar_ros package installed on the Pi, which was non-trivial. ROS Noetic is best supported (easiest to install) on Ubuntu 20.04 or Debian Buster, both of which could be loaded on the Pi 4. However, both had problems accessing the Pi's GPIO or the Camera Module 3.

Hence, we had to build a barebones ROS Noetic installation and rplidar_ros package from source on the latest but unsupported RPi OS (Debian Bookworm); this involved downloading dependencies from online package servers, as well as fixing several compilation errors.

P3. Camera Stream Server

Provided Python 3 program which starts a camera stream web server at *http://<Pi's IP Address>:8000*, where a contour-only camera feed from Alex's Camera Module 3 can be viewed for 10s when activated.

P4. IP Address Output (not directly used during mission)

Using a cronjob, the Pi automatically runs a Python3 script (Appendix C) on boot. This script initializes the I2C bus and SSD1315 display using the `adafruit-circuitpython-ssd1306` library.

Afterwards, it runs a loop, where it collects relevant system information (most importantly, IP address) and draws it on the display every second.

**Section 6.2 Team Laptop**

L1. SSH Remote Session into Pi

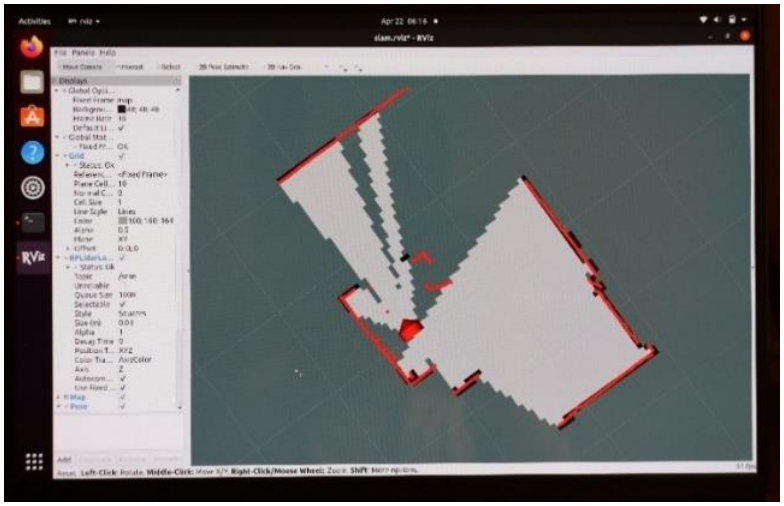
Started using the command `ssh pi@<Pi's IP Address>`, which opens an SSH remote session into the Pi from which P1 is started. This enables us to wirelessly (and securely) send keystrokes to and view the output of P1 from the team laptop.

L2. HectorSLAM Mapping + RViz Visualisation

Started using the command `roslaunch rplidar_ros view_slam.launch`. This launches a ROS node which subscribes to the /slam topic and produces a 2D map of the environment using the HectorSLAM algorithm.

The command also creates an RViz window which visualizes the map. We configured the map to depict the robot's position and heading with an arrow, sized to represent the robot chassis accurately.

To achieve this, we needed ROS Noetic and the hector_slam package installed on the team laptop, which was easier to accomplish than for the Pi; we installed Ubuntu 20.04 which supports (pain-free) ROS Noetic installation.



*RViz Map Visualisation with Robot Pose*

L3. Web Browser

Browser used to access the camera stream website served by the Pi over the network.

L4. roscore (not directly used during mission)

Started using the command roscore on the ROS Master (team laptop). Required for ROS Noetic nodes to communicate across the network.

L5. More SSH Remote Sessions into Pi (not directly used during mission)

Used to initialize the Pi's programs (P2, P3) remotely during setup and set hostnames.

## Section 7 Lessons Learnt & Conclusion

One important lesson we learned was to spend more time testing the robot under real-world conditions. Due to insufficient testing, we were only able to pinpoint certain issues with the robot (claw problems, lack of torque with 2 motors when holding astronaut) after our trial run. If we had tested under realistic conditions more often, these issues might have been prevented beforehand.

Another important lesson we learned was the utility you can get from splitting compute tasks between hardware. In CG1111A, we used only one microcontroller and program to achieve everything. But for this project, we had multiple computers each running programs to handle functions well-suited for them, communicating together to achieve a more complex task.

However, we also learned about the considerations that come into play when compute is decentralized – and the need to check for data corruption and network instability.

As for mistakes, we made plenty; once, we mistakenly assumed that the test movement code we had uploaded to the Mega was not functional, when it was. As a result, our robot drove off our table and crashed to the ground, requiring hours of repair and reassembly.

