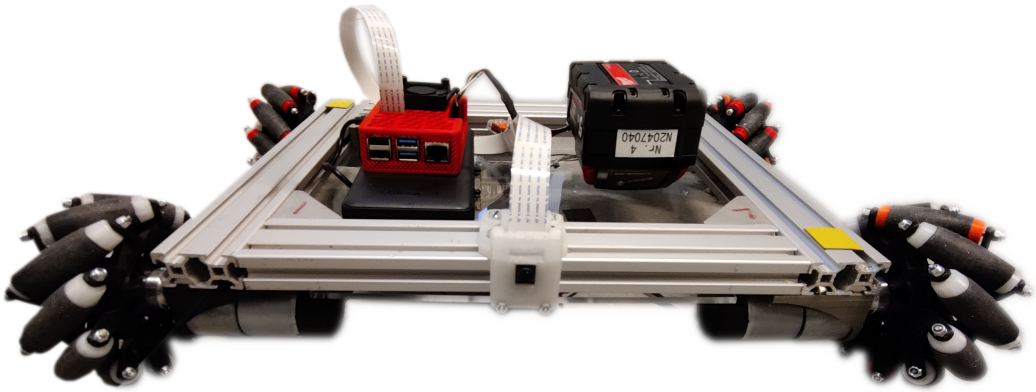
Mecanum Robot

Real-time Computer Technology



Technical report
Aalesund, autumn 2020

NTNU

CANDIDATES (last name, first name):
Osnes, Peter H.
Aakervik, Ruben
Storvik, Kevin M.

DATE:	SUB.CODE:	GROUP (name/num):	PAGES/APP.DIX:
27.11.20	IE303812	Mecanum Robot / Group 1	61 / 103

LECTURER:
Ivar Blindheim

TITLE:
Mecanum Robot

SUMMARY:
This report is about the techniques and advancements we made to achieve multi-threading within the programming language, Python. Through the past years with this course, Java has been utilised to practice this concept. However, with Python becoming more and more attractive on the job market, we decided that we wanted to achieve this in Python. Therefore, it was agreed that this project would be executed in Python. This has been a lesson for us and granted us the knowledge we will bring to others when in need of threading in Python without having to learn a new language.

Contents

1 Terminology

The first part of this rapport will cover brief descriptions of terminologies that are used throughout the project. These are meant for the reader to have at least a basic understanding of what is to be covered, even if they are new to the concept

1.1 Glossary

Visual Studio Code

Visual Studio Code is a streamlined code editor with support for development operations like debugging, task running, and version control.

Python

"Python is a programming language that lets you work quickly and integrate systems more effectively." [Wikd]
Python as a language aims to help programmers write logical code that's easy to read for small and large-scale projects. It is an interpreted, high-level and general-purpose programming language, that includes many types of programming paradigms such as procedural, object-oriented and functional programming.
Python has a massive standard library which makes it very easy to start off, and support for extra modules is extensive.
[Wiki]

Java

"Java is a class-based, object-oriented programming language that is designed to have as few implementation dependencies as possible." [Wikd]
Java is a general-purpose programming language intended to let application developers *write once, run anywhere*, meaning that compiled Java code can run on all platforms that support Java without the need for recompilation. Its syntax is very similar to C and C++ and is often used for client-server web applications.

Global Interpreter Lock GIL in Python works as a lock, in that it allows only a single thread to hold control of the interpreter at a time. This means that only one thread can be in execution state at any point in time. CPU-bound and multi-threaded code can suffer performance bottleneck because of this. [Wikc]

Context switch Context switch, within computing, refers to a process where we store the state of a process or thread and resume execution at a later point. Allowing multiple processes to share a single Central Processing Unit (CPU). This is essential in systems that use multitasking operations. [Wikb]

Mecanum Wheels Mecanum wheels refers to a set of omni-directional wheels that is designed to allow movement in any direction. Rubberized rollers are attached along the entirety of the rim at an angle. Each wheel is independent from each other which is what permits the omni-directional movement. This gives the mecanum wheels the unique ability to move with minimal need for space, in any direction.

1.2 Acronyms

VSCode	Visual Studio Code is a streamlined code editor with support for development operations like debugging, task running, and version control.
GIL	Global Interpreter Lock, synchronization mechanism that allows only a single thread to execute at a time.
RPI	Raspberry Pi.
GUI	Graphical User Interface.
TCP	Transport Control Protocol.
UDP	User Datagram Protocol.
PLA	PolyLactic Acid, a hard and brittle 3D-printing filament.
TPU	Thermoplastic PolyUrethane, a flexible rubber-like 3D-printing filament.
FPS	Frames per second.
GPU	Graphics processing unit.
CPU	Central Processing Unit.
I/O	Input/Output.
IDE	Integrated Development Environment.
RGB	Red, green and blue, referring to the primary colours.

2 Introduction

As an introduction, we will cover a few topics about the background, problem, primary goal, limitations and finally, the approach to this project.

2.1 Background

Background of this course is to achieve real-time programming within Java. Java has long been known for its wide-spread and well-documented library and is a language still used to modernize and future-proof everything around the globe to date. The concept of threading is to make it possible to do more than one thing at a time. With this, you can start fully utilize the processing power. This includes reaching time-breaking records when it comes to calculations and make scripts that would take hours if not threaded.

2.2 Problem formulation

Java has long been known for being a standard programming language that many will come across. Being one of the most popular programming languages in use according to GitHub, it got many competitors such as C, C++ and Python. In recent years, Python seen an increase in popularity and demand on the job market, due to being relatively easy to pick up and learn. We then set out to learn Python and research whether Python could compete against Java when it comes to threading.

2.3 Objectives

The main objective is to achieve real-time threading using Python. This is to challenge ourselves to both learn the language and how threading works.

2.4 Limitations

The most significant limitations we have met so far is how Python executes code. The GIL is the main restriction we've met upon when it comes to real-time threading.

2.5 Approach

To approach this, we started researching how Python behaves, threading concepts, and compare its Threading with Java.

3 Theoretical basis

In this section, we are going to talk about the theoretical basis of our project, divided into subsections as seen below. We will cover the programming languages and software used, and how our mecanum wheels work.

3.1 Programming language

Python is an interpreted, object-oriented, high-level programming language. Its built in data structures, combined with dynamic typing and dynamic binding, make it very useful for Rapid Application Development, as well as for use as a scripting or glue language to connect existing components together. Python also supports modules and packages.

3.1.1 Algorithm

An algorithm is a procedure or formula for solving a problem, based on conducting a sequence of specified actions. A computer program can be viewed as an elaborate algorithm. In mathematics and computer science, an algorithm usually means a small procedure that solves a recurrent problem.

3.2 Image Processing

Image processing is a way to digitally alter images through algorithms. This is usually done through some kind of software library, to simplify the use in a project. It can be used for many of purposes, like object detection, facial feature detection, color manipulation, and much more. For use in this project we have opted for OpenCV.

3.2.1 OpenCV

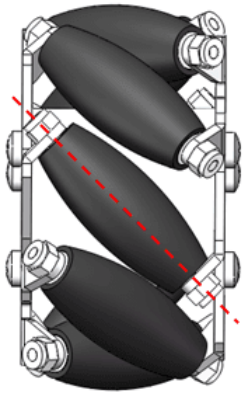
"OpenCV (Open Source Computer Vision Library) is an open source computer vision and machine learning software library. OpenCV was built to provide a common infrastructure for computer vision applications and to accelerate the use of machine perception in the commercial products. Being a BSD-licensed product, OpenCV makes it easy for businesses to utilize and modify the code." [Ope]

3.3 Mecanum-Robot

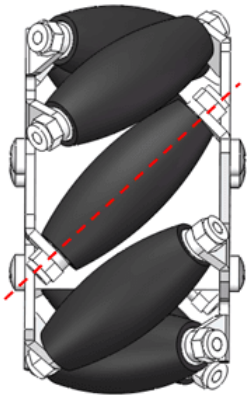
3.3.1 Base

The base of the robot was received from a previous project. With similar project design, they achieved multi-threading through Java and passed the course. The base consisted of a square frame built with aluminium profiles, with a clear acrylic sheet used as a bottom. We then decided to use this base as the main structure to attach the wheels, battery and the RPI.

3.3.2 Wheels



Left mecanum wheel rollers rotated clockwise



Right mecanum wheel rollers rotated counter-clockwise

The project is most visually attractive due to our precious Mecanum Wheels. Mecanum wheels are a tireless design, where the tire is replaced with rollers placed on the circumference of the wheel. The rollers are not placed perpendicular to the axle, but at a 45-degree angle, on one pair 45 degrees clockwise and the other 45 degrees counter-clockwise, namely left and right mecanum wheels. This wheel-design is designed to make the base move in 8 different directions. The design of these wheels was found on the web and 3D-printed using the various 3D-printers available at NTNU.

3.3.3 Motor setup

For the robot we used two of the BasicMicro RoboClaw 2x15A, a dual-channel brushed DC-motor controller, to control our four Pololu 131:1 brushed DC motors. The motor controllers are managed by multiple different methods, notably USB, RC radio systems, analogue voltages and serial signals, both simple serial and a packet serial mode.

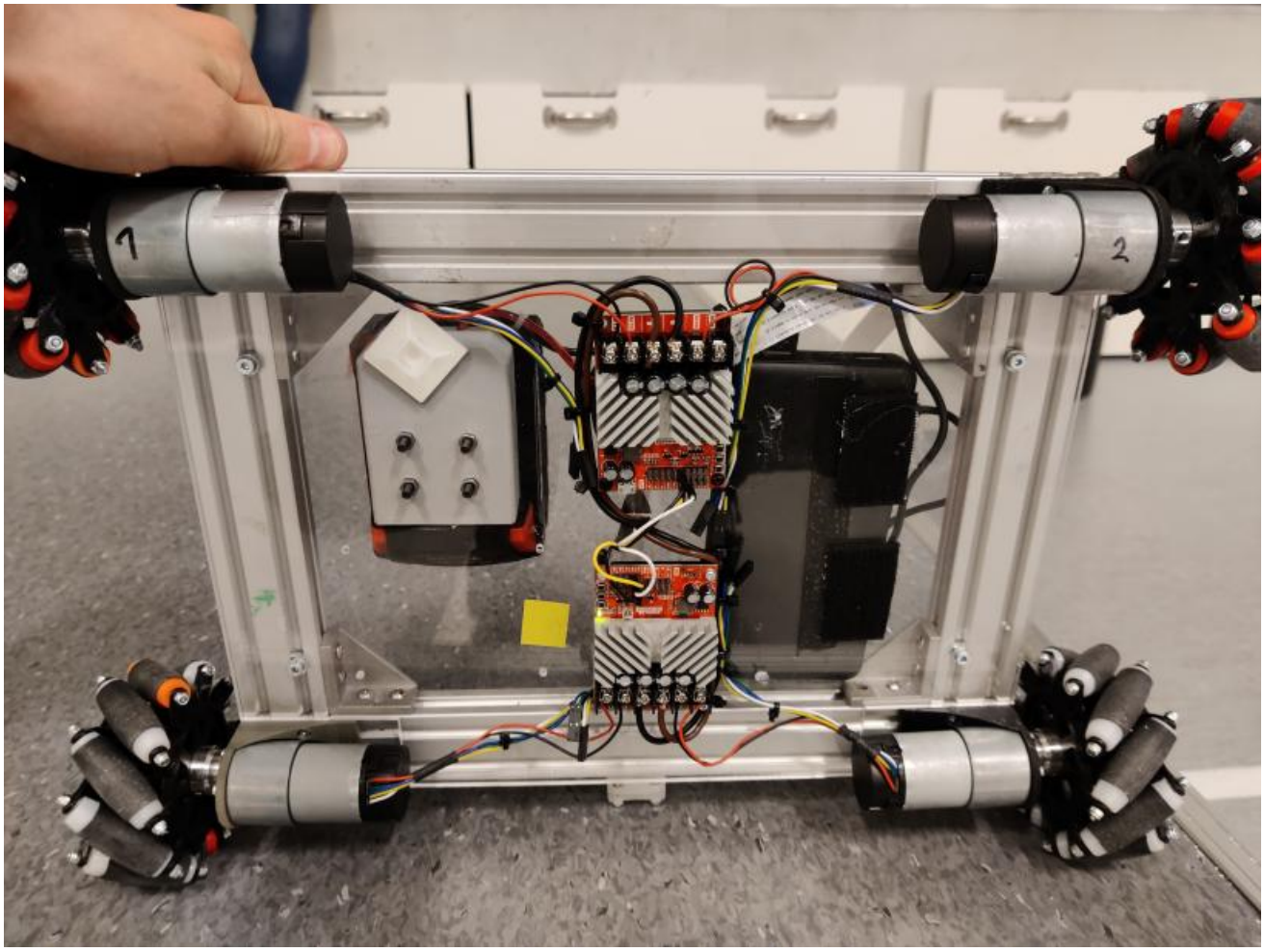


Figure 1: Components mounted to the robot: motors in the four corners, battery center-left, both motor controllers center and RPI and its battery pack to the right

3.3.4 Mecanum Movement

The movement-mechanics of the Mecanum wheels is very unlike that of normal wheels. Because of the way the rollers are mounted to the wheel, the force the wheels exert on the ground is diagonal. By combining the diagonal forces exerted by the four wheels in specific patterns, you achieve the ability to drive the robot in eight directions, that is right/left, forwards/backwards and diagonal on a 2D-plane, as well as rotating around the centre of one axle, rotating around one of the wheels and rotating around the centre of the robot.

Wheel radius	= R
Angular velocity of robot	= ω_0
Angular velocity of wheels	= $\omega_1 \ldots \omega_4$
Speed of robot in x and y-direction	= V_x, V_y
Width from center of robot to center of wheel	= L1
Length from center of robot to center of wheel	= L2
Angle of rollers	= $\alpha = 45$
Speed of rollers on the wheels	= $V_{g1 \ldots 4}$
Global coordinates at center of platform	= O
Local coordinates at center of wheel	= $O_{1 \ldots 4}$

Table 1: Mecanum equation table

3.3.4.1 Movement equation

In the global coordinate, the speed at the center of wheel 1 (front left) is:

$$V_{o1x} = V_x - \omega_0 \cdot L_1 \tag{1}$$

$$V_{o1y} = V_y - \omega_0 \cdot L_2 \tag{2}$$

In the local coordinate of wheel 1, the speed is:

$$V_{o1x} = -V_{g1} \cdot cos\alpha \cdot +\omega_1 \cdot R \tag{3}$$

$$V_{o1x} = V_{g1} \cdot sin\alpha \tag{4}$$

If we combine the previous equations, we get:

$$V_x - \omega_0 \cdot L_1 = -V_{g1} \cdot cos\alpha + \omega_1 \cdot R \tag{5}$$

$$V_y - \omega_0 \cdot L_2 = V_{g1} \cdot sin\alpha \tag{6}$$

From this we can rearrange the equation to find ω:

$$\omega_1 = \frac{1}{R}\left[1\frac{1}{\tan\alpha} - (L_1 + \frac{L_2}{\tan\alpha})\right]\begin{bmatrix} V_x \\ V_y \\ \omega_0 \end{bmatrix} \tag{7}$$

Which leads us to this final equation for calculating the velocity of all 4 wheels:

$$\begin{bmatrix} \omega_1 \\ \omega_2 \\ \omega_3 \\ \omega_4 \end{bmatrix} = \frac{1}{R}\begin{bmatrix} 1 & 1 & -(L_1 + L_2) \\ 1 & -1 & (L_1 + L_2) \\ 1 & -1 & -(L_1 + L_2) \\ 1 & 1 & (L_1 + L_2) \end{bmatrix}\begin{bmatrix} V_x \\ V_y \\ \omega_0 \end{bmatrix} \tag{8}$$

[stu]

3.3.5 Camera

The eyes of the robot is the Raspberry Pi Camera Module. We use the camera for both streaming and finding out whenever the robot has encountered an object and will try stop, reverse and turn away from the object depending on where he managed to notice it first.

3.3.6 System

The system can be divided into three parts, main, server and client system. The main system manages the GUI logic, showing and sending commands to the client. The main consists of a class and one main method. This class is the TextInputBox, where the user may send speed inputs to the client. Its only purpose is to get inputs and set the new speed received by the user.

The main method however, controls the entirety of the GUI logic. Here we set the different variables, check for keyboard inputs and what should be done when a certain input is received. This script is run through the main loop, also known as how the Python interpreter reads a source file. This is by defining what the interpreter should do once you attempt to run the script. At the end of our file, we then make an if-statement `if __name__ == '__main__'` The code within this statement will then be executed when a call is made to run your file.

3.4 Communication

3.4.1 TCP

To establish communication between server and client, we use a transport control protocol. This protocol lets us always remain connected and safely transfer packages between our end-points. To keep this communication consistent, we need a reliable WiFi network. We use this to transfer commands from the main, through the server to the client. The client, on the other hand, sends an encoded serialised image over the net which gets decoded at the endpoint and showed to the user.

3.4.1.1 Client

The client will represent the one who connects to the server—often defined as a class which holds methods of handling sending and reading messages from the server. Clients are known as a desktop computer or workstation that is capable of obtaining information and application from a server.

3.4.1.2 Server

The server will represent hardware or software which provides functionality to other programs or devices. In this example, these are clients.

3.4.1.3 Limitations

One of the limitations encountered when using this protocol would be treating the transport control protocol as a message protocol. TCP is a streaming protocol and must be set up correctly as one to work. Ignoring this will result in incomplete 'messages' and could lead to an exception in your code. To counter this, applications are reasonable for checking that all data has been sent. If only some of the data was transmitted, the application needs to attempt the delivery of the remaining data.

To circumvent this restriction; realise the use of sendall() instead of send(). The sendall() method continues to send data from bytes until either all data has been sent or an error occurs. On error, an exception is raised, and there is no way to determine how much data was successfully sent. [Foub]

3.5 Threading

3.5.1 Basics of threading

The simplest explanation of what a thread is would be to call it a separate flow of execution. The program will have two things happening at once. In reality, the threads are not indeed running at the same time though, but rather in parallel on multiple cores or through context switch. This process is happening so fast that for our purposes, they are running at the same time. What is happening with threading is that the operating system knows each thread, and therefore can interrupt it at any time to run another thread in its place.

In Java, we have the option of using both parallel execution and context switch. Parallel execution allows the threads to be run on multiple cores of the CPU concurrently in parallel. On the other hand, context switch uses a process where a task stores its system state so that it can be paused and another task can start. This is technically "fake" concurrency since tasks are not actually happening at the same time, but instead switching between tasks at an extremely rapid pace.

Context Switch is therefore very CPU load heavy since it requires each operation to store it's memory while waiting for other tasks to finish. Python, because of the GIL, can not actually use parallel execution with threading since only one CPU core can run at a time, meaning only one thread can run at a time. Because of this, when multi-threading is involved, Python is limited to only context switch or multiprocessing. [Anda]

3.5.2 Multi-threading and multi-processing

3.5.2.1 Multi-threading

Multi-threading is a CPU's ability to execute multiple threads concurrently, using the shared resources of one or multiple cores. This is achieved using thread-level parallelism. When using multi-threading, the memory heap within the process is shared. This means that any thread is able to modify or access objects in the same memory heap. Since it contains fewer resources and doesn't carry the memory space, a thread is exceptionally lightweight. On the other hand, this makes object management mandatory to keep the shared object consistent under multiple threads. This requires language-specific measures. [Wikg]

3.5.2.2 Multi-processing

Multiprocessing, in our case, refers to systems where multiple processes are running on a separate CPU or core and execute concurrently. This achieves true parallel execution using more than one processor. In multiprocessing, instead of threads, we create new processes. These processes usually carry a set of run-time resources, including its memory heap. This is then copied for each of the generated sub-processes. Therefore the overhead is increased when using multiprocessing. Although this also makes object management unnecessary, since each process only accesses its copy of objects. [Wikf]

3.5.3 Python and Java differences

There are several differences between how Java and Python handle concurrent programming. Java supports the ability to run multi-threading running on multiple cores. This means that Java achieves true parallel execution. It can then support a certain number of threads for a certain number of cores, before having to use context switch, increasing the overhead, and decreasing the performance for CPU intensive operations.

A large number of threads mostly makes sense when the operation is I/O-related, such as servers. Shared object management in Java is handled using support for the fine-grained lock for each resource or object. A keyword like "synchronised", "final", "volatile" are used for controlling the shared objects.

"Synchronised": Threads that need access to a synchronised object have to obtain the lock for the object. Guaranteeing that only one thread can access the object each time, avoiding any data inconsistency.

"Final": Final marked fields are read-only, meaning they can only be initialized in the constructor. This allows threads to freely access or read the field since they can't be changed.

"Volatile": Changes made in volatile fields will be visible for the other threads. Therefore the changes made in local caches are pushed to the main memory, keeping the local caches consistent for the volatile fields.

As mentioned above, Python "fakes" its concurrency when using multi-threading. This is why Java is more commonly used for applications where multi-threading will be used. This does not mean that Python multi-threading can't be used, however. Because of the

GIL though Python is in effect, single-threaded. It can only run on a single CPU core at a time. On the other hand, using GPU with Python is relatively simple compared to Java. If given a powerful GPU, with many cores, Python will probably outperform Java with a powerful CPU, using multiprocessing.

Generally, we want to avoid using multi-threading in Python when doing high computation operations because we are forced to use the context switch. In comparison, it works fine for I/O-related jobs, like in the case of this project. To achieve true parallel execution in Python, we would have to use multiprocessing rather than multi-threading. Operations that are CPU-intensive is better done using multiprocessing in Python. [Xu]

3.5.4 I/O-bound and CPU-bound problems

We've talked some about I/O or CPU related problems and operations going through threading. Let's explain what these actually entail.

3.5.4.1 I/O-bound problems

These generally occur when the program is frequently waiting for input/output from an external source, causing it to slow down. If your program is interacting with something that is way slower than your CPU, i.e. network connections or file system, it can cause the program to spend most of its time waiting.