Apart from being careless with the robot, another mistake we made was leaving much of the work to be done the night before our trial run. Both teleoperation and the rplidar_ros module were broken, so we almost failed to use ROS and Hector SLAM as intended. Although we got it working at 3am (with a long source compilation along the way), it meant we only had 2 hours to test the mapping and practice operation before our trial run.

However, we are proud of our work and believe that we have developed interesting hardware and software features proven to deliver performance in this search and rescue mission.

## References

Artit, N., Jirakarn, S., Peeyaphoom, T., Jiraphan, I., Noppadol, P., & Aran, B. (2023). *2023 – TDP – iRAP ROBOT – RoboCupRescue – Robot*. TDP – Robocup TDP System. https://tdp.robocup.org/tdp/2023-tdp-irap-robot-robocuprescue-robot/

Kotaro, K., Hayato, M., Sota, S., & Noritaka, S. (2023). *2023 – TDP – Nitro – RoboCupRescue – Robot*. TDP – Robocup TDP System. https://tdp.robocup.org/tdp/2023-tdp-nitro-robocuprescue-robot/

# Appendix A – Arduino Mega Firmware (C++)

```
#include <avr/io.h>
#include <avr/interrupt.h>
#include <AFMotor.h>

#define S0_PORT PORTG
#define S0_DDR DDRG
#define S0_BIT 0

#define S1_PORT PORTL
#define S1_DDR DDRL
#define S1_BIT 6

#define S2_PORT PORTC
#define S2_DDR DDRC
#define S2_BIT 0

#define S3_PORT PORTG
#define S3_DDR DDRG
#define S3_BIT 2

#define SENSOR_PORT PORTD
#define SENSOR_DDR DDRD
#define SENSOR_PIN PIND
#define SENSOR_BIT 1

AF_DCMotor br(1), bl(2), fr(4), fl(3);

volatile unsigned int green, red, edges, lastEdge, ms, pulse;

ISR(INT1_vect) {
  unsigned int now = ms * 1000 + TCNT2 * 4;
  long delta = (long)now - (long)lastEdge;
  pulse = (delta < 0) ? delta + 0xFFFF : delta;
  lastEdge = now;
  edges++;
}

ISR(TIMER2_COMPA_vect) {
  ms++;

  //runs every 33ms
  if (!(ms % 33)) {
    //update reading, alternate colour filter
    if ((ms / 33) % 2) {
      green = (edges >= 3) ? pulse : 65535;
      setColourFilter(0, 0);
    } else {
      red = (edges >= 3) ? pulse : 65535;
      setColourFilter(1, 1);
    }
    edges = 0;
  }
}

void setupMotors() {
  //configure motor PWM pins as outputs
  DDRB |= _BV(PB5);             // D11
  DDRE |= _BV(PE5) | _BV(PE3);  // D5, D3
  DDRH |= _BV(PH3);             // D6
```

```
  //enable phase-correct PWM output for motor PWM pins (TOP = 0xFF)
  TCCR1A = (1 << COM1A1) | (0 << COM1A0) | (1 << WGM10);
  TCCR1B = (0 << WGM13) | (0 << WGM12) | (0 << CS12) | (1 << CS11) | (1 << CS10);
  TCCR3A = (1 << COM3A1) | (1 << COM3C1) | (1 << WGM30);
  TCCR3B = (0 << WGM33) | (0 << WGM32) | (0 << CS32) | (1 << CS31) | (1 << CS30);
  TCCR4A = (1 << COM4A1) | (1 << WGM40);
  TCCR4B = (0 << WGM43) | (0 << WGM42) | (0 << CS42) | (1 << CS41) | (1 << CS40);
}

void setupServos() {
  //configure servo PWM pins as outputs
  DDRL |= _BV(PL5) | _BV(PL4) | _BV(PL3);  //D44, D45, D46

  //enable phasecorrect PWM output for servo PWM pins
  //(TOP=20000, pre-scaled clk=2MHz, meaning 50Hz output)
  TCCR5A = (1 << COM5A1) | (1 << COM5B1) | (1 << COM5C1) | (1 << WGM11);
  TCCR5B = (1 << WGM13) | (1 << CS11);
  ICR5 = 20000;

  //start servos in neutral position (1500us pulse)
  OCR5A = 1500;
  OCR5B = 1500;
  OCR5C = 1500;
}

void setupColour() {
  //set colour sensor setting pins as outputs
  S0_DDR |= _BV(S0_BIT);
  S1_DDR |= _BV(S1_BIT);
  S2_DDR |= _BV(S2_BIT);
  S3_DDR |= _BV(S3_BIT);

  //set colour sensor data pin as input
  SENSOR_DDR &= ~_BV(SENSOR_BIT);

  //set colour sensor frequency scaling to 20%
  S0_PORT |= _BV(S0_BIT);
  S1_PORT &= ~_BV(S1_BIT);

  //enable INT1 interrupt for rising and falling edges