Speeding up this process involves making it so that the time spent waiting is overlapped with each other. This can be achieved by something like threading (like we do), asyncio version, or synchronous version. Multiprocessing can also be used, but it's not really made for this type of problem, as threading and asyncio will be faster. [Andb]

3.5.4.2 CPU-bound problems

On the other hand, CPU-bound problems do minimal I/O operations, and the time it takes to execute is how fast it can process the required data. How quickly the program then runs is affected by how quickly it can do CPU operations.

To speed up this type of program, we will have to find ways to do more computation at the same time. This is where multiprocessing truly shines. Multiprocessing allows us to take a massive CPU workload problem and let multiple CPUs works on it at the same time. This is the way to achieve true parallel execution in Python. [Andb]

3.6 Modules

Below we will introduce the most important modules that we have used during this project.

3.6.1 Pygame

Pygame is a cross-platform set of Python modules designed for writing video games. It includes computer graphics and sound libraries designed to be used with the Python programming language.[Wikh]

3.6.2 socket

This module provides access to the BSD socket interface. It is available on all modern Unix systems, Windows, MacOS, and probably additional platforms.

The Python interface is a straightforward transliteration of the Unix system call and library interface for sockets to Python's object-oriented style: the socket() function returns a socket object whose methods implement the various socket system calls. Parameter types are somewhat higher-level than in the C interface: as with read() and write() operations on Python files, buffer allocation on receive operations is automatic, and buffer length is implicit on send operations.[Fouc]

3.6.3 cv2

Cv2 module is OpenCV, as previously mentioned on page 10

3.6.4 threading

This module constructs higher-level threading interfaces on top of the lower level `_thread` module. See also the queue module.[Anda]

3.6.5 thread

This module provides low-level primitives for working with multiple threads (also called light-weight processes or tasks) — multiple threads of control sharing their global data space. For synchronization, simple locks (also called mutexes or binary semaphores, similar to Java) are provided. The threading module provides an easier to use and higher-level threading API built on top of this module. [Foud]

3.6.6 queue

The queue module implements multi-producer, multi-consumer queues. It is especially useful in threaded programming when information must be exchanged safely between multiple threads. The Queue class in this module implements all the required locking semantics. This class is similar to Java's Queue module.[Foue]

3.7 Program

3.7.1 Visual Studio Code

Visual studio Code is a streamlined code editor with support for development operations like debugging, task running, and version control. It aims to provide just the tools a developer needs for a quick code-build-debug cycle and leaves more complex workflows to a fuller featured IDEs, such as Visual Studio IDE. [Mic]

3.7.2 GitKraken

GitKraken is a powerful and elegant multi-platform graphical interface for Git developed by Axosoft, as an alternative to the command line. In a very intuitive and delicate way, you can keep track of your repositories, branches, tags, create new and have all the history of your work and commits. This is great for teams working together on a software project. [Axo]

3.7.3 Overleaf

Overleaf is used as the preferred report editor tool following the LaTeX standard of reports and allows for online report creation and collaboration. [Ove].

4 Materials and methods

While the materials are elementary to cover, we need to go into some depth about why we chose to achieve objectives and how we did them. Therefore, this in section, we will peer into our choices, how we accomplished our goals, and explain our methods.

4.1 Material List

Added as an attachment

4.2 Project Organisation

The project organisation were built initially upon each member learning and writing threaded code. The result of this was that each member was assigned a task. A completion date of each task depended on the importance. The result of this was as referenced below;

Kevin	Peter	Ruben
Researching design	Researching GUI	3D-modelling
Researching method of thread-implementation	Testing various GUI libraries	Creating rods to fit wheels
Testing motordrivers	Researching threading for Python	Built and improved wheels
Programming motordrivers	Testing threading in Python	Created better wheel mounts
Making first version of controller design	Implementing threading	Created camera mount
Improving controller design	Bugfixing threading	Mounted components to the platform
Adding TCP communication	Researching automated movement	Coding camera-stream
Fail-proofing communication	Researching image analysis in Python	Coding GUI
Merging threading, communication and controller	Implemented simple object detection	Coding variable speed
Testing threading concepts	Implemented corner detection	Implementing gamepad support
		Researching 360-degree movement for robot

4.3 Communication protocol

The protocol we decided to go for was TCP. Initially, we wanted a solution where we could take advantage of UDP since we continuously send and receive either CMDs or frames. However, the issue with this approach is that each time we receive something through UDP, we have to confirm that the packet formatted as designed. This means that the package has to be encoded in a way which only we will decode it. If some errors or disturbances occur midst transmission, we could end up corrupted and receive a bizarre frame.

The solution to this was to stray away from UDP and instead use TCP. What is so great with TCP is that it needs a stable connection to work. This means that as soon as we get disconnected due to some timeout or network loss if we do not code a way to handle this, an exception will occur. If an exception occurs, usually the program exits, and you have to rerun the code. Using this method, we could always ensure connection if the end-points were within the same network and in range. Doing so made it possible to drive the car around wherever we wanted to go as long as the network-range covered the end-points.

4.4 Network Setup

Usually, we would require a router to create a local WiFi. The issue doing so is that we would be left disconnected to the internet if we connected to this local network. A simple solution to this was that we used one of our phones as a WiFi Hotspot. This works exactly like a router, except the diverse and advanced options you get on a router such as port-forwarding, firewall modification and so on.

A solution like this required one of us to always be there if we were going to use the robot. This is something we wished to solve with being able to have a static IP on the Eduroam network, which is easier said than done.

4.5 Image processing

For this project, image processing is used to process what the camera on the vehicle captures, and use this both for improving the speed of the program, and also create the baseline for autonomous movement.

To start with we use OpenCV to change the feed from coloured to grey-scale. This takes a coloured frame, and a converts it to scales of grey, from the darkest black, 0, to the brightest white, 255. This will achieve two important things for us in this project.

First, it reduces the size of what has to be sent between the server and the client, which will significantly increase the frame rate of the footage received. This is vital since we want the feed from the camera to be as quick as possible. What the user on the server-side will see captured from the camera will therefore be a black-and-white video-feed.

The second purpose of the grey-scaling is for us to achieve object detection. This is done by further processing the frames from the feed to canny edge detection. From the grey-scale image, we perform Gaussian Blur, which will reduce noise and reduce details. Doing this allows a better result for the next step of the processing. Next is what is referred to as Canny Edge detection. This is an algorithm that enables us to detect a wide range of edges in the images.

The Process of Canny edge detection algorithm can be broken down to 5 different steps:

1. Apply Gaussian filter to smooth the image in order to remove the noise
2. Find the intensity gradients of the image
3. Apply non-maximum suppression to get rid of spurious response to edge detection
4. Apply double threshold to determine potential edges
5. Track edge by hysteresis: Finalize the detection of edges by suppressing all the other edges that are weak and not connected to strong edges.[Wika]

The code to achieve this is deceptively simple. OpenCV is used to achieve all the algorithms used here.

The process goes like this:

Figure 2: Original image

This is the image captured by the camera.

Figure 3: Gray-scale

Here we have done the first conversion, grey-scale. The pixels in the image is now only shades of grey on a scale of 0-255 in intensity. This is what the user will see from the video feed.

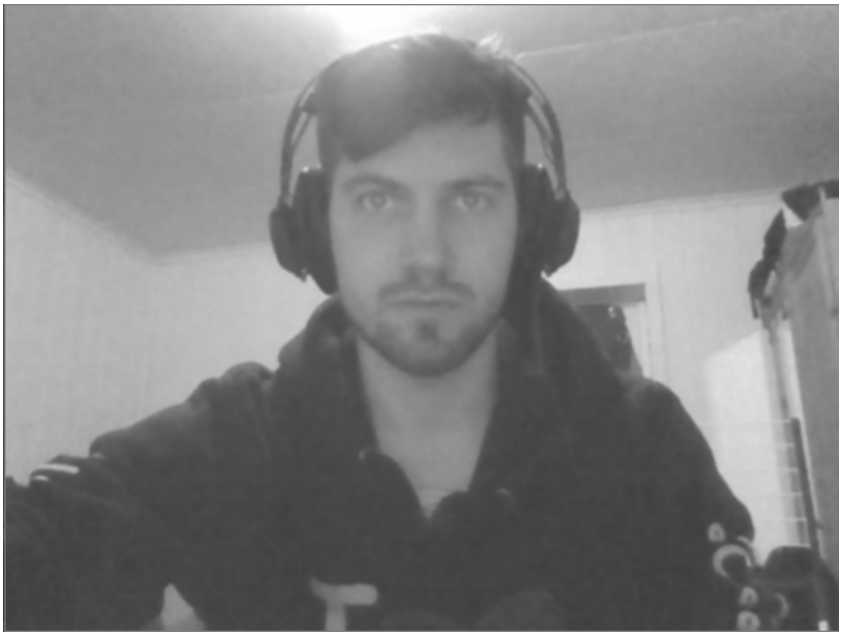


Figure 4: Gaussian Blur

Next we have applied the Gaussian Blur filter. The details of the image has been smoothed over and the noise is reduced.

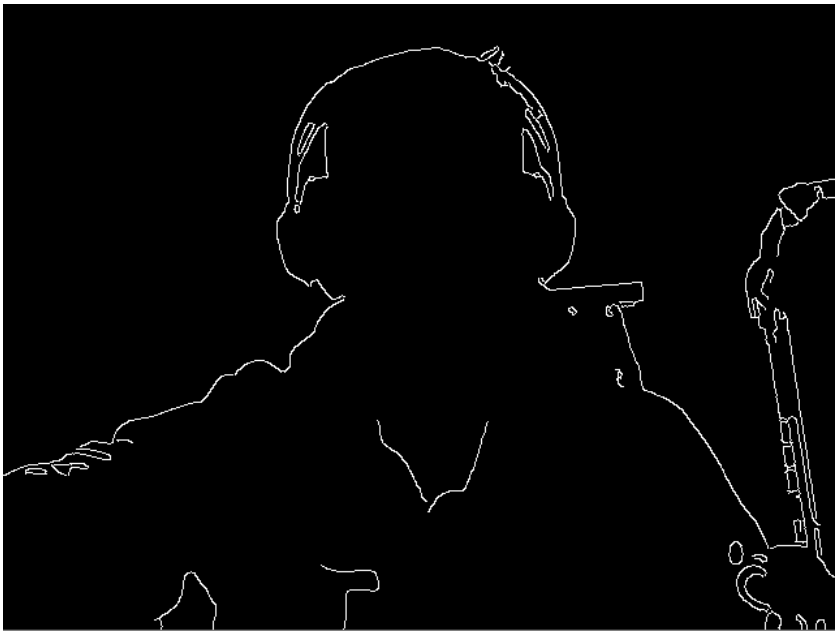


Figure 5: Canny edge

Lastly, we do the Canny Edge Detection. This is the part we will use for object detection. As seen above, we now only get lines where the contrast of the image is large. This allows us to see where something is in the image by the edges of that object.

Figure 6: Region of Interest

Here is what the program will recognize as being in the frame, the user will still get the full video feed, but the robot will not respond to things outside our Region of Interest (ROI). The specific region we are interested in is the very bottom. As can be seen in the image above, most of the image is pure black, while at the lower end, we see the canny edge detection still working. This part is split into two separate rectangles that meet at the middle. The reason for splitting the regions in two is so that we can add response from the program for when something enters either part. If an edge is detected in the left part, the vehicle can be told to take a right, while the opposite for if an edge shows up in the right part. This also allows to detect something right in front, if it detects edges in both regions at the same time.

We chose to do object detection in this way since it relates to Image Analysis course, which two of us on the project also took during this semester. We will discuss other ways of doing it later, but this was the reasoning for why we took this approach for our project.

4.6 Python Programming

Due to lack of knowledge or experience, we started coding the program to match what we thought the code would look like–which in the end did not match at all. The foremost goal was to have two plans which operate parallel on both end-points. To achieve this, we split the main loop into three categories, or in this case, three threads. These threads are assigned one task and will continuously do this one task until told otherwise.

An important thing to note whilst programming in Python is its indentation-based code-execution. Typically, you would not bother that much with the importance of keeping everything aligned and straight. There are of course code-styles that a specific programming language wishes you to follow, but you do not need to follow this exact style. After some hours of programming, you are going to find out that keeping a stable code-style is going to help you out a lot in both readabilities and making the code look less confusing.

Python is not as beautiful to look at when you think about a well documented and perfect code. Compared to Java, it is considered hollow. This glorifies the importance of keeping the code sufficiently documented. To ensure readability for this report, we decided to follow a template code-style to comment the code.[Foua]

Anyway, to start the program, we went with the approach that you can begin either end-point at first, then start the other. Initially, we wanted to start the server and send a signal for the client to start. Unfortunately, we did not manage to get this done due to lack of time, and it was not a high-priority task. More in-depth of how our code works will be reviewed in the section below. 4.7.4

4.6.1 Creating threads

Creating and running threads can be similar to running several different programs concurrently, but with several benefits.

1. The data space is shared between threads within a process. This allows for communication sharing and communication between multiple thread.

2. Threads being like light-weight processes works very well for us since they do not require much memory overhead. This helps us since it makes it easier for the communication between server and client to have the speed we require.

3. Threads can also be both, temporarily put on hold while other threads are running, or simply interrupted.

In this project we use the threading module available in Python to create the threads. This works by using provided function calls which we use to create new threads. The `__init__` function initializes the data represented with the new thread, and the run function is in charge of the threads behavior as the threads starts its execution. Therefore to create a new thread we need to:

1. Create a sub class of the thread class.

2. Override the `__init__` function of the thread class. This method starts up the data specific to the thread

3. Override the run method to define the behavior of the thread.

We opted for not using a threadpool executor since the threads we are running are simply paused during the lifetime of the program. This meant that a threadpool was not necessary because we did not require a lot of separate threads.

```
# ---------- MAIN LOOP ----------
if __name__ == '__main__':
    # Set basic logging configuration.
    logging.basicConfig(format='%(asctime)s - %(message)s', datefmt='%
                        level=logging.DEBUG)
    logging.debug("Started program")
    # Initialize the client object.
    client = Client()
    logging.debug("Created Client")
    client.start()
    logging.debug("Started Client")

    # Set up the threading environment with threadPoolExecutor.
    pipeline = queue.Queue(maxsize=5)
    event = threading.Event()

    thread1 = threading.Thread(target=client.handle_read, args=(queue,
    thread2 = threading.Thread(target=client.handle_send, args=(queue,

    thread1.start()
    thread2.start()

    thread1.join()
    thread2.join()
```

Figure 7: Threading code

4.7 Primary files

In the following subsections below, we will talk about what each files does and how it operates.

4.7.1 main.py

To start explaining how we created the algorithm for the robot, let us start with the main file, `src\main.py`.

```
# ---------- MAIN LOOP ----------
if __name__ == '__main__':
    # Set basic logging configuration.
    logging.basicConfig(format='%(asctime)s - %(message)s',
                        level=logging.DEBUG)

    # Initialize the server object.
    server = Server()
    obsdet = ObstacleDetection()

    #pipeline = queue.Queue(maxsize=5)
    event = threading.Event()

    thread1 =
        threading.Thread(target=rungame, args=(queue, event))
    thread2 =
        threading.Thread(target=server.start, args=(queue, event))
    thread3 =
        threading.Thread(target=obsdet.start, args=(queue, event))

    thread1.start()
    thread2.start()
    thread3.start()

    thread1.join()
    thread2.join()
    thread3.join()
```

Listing 1: Main start example

We have covered the `if __name__ == '__main__'` previously in this report, and here is an example where it has been used. When you start this script, if it is a class, it

will search for this particular loop and start it. If this loop does not exist, the script will not run anything at all. An example of this could be seen attempting to run the `src\utils\server.py`. Trying this, you are met with a Python Terminal that says it has started the code and finished the execution, only because it found no errors and no `if __name__ == '__main__'` to run.

To describe more in the depth of what this loop does; it starts by setting up our logging configuration, which is an imported library at the top of the code. This library aims to give developers more detailed information in the terminal, compared to the standard ways of printing text using `print("Hello World")`. This lets us categorise different prints as either error, debug, exception, info, etc. A great and easy to use addition which removes the necessity to go back into the code to remove unwanted prints when debugging.

That being said, here on line 4, we set the logging configuration to only print messages info and debug. We also format the message to give us the date and time the message was sent, down to milliseconds. This provided us intel on how fast the code would go through the code as well, which we wanted at least be 15-30fps. In seconds, this means that all the threads go through a whole cycle between 66.6 to 33.3 milliseconds.

Further down the main, we initialise both the server and obstacle-detection objects. These will be thrown into the thread creators and run as different threads along with the main game. More on how threading works will be on the next section. For now, let us talk about what the first thread does.

4.7.1.1 Rungame

Rungame consists of three main components. Being the source of this project, it represents the user interface, simulation and has the responsibility of sending commands through the server. We decided to go for this solution due to the lack of knowledge for designing a user interface. Initially, we set out to make the user interface with a GUI library called FLASK, only to be struck by a rough road that led to many other issues in the project.

The result of this method is a little messy and needs a rework for code optimisation purposes, but as for now, it works as intended. We will refer to our diagram on page 31 for the main loop when explaining the results of our code, and shall explain our steps below. [Main loop code example]

```python
def rungame(queue: Queue, event: Event) -> None:
    """documentation"""
    # Initialise variables
    ...
    run = True
    # Initialise different objects
    ...
    while run:
        # -----------------------------------------------
        # ---------Check if connection is active---------
        ...
        # Get the current events happening on screen
        # and treat each individual input as designed
        ...
        # Check for joystick inputs
        ...
        # Check for current keys registered.
        ...
        # Check both arrow and wasd keys for flags.
        # If no flags on them, check joystick.
        # If connected, send cmd to client.
        ...
        # Check move and stop for flags
        # If none are true, no new inputs are detected and
        # thus we stop the robot
        ...
        # If connected, fill the background with the videostream
        # If not, just fill it with black
        ...
    # Send warning-flags to stop the server and client.
    # After this has been done, close the game properly.
    logging.debug("Closing game...")
    pygame.quit()
```

Listing 2: Main loop code example

As seen above, the method consists a loop which have exit points such to whenever an exception or a timeout occurs. It contains many checks to statements which we will not enter if the connection variable contains `False`. This is important to the logic due to the script knowing whether we are connected and should send or not. If it attempts to send while connection shows `False`, we will receive an exception, thus exiting the program.

4.7.1.2 User Interface

The user interface is a very simple GUI, made up of a few components in a `pygame-window`. If the connection between the server and the client is active, the server receives an image-frame from the camera mounted at the front of the robot, via the TCP-socket. This frame is continuously updated and used as the background in the interface. The user interface also has a `textbox` which displays the current speed of the robot, as well as a indicator which displays what direction the robot is currently being driven in.

Even without the connection to the client, the interface is used as a controller-simulator. The direction-indicator is available regardless of whether the server has a connection or not, as the indicator for direction is created within the user interface, server-side.

4.7.1.3 Sending CMDs

A key feature to this script, besides the GUI, is the Pygame module. The module not only lets us create a user interface and games, but also gives us the possibility to use keyboard inputs to control the robot. Similarly to keyboard inputs, we have added compatibility for controllers. Without any change in the current code, we can seamlessly change between keyboard and joystick inputs. This is all thanks to the module, and without it, we would have struggled to get a GUI which got these input checks implemented.

4.7.2 server.py

This file holds the procedure or receiving and sending commands through TCP sockets. It consists of two essential functions, `send` and `start`, and some others which makes canny and frame image available and stopping/closing the server connection whenever an exception occurs or the socket connection times out.

Once again, we will refer to the diagram on page 41 showed further below. To aid in understanding how we created this logic, I will take some code examples from the Server class. [Main loop code example]

```python
class Server(object):
    """summary"""
    def __init__(self):
        # - - Initialize variables, logging configuration, - -
        # - - - - - - - - - - socket and bind. - - - - - - - -
        ...

    def stop(self):
        # Stops server

    def isconnected(self) -> bool:
        # Returns connected state

    def close(self) -> NoReturn:
        # Closes server

    def get_canny(self) -> any:
        # Returns canny-image.

    def get_frame(self, resolution) -> any:
        # Returns frame.

    def start(self, queue: Queue, event: Event) -> None:
        # Starts the server and receives incoming packets.
        # Formats incoming packets as current frame and canny.

    def send(self, message: any) -> NoReturn:
        # Accessed by rungame, sends CMD to client.
```

Listing 3: Main loop code example

Here we see a draft of how the class is implemented and shall explain how both the essential methods works to make this a server. Our first interaction with this class is that the second thread gets assigned `server.start` as a task. Without this, the server will not start, and we will not be able to communicate with the client. We will go through this method step-wise and explain how it works.

4.7.2.1 Server.start

```
def start(self, queue: Queue, event: Event) -> None:
        """
        The method starts by listening for a client to connect.
        When a client connects, start receiving data from said client.
        After data is received, unpacks it and formats it to suit
        the GUI format. This also includes reversing the received array,
        rotating it, flipping it and sets it as the current given frame.

        Args:
            queue (Queue): [description]
            event (Event): [description]
        """
        self.server.listen()
        self.logger.info(f"Server is listening on {self.ADDR}")
```

Listing 4: Server.start method

First of all, when the server gets initialised, it will bind itself to the socket upon its IP address. By doing during the initialisation, the second thread can now start to listen on the given address. The `self.server.listen()` means that it will start listening for any activity on the socket. If any client wants to connect during this stage, the server will accept the first connection and proceed with accepting incoming packets and decoding them. Below shows how we accept a connection to the server.

4.7.2.2 Accepting connection

```
while not event.is_set() and not self.stop:
    data = b''
    try:
        self.conn, self.addr = self.server.accept()
        logging.debug("connection accepted.")
        self.connected = True
        logging.debug("self.connected = True.")
        logging.info(f"[NEW CONNECTION] {self.addr} connected.")
```

Listing 5: Accepting connection

You will notice that we have solved our connection with a while loop-statement. `Event.is_set()` will be explained in threading, and `not self.stop` is our internal flag that notifies the server when to stop and break out of the while loop. Right after entering

this loop, we create a variable named `data` that contains an empty string of bytes. This is to ensure that before we start receiving and decoding data bytes, they represent nothing.

4.7.2.3 Receiving and decoding

Further down, we will start receiving and decode the incoming packets.

```
while not event.is_set() and not self.stop:
    self.conn.settimeout(2)
    while len(data) < self.payload_size:
        data += self.conn.recv(self.HEADER)
    packed_msg_size = data[:self.payload_size]
    data = data[self.payload_size:]
    msg_size = struct.unpack("L", packed_msg_size)[0]
    # logging.debug(f"recieved: {msg_size}")
    while len(data) < msg_size:
        data += self.conn.recv(self.HEADER)
    frame_data = data[:msg_size]
    data = data[msg_size:]
    pickledframe = pickle.loads(frame_data)
    frame = cv2.flip(pickledframe, 0)
    frame = cv2.flip(frame, 1)
    frame = numpy.rot90(frame)
    frame = frame[::-1]
    frame = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
    frame = pygame.surfarray.make_surface(frame)

    canny = cv2.flip(pickledframe, 0)
    canny = cv2.flip(canny, 1)
    # canny = numpy.rot90(canny)
    canny = cv2.cvtColor(canny, cv2.COLOR_BGR2RGB)
    blur  = cv2.GaussianBlur(canny, (5,5),1)
    canny = cv2.Canny(blur,50,150)

    self.canny = canny
    self.frame = frame
```

Listing 6: Receiving and decoding packets

For the untrained eye, a lot is going on here, and it might be challenging to read at first. As previous code examples, let us introduce you to how we receive and decode the incoming packets, which in this example represents frames from the client. We start by

nesting everything in another while-loop statement as previous code example Accepting connection. This is to make it possible to receive packets continuously and not having to wait for a new connection each time we want to receive a single packet. We encode our frames in a format in which we expect the frames to arrive.

While our previously initialised variable `data` is less than the `self.payload_size`, we shall continuously add data to our variable until it matches the wished size. When this criterion is met, all incoming data after will be received as the sent frame or *message*. We also use a library to encode and decode our packets, named *Pickle*. This library will in future versions changed to something more strict due to a security issue when sending pickle-coded packets across the network.

As we receive our image frame from the data, we then decode it with Pickle (as seen on line 13 on Receiving and decoding packets). Now starts the process of formatting the frame correctly to satisfy the wanted array-dimension that `pygame.Surface` uses. If we do not initiate this process, you will get some *trippy* and *wicked* results looking at the video stream. After the frame has been decoded, we must do a number of changes to the frame to display it in our user interface correctly. These steps are shown from line 14 to 29 and to visualise it better we can make a shape it into a numbered list. After decoding, the following procedure is:

1. Flip the image horizontally
2. Flip the image vertically
3. Rotate the image 90 degrees
4. Rearrange the whole array that represents the frame
5. Rearrange the colour-coding format we received
6. Prepare the user interface frame
7. Copy the loaded frame from the decoded pickle
8. Flip the image horizontally
9. Flip the image vertically

10. Rearrange the colour-coding format we received
11. Apply Gaussian blur to the image
12. Apply Canny Edge to the blurred image
13. Finally, set the frame and canny image as the current frames available.

Lastly, now that we are done with the procedure, we backtrack to the start of the loop and do it all over again. This occurs within the previously mentioned 33.3 to 66.6 milliseconds, which is entirely within our desired loop time. If this where someone interrupted with an exception or error of some sorts, we will break out of the loop and into our exception handler as seen here.

4.7.2.4 Exception handling

```
except socket.timeout:
    event.set()

self.close()
```

Listing 7: Expected exception

Here, the only exception we expect to find is a `socket.timeout`. One could argue that it should catch all exceptions since we cannot guarantee that only `socket.timeout` is the single possible one. However, we see that it is not likely that the system should continue if we receive such exceptions, therefore leading into a crash and ending all the processes. If a `socket.timeout` were to occur, we set the event flag that signals the threads to terminate.

4.7.3 obstacledetection.py

Third and last of the main threads, is the one who detects obstacles and overrides the user send essential commands. We considered this as one of the main goals of this project, which still needs to be correctly implemented. Obstacle Detection was created as a class that consists of methods like converting frame to canny (if necessary), creating two regions of interest and checking these accordingly.

To start the detection procedure, we initialise the object in `main.py` [Main start example] and then assign the start method to the thread. Our current plan of detecting and flag-

ging signals is letting the class itself warn whenever something is seen within the regions of interest. Below is a rough sketch of how this class was created.

```
class ObstacleDetection:
    def __init__(self):
        # Initialise variables, in this case, an empty canny array.

    def CannyEdge(self, image):
        # Takes an image input and returns it as a canny.

    def region_of_interest(self, image, resolution, percent) -> any:
        # Creates a region of interest within the left corner
        # of the image

    def region_of_interest2(self, image, resolution, percent) -> any:
        # Creates a region of interest within the right corner
        # of the image

    def check_roi(self, image) -> bool:
        # Check if any values within the region contains an
        # obstacle. If so, return True. Else, return False.

    def set_canny(self, canny):
        # Sets the current available canny image as the
        # canny image to analyse.

    def start(self, queue: Queue, event: Event):
        # This is the primary function of the class,
        # which uses its previously mentioned methods
        # and utilises them to check for obstacles in
        # the given canny image.
```

Listing 8: Obstacle Detection

An immaculate class compared to the others and serves as a helping service dog to our half-blind vehicle. Let us talk about how the start method works.

4.7.3.1 obstacledetection.start

This exact method is run by our third and final thread. However, it does not proceed to action without an available canny frame. We will go through the process below in

numbered steps.

1. Set local variables.
    - Resolution.
    - Height percentage of resolution to analyse.
2. Wait for two seconds to ensure the camera is up and running.
3. Check if canny is available. If not, repeat the check. If so;
    - Create regions of interest.
    - Try to display canny image, can be toggled off and on.
    - Check each region of interest.
    - Give output based on ROI results.
    - Return to start and repeat the process.

These steps are what we will call the prototype of our detection algorithm. Similar to previous threaded methods, we use a while loop-statement to ensure that we always fetch a new image whenever available. There are other ways to obtain detection with better results, but for our project, this will suffice. Different ways to achieve similar results were not as simple as what we were able to accomplish. As the last input about this section, we add the start method code below.

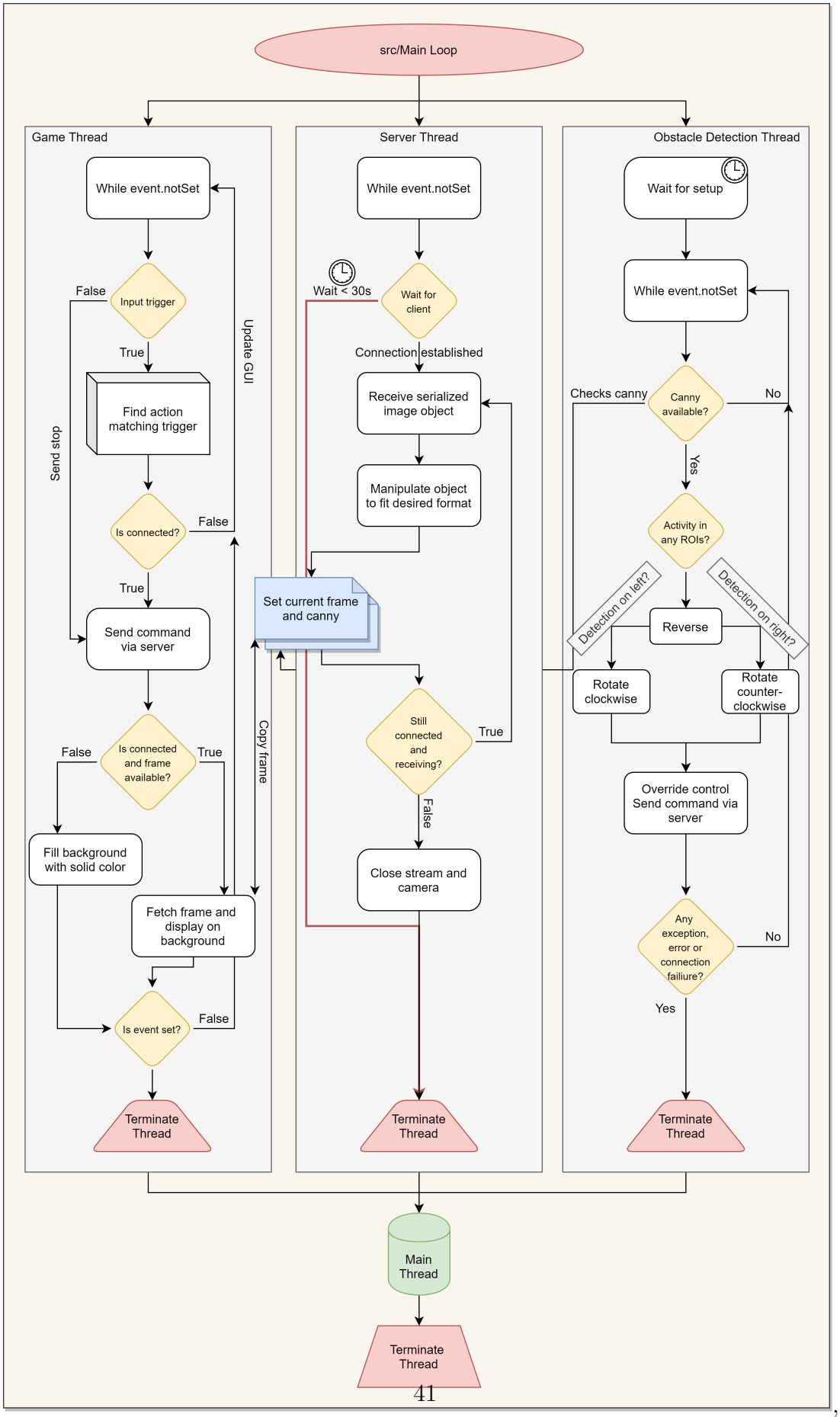
```
1  def start(self, queue: Queue, event: Event):
2      resolution = (256, 144)
3      percent = 1.2
4      sleep(2)
5      while not event.is_set():
6          cv2.waitKey(50)
7          if self.canny.any() > 0 or self.canny != None:
8              cropped_image = self.region_of_interest(self.canny,
    resolution, percent)
9              cropped_image2 = self.region_of_interest2(self.canny,
    resolution, percent)
10             combo_image = cropped_image + cropped_image2
11
```

```
12              try:
13                  cv2.imshow("ROI", combo_image)
14              except:
15                  pass
16
17              # Checking activity in the rois
18              if self.check_roi(cropped_image) and self.check_roi(
    cropped_image2):
19                  print('Back up')
20                  pass
21                  # TODO Turn around 90 degrees
22              elif self.check_roi(cropped_image):
23                  print('left')
24                  pass
25                  # TODO Turn 45 degrees to the right
26              elif self.check_roi(cropped_image2):
27                  print('right')
28                  pass
29                  # TODO Turn 45 degrees to the left
30              else:
31                  print('nothing')
32                  pass
33                  # TODO Nothing detected, do nothing
```

Listing 9: Obstacledetection.start method

4.7.4 Loop diagram for server end-point

On the next is the diagram created to refer to our threads within the server end-point. As seen and mentioned beforehand, it consists of three threads running in parallel with each other. We have tested a couple of other methods of implementing this, such as creating more classes to inherit different tasks, resulting in more of a confusing mess and spaghetti code.


src/Main Loop
Game Thread
Server Thread
Obstacle Detection Thread
While event.notSet
While event.notSet
Wait for setup
False
Input trigger
Wait < 30s
Wait for client
While event.notSet
True
Connection established
Find action matching trigger
Receive serialized image object
Checks canny
Canny available?
No
Send stop
Update GUI
Manipulate object to fit desired format
Yes
Is connected?
False
Activity in any ROIs?
True
Set current frame and canny
Detection on left?
Detection on right?
Reverse
Send command via server
Still connected and receiving?
True
Rotate clockwise
Rotate counter-clockwise
False
Is connected and frame available?
False
True
Copy frame
False
Override control Send command via server
Fill background with solid color
Close stream and camera
Any exception, error or connection failiure?
No
Fetch frame and display on background
Yes
Is event set?
False
Terminate Thread
Terminate Thread
Terminate Thread
Main Thread
Terminate Thread

4.7.5 client.py

On these following sub-sections, we are going to talk about how the client end-point works and why we decided to for this strategy. Much like Main start example, this script has a main loop which initialises the threads, and despite being different, they look quite similar.

```
# ---------- MAIN LOOP ----------
if __name__ == '__main__':
    # Set basic logging configuration.
    logging.basicConfig(format='%(asctime)s - %(message)s', datefmt='%d
    -%b-%y %H:%M:%S',
                        level=logging.DEBUG)
    logging.debug("Started program")
    # Initialize the client object.
    client = Client()
    logging.debug("Created Client")
    client.start()
    logging.debug("Started Client")

    pipeline = queue.Queue(maxsize=5)
    event = threading.Event()

    thread1 =
        threading.Thread(target=client.handle_read, args=(queue, event))
    thread2 =
        threading.Thread(target=client.handle_send, args=(queue, event))

    thread1.start()
    thread2.start()

    thread1.join()
    thread2.join()
```

Listing 10: Client main example

The critical difference between these loops, is that on the client side we only assign threads to handle methods is within the client. As the client is initialised, we obtain the connection to the motor controllers and therefore only need two threads. At the bottom of this section, the diagram of how the remote thread loop works. This can be found on page 47

The client consists of a class with two primary methods, `handle_send` and `handle_read`. To start the connection with the server, we made a start method in this class which is similar to the start method on the Server.start method. To get an overview of this class, we have made a quick draft of how the class looks below and will go through how it works.

```
class Client(object):
    def __init__(self):
        # - - Initialize variables, logging configuration, - -
        # - - - - - - - - - - - socket and bind. - - - - - - - - -
        ...

    def start(self) -> NoReturn:
        # Starts the camera, initializes the socket and connects to it.
        # When done, sets the connection to true.

    def disconnect(self) -> NoReturn:
        # Shuts the videocapture down and closes the socket.

    def handle_send(self, queue: Queue, event: Event) -> NoReturn:
        # The thread get called to loop here continuously unless
        # flagged not to. Its only purpose is to read and process the
        # image from the raspberry pi camera module. After it is read,
        # convert it to GRAY to reduce amount of data sent.

    def handle_read(self, queue: Queue, event: Event) -> NoReturn:
        # The thread gets called to loop here continuously unless
        # flagged not to. Its only purpose is to read and process data
        # that comes in from the server. The CMDs received from the
        # server comes in form of 'aswdqe' or 'stop'. After received,
        # the msg is processed and passed into the if-statements to
        # check what direction/move the robot should do next.
```

Listing 11: Client code draft

4.7.5.1 Receiving CMDs

The first important step in this client is to be able to connect and receive commands from the server. Regardless of having a video stream, this was implemented first and recently reworked due to motor controller version change. This ended up being quite unfortunate as we struck upon some hardware issues and spent too much time on updating how to send CMDs to the controller.

Unfortunately, due to some errors we had to archive our custom motor controller class as it did not work with the new version of the motor controller. How we receive and treat incoming CMDs might not look as pretty as it should, but works as intended.

```
def handle_read(self, queue: Queue, event: Event) -> NoReturn:
    while not event.is_set():
    try:
        data = self.socket.recv(self.HEADER)
        msg = pickle.loads(data)
        if msg == 'w':
            # Command motors to move forward
        elif msg == 'a':
            # Command motors to move left
        elif msg == 's':
            # Command motors to move backward
        elif msg == 'd':
            # Command motors to move right
        elif msg == 'wd':
            # Command motors to move northeast direction
        elif msg == 'wa':
            # Command motors to move northwest direction
        elif msg == 'sd':
            # Command motors to move southeast direction
        elif msg == 'sa':
            # Command motors to move southwest direction
        elif msg == 'q':
            # Command motors to rotate counter-clockwise
        elif msg == 'e':
            # Command motors to rotate clockwise
        elif msg == 'stop':
            # Command motors to completely stop

        if 'speed' == msg[0]:
            self.SPEED = msg[1]
            # Change current speed to the newly received speed.
        elif msg == '!DISCONNECT':
            # Disconnect and close the client.

    except Exception as e:
        logging.debug(f"Exception occurred: {e}")
        event.set()
```

Listing 12: Handle-read code example

As seen above, it consists of a long `elif` assembly to ensure that the various CMDs get correctly treated. It also has the possibility to change speed and disconnect if the server sends a CMD that matches the if-statement criterion. Since we are dealing with a thread handling this method, the while loop-statement with `event.is_set()` can be found here. This statement is used again to ensure that we continuously continue to receive CMDs.

Additionally, we catch any exceptions that might occur. We expect these exceptions to come from an error reading the data we received, or connection might have been lost, thus leading to an exception being raised. If so, we set the event and flag the threads to terminate.

4.7.5.2 Sending frames

Our second but equally important method is how we send frames to the server. In contrast to the previously mentioned method, handle-read, this consists of only a few lines of code. Let us have a look at how it is built.

```
1  def handle_send(self, queue: Queue, event: Event) -> NoReturn:
2      while not event.is_set() and self.connected:
3          try:
4              cv2.waitKey(100)
5              _, self.frame = self.cap.read()
6              frame = cv2.cvtColor(self.frame, cv2.COLOR_BGR2GRAY)
7              data = pickle.dumps(frame)
8              self.socket.sendall(struct.pack("L", len(data)) + data)
9          except Exception as e:
10             self.logger.exception(f"[ERROR] Closing.. {e}")
11             event.set()
12             break
```

Listing 13: Handle-send code example

Just like any other threaded method, we continuously send frames within a while loop-statement. This can only be interrupted by any exception and operates as the optic nerve that sends images to the brain. We have in addition to the while loop, making sure that we have a connection to the server before we attempt to send any images.

It is important to notice on line 4, `cv2.waitKey(100)`, that we have set the delay on how many frames we get per second to be strictly once every 100 milliseconds. This was initially a delay imposed to ensure that we did not push the network bandwidth to its

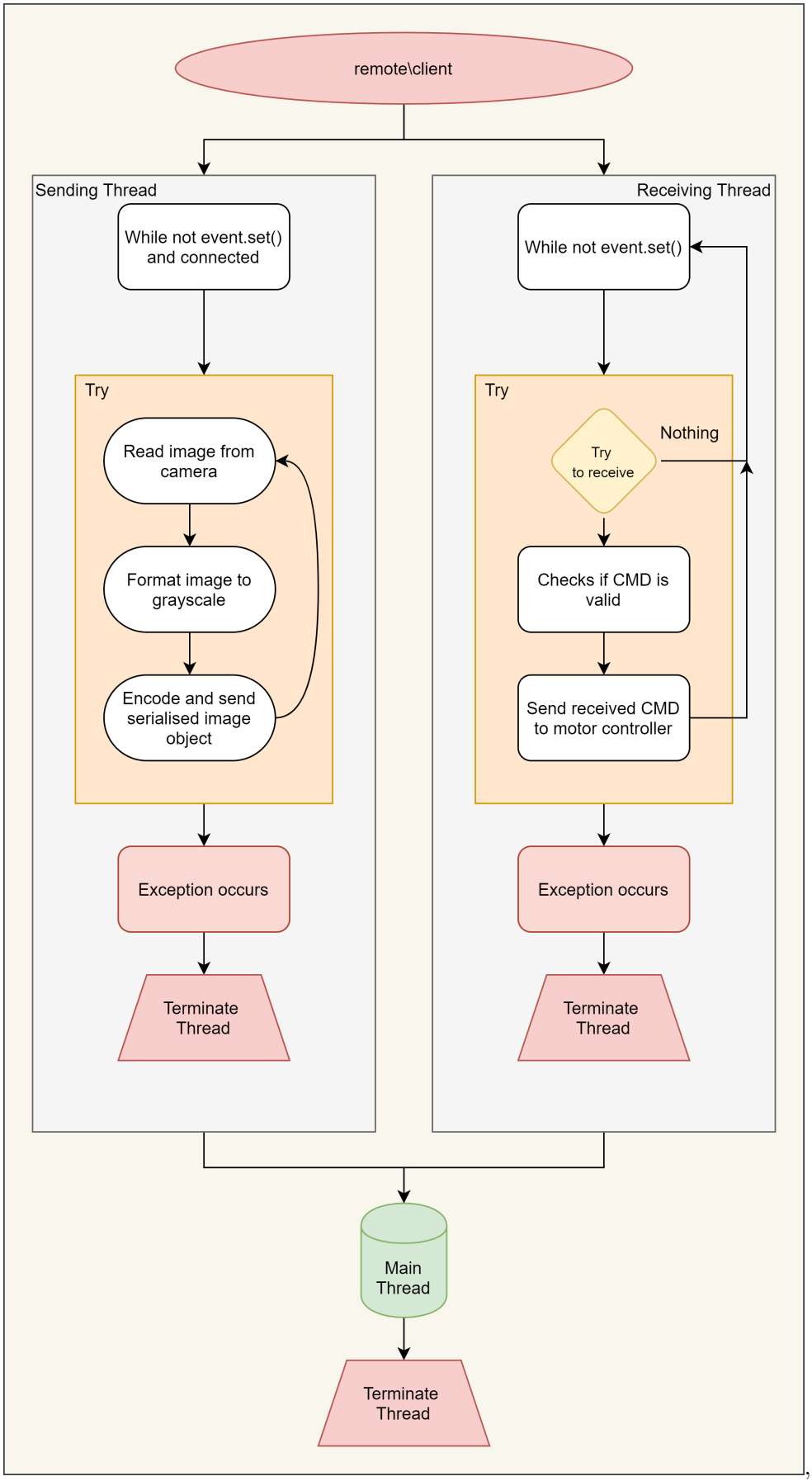
limits, and get a consistent flow of images to the server.