  EICRA |= _BV(ISC10);
  EIMSK |= _BV(INT1);

  //enable Timer2 Overflow Interrupt (fires every 1ms)
  TCCR2A = (1 << WGM21);
  TCCR2B |= (1 << CS22);
  TIMSK2 = (1 << OCIE2A);
  OCR2A = 249;
  TCNT2 = 0;

  setColourFilter(0, 0);
}

void setupUART2(uint32_t baud) {
  uint16_t ubrr = (F_CPU / 4 / baud - 1) / 2;
  UCSR2A = _BV(U2X2);                // doublespeed
  UCSR2B = _BV(RXEN2) | _BV(TXEN2);    // enable RX & TX
  UCSR2C = _BV(UCSZ21) | _BV(UCSZ20);  // 8N1
  UBRR2H = ubrr >> 8;
  UBRR2L = ubrr;
}
```

```
uint8_t UART2Available() {
  return (UCSR2A & _BV(RXC2));
}

uint8_t UART2BlockingRead() {
  while (!(UCSR2A & _BV(RXC2))) {}
  return UDR2;
}

void UART2BlockingWrite(uint8_t c) {
  while (!(UCSR2A & _BV(UDRE2))) {}
  UDR2 = c;
}

void delayMs(int delay_ms) {
  unsigned int last_time = ms;
  long delta = 0;

  while (delta < delay_ms) {
    delta = (long)ms - (long)last_time;
    delta = (delta < 0) ? (delta + 0xFFFF) : delta;
  }
}

void setColourFilter(bool s2, bool s3) {
  if (s2)
    S2_PORT |= (1 << S2_BIT);
  else
    S2_PORT &= ~(1 << S2_BIT);

  if (s3)
    S3_PORT |= (1 << S3_BIT);
  else
    S3_PORT &= ~(1 << S3_BIT);
}

uint8_t getColourStatus() {
  const unsigned int threshold = 6000;
  if ((green < threshold || red < threshold) && green < red)
    return 0b01;  //green
  else if ((green < threshold || red < threshold) && red < green)
    return 0b00;  //red
  else
    return 0b10;  //black/unknown
}

void translate(bool dir, uint8_t speed) {
  //dir true = forward, false = backward
  br.run(dir ? FORWARD : BACKWARD);
  bl.run(dir ? BACKWARD : FORWARD);
  fr.run(dir ? FORWARD : BACKWARD);
  fl.run(dir ? BACKWARD : FORWARD);
  OCR1A = speed;
  OCR3C = speed;
  OCR3A = speed;
  OCR4A = speed;
}

void rotate(bool dir, uint8_t speed) {
  //dir true = left, false = right
```

```
  br.run(dir ? FORWARD : BACKWARD);
  bl.run(dir ? FORWARD : BACKWARD);
  fr.run(dir ? FORWARD : BACKWARD);
  fl.run(dir ? FORWARD : BACKWARD);
  OCR1A = speed;
  OCR3C = speed;
  OCR3A = speed;
  OCR4A = speed;
}

void stop() {
  OCR1A = 0;
  OCR3C = 0;
  OCR3A = 0;
  OCR4A = 0;
}

void openClaw() {
  OCR5B = 1800;
  OCR5C = 1300;
}

void closeClaw() {
  OCR5B = 1100;
  OCR5C = 2000;
}

void holdMedpack() { OCR5A = 1050; }
void depositMedpack() { OCR5A = 1650; }

void setup() {
  cli();
  setupMotors();
  setupServos();
  setupColour();
  setupUART2(115200);
  sei();

  //start robot with claw open and holding medpack
  openClaw();
  holdMedpack();
}

void loop() {
  int last_time = ms;

  //wait for new byte from Pi
  //stop if no byte for >500ms
  while (!UART2Available()) {
    long delta = (long)ms - (long)last_time;
    delta = (delta < 0) ? (delta + 0xFFFF) : delta;

    if (delta > 500)
      stop();
  };

  //perform action based on received byte
  uint8_t cmd = UART2BlockingRead();
  switch (cmd) {
    case 0: holdMedpack(); break;
    case 1: translate(true, 255); break;
```

```
    case 2: stop(); break;
    case 3: rotate(true, 255); break;
    case 4: rotate(false, 255); break;
    case 5: translate(false, 255); break;
    case 6: openClaw(); break;
    case 7: closeClaw(); break;
    case 8: depositMedpack(); break;

    case 9:
      rotate(true, 255);
      delayMs(25);
      stop();
      break;
    case 10:
      rotate(false, 255);
      delayMs(25);
      stop();
      break;
    case 11:
      translate(true, 255);
      delayMs(25);
      stop();
      break;
    case 12:
      translate(false, 255);
      delayMs(25);
      stop();
      break;

    default: break;
  }

  //send classified colour in response
  uint8_t col = getColourStatus();
  UART2BlockingWrite(col);
}
```