However, this has been tested all the way down to a delay on 25 milliseconds with decent results. This concludes that we can run it approximately as fast as we wish for, as long the bandwidth supports us. As for capturing and sending, the method reads the image from the camera and in instant converts it to grey-scale.

This saves us a lot of bandwidth and provides us with a massive performance boost due to sending a third of the original data instead of the RBG variant. This frame then gets encoded by our earlier mentioned Pickle module and sent through the socket.

4.7.6 Loop diagram for server end-point

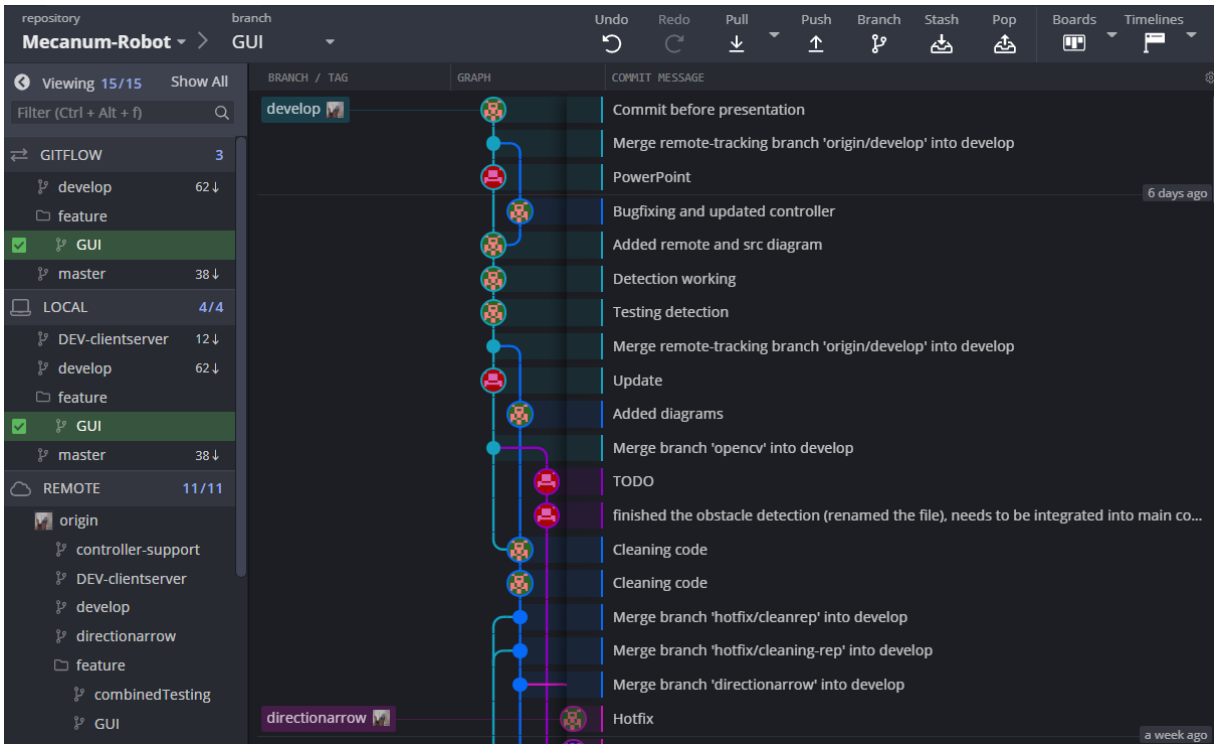
This next section yields the diagram created to refer to our threads within the client end-point. In contrast to the server end-point, it consists of two threads running in parallel with each other. We have tested a couple of different methods of implementing this as well, such as the motor controller class, which was discarded due to rework and the approaching deadline of the project. Even if this was an issue to fix, the client works as intended.


remote\client
Sending Thread
While not event.set() and connected
Try
Read image from camera
Format image to grayscale
Encode and send serialised image object
Exception occurs
Terminate Thread
Receiving Thread
While not event.set()
Try
Try to receive
Nothing
Checks if CMD is valid
Send received CMD to motor controller
Exception occurs
Terminate Thread
Main Thread
Terminate Thread

4.8 Git

We have used GIT to achieve version control and manage to test out many concepts at the same time, whilst avoiding ruining or overwriting code on the master branch. The branching system was divided into three main categories, in which we all three did our research and testing on these branches. When we achieved any significant results we could use within our develop or master branch, we merged the branches then went off to a new branch to avoid interrupting the master or create a branch.

This is a procedure used to prevent merge-conflicts when coding together. A merge conflict occurs when two actors change the same line of code, then attempt to push both together, which results in conflicts happening due to two changes on overlapping on the same line.



5 Results

Overall we are quite pleased with our results. We did not get to implement all that we wanted, like autonomous movement or a proper GUI, mostly because of time constraints. What we did manage to complete was, the communication between the vehicle and the server, the mecanum wheels and the frame of the vehicle, and the threading needed. We will go into more specific details below to explain the results of the parts of the project.

5.1 Robot

Starting off, we'll look at the robot itself, including the frame and wheels.

5.1.1 Frame

The frame we used to build the mecanum robot on was a frame previously used in this course, created by students who have taken this course earlier, and was made from aluminium profiles and an acrylic bottom plate. The platform worked okay for the robot we created, but it did cause some minor problems. Unfortunately, the frame was not completely square. This lead to one of the motor mounts not being fastened squarely compared to the other motors, which led to the robot not running straight.

The motors were fastened to frame with the original brackets, 2mm thick aluminium L-brackets. After driving the robot with the original brackets for some time, we designed a new set which we 3D-printed.

5.1.2 Mecanum wheels

The different components of the wheels were mostly 3D-printed in PLA-filament, a hard but brittle thermoplastic. The rollers were mounted to the rims using 3mm threaded rod and m3 nuts. The wheels were then mounted to the motors using a mounting kit initially designed for mounting scooter-wheels.

5.2 Communication

Our final way of communicating between end-points, turned out to work as intended. Communication was implemented half-way through the project and is the only way to talk to each end-point. We can connect multiple clients if wanted, meaning we can expand this project to operate multiple units.

5.3 Client Server Logic

As mentioned earlier in methods, the results of our end-point logic were not as we wished it to be when it comes to starting both end-points. As a step-wise introduction to making the logic more comfortable to read, the procedure we had to engage looks like this:

1. Ensure that both end-points are powered and turned on. This includes the battery to the motor controller as well.
2. Check if both are connected to the same WiFi.
3. On the server-side, start Anaconda and launch VSCode.
4. When VSCode is ready, start the `src\main.py`
5. Within thirty seconds on the remote, start the `remote\client.py`
6. The video stream should now be available, and you can operate the vehicle.

5.4 Threading Logic

Our threading resulted in concurrency within our program. The robot can both send its footage to the server, and receive commands at the same time. It also allowed for image processing to happen while sending the footage, and the framerate of the obtained footage was significantly increased thanks to threading.

5.5 Image processing

Image processing was some of the later parts added to the project. This resulted in not getting all that we wanted in the final product. The grey-scale conversion and Canny edge detection work well, both doing exactly what they were designed to do.

The grey-scale results frames being sent much quicker. The data being sent was about three times less because of the conversion, which resulted in significantly increased performance.

While the concept and most of the code for using the Canny edge detector were completed, we ran out of time to implement autonomous movement. As a result, our vehicle can only be controlled manually by either a keyboard or controller.

5.6 Limitations

5.6.1 RoboClaw

The limitations of the motor controller are that documentation is sparse. We struck upon issues such as 'blanking' and trying to understand how to configure the controller when a universal serial bus connection was not an alternative. We wasted a lot of time trying to configure the controller manually. If the documentation provided enough information covering specifically the critical components in how to configure and use the controller, we would have more time to implement, for example, autonomous movement. Else wise, this is an alright controller to start using for fundamental to intermediate tasks.

6 Discussion

In this section, we will be discussing how the project as a whole went, specific parts, and look at weaknesses, strengths and potential improvements that apply to the assignment.

6.1 Set up

When it comes to the setup, we went with having a router, laptop and our robot for the hardware setup. Requirements for our structure are the following; have an IDE, a router available, and a platform with motors on. This includes having two Roboclaw Motor Controller within our grasp. Being able to work with all these components in one room made it far easier to cooperate as a team while having an overview of how far we are away from the goal.

6.2 Communication

The communication was a task we postponed for quite some time. We started by researching whether we should use TCP or UDP. After testing both these communication protocols out, we decided that we had no use for UDP. We have both protocols as separate files when it comes to the version control, and can confirm that these two work as intended. The reason we went for TCP was due to us wanting to have a stable connection and not wanting to struggle with potential packet errors you could get with UDP.

6.3 Robot

6.3.1 Frame

The original brackets that were packaged with the motors were not sturdy enough; the weight of the robot itself was sufficient for the brackets to start bending, giving the robot a lot of negative camber. To attempt to resolve this problem, we downloaded the motors dimension sheet and used these measurements to create our own motor mounts which we 3D-printed. These mounts were a lot sturdier but did not completely resolve the camber-problem. The mounts we made of the same design as the original L-shaped brackets, this time thicker and with some extra material to stiffen the mount. If we had designed the new mounts such that the motors had more than just a single point of connection to the frame, we would avoid the weight of the robot torquing the wheels.

6.3.2 Mecanum wheels

A downfall of printing all the parts in PLA was that the friction between the wheels and the ground was very low, as PLA is quite slippery. We tried to solve this by adding heat shrink to the outside of the rollers, the only part touching the floor, which helped somewhat, but still not to the level of friction we wanted to achieve. A way to accomplish this could be to 3D-print the rollers with TPU-filament, which is a more rubber-like material.

6.4 Program and design

We feel like we have to defend our reasoning behind choosing Python over Java for the execution of our project. We will go into more of the technical side of the argument, but first, we are going to talk about the Literate Programming Approach.

Literate programming is quite simple in concept, if not in execution. It is all about explaining your code while creating it, not just making it readable by machines or other developers, but also by others with less experience within programming. We students at NTNU has been thought since our first year to comment our code well so that it's easier to come back to later, or for others to understand your code without having to read every function in detail. We are aware of this and try to employ it in our code as much as possible. It is often easy to postpone the commenting until the code is finished, and often than not having enough time before a deadline needs to be met.

In general, we refer to this as code style. Only applying techniques for understanding the code better, this includes using indentations, among others. Now, what if our code is not just to be presented to other developers, but to a group of researchers, or a staff within a company, or a new group of student at university, where not everyone is proficient with programming? In this case, we would probably need even more than just comments explaining what each function do. We would need graphs, pictures, detailed comments, and yet it is to be utterly executable within the environment.

Literate programming, therefore, excels in areas such as demonstrations, teaching, collaboration, research, and presentation. And Python already supports such environments through IDE's such as LEO and Jupyter Notebook. This could become more and more important in the future since programming is working its way into just about every single market in the world. Being able to present the ideas of a program is not just about mak-

ing it work anymore, but working together with other parts of the industry, i.e. business. When the goal might not just be application development, the need to consider the skills and needs of other collaborators will be necessary.

Therefore we believe Python is very worth learning, with its lower barrier to entry, its full range of use, and how it helps build useful coding techniques early.

[Wike]

On the technical side, however, we spared us a lot of time creating classes and assigning threads. We also avoided having to communicate between two languages, or having to rewrite a python script into Java. The motor controller is designed in C and later been converted to Python to make it more up-to-date for young engineers to try. This subsequently ended up motivating us more to attempt a real-time threading project in pure Python.

We also managed to learn a new side of Python and how the world of threading can create an impressive performance boost in many instances. A great example of this is when we had a hard time with the frames from the camera. The client struggled with sending images fast enough, as well as reading them. We ended up having struggling to get more than 10-20 frames on average. Having such a low framerate is not too bad, but leaves us at a limit on how fast the system could go. After implementing threading and leaving a thread to read, and another one to read the camera. This led us to a massive framerate boost, giving us up to 200 frames per second. Utilising threading, when working around image analysis, therefore made it clear that this was the approach to use.

However, we shall not look the other direction when it comes to weaknesses for the programming languages. Compared to Java, Python has yet a long way to go when it comes to having easy to read the documentation. There is no standard documentation style in Python that looks, and operates, as good as JavaDoc. Some might argue that the code-style in general look cleaner in Java and that certain operations are much faster to do in that language. Multi-threading has also been one of the key strengths in Java throughout the years, where Python falls a bit short struggling with the GIL restricting the interpreter only to execute one code line at the time.

6.5 Further work

Even though the project is at the deadline and we have achieved wanted results, there is still a lot we could improve on this project. To advance in this project, we have a list of design and program improvements to make it all better.

- Stronger platform structure
- Better gearboxes
- Use encoders to travel set distances
- Better friction on wheels
- Better motor drivers
- Rework GUI
- Fully implement autonomous movement

6.5.1 Better Simulation

There are some small things that could have been implemented in the user interface, which doubles as the simulator. We could have added a model to represent the robot, allowing the user to drive a movable model instead of only having an arrow pointing in the direction the robot would have driven if it had been connected to the system. In a previous version of the simulator, we used a solid red cube that moved across the screen according to the inputs it received. We ended up replacing this with the direction-indicator when we instead should have made some simple tweaks to keep both of these functions so that we could still have the model acting as the robot when driven. At the same time, the server is disconnected and keep the direction-indicator for use when operating the robot.

Using the already stored values like speed and drive time, which indicates how long the robot is to operate in the direction given by the incoming commands, the simulator could be developed further to allow it to function as a digital twin.

6.5.2 Image Analysis

The image analysis part of this project, technically, does precisely what it is supposed to do. However, except certain parts of it never got completed. Gray-scaling for the sake

of speeding up the data transfer between the client and server works as intended. Still, the autonomous control never got past the point of printing out a reaction to something entering the ROIs. So there are improvements for what can be done with this project.

For object detection in general, there are many ways to accomplish what we set out to do. As mentioned earlier, we chose to use only Canny Edge detection to detect what is in the view of the camera, but there are other ways. We could have used already existing haarcascade files, where deep learning has been used to teach a program to recognize particular objects. This could have been used for more accurate detection of objects, which could lead to better execution of autonomous movement. For example, we could have made it so the vehicle would follow after certain objects it recognized, like feet or shoes. A possible way to take the project then would be to follow someone walking in front of it or follow another vehicle.

The simplicity of our program can also be a strength, however. With a simple Canny Edge detector, there are many options we can explore for further improvement. Following a track, moving according to what is in its path, crash prevention, and more is achievable with the very easy to use the Canny Edge algorithm. This makes our solution quite adaptable, and with almost no requirements outside of the camera, we already have. The options available to us might also be a reason for why we didn't get time to implement it properly, since we couldn't decide on what, exactly, we wanted the vehicle to do.

6.6 Group Experience

One of the challenges the group had during the project was the simple fact that only one of us had previous experience with Python. This meant that while researching threading and how to complete the tasks we wanted, we also had to learn Python. This made particular objectives take longer than initially planned, and caused some tasks to be scrapped or simplified.

Splitting the team to focus on different parts, however, worked exceptionally well. While one worked on the main, code, another researched how to thread in Python, and the third focused on the construction of the robot. All while we helped each other and worked together to fit the individual parts together.

Overall the group worked very well together, although we spent too much time in the beginning planning concepts and ideas, without landing on something concrete and exe-

cutable quickly. This did impede our progress some, which again, resulted in tasks having to be dropped or reduced so we could finish the most critical parts of the assignment.

6.6.1 Group Dynamic

While no one on the group had worked together on a project together before, we ended up working quite well together. Difference in previous experience helped us designate what task that person should do. For example, the only one that knew Python already was set to work on the main structure of the program, and the other was assigned to design the wheels due to experience in both 3D modelling and printing.

6.6.2 Learning outcome

All in all, we have learned a lot by challenging ourselves to do this project purely with Python. Everyone's work intertwined with each other and we always stepped in to help each other when we got stuck upon an issue. Learning Python was a valuable experience each of us will have use for later. After all, Python is one of the most leading programming languages coming up, and we covered how practical it is in Programs and Design earlier in this section.

7 Conclusion - Experience

Lastly, we will put it all together. In conclusion, we are going to draw a picture from the previous sections on how our project turned out. We will conclude with some additional advice for anyone attempting similar projects in the future.

7.1 Project Conclusion

While we did not manage to finish everything we set out to do with our robot, we ended up being quite pleased with the result. What made this robot unique, its wheels, are prominently on display working as intended. Its omnidirectional movement makes it easy to manoeuvre through most spaces, as long as it physically can fit. This freedom of movement allows us to operate the vehicle how we want, which can be expanded on in many ways.

7.1.1 Project Goals

The goals we completed were the most vital of the project. The multi-threading, control over the robot, the framework of the physical robot, where all finished. We prioritized these parts because they where most needed for testing and making the project work.

There where however, some goals we did not complete. The biggest was autonomous movement. This was part of our original design for the project, therefore, the biggest loss. We also wanted better overall quality on the physical pieces for the robot. The wheels where 3D-printed at school, so they have some traction issues. One part of the metal holding the motors where bent slightly, which resulted in some problems with directional control.

7.1.2 Advice for future work

For future improvements to the project we recommend looking into vector based movement control of the wheels. Currently the robot can not move in more than one of its eighth directions at a time. This makes for more of a staccato type movement. Using vectors however, allows for the vehicle to, for example, "drift" around an object, so that the camera all ways is pointing towards the object. Another thing to look into would be implementing *on the fly* adjustable speed-adjustment

Autonomous movement is also on the list of things to improve upon. Mostly because we did not get to add it. But we also don't really know how well our solution would have worked since we did not get that far.

Bibliography

[Anda] Jim Anderson. *An Intro to Threading in Python*. URL: https://realpython.com/intro-to-python-threading/#conclusion-threading-in-python. (accessed: 24.09.2020).

[Andb] Jim Anderson. *Speed Up Your Python Program With Concurrency*. URL: https://realpython.com/python-concurrency/. (accessed: 25.11.2020).

[Axo] Medium Axosoft. *What is GitKraken? Why Students Should Use GitKraken?* URL: https://medium.com/@hussnainfareed/what-is-gitkraken-why-students-should-use-gitkraken-ccbc1f36c187. (accessed: 16.11.2020).

[Foua] Python Software Foundation. *Documenting Python Code: A Complete Guide*. URL: https://realpython.com/documenting-python-code/. (accessed: 14.09.2020).

[Foub] Python Software Foundation. *Low-level networking interface*. URL: https://docs.python.org/3/library/socket.html#socket.socket.send. (accessed: 16.11.2020).

[Fouc] Python Software Foundation. *Low-level networking interface*. URL: https://docs.python.org/3/library/socket.html. (accessed: 13.11.2020).

[Foud] The Python Software Foundation. $_{t}hread^{-}Low-levelthreadingAPI$. URL: https://docs.python.org/3/library/_thread.html#module-_thread. (accessed: 24.09.2020).

[Foue] The Python Software Foundation. *queue — A synchronized queue class*. URL: https://docs.python.org/3/library/queue.html#module-queue. (accessed: 24.09.2020).

[Mic] Microsoft. *Visual Studio Code FAQ*. URL: https://code.visualstudio.com/docs/supporting/FAQ. (accessed: 16.11.2020).

[Ope] OpenCV. *About Open CV*. URL: https://opencv.org/about/. (accessed: 25.09.2020).

[Ove] Overleaf. *About us*. URL: https://www.overleaf.com/about. (accessed: 20.11.2020).

[stu] Seeed studio. *Mecanum wheeled robot*. URL: https://seeeddoc.github.io/4WD_Mecanum_Wheel_Robot_Kit_Series/. (accessed: 24.09.2020).

[Wika] Wikipedia. *Canny edge detector*. URL: https://en.wikipedia.org/wiki/Canny_edge_detector. (accessed: 26.09.2020).

[Wikb] Wikipedia. *Context switch*. URL: https://en.wikipedia.org/wiki/Context_switch. (accessed: 24.09.2020).

[Wikc] Wikipedia. *Global interpreter lock*. URL: https://en.wikipedia.org/wiki/Global_interpreter_lock. (accessed: 19.09.2020).

[Wikd] Wikipedia. *Java (programming language)*. URL: https://en.wikipedia.org/wiki/Java_(programming_language). (accessed: 11.11.2020).

[Wike] Wikipedia. *Literate Programming approach*. URL: https://en.wikipedia.org/wiki/Literate_programming. (accessed: 27.09.2020).

[Wikf] Wikipedia. *Multiprocessing*. URL: https://en.wikipedia.org/wiki/Multiprocessing. (accessed: 24.09.2020).

[Wikg] Wikipedia. *Multithreading (computer architecture)*. URL: https://en.wikipedia.org/wiki/Multithreading_(computer_architecture). (accessed: 24.09.2020).

[Wikh] Wikipedia. *Pygame*. URL: https://en.wikipedia.org/wiki/Pygame. (accessed: 05.11.2020).

[Wiki] Wikipedia. *Python (programming language)*. URL: https://en.wikipedia.org/wiki/Python_(programming_language). (accessed: 19.11.2020).

[Xu] Liyan Xu. *Summary of Multithreading and Multiprocessing in Java and Python*. URL: https://liyanxu.blog/2019/04/19/summary-of-multithreading-and-multiprocessing-in-java-python/. (accessed: 24.09.2020).

Appendix

A Roboclaw Manual

RoboClaw Series Brushed DC Motor Controllers

RoboClaw Solo
RoboClaw 2x5A
RoboClaw 2x7A
RoboClaw 2x15A
RoboClaw 2x30A
RoboClaw 2x45A
RoboClaw 2x45A ST
RoboClaw 2x60A
Roboclaw 2x60HV
Roboclaw 2x120A
Roboclaw 2x160A
Roboclaw 2x200A

User Manual

Firmware 4.1.34 and Newer
Hardware V3, V4, V5, V6 and V7
User Manual Revision 5.7

Liability Statement

By purchasing and using Basicmicro's products you acknowledge and agree to the following: Basicmicro has no liabilities (and, to the Basicmicro's knowledge, there is no basis for any present or future action against the company giving rise to any liability) arising out of any injury to individuals or property as a result of ownership, possession or use of any product designed or sold by Basicmicro. No product from Basicmicro should be used in any medical devices and/or medical situations. No product should be used in any life support situations.

Contents

Warnings

There are several warnings that should be noted before getting started. Damage can easily result by not properly wiring RoboClaw. Harm can also result by not properly planning emergency situations. Any time mechanical movement is involved the potential for harm is present. The following information can help avoid damage to RoboClaw, connected devices and help reduce the potential for harm or bodily injury.

Disconnecting the negative power terminal is not the proper way to shut down a motor controller. Any connected I/O to RoboClaw will create a ground loop and cause damage to RoboClaw and attached devices. Always disconnect the positive power lead first.

Brushed DC motors are generators when spun. A robot being pushed or coasting can create enough voltage to power RoboClaws logic intermittenly creating an unsafe state. Always stop the motors before powering down RoboClaw.

RoboClaw has a minimum power requirement. Under heavy loads, without a logic battery, brownouts can happen. This will cause erratic behavior. A logic battery should be used in these situations.

Never reverse the main battery wires, Roboclaw will be permenantly damaged.

Never disconnect the motors from RoboClaw when under power. Damage will result.

Introduction

Motor Selection

When selecting a motor controller several factors should be considered. All DC brushed motors will have two current ratings, maximum stall current and continuous current. The most important rating is the stall current. Choose a model that can support the stall current of the motor selected to insure the motor can be driven properly without damage to the motor controller.

Stall Current

A motor at rest is in a stall condition. This means during start up the motors stall current will be reached. The loading of the motor will determine how long maximum stall current is required. A motor that is required to start and stop or change directions rapidly but with light load will still require maximum stall current often.

Running Current

The continuous current rating of a motor is the maximum current the motor can run without overheating and eventually failing. The average running current of the motor should not excede the continuous current rating of the motor.

Shut Down

To shut down a motor controller the positive power connections should be removed first after the motors have stopped moving. Powering off in an emergency, a properly sized switch or contactor can be used. A path to ground for regeneration energy to return to the battery should always be provided. This can be accomplish by using a power diode with proper ratings to provide a path across the switch or contactor when in an open circuit state.

Run Away

During development of your project caution should be taken to avoid run away conditions. The wheels of a robot should not be in contact with any surface until all development is complete. If the motor is embedded, ensure you have a safe and easy method to remove power from RoboClaw as a fail safe.

Wire Lengths

Wire lengths to the motors and from the battery should be kept as short as possible. Longer wires will create increased inductance which will produce undesirable effects such as electrical noise or increased current and voltage ripple. The power supply/battery wires must be as short as possible. They should also be sized appropriately for the amout of current being drawn. Increased inductance in the power source wires will increase the ripple current/voltage at the RoboClaw which can damage the filter caps on the board or even causing voltage spikes over the rated voltage of the Roboclaw, leading to board failure.

Power Sources

A battery is recommended as the main power source for the motor controller. Some power supplies can also be used without additional hardware if they have built in voltage clamps or if used with very low current motors. Most Linear and Switching power supplies are not capable of handling the regeneration energy generated by DC motors. Switching power supplies will momentarily reduce voltage and/or shut down, causing brown outs which will leave the controller in an unsafe state. The RoboClaw's minimum and maximum voltage levels can be set to prevent some of these voltage spikes, however this will cause the motors to brake when slowing down in an attempt to reduce the over voltage spikes. This will also limit power output when accelerating motors or when the load changes to prevent undervoltage conditions. Voltage clamp solutions may be required for higher power motors when using power supplies.

Logic Power

When powering external devices from RoboClaw ensure the maximum BEC output rating is not exceeded. This can cause RoboClaw to suffer logic brown out which will cause erratic behavior. Some low quality encoders can cause excessive noise on the +5VDC rail of the RoboClaw. This excessive noise will cause unpredictable behavior.

Encoders

RoboClaw features dual channel quadrature/absolute decoding. When wiring encoders make sure the direction of spin is correct to the motor direction. Incorrect encoder connections can cause a run away state. Refer to the encoder section of this user manual for proper setup.

Data Sheets

Please refer to the data sheet for the specific model of RoboClaw being used. The data sheet contains information specific to each model of RoboClaw. This manual is a general overview of RoboClaw usage and does not contain detailed information such as pinouts for each model of RoboClaw.

Getting Started

Initial Setup

RoboClaw offers several methods of control. Each control scheme has several configuration options. The following quick start guide covers the basic initialization of RoboClaw. Most control schemes require very little configuration. The control options are covered in detail in this manual. For model specific information please refer to the data sheet for each model of RoboClaw. The following is a basic setup procedure.

1. Read the Introduction and Hardware Overview sections of this manual. It is important to ensure the RoboClaw model chosen is rated to drive the selected motors. RoboClaw must be paired by the motor stall current ratings. Not running current.

2. Before configuring RoboClaw. Make sure a reliable power source is available such as a fully charged battery. See Wiring section of this manual for proper wiring instructions.

3. The RoboClaw's main modes can be configured using Motion Studio or on-board buttons. Motion Studio is the preferred method of configuration with additional options not available using the on-board buttons. However these additional options are not critical to RoboClaw's basic operation. This manual covers both configuration methods.

4. Once the configuration is complete see Wiring section of this manual. The basic wiring diagram should only be used for basic testing purposes. The Safety Wiring diagram is recommended for safe and reliable operation.

Encoder Setup

RoboClaw supports several encoder types. All encoders require tunning to properly pair with the selected motors. The auto tune function can automatically tune for most all combinations. However some manual adjustment maybe required. The final auto tune settings can be adjusted for optimal performance.

1. Once Initial Setup is complete attach an encoder to your motor and wire as shown in the encoder section of this manual. Make sure the encoder can be powered from a 5VDC power source.

2. After the encoder is wired double check the wiring. Then proceed to the auto tune function in the Encoder section of this manual.

3. Auto tune will work in most all cases. Some manual tweaks may be necessary. If additional assistance is required contact support at support@basicmicro.com

Hardware Overview

I/O

RoboClaw's I/O is setup to interface to both 5V and 3.3V logic. This is accomplished by internally current limiting and clipping any voltages over 3.3V. RoboClaw outputs 3.3V which will work with any 5V or 3.3V logic. This is also done to protect the I/O from damage.

Headers

RoboClaws share the same header and screw terminal pinouts across models in this user manual. The main control I/O are arranged for easy connectivity to control devices such as R/C controllers. The headers are also arranged to provide easy access to ground and power for supplying power to external controllers. See the specific model of RoboClaw's data sheet for pinout details.

Control Inputs

S1, S2, S3, S4 and S5 are setup for standard servo style headers I/O(except on ST models), +5V and GND. S1 and S2 are the control inputs for serial, analog and RC modes. S3 can be used as a flip switch input when in RC or Analog modes. Additionally, in all modes, S3, S4 and S5 can be used as emergency stop inputs or as voltage clamp control outputs. When set as E-Stop inputs they are active when pulled low and have internal pullups so they will not accidentally trip when left floating. S4 and S5 can also optionally be used as home signal inputs. The pins closest to the board edge are the I/0s, center pin is the +5V and the inside pins are ground. Some RC receivers have their own supply and will conflict with the RoboClaw's 5v logic supply. It may be necessary to remove the +5V pin from the RC receivers cable in those cases.

Encoder Inputs

EN1 and EN2 are the inputs from the encoders on pin header versions of RoboClaw. 1B, 1A, 2B and 2A are the encoders inputs on screw terminal versions of RoboClaw. Channel A of both EN1 and EN2 are located at the board edge on the pin header. Channel B pins are located near the heatsink on the pin header. The A and B channels are labeled appropriately on screw terminal versions.

When connecting the encoder make sure the leading channel for the direction of rotation is connected to A. If one encoder is backwards to the other you will have one internal counter counting up and the other counting down. Refer to the data sheet of the encoder you are using for channel direction. Which encoder is used on which motor can be swapped via a software setting.

Logic Battery (LB IN)

The logic side of RoboClaw can, optionally, be powered from a secondary battery wired to LB IN. The positive (**+**) terminal is located at the board edge and ground (**-**) is the inside pin closest to the heatsink. On older RoboClaws that have it, remove the LB-MB jumper if a secondary battery for logic will be used.

BEC Source (LB-MB)

RoboClaw logic requires 5VDC which is provided from the on board BEC circuit. On older model RoboClaws the BEC source input is set with the LB-MB jumper. Install a jumper on the 2 pins labeled LB-MB to use the main battery as the BEC power source. Remove this jumper if using a separate logic battery. On models without this jumper the power source is selected automatically.

Encoder Power (+ -)

The pins labeled + and - are the source power pins for encoders. The positive (+) is located at the board edge and supplies +5VDC. The ground (-) pin is near the heatsink. On ST models all power must come from the 5V screw terminals and the GND screw terminals.

Main Battery Screw Terminals

The main power input can be from 6VDC to 34VDC on a standard RoboClaw and 10.5VDC to 60VDC on an HV (High Voltage) RoboClaw. The connections are marked **+** and **-** on the main screw terminal. The plus (**+**) symbol marks the positive terminal and the negative (**-**) marks the negative terminal. The main battery wires should be as short as possible.

Do not reverse main battery wires. Roboclaw will be permenantly damaged.

Main Battery Disconnect

The main battery should have a disconnect in case of a run away situation and power needs to be cut. The switch must be rated to handle the maximum current and voltage from the battery. This will vary depending on the type of motors and or power source you are using. A typically solution would be an inexpensive contactor which can be sourced from sites like Ebay. A power diode rated for the maximum current the battery will deliver should be placed across the switch/contactor to provide a path back to the battery when disconnected while the motors are spinning. The diode will provice a path back to the battery for regenerative power even if the switch is opened. The diode should be rated for 1/10th the maximum current expected.

Motor Screw Terminals

The motor screw terminals are marked with M1A / M1B for channel 1 and M2A / M2B for channel 2. For both motors to turn in the same direction the wiring of one motor should be reversed from the other in a typical differential drive robot. The motor and battery wires should be as short as possible. Long wires can increase the inductance and therefore increase potentially harmful voltage spikes.

Easy to use Libraries

Source code and Libraries are available on the Basicmicro website. Libraries are available for Arduino(C++), C# on Windows(.NET) or Linux(Mono) and Python(Raspberry Pi, Linux, OSX, etc).

Motion Studio Overview

Motion Studio

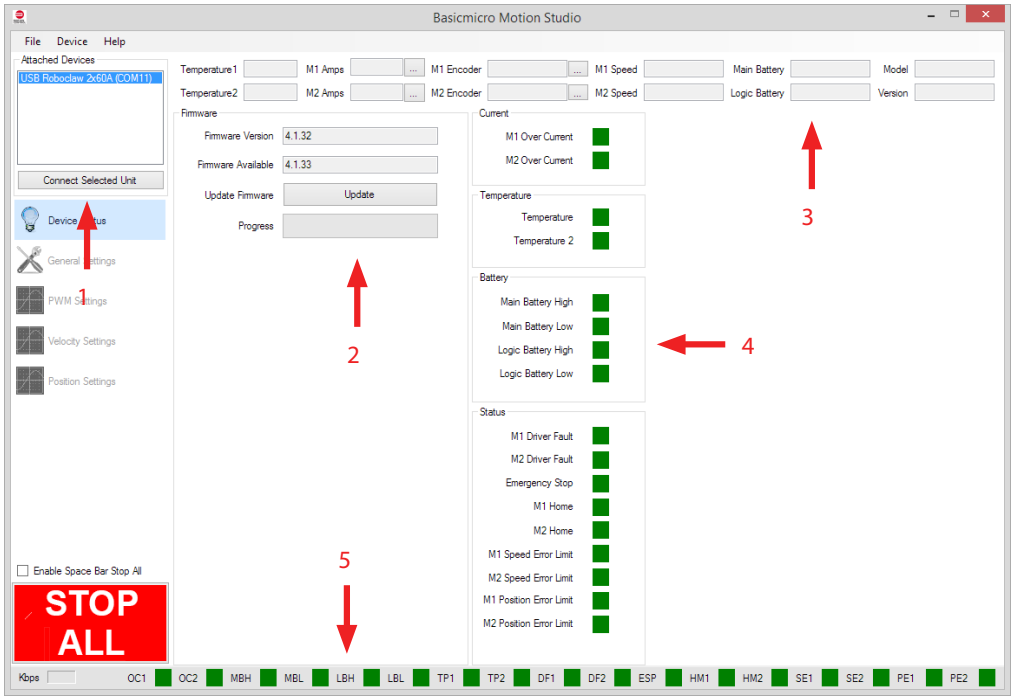
The BasicMicro Motion Studio software suite is design to configure, monitor and maintain RoboClaw. It's used to configure all the available RoboClaw modes and options. Motion Studio can be used to monitor and control RoboClaw. It can be download from https://www.basicmicro.com. Once installed, each time Motion Studio is run it will check for the latest version online.

Connection

This is the first screen shown when first running Motion Studio. From this screen you can select a detected RoboClaw and connect (1). More than one RoboClaw can be connected at a time. Box (1) is where the desired RoboClaw is selected.

After the RoboClaw is detected and it's firmware version is checked (2), if a newer firmware version is available it can be updated by clicking the Update Firmware button (2).

Fields (3,4,5) display current values and status. The fields at the top of the screen (3) show the current value of each monitored parameter and are updated live once a RoboClaw is connected. Status indicators (4,5) indicate the current condition of the named monitor parameter. Green indicates operationing within the defined parameter. Yellow indicates a warning. Red indicates a fault.



Device Status

Once a RoboClaw is connected, the connection screen becomes active (1) and is now the Device Status screen. All status indicators (3,4) and monitored parameter fields (2) will update to reflect the current status and values of the connected RoboClaw.

When a RoboClaw is connected the Stop All (5) button becomes active. There is a small check box to activate the Stop All function by using the space bar on the keyboard. This is a safety feature and is the quickest method to stop all motor movements when using Motion Studio.

Device Status Screen Layout

Label	Function	Description
1	Window Selection	Used to select which settings or testing screen is currently displayed.
2	Monitored Parameters	Displays continuously updated status parameters.
3	Status Indicators	Displays current warnings and faults.
4	Status Indicators	Displays abbreviated status of warnings and faults. Visible at all times.
5	Stop All	Stops all motion. Can activate from keyboard space bar.

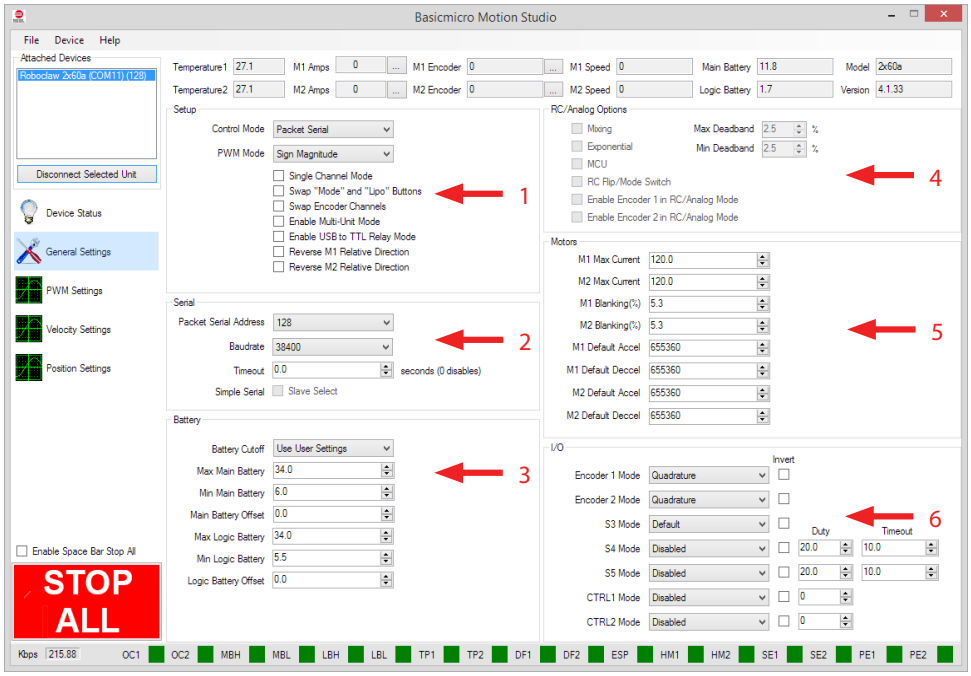
Status Indicator (4)

The status indicators shown at the bottom of the screen are an abbreviated duplication of the main status indicators shown on the device status screen.

Label	Description
M1OC	Motor 1 over current.
M2OC	Motor 2 over current.
MBHI	Main battery over voltage.
MBLO	Main battery under voltage.
LBHI	Logic battery over voltage.
LBLO	Logic battery under voltage.
TMP1	Temperature 1
TMP2	Optional temperature 2 on some RoboClaw models.
M1DF	Motor driver 1 fault.
M2DF	Motor driver 2 fault.
ESTP	Emergency stop. When active.
M1HM	Motor 1 homed or limit switch active. When option in use.
M2HM	Motor 2 homed or limit switch active. When option in use.
SE1	Motor 1 speed error limit
SE2	Motor 2 speed error limit
PE1	Motor 1 position error limit
PE2	Motor 2 position error limit

General Settings

The general settings screen can be used to configure RoboClaw. This includes modes, mode options and monitored parameters. For detailed explanations see the Configuration with Motion Studio section of this manual.



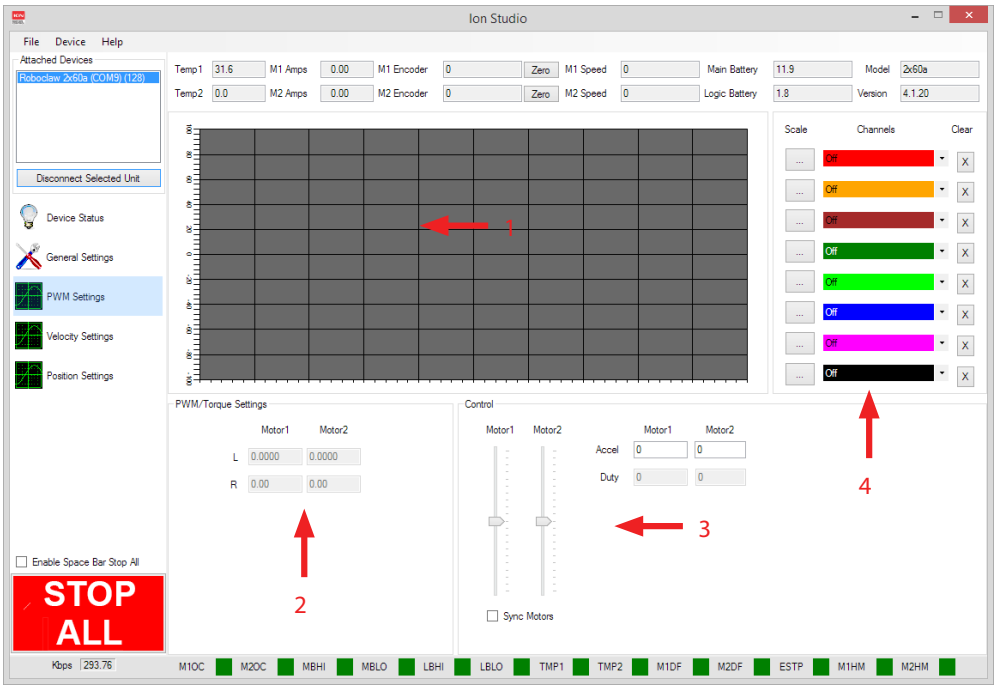
Configuration Options

Each control mode will have several configuration options. Some options will appear grayed out to indicate the option is not available for the selected mode. Settings are only stored in RAM until the user writes the settings to the RoboClaw's non-volatile memory. Select Write Settings under Device in the menu bar.

Label	Function	Description
1	Setup	Main confirguration options and main control mode selection drop down.
2	Serial	Settings for serial modes. Set packet address, baudrate and slave select.
3	Battery	Voltage setting options for main battery and logic batteries.
4	RC/Analog Options	Configure RC and Ananlog control options.
5	Motors	Motor current, accel and deccel settings.
6	I/O	Set encoder input type. Set S3, S4 and S5 configuration options. Enabling output pins on certain models of RoboClaw.

PWM Settings

The PWM settings screen is used to control RoboClaw for testing. Sliders are provided to control each motor channel. This screen can also be used to determine the QPPS of attached encoders.



(1) Graph

Function	Description
Grid	Displays channel data with 100mS update rate and one second horizontal divisions.

(2) PWM/Torque Settings

Function	Description
L	MCP only. Motor Inductance in Henries.
R	MCP only. Motor resistance in Ohms.