# Appendix B – Control Program (C++)

```
#include <ncurses.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <wiringPi.h>
#include <wiringSerial.h>

int main(int argc, char *argv[])
{
   int ch = 2, c;
   initscr();
   cbreak();
   noecho();

   int fd;
   if ((fd = serialOpen("/dev/ttyS0", 115200)) < 0)
   {
      fprintf(stderr, "Unable to open serial device: %s\n", strerror(errno));
      return 1;
   }

   if (wiringPiSetup() == -1)
   {
      fprintf(stdout, "Unable to start wiringPi: %s\n", strerror (errno)) ;
      return 1;
   }

   timeout(100);
   while (true)
   {
      char c = getch();

      switch (c)
      {
         case 'w': ch = 1; break;          //forward (100ms)
         case 'a': ch = 3; break;          //left (100ms)
         case 's': ch = 5; break;          //back (100ms)
         case 'd': ch = 4; break;          //right (100ms)

         case 'q': ch = 6; break;          //open claw
         case 'e': ch = 7; break;          //close claw
         case 'z': ch = 0; break;          //hold medpack
         case 'c': ch = 8; break;          //deposit medpack

         case 'j': ch = 9; break;          //left (25ms)
         case 'l': ch = 10; break;         //right (25ms)
         case 'i': ch = 11; break;         //forward (25ms)
         case 'k': ch = 12; break;         //back (25ms)

         case '=':                         //quits program
            endwin();
            return 0;
```

```
            break;

        default: ch = 2; break;                    //stop
      }

     serialPutchar(fd,(uint8_t)ch);

      while (serialDataAvail(fd))
      {
        char received = serialGetchar(fd);

        if (received == 0)
          printw("Red  \r");
        else if (received == 1)
          printw("Green\r");
        else
          printw("Black\r");
      }
   }

   endwin();
   return 0;
}
```

# Appendix C – OLED Program (Python 3)

```
import math
import time
import subprocess
from PIL import Image, ImageDraw, ImageFont
from board import SCL, SDA
import busio
import adafruit_ssd1306

i2c = busio.I2C(SCL, SDA)
disp = adafruit_ssd1306.SSD1306_I2C(128, 64, i2c)
disp.fill(0)
disp.show()

width = disp.width
height = disp.height
image = Image.new("1", (width, height))
draw = ImageDraw.Draw(image)
draw.rectangle((0, 0, width, height), outline=0, fill=0)
font = ImageFont.load_default()

padding = -2
top = padding
bottom = height - padding
x = 0

while True:
    draw.rectangle((0, 0, width, height), outline=0, fill=0)

    cmd = "hostname -I | cut -d' ' -f1"
    line1 = subprocess.check_output(cmd, shell=True).decode("utf-8")

    cmd = "top -bn1 | grep load | awk '{printf \"Load: %.2f\", $(NF-2)}'"
    line2 = subprocess.check_output(cmd, shell=True).decode("utf-8")

    cmd = "free -m | awk 'NR==2{printf \"Memory: %.0f%%\", $3*100/$2}'"
    line3 = subprocess.check_output(cmd, shell=True).decode("utf-8")

    cmd = 'df -h | awk \'$NF=="/"{printf "Disk: %s", $5}\''
    line4 = subprocess.check_output(cmd, shell=True).decode("utf-8")

    draw.text((x, top + 0),  line1, font=font, fill=255)
    draw.text((x, top + 8),  line2, font=font, fill=255)
    draw.text((x, top + 16), line3, font=font, fill=255)
    draw.text((x, top + 25), line4, font=font, fill=255)

    disp.image(image)
    disp.show()
    time.sleep(1)
```