(3) Control

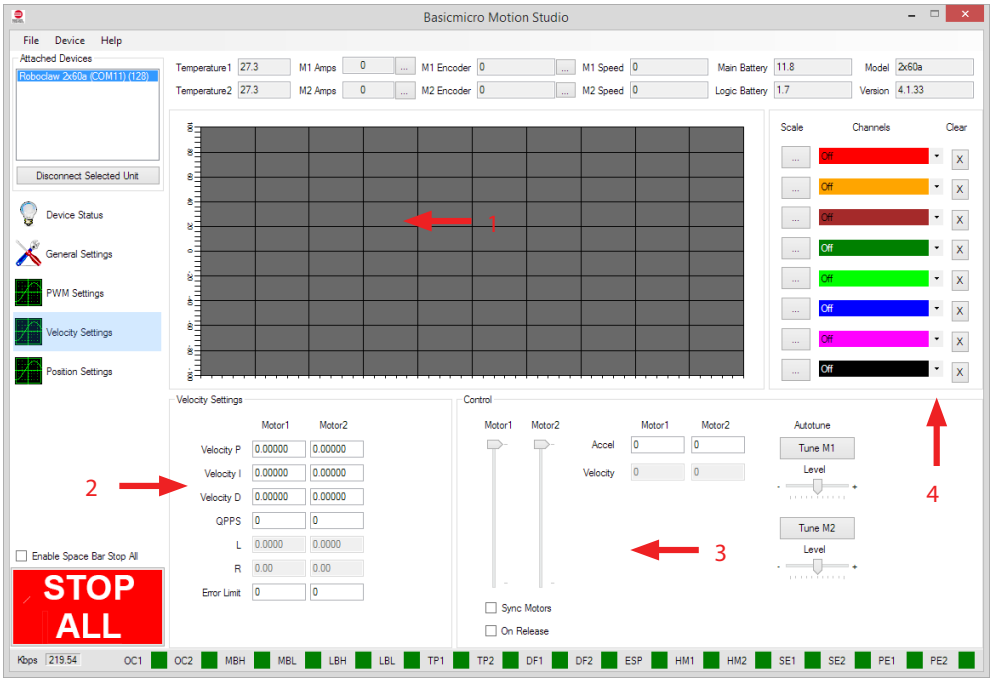
Function	Description
Motor 1	Controls motor 1 duty percentage forward and reverse.
Motor 2	Controls motor 2 duty percentage forward and reverse.
Sync Motors	Synchronises Motor 1 and Motor 2 Sliders.
Accel	Acceleration rate used when moving the sliders.
Duty	Displays the numberic value of the motor slider in 10ths of a Percent (0 to +/-1000).

(4) Graph Channels

Function	Description
Scale	Sets vertical scale to fit the range of the specified Channel.
Channels	Select data to display on the channel. The channel is graphed in the color shown. Channel options: • M1 or M2 Setpoint - User input for channel • M1 or M2 PWM - Motor PWM output • M1 or M2 Speed - Motors Encoder Velocity • M1 or M2 ISpeed - Motor instantaneous speed (1/300 second) • M1 or M2 Speed Error - Motor speed error relative to target speed • M1 or M2 Position - Motors Encoder Position • M1 or M2 Position Error - Motor position error relative to target position • M1 or M2 Current - Motor running current • Temperature • Main Battery Voltage • Logic Battery Voltage
Clear	Clears channels graphed line.

Velocity Settings

The Velocity settings screen is used to set the encoder and PID settings for speed control. The screen is also used for testing and plotting.



(1) Graph

Function	Description
Grid	Displays channel data with 100mS update rate and one second horizontal divisions.

(2) Velocity Settings

Function	Description
Velocity P	Proportional setting for PID.
Velocity I	Integral setting for PID.
Velocity D	Differential setting for PID.
QPPS	Maximum speed of motor using encoder counts per second.
L	MCP only. Motor Inductance in Henries.
R	MCP only. Motor resistance in Ohms.
Error Limit	Maximum allow velocity error.

(3) Control

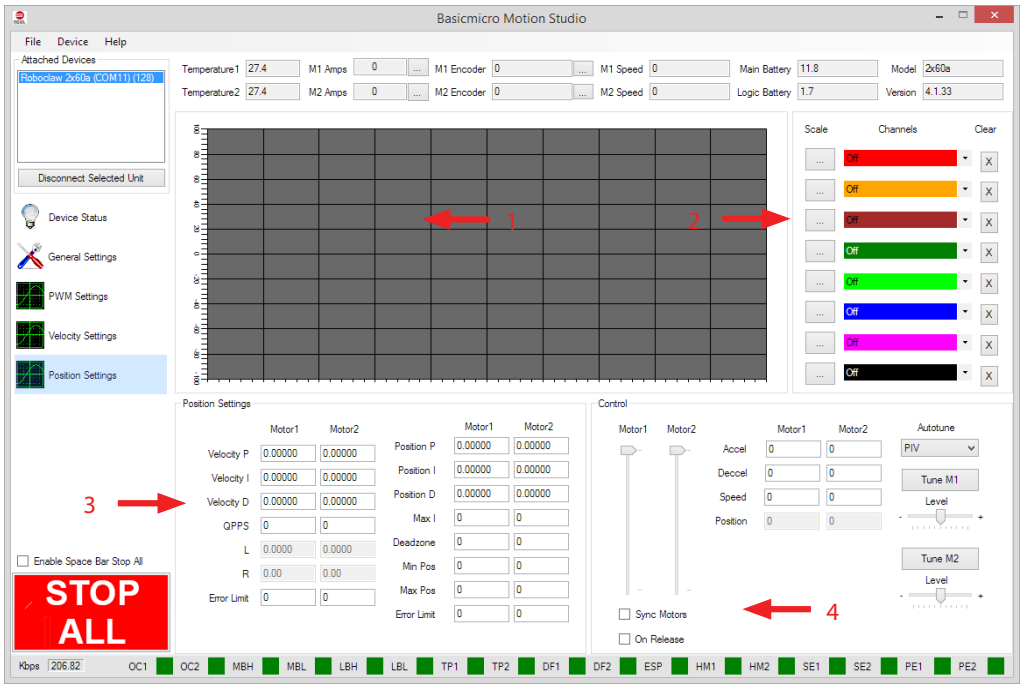
Function	Description
Motor 1	Motor 1 velocity control (0 to +/- maximum motor speed).
Motor 2	Motor 2 velocity control (0 to +/- maximum motor speed).
Sync Motors	Synchronises Motor 1 and Motor 2 Sliders.
On Release	Will not update new speed until the slider is released.
Accel	Acceleration rate used when moving the sliders.
Velocity	Shows the numeric value for the sliders current position.
Tune M1	Start motor 1 velocity auto tune.
Level	Adjust auto tune 1 values agressiveness. Sllide left for softer control.
Tune M2	Start motor 2 velocity auto tune.
Level	Adjust auto tune 2 values agressiveness. Sllide left for softer control.

(4) Graph Channels

Function	Description
Scale	Sets vertical scale to fit the range of the specified Channel.
Channels	Select data to display on the channel. The channel is graphed in the color shown. Channel options: • M1 or M2 Setpoint - User input for channel • M1 or M2 PWM - Motor PWM output • M1 or M2 Speed - Motors Encoder Velocity • M1 or M2 ISpeed - Motor instantaneous speed (1/300 second) • M1 or M2 Speed Error - Motor speed error relative to target speed • M1 or M2 Position - Motors Encoder Position • M1 or M2 Position Error - Motor position error relative to target position • M1 or M2 Current - Motor running current • Temperature • Main Battery Voltage • Logic Battery Voltage
Clear	Clears channels graphed line.

Position Settings

The Position settings screen is used to set the encoder and PID settings for position control. The screen is also used for testing and plotting.



(1) Graph

Function	Description
Grid	Displays channel data with 100mS update rate and one second horizontal divisions.

(2) Graph Channels

Function	Description
Scale	Sets vertical scale to fit the range of the specified Channel.
Channels	Select data to display on the channel. The channel is graphed in the color shown. Channel options: • M1 or M2 Setpoint - User input for channel • M1 or M2 PWM - Motor PWM output • M1 or M2 Velocity - Motors Encoder Velocity • M1 or M2 Position - Motors Encoder Position • M1 or M2 Current - Motor running current • Temperature • Main Battery Voltage • Logic Battery Voltage
Clear	Clears channels graphed line.

(3) Position Settings

Function	Description
Velocity P	Proportional setting for velocity PID.
Velocity I	Integral setting for velocity PID.
Velocity D	Differential setting for velocity PID.
QPPS	Maximum speed of motor using encoder counts per second.
L	MCP only. Motor Inductance in Henries.
R	MCP only. Motor resistance in Ohms.
Position P	Proportional setting for position PID.
Position I	Integral setting for position PID.
Position D	Differential setting for position PID.
Max I	Maximum integral windup limit.
Deadzone	Zero position deadzone. Increases the "stopped" range.
Min Pos	Minimum encoder position.
Max Pos	Maximum encoder position.

(4) Control

Function	Description
Motor 1	Motor 1 velocity control (0 to +/- maximum motor speed).
Motor 2	Motor 2 velocity control (0 to +/- maximum motor speed).
Sync Motors	Synchronises Motor 1 and Motor 2 Sliders.
On Release	Will not update new speed until the slider is released.
Accel	Acceleration rate used when moving the sliders.
Deccel	Decceleration rate used when moving the sliders.
Speed	Speed to use with slide move.
Position	Numeric value of slider motor position.
Autotune	Method used. PD = Proportional and Differential. PID = Proportional Differential and Integral. PIV = Cascaded Velocity PD + Position P.
Tune M1	Start motor 1 velocity auto tune.
Level	Adjust auto tune 1 values agressiveness. Sllide left for softer control.
Tune M2	Start motor 2 velocity auto tune.
Level	Adjust auto tune 2 values agressiveness. Sllide left for softer control.

Firmware Updates

Motion Studio Setup

Download and install the Motion Studio application. Win7 or newer is required. When opening Motion Studio, it will check for updates and search for a USB Windows Driver to verify installation. If the USB driver is not found, Ion Studio will install it.

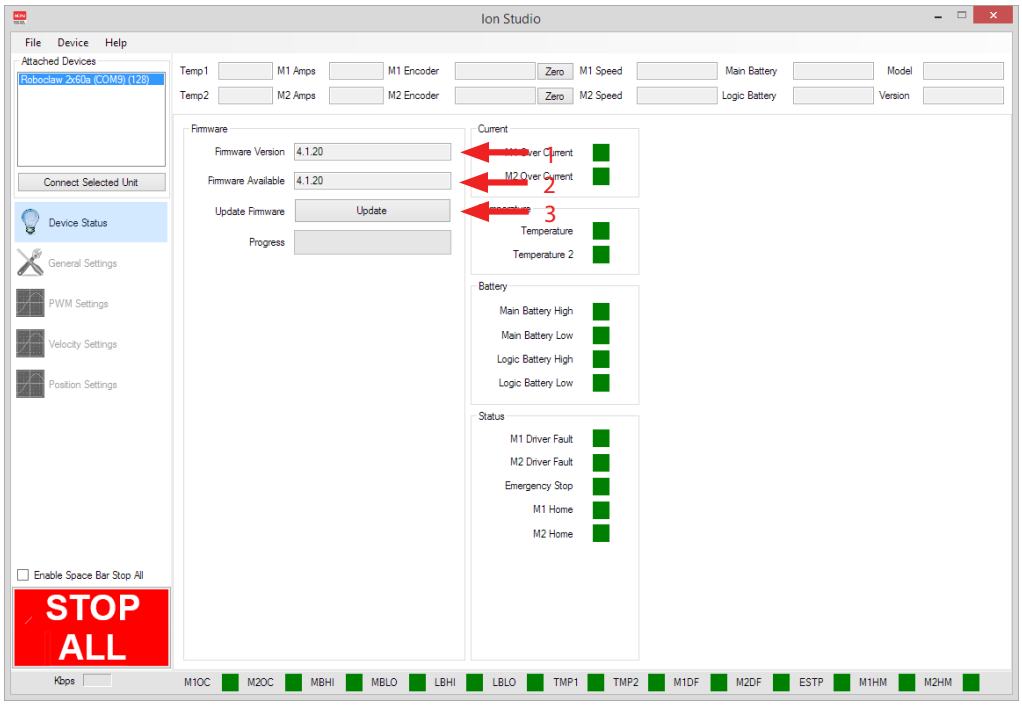
1. Open the Motion Studio application.

2. Apply a reliable power source such as a fully charge battery to power the motor controller.

3. Connect the powered motor controller to a USB port on your computer with Motion Studio already open.

Firmware Update

Once Motion Studio detects the motor controller it will display the current firmware version in the Firmware Version field (1). Each time Motion Studio is started it will check for a new version of its self which will always include new firmware. If an update is required Motion Studio will download the latest version and display it in the firmware available field (2).



1. When a new version of firmware is shown click the update button (3) to start the process.

2. Motion Studio will begin to update the firmware. While the firmware update is in progress the onboard LEDs will begin to flash. The onboard flash memory will first be erased.

3. Once the firmware update is complete the motor controller will reset. Click the "Connect Selected Unit" button to re-connect.

Control Modes

Setup

RoboClaw has several functional control modes. There are two methods to configure these modes. Using the built-in buttons or Motion Studio. This manaul covers both methods of configuration. Motion Studio offers greater options for each mode and can be easier to configure the RoboClaw in several situations. However the built-in buttons are more than adequate in most all modes. Refer to the configuration section of this manual for mode setup instructions using Motion Studio or the built-in buttons.

There are 4 main modes with several variations. Each mode enables RoboClaw to be controlled in a very specific way. The following list explains each mode and the ideal application.

USB Control

USB can be used in any mode. When RoboClaw is in packet serial mode and another device, such as an Arduino, is connected commands from the USB and Arduino will be executed and can potentially over ride one another. However if Roboclaw is not in packet serial mode, motor movement commands will not function. USB packet serial commands can then only be used to read status information and set configuration settings.

RC

Using RC mode RoboClaw can be controlled from any hobby RC radio system. RC input mode also allows low powered microcontrollers such as a Basic Stamp to control RoboClaw. RoboClaw expects servo pulse inputs to control the direction and speed. Very similar to how a regular servo is controlled. RC mode can use encoders if properly setup(See Encoder section).

Analog

Analog mode uses an analog signal from 0V to 2V to control the speed and direction of each motor. RoboClaw can be controlled using a potentiometer or filtered PWM from a microcontroller. Analog mode is ideal for interfacing RoboClaw with joystick positioning systems or other non microcontroller interfacing hardware. Analog mode can use encoders if properly setup(See Encoder section).

Simple Serial

In simple serial mode RoboClaw expects TTL level RS-232 serial data to control direction and speed of each motor. Simple serial is typically used to control RoboClaw from a microcontroller or PC. If using a PC, a MAX232 or an equivilent level converter circuit must be used since RoboClaw only works with TTL level inputs. Simple serial includes a slave select mode which allows multiple RoboClaws to be controlled from a signal RS-232 port (PC or microcontroller). Simple serial is a one way format, RoboClaw can only receive data. Encoders are not supported in Simple Serial mode.

Packet Serial

In packet serial mode RoboClaw expects TTL level RS-232 serial data to control direction and speed of each motor. Packet serial is typically used to control RoboClaw from a microcontroller or PC. If using a PC a MAX232 or an equivilent level converter circuit must be used since RoboClaw only works with TTL level input. In packet serial mode each RoboClaw is assigned a unique address. There are 8 addresses available. This means up to 8 RoboClaws can be on the same serial port. Encoders are support in Packet Serial mode(See Encoder section).

Configuration Using Motion Studio

Mode Setup

Download and install the Motion Studio application from https://www.basicmicro.com. A PC with Windows 7 or newer is required. Motion Studio will check for a newer version each time it is ran. It will then search for the USB RoboClaw Windows Driver to verify installation. If the USB driver is not found Ion Studio will install it.

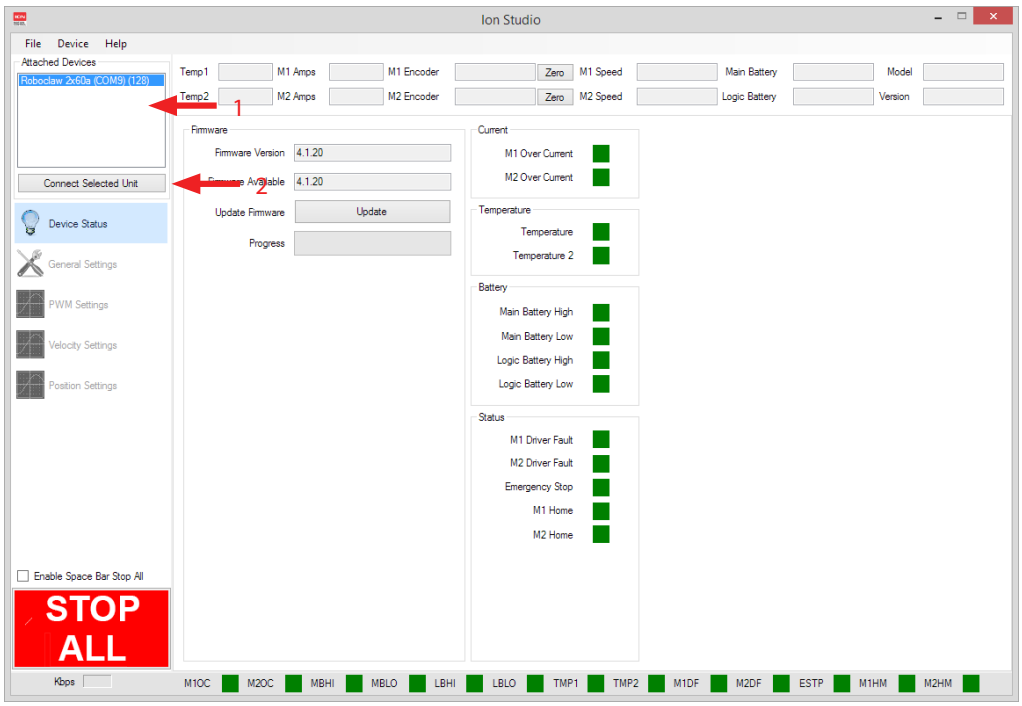
1. Open the Motion Studio application.

2. Apply a reliable power source such as a fully charged battery to power up RoboClaw.

3. Connect the powered RoboClaw to a USB port on your computer with Motion Studio already open. The RoboClaw USB driver may need to be installed before Motion Studio will automatically handling installing the required driver.

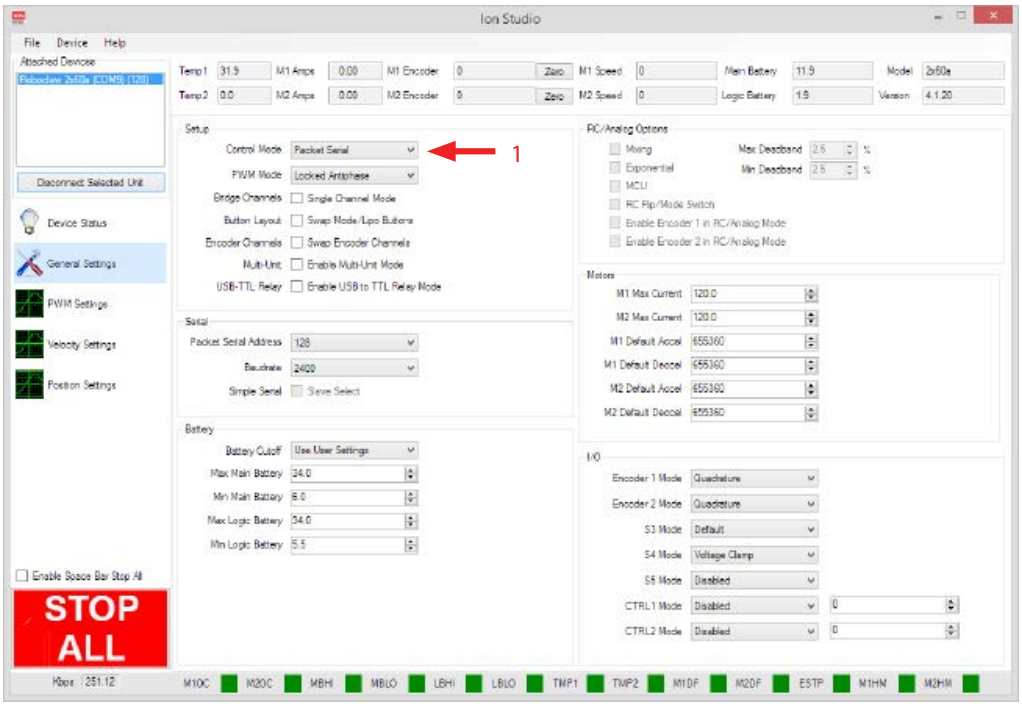
4. When RoboClaw is detected, it will appear in the Attached Device window (1).

5. Once RoboClaw appears in the Attached Device window (1), click the connect button (2).



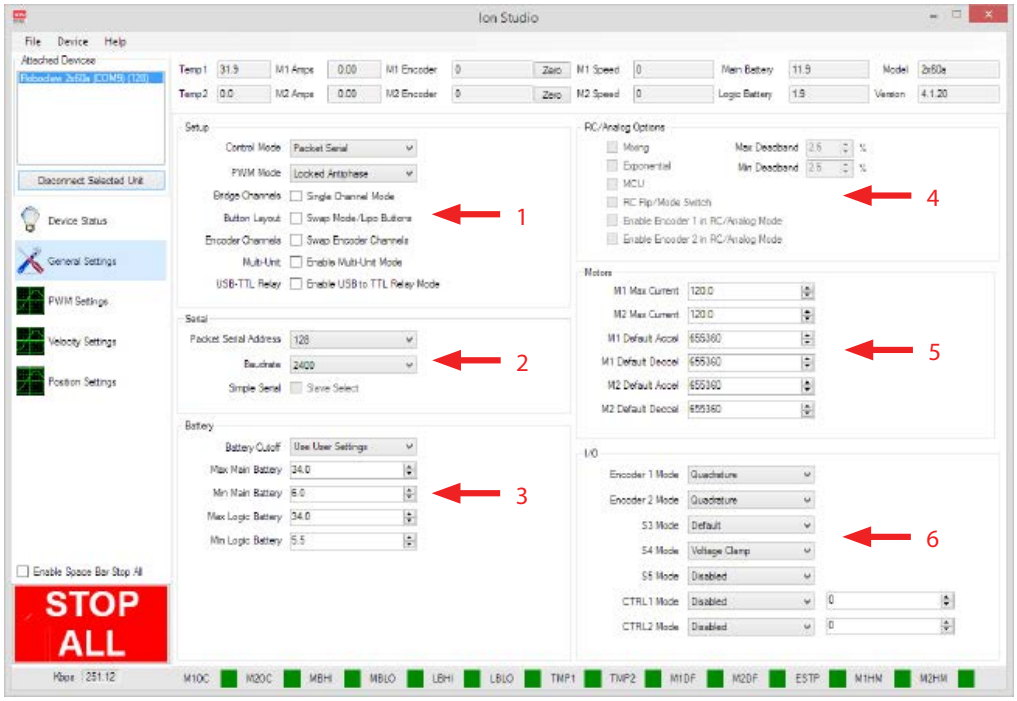
Control Mode Setup

Select the Control Mode drop down (1). There are 4 main modes. See the Control Modes section of this manual for a detailed explanation of each available mode.



Control Mode Options

The general settings screen is used to configure RoboClaw. Each control mode will have several configuration options. Grayed out options are not available for the selected mode. Once all settings are configured they must be saved to RoboClaw. This is done by selecting Write Settings from the Device menu in the menu bar.



(1) Setup
Main drop down for setting the control modes and confirguration options.

Function	Description
Control Mode	Drop down to set main control mode. Some options may grey out if not available in the selected mode.
PWM Mode	Drop down to set the main MOSFET driving scheme. This option should never be change but in rare circumstances.
Bridge Channels	Used to bridge motor channe 1 and 2. This option must be set before physically bridging the channels. Or damage will result.
Button Layout	Swaps Mode and LIPO button interface. Only affects hardware V5 and RoboClaw 2x15, 2x30 and 2x45.
Encoder Channels	This option will swap encoder channels. Pair encoder 1 to motor channel 2 and encoder 2 to motor channel 1.
Multi-Unit	Sets S2 pin to open drain. Allows multiple Roboclaws to be controlled from a single serial port.
USB-TTL Relay	Enables RoboClaw to pass data from USB through S1 (RX) and S2 (TX). Allows several RoboClaws to be networked from one USB connection. All connected RoboClaw's baud rates must be set to the same.
Reverse M1 Relative Direction	Reverse the direction of motor channel 1.
Reverse M2 Relative Direction	Reverse the direction of motor channel 2.

(2) Serial
Settings for serial modes. Set packet address, baudrate and slave select.

Function	Description
Packet Serial Address	Sets RoboClaw address for packet serial mode. Allows multiple Roboclaws to be controlled from a single Serial port.
Baudrate	Sets the baudrate in all serial modes.
Simple Serial	Sets simple serial mode with slave select. Set pin S2 high to enable the attached RoboClaw. Pull S2 low and all commands will be ignored.
Timeout	Sets the time in seconds before motors will stop if serial communication stops. Default is disabled (0.0 seconds).

(3) Battery
Main and logic battery voltage settings. Sets cut off and protection limits.

Function	Description
Battery Cut Off	Sets main battery cut off based on LiPo cell count. Can also be set to auto detect or User Settings for manual configuration. Auto detect requires a properly charged battery. User Settings allows editing of the voltage values manually. See Battery Settings.
Max Main Battery	Sets main battery maximum voltage. If the main battery voltage goes above the set maximum value running motors will go into brake mode.
Min Main Battery	Sets main battery minimum voltage. If the main battery voltage falls below the set minimum value running motors will go into freewheel.
Main Battery Offset	Value is used to correct for offset error in voltage readings. Range is +/- 1V.
Max Logic Battery	Sets logic battery maximum voltage. If logic battery voltage goes above the maximum set value RoboClaw will shut down until the voltage is corrected and a reset.
Min Logic Battery	Sets logic battery minimum voltage. If logic battery voltage goes below the minimum set value RoboClaw will shut down until the voltage is corrected and a reset.
Logic Battery Offset	Value is used to correct for offset error in voltage readings. Range is +/- 1V.

(4) RC/Analog Options
Configure RC and Ananlog control options. Set control type in RC and Analog modes.

Function	Description
Mixing	Mixes S1 and S2 inputs for control of a differentially steered robot. S1 controls direction (forward / reverse) and speed. S2 controls turning left or right with speed. Simliar to how a RC car would be controlled. Turn this mode off for tank style control.
Exponential	Enable increased control range at slow speed.
MCU	Disables auto calibrate. Allows slow MCU to send R/C pulses at lower than normal R/C rates.
RC Flip/Mode Switch	R/C pulse switched. Use radio channel to toggle and change all motor direction. Used when a robot is flipped upside down to reverse steering control.
Enable Encoder 1 in RC/ AnalogMode	Enables encoder 1 to be used in RC or Analog mode. Will control motor by speed or position depending on which PID control is set. The range of speed is mapped to the RC control using the QPPS value as the maximum speed. The position range is controlled by maximum and minimum position settings.
Enable Encoder 2 in RC/ AnalogMode	Enables encoder 1 to be used in RC or Analog mode. Will control motor by speed or position depending on which PID control is set. The range of speed is mapped to the RC control using the QPPS value as the maximum speed. The position range is controlled by maximum and minimum position settings.
Max Deadband	Sets maximum range of control signal seen as 0 (Stopped).
Min Deadband	Sets minimum range of control signal seen as 0 (Stopped).

(5) Motors
Motor current limit settings. The accel and deccel settings apply to RC, Analog and commands with no Accel and Deccel arugements.

Function	Description
M1 Max Current	Sets maximum motor current for channel 1. Can not exceed RoboClaw rated peak current.
M2 Max Current	Sets maximum motor current for channel 2. Can not exceed RoboClaw rated peak current.
M1 Blanking	Sets the PWM percentage of current readings that are blanked for noise. The range is 0-20%. The default is 5.3%.
M2 Blanking	Sets the PWM percentage of current readings that are blanked for noise. The range is 0-20%. The default is 5.3%.
M1 Default Accel	Sets the ramp rate of acceleration for motor channel 1. A value of 1 to 655,360 can be used. Value of 0 sets the internal default for Accel and Deccel. Value of 655,360 equals 100mS full forward to reverse ramping rate.
M1 Default Deccel	Sets the ramp rate of decceleration for motor channel 1. A value of 1 to 655,360 can be used. Value of 0 sets the internal default for Accel and Deccel. Value of 655,360 equals 100mS full forward to reverse ramping rate.
M2 Default Accel	Sets the ramp rate of acceleration for motor channel 2. A value of 1 to 655,360 can be used. Value of 0 sets the internal default for Accel and Deccel. Value of 655,360 equals 100mS full forward to reverse ramping rate.
M2 Default Deccel	Sets the ramp rate of decceleration for motor channel 2. A value of 1 to 655,360 can be used. Value of 0 sets the internal default for Accel and Deccel. Value of 655,360 equals 100mS full forward to reverse ramping rate.

(6) I/O
Set encoder input type. Set S3, S4 and S5 configuration options. Enabling output pins on certain models of RoboClaw. Set limit, homing, voltage clamp, E-stop options.

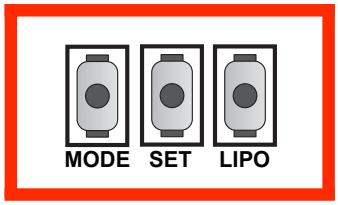
Function	Description
Encoder 1 Mode	Sets encoder type for encoder 1.
Encoder 2 Mode	Sets encoder type for encoder 2.
S3 Mode	Sets the default function for S3.
S4 Mode	Sets the default function for S4.
S5 Mode	Sets the default function for S5.
S4/S5 Duty	Sets the auto-homing duty percentage. When velocity or postion control is configured duty sets the speed percentage.
S4/S5 Timeout	Sets the auto-homing timeout in seconds.
CTRL1 Mode	Enables output pins on certain models of RoboClaw. A value of 0 to 4095 can be used to set the pin's default PWM output. Value can be changed by commands during run time.
CTRL2 Mode	Enables output pins on certain models of RoboClaw. A value of 0 to 4095 can be used to set the pin's default PWM output. Value can be changed by commands during run time.
CTRL1/2 Duty	Sets the initial output state of CTRL1 or CTRL2. Range is 0 to 4095. 0 is off and 4095 is fully on.

Configuration with Buttons

Mode Setup

The 3 buttons on RoboClaw are used to set the different configuration options. The MODE button sets the interface method such as Serial or RC modes. The SET button is used to configure the options for the mode. The LIPO button doubles as a save button and configuring the low battery voltage cut out function of RoboClaw. To set the desired mode follow the steps below.

1. Press and release the MODE button to enter mode setup. The STAT2 LED will begin to blink out the current mode. Each blink is a half second with a long pause at the end of the count. Five blinks with a long pause equals mode 5 and so on.
2. Press SET to increment to the next mode. Press MODE to decrement to the previous mode.
3. Press and release the LIPO button to save this mode to memory.



Modes

Mode	Function	Description
1	R/C mode	Control with standard R/C pulses from a R/C radio or MCU. Controls a robot like a tank. S1 controls motor 1 forward or reverse and S2 controls motor 2 forward or reverse.
2	R/C mode with mixing	Same as Mode 1 with mixing enabled. Channels are mixed for differentially steered robots (R/C Car). S1 controls forward or reverse and S2 controls left or right.
3	Analog mode	Control using analog voltage from 0V to 2V. S1 controls motor 1 and S2 controls motor 2.
4	Analog mode with mixing	Same as Mode 3 with mixing enabled. Channels are mixed for differentially steered robots (R/C Car). S1 controls forward or reverse and S2 controls left or right.
5	Standard Serial	Use standard serial communications for control.
6	Standard Serial with slave pin	Same as Mode 5 with a select pin. Used for networking. RoboClaw will ignore commands until pin goes high.
7	Packet Serial Mode - Address 0x80	Control using packet serial mode with a specific address for networking several motor controllers together.
8	Packet Serial Mode - Address 0x81	
9	Packet Serial Mode - Address 0x82	
10	Packet Serial Mode - Address 0x83	
11	Packet Serial Mode - Address 0x84	
12	Packet Serial Mode - Address 0x85	
13	Packet Serial Mode - Address 0x86	
14	Packet Serial Mode - Address 0x87	

Mode Options

Each mode will have several possible configuration settings. The settings need to be setup after the initial mode is selected. Follow the steps below.

1. After the desired mode is set and saved press and release the SET button for options setup. The STAT2 LED will begin to blink out the current option setting.
2. Press SET to increment to the next option. Press MODE to decrement to the previous option.
3. Once the desired option is selected press and release the LIPO button to save the option to memory.

RC and Analog Mode Options

Option	Function	Description
1	TTL Flip Switch	Logic level switch. Toggle to change all motor direction. Used when a robot is flipped upside down to reverse steering control.
2	TTL Flip and Exponential Enabled	Option 1 combined with increased control range at slow speed.
3	TTL Flip and MCU Enabled	Disables auto calibrate. Allows slow MCU to send R/C pulses at lower than normal R/C rates.
4	TTL Flip and Exp and MCU Enabled	Option 2 and 3 combined.
5	RC Flip Switch	R/C pulse switched. Use radio channel to toggle and change all motor direction. Used when a robot is flipped upside down to reverse steering control.
6	RC Flip and Exponential Enabled	Option 5 combined with increased control range at slow speed.
7	RC Flip and MCU Enabled	Disables auto calibrate and auto stop due to R/C signal loss. Allows slow MCU to send R/C pulses at lower than normal R/C rates.
8	RC Flip and Exponential and MCU Enabled	Option 6 and 7 combined.

Standard Serial and Packet Serial Mode Options

Option	Baud Rate	Description
1	2400bps	Standard RS-232 serial data rate.
2	9600bps	Standard RS-232 serial data rate.
3	19200bps	Standard RS-232 serial data rate.
4	38400bps	Standard RS-232 serial data rate.
5	57600bps	Standard RS-232 serial data rate.
6	115200bps	Standard RS-232 serial data rate.
7	230400bps	Standard RS-232 serial data rate.
8	460800bps	Standard RS-232 serial data rate.

Battery Cut Off Settings

The RoboClaw is able to protect the main battery by utilizing a battery voltage cut off. The cut off voltage will vary depending on the size of battery used. The table below shows the battery option setting with the type of battery it will protect and at what voltage the cutoff will kick in. The battery settings can be set by following the steps below.

1. Press and release the LIPO button. The STAT2 LED will begin to blink out the current setting.
2. Press SET to increment to the next setting. Press MODE to decrement to the previous setting.
3. Once the desired setting is selected press and release the LIPO button to save this setting to memory.

Battery Options

Option	Setting	Description
1	Disabled	6VDC is the default cut off when disabled.
2	Auto Detect	Battery must not be overcharged or undercharge! See Battery Settings.
3	3 Cell	9VDC is the cut off voltage.
4	4 Cell	12VDC is the cut off voltage.
5	5 Cell	15VDC is the cut off voltage.
6	6 Cell	18VDC is the cut off voltage.
7	7 Cell	21VDC is the cut off voltage.
8	8 Cell	24VDC is the cut off voltage.

Battery Settings

Automatic Battery Detection on Startup

Auto detect will sample the main battery voltage on power up or after a reset. All Lipo batteries, depending on cell count will have a minimum and maximum safe voltage range. The attached battery must be within this acceptable voltage range to be correctly detected. Undercharged or overcharged batteries will cause false readings and RoboClaw will not properly protect the battery. If the automatic battery detection mode is enabled using the on-board buttons, the Stat2 LED will blink to indicate the battery cell count that was detected. Each blink indicates the number of LIPO cells detected. When automatic battery detection is used the number of cells detected should be confirmed on power up.

Undercharged or overcharged batteries can cause an incorrect auto detection voltage.

Manual Voltage Settings

The minimum and maximum voltage can be set using the Motion Studio application or packet serial commands. Values can be set to any value between the boards minimum and maximum voltage limits. This feature can be useful when using a power supply to power RoboClaw. The minimum voltage should be set to 2V below the power supply voltage to protect against dips due to heavy load. The maximum voltage should be set to 2V above the power supply voltage to protect against regenerative voltage spikes. However when the minimum or maximum voltages are reached RoboClaw will go into either braking or freewheel mode. This feature will only help to protect a power supply not correct regenerative voltages issues. A voltage clamping circuit is required to correct any regenerative voltage issues when a power supply is used as the main power source. See Voltage Clamping.

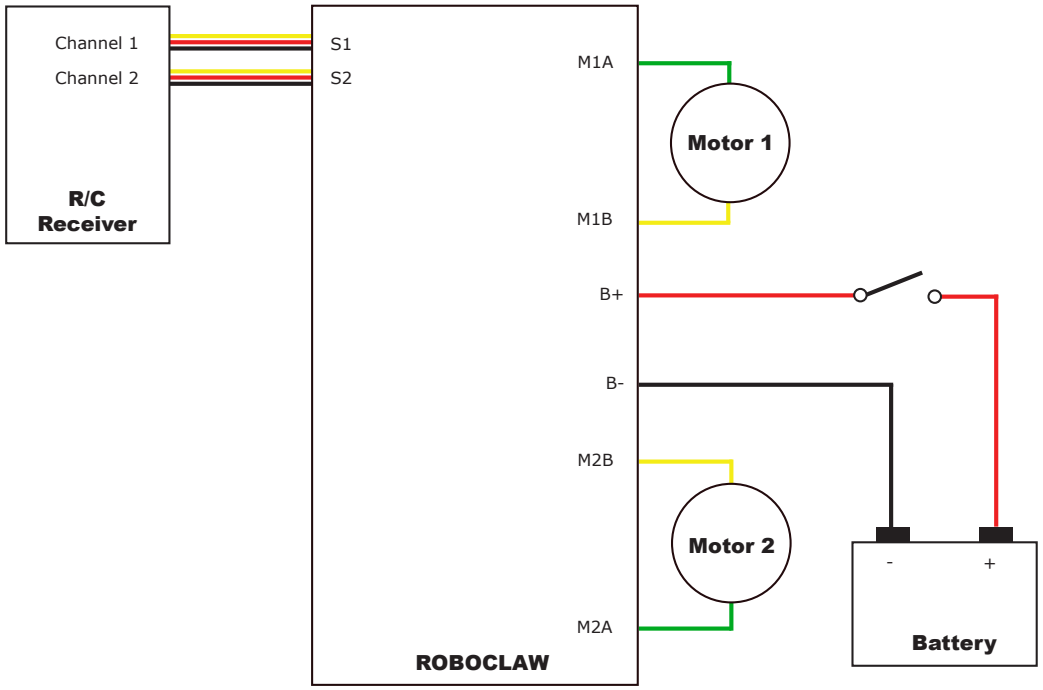
Wiring

Basic Wiring

RoboClaw has many control modes and each mode may have unique wiring requirments to ensure safe and reliable operation. The diagram below illustrates a very basic wiring configuration used in a small motor system where safety concerns are minimal. This is the most basic wiring configuration possible. Any wiring of RoboClaw should include a main battery shut off switch, even when safety concerns are minimal. Never underestimate a motorized system in an uncontrolled condition.

In addition, RoboClaw is a regenerative motor controller. If the motors are moved when the system is off, it could cause potential erratic behavior due to the regenerative voltages powering the system. A return path to the battery should always be supplied if the system can move when main power is disconnected or a fuse is blown.

For model specific pinout information please refer to the data sheet for the model being used.



Never disconnect the negative battery lead before disconnecting the positive!

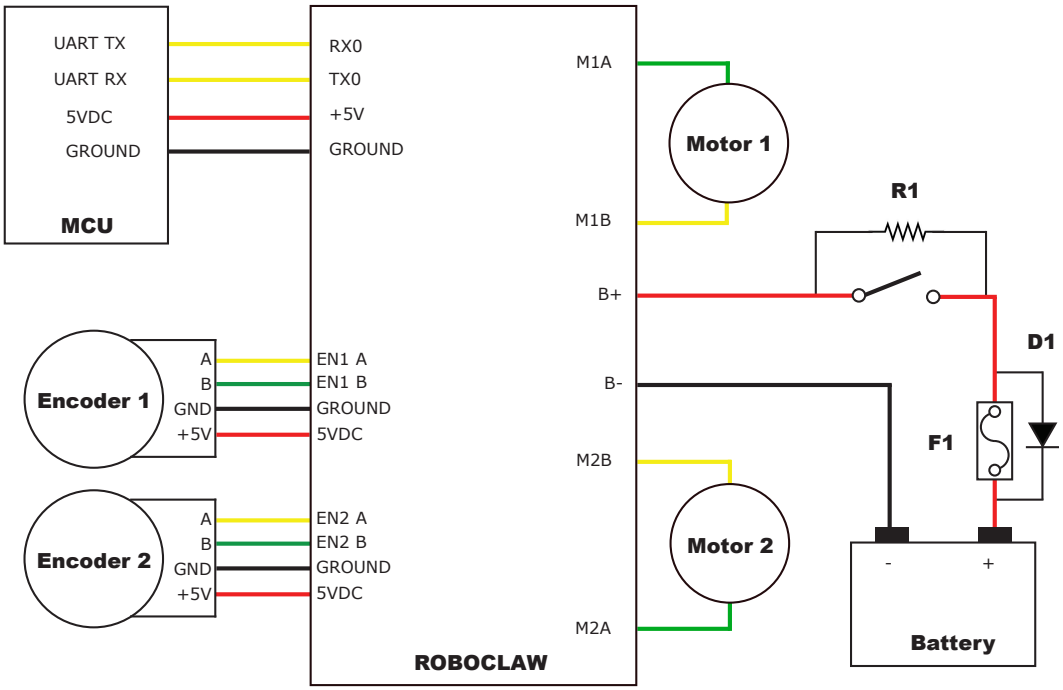
Safety Wiring

In all system with movement, safety is a concern. The wiring diagram below illustrates a system wired for minimum safety requirements. An external main power cut off is required for safety. When the RoboClaw is switched off or the fuse is blown, a high current diode (D1) is required to create a return path to the battery for any regenerative voltages. The use of a pre-charge resistor (R1) is required to avoid high inrush currents and arcing. A pre-charge resistor (R1) should be 1K, 1/2Watt for a 60VDC motor controller which will give a pre-charge time of about 15 seconds. A lower resistances can be used with lower voltages to decrease the pre-charge time.

Encoder Wiring

A wide range of sensors are supported including quadrature encoders, absolute encoders, potentiometers and hall effect sensors for closed loop operation. The encoder pins are not exclusive to supporting encoders and have several functions available. See Encoder section of this manual for additional information.

For model specific pinout information please refer to the data sheet for the model being used.

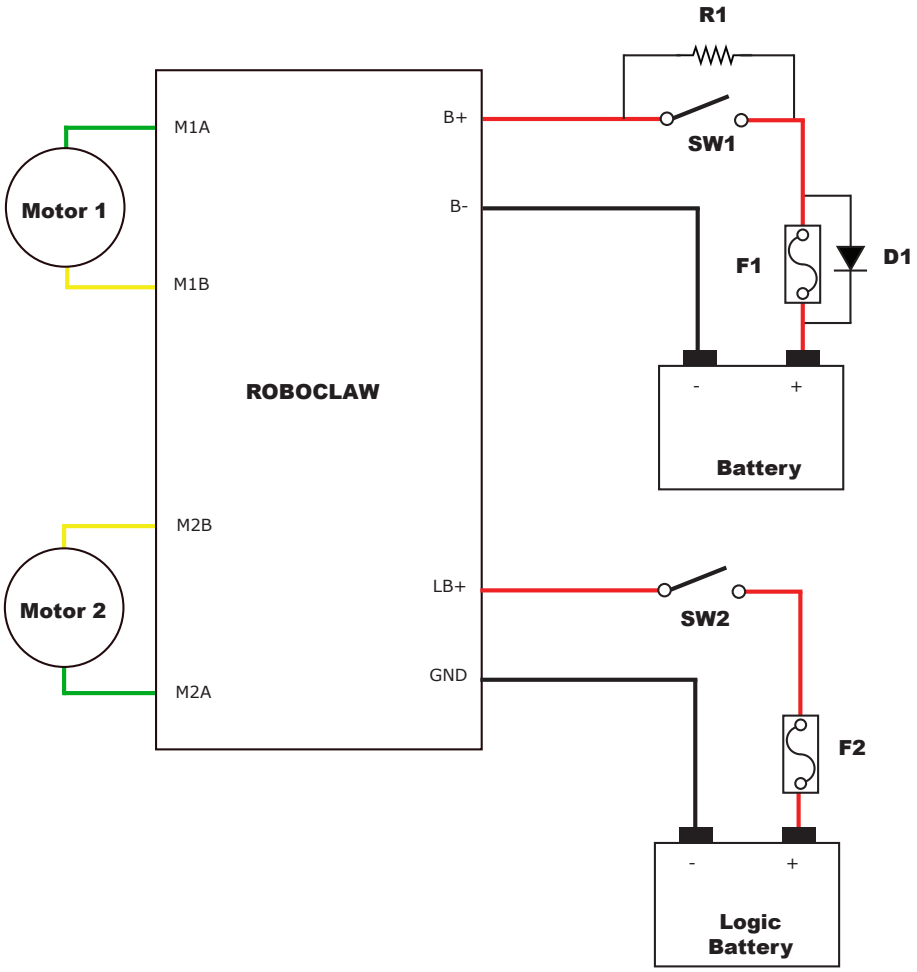


Logic Battery Wiring

An optional logic battery is supported. Under heavy loads the main power can suffer voltage drops, causing potential logic brown outs which may result in uncontrolled behavior. A separate power source for the motor controllers logic circuits, can remedy potential problems from main power voltage drops. The logic battery maximum input voltage is 34VDC with a minimum input voltage of 6VDC. The 5V regulated user output is supplied by the secondary logic battery if supplied. The mAh of the logic battery should be determined based on the load of attached devices powered by the regulated 5V user output.

Logic Battery Jumper

A logic battery is used in the configurtion below. Some older models of RoboClaw have a jumper to set the logic battery. On models where the LB-MB header is present the jumper must be removed when a logic battery is used. If the header for LB-MB is not present, then the RoboClaw will automatically set the logic battery power source.
For model specific pinout information please refer to the data sheet for the model being used.

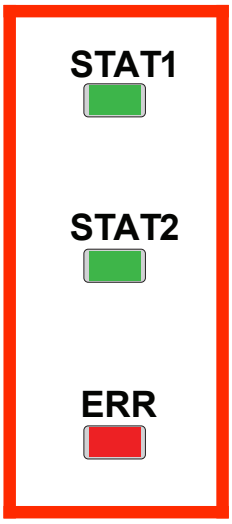


Status LEDs

Status and Error LEDs

RoboClaw includes 3 LEDs to indicate status. Two green status LEDs labeled STAT1 and STAT2 and one red error LED labeled ERR. When the motor controller is first powered on all 3 LEDs should blink briefly to indicate all LEDs are functional.

The LEDs will behave differently depending on the mode. During normal operation the status 1 LED will remain on continuously or blink when data is received in RC Mode or Serial Modes. The status 2 LED will light when either drive stage is active.



Message Types

There are 3 types of messages RoboClaw can indicate. The first type is a fault. When a fault occurs, both motor channel outputs will be disabled and RoboClaw will stop any further motions until the unit is reset, or in the case of non-latching e-Stops, the fault state is cleared. The second message type is a warning. When a warning occurs both motor channel outputs will be controlled automatically depending on the warning condition. As an example if an over temperature of 85c is reached RoboClaw will reduce the maximum allowed current until a safe temperature is reached. The final message type is a notice. Currently there is only one notice indicated.

LED Blink Sequences

When a warning or fault occurs RoboClaw will use the LEDs to blink a sequence. The below table details each sequence and the cause.

LED Status	Condition	Type	Description
All three LEDs lit.	E-Stop	Fault	Motors are stopped by braking.
Error LED lit while condition is active.	Over 85c Temperature	Warning	Motor current limit is recalculated based on temperature.
Error LED blinks once with short delay. Other LEDs off.	Over 100c Temperature	Fault	Motors freewheel while condition exist.
Error LED lit while condition is active.	Over Current	Warning	Motor power is automatically limited.
Error LED blinking twice. STAT1 or STAT2 indicates channel.	Driver Fault	Fault	Motors freewheel. Damage detected.
Error LED blinking three times.	Logic Battery High	Fault	Motors freewheel until reset.
Error LED blinking four times.	Logic Battery Low	Fault	Motors freewheel until reset.
Error LED blinking five times.	Main Battery High	Fault	Motors are stopped by braking until reset.
Error LED lit while condition is active.	Main Battery High	Warning	Motors are stopped by braking while condition exist.
Error LED lit while condition is active.	Main Battery Low	Warning	Motors freewheel while condition exist.
Error LED lit while condition is active.	M1 or M2 Home	Warning	Motor is stopped and encoder is reset to 0
All 3 LED cycle on and off in sequence after power up.	RoboClaw is waiting for new firmware.	Notice	RoboClaw is in boot mode. Use IonMotion PC setup utility to clear.

Inputs

S3, S4 and S5 Setup

RoboClaw S3, S4 and S5 inputs support the use of home switches, limit switches, Voltage Clamping and E-Stops. A limit switch is used to detect the travel limits. Travel limits are typically used on a linear slide to detect when the assembly has reached the end of travel. A home switch is used to create a known start position. In some situations both may be required.

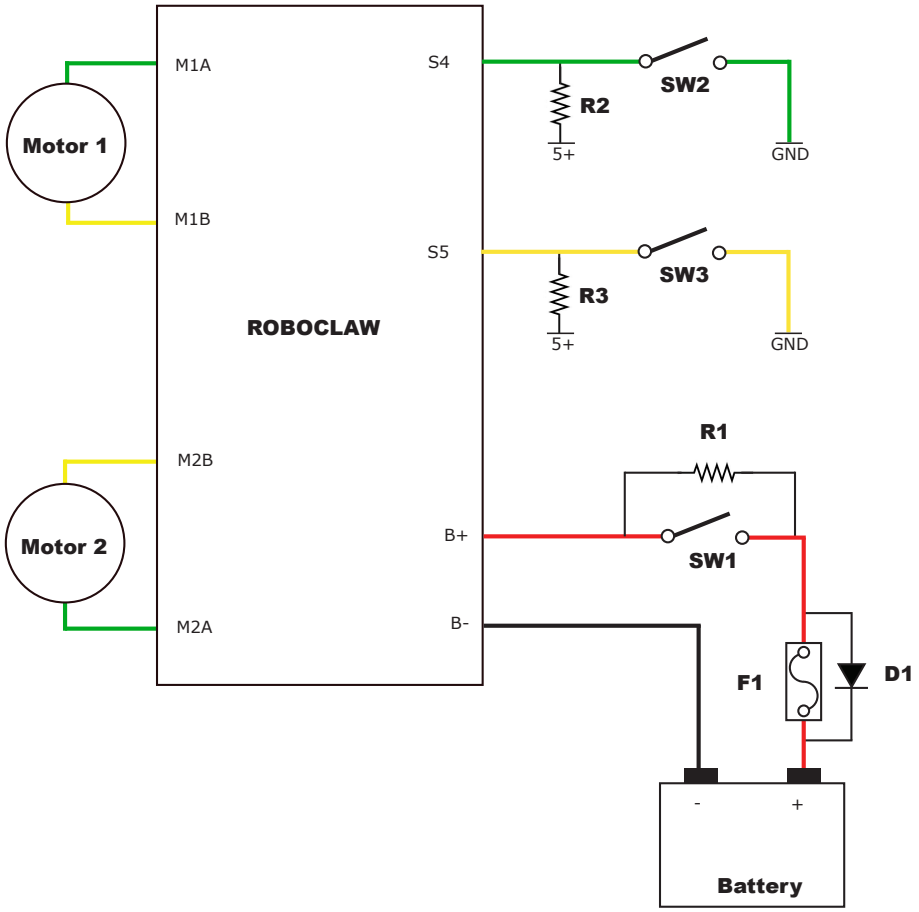
Open Motion Studio and select the S3, S4 or S5 options drop down. There are several options to choose from. Each option is explained below. After setting S3, S4 and S5 options save the settings before exiting Motion Studio.

Option	Description
Disable	Disables S4 and S5. Set by default.
E-Stop(Latching)	All stop until RoboClaw is reset.
E-Stop	All stop until switch released.
Voltage Clamp	Used to control a voltage clamp circuit. Dumps the regenerative voltages. For use with power supplies.
Motor Home(Auto)	After starting up the motor will move backwards until the timeout is reached or the switch is triggered. If during startup the switch is already triggered the motor will move forwards for 3 seconds. During normal operation if the switch is triggered the motor will stop and the encoder is zeroed/
Motor Home(User)	User controls direction of motor to reach switch.
Limit(Forward)	Motor moves forward rotation until switch tripped.
Limit(Reverse)	Motor moves in reverse rotation until switch tripped.
RS-485 Direction Output (S3 only)	Outputs a read/write signal for RS-485 adapters.
Encoder Enable/Disable (S3 only)	Toggles encoder support in RC and analog mode. Allows the user to disable/enable encoders as needed.
Motor Home(Auto)/ Limit(Fwd)	Used when two limit switches are connected to the same input. Same powerup behavior as Motor Home (Auto). Once the home switch is triggered the motor will only be able to move forward until the switch is released. When moving forward the other limit switch stops the motor and the motor will only be able to move backwards until the switch is released.
Motor Home(User)/ Limit(Fwd)	Used when two limit switches are connected to the same input. When the switch is triggered moving backwards the motor stops and the encoder is zeroed. The motor will only move forward until the switch is released. When moving forward the other switch stops the motor. The motor can only move backwards until the switch is released.
Motor Limit(Both)	Used when two limit switches are connected to the same input. When moving backwards the motor will be stopped and only allow forwards movement until the switch is released. The opposite effect occurs with the other switch when moving forward.
CTRL1/CTRL2 (some models)	Output can be set by user command.
Brake	Activates when motor stops. Releases when motor starts.

Limit / Home / E-Stop Wiring

S4 controls motor channel 1 and S5 controls motor channel 2. A pull-up resistor to 5VDC or 3.3VDC should be used if wire lengths exceed 6" (150mm). The circuit below shows a NO (normally open) style switch. Connect the NO to S4 or S5 and the COM end to a power ground shared with RoboClaw.

For model specific pinout information please refer to the data sheet for the model being used.



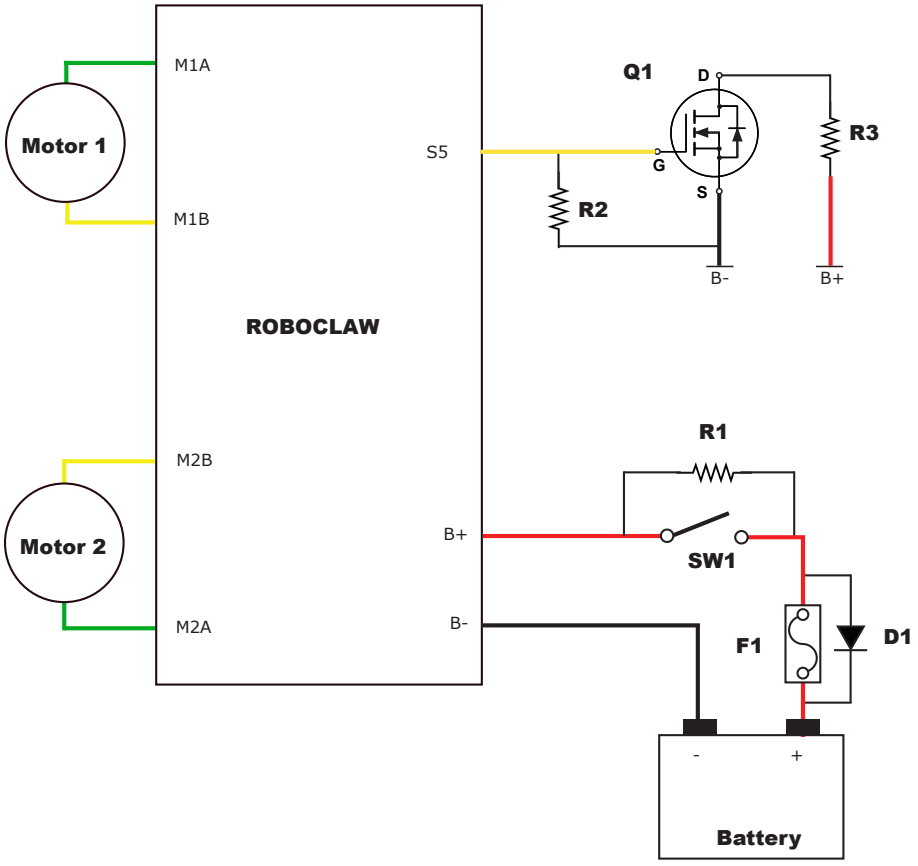
Regenerative Voltage Clamping

Voltage Clamp

When using power supplies regenerative voltage spikes will need to be dissipated. This can be done with a simple circuit shown below and using a V-Clamp pin to activate it. A solid state switch (Q1) and large wattage resistor (R3) are required. The regenerative voltage will be dissipated as heat. An example of a large resistor would be in the range of 10 Ohms at 50 watts for small motors and down to 1 Ohms or lower for larger motors. 50 watts will likely cover most situation smaller wattage resistor may work.

Voltage Clamp Circuit

Wire the circuit as shown below. Q1 should be a 3V logic level MOSFET. An example would be BUK965R8. Which is rated to a maximum voltage of 100VDC. You may need to change the MOSFET used based on the application. R2 (10K) will keep the MOSFET off when not in use. R3 will need to be adjusted based on the initial test results. Start with a 10 Ohms 50 Watt. If after testing the voltage spike is still too great reduce the resistor to a 5 Ohms and so on. For model specific pinout information please refer to the data sheet for the model being used.



Voltage Clamp Setup and Testing

Open Motion Studio and set S5 to the Voltage Clamp option in the drop down and save the setting before exiting the application (see user manual).

The circuit shown will need to be tuned for each application to properly dissipate the regenerative voltages. Testing should consist of a running the motor up to 25% of its speed and then quickly slowing down without braking or e-stop while checking the voltage spike. Repeat, by increasing the speed and power by 5% and checking the voltage spikes again. Repeat this process until 100% power is achieved without a major spike or until the voltage clamp is not dissipating the voltage spikes. If over voltages are not completely clamped, either a lower Ohm resistor is required or additional capacitance is required. Add a capacitor of 5000uF to 10000uF or more across B+ and B-.

Bridge Mode

Bridging Channels

RoboClaw's dual channels can be bridge to run as one channel, effectively doubling its current capability for one motor. RoboClaw will be damaged if it is not set to bridged channel mode before wiring.

Download and install Motion Studio application. Connect the motor controller to the computer using an available USB port. Run Motion Studio and in general settings check the option to "Bridge Channels". Then click “Write Settings” in the Device menu at the top of the window.

When operating in bridged channel mode the total peak current output is combined from both channels. The peak current run time is dependant on heat build up. Adequate cooling must be maintained.

Bridged Channel Wiring

When bridged channel mode is active the internal driver scheme for the output stage is modified. The output leads must be wired correctly or damage will result. Each RoboClaw varies on the correct wiring. See each models data sheet for wiring schematic. Motion Studio will display the proper wiring required for each model when bridged mode is selected.

Bridged Motor Control

When RoboClaw is set to bridged mode all motor control commands for M1 will control the attached motor. All commands for M2 will be ignored. In RC and Analog modes S1 will control the motor and S2 will be ignored.

USB Control

USB Connection

When RoboClaw is connected, it will automatically detect it has been connected to a powered USB master and will enable USB communications. USB can be connected in any mode. When the Roboclaw is not in packet serial mode USB packet serial commands can be used to read status information and set configuration settings, however motor movement commands will not function. When in packet serial mode if another device such as an Arduino is connected to S1 and S2 pins and sending commands to the RoboClaw, both those commands and USB packet serial commands will execute.

USB Power

The USB RoboClaw is self powered. This means it receives no power from the USB cable. The USB RoboClaw must be externally powered to function.

USB Comport and Baudrate

The RoboClaw will be detected as a CDC Virtual Comport. When connected to a Windows PC a driver must be installed. The driver is available for download from our website. On Linux or OSX the RoboClaw will be automatically detected as a virtual comport and an appropriate driver will be automatically loaded.

Unlike a real comport the USB CDC Virtual Comport does not need a baud rate to be set. It will always communicate at the fastest speed the master and slave device can reach. This will typically be around 500kbps.

RC Control

RC Mode

RC mode is typically used when controlling RoboClaw from a hobby RC radio. This mode can also be used to simplify driving RoboClaw from a microcontroller using servo pulses. In this mode S1 controls the direction and speed of motor 1 and S2 controls the direction and speed of motor 2.

RC Mode With Mixing

This mode is the same as RC mode with the exception of how S1 and S2 controls the attached motors. When used with a differentially steered robot, mixing mode allows S1 to control the speed forward and backward and S2 to control steering left and right.

RC Mode with feedback for velocity or position control

RC Mode can be used with encoders. Velocity and/or Position PID constants must be calibrated for proper operation first. Once calibrated values have been set and saved into Roboclaws eeprom memory, encoder support using velocity or position PID control can be enabled. Use Motion Studio control software or Packet Serial commands, enable encoders for RC/Analog modes (See Configuration Using Ion Studio).

RC Mode Options

Option	Function	Description
1	TTL Flip Switch	Flip switch triggered by low signal.
2	TTL Flip and Exponential Enabled	Softens the center control position. This mode is ideal with tank style robots. Making it easier to control from an RC radio. Flip switch triggered by low signal.
3	TTL Flip and MCU Enabled	Continues to execute last pulse received until new pulse received. Disables Signal loss fail safe and auto calibration. Flip switch triggered by low signal.
4	TTL Flip and Exponential and MCU Enabled	Enables both options. Flip switch triggered by low signal.
5	RC Flip Switch Enabled	Same as mode 1 with flip switch triggered by RC signal.
6	RC Flip and Exponential Enabled	Same as mode 2 with flip switch triggered by RC signal.
7	RC Flip and MCU Enabled	Same as mode 3 with flip switch triggered by RC signal.
8	RC Flip and Exponential and MCU Enabled	Same as mode 4 with flip switch triggered by RC signal.

Pulse Ranges

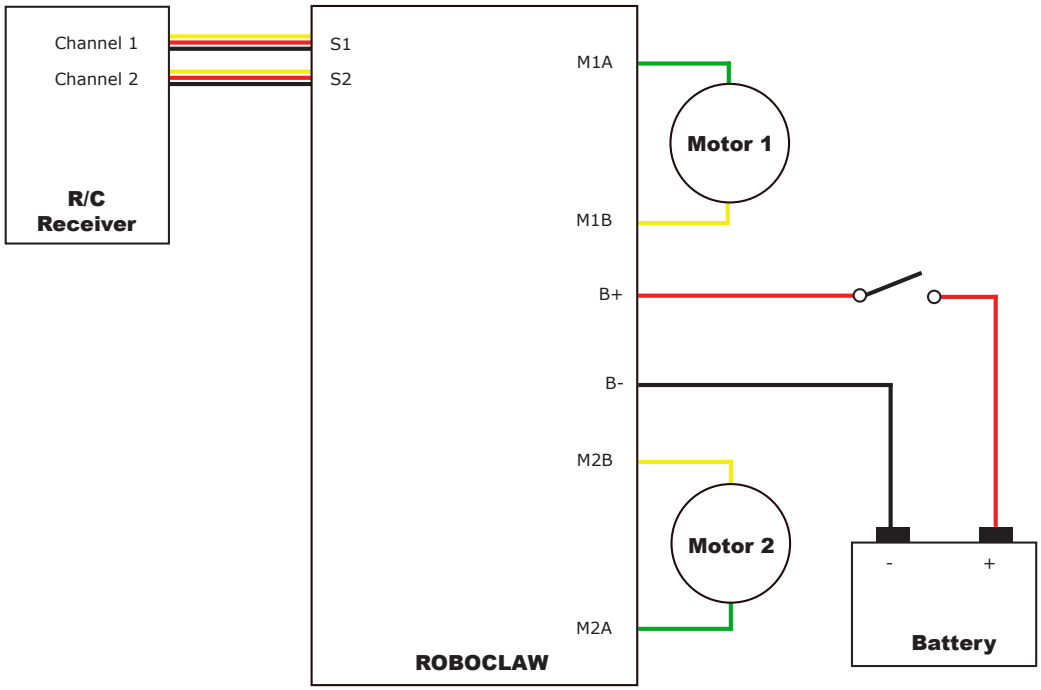
The RoboClaw expects RC pulses on S1 and S2 to drive the motors when the mode is set to RC mode. The center points are calibrated at start up(unless disabled by enabling MCU mode). The RoboClaw will auto calibrate these ranges on the fly unless auto-calibration is disabled. If pulses below the minumum or over the maximum are received the range will be re-calibrated.

Pulse	MCU Mode Enabled	MCU Mode Diabled
Stopped	1520µs	Start Up Auto Calibration
Full Reverse	1120µs	+400µs
Full Forward	1920µs	-400µs

RC Wiring Example

Connect the RoboClaw as shown below. Set mode 1 with option 1. Before powering up, center the control sticks on the radio transmitter, turn the radio on first, then the receiver, then RoboClaw. It will take RoboClaw about 1 second to calibrate the neutral positions of the RC controller. After RC pulses start to be received and calibration is complete the Stat1 LED will begin to flash indicating signals from the RC receiver are being received.

For model specific pinout information please refer to the data sheet for the model being used.



Analog Control

Analog Mode

Analog mode is used when controlling RoboClaw from a potentiometer or a filtered PWM signal. In this mode S1 and S2 are set as analog inputs. The center points are calibrated at start up unless disabled by enabling MCU mode which sets the center at 1v. The RoboClaw will auto calibrate these ranges on the fly unless auto-calibration is disabled. If a voltage smaller than .5v or larger than 1.5v is detected the new voltage will be set as the minimum/maximum.

Analog Mode With Mixing

This mode is the same as Analog mode with the exception of how S1 and S2 control the attached motors. When used with a differentially steered robot, mixing mode allows S1 to control the speed forward and backward and S2 to control steering left and right.

Analog Mode with feedback for velocity or position control

Analog Mode can be used with encoders. Velocity and/or Position PID constants must be calibrated for proper operation. Once calibrated values have been set and saved into Roboclaws eeprom, encoder support using velocity or position PID control can be enabled. Use Ion Studio control software or PacketSerial commands to enable encoders for RC/Analog modes (see Configuration Using Ion Studio).

Analog Mode Options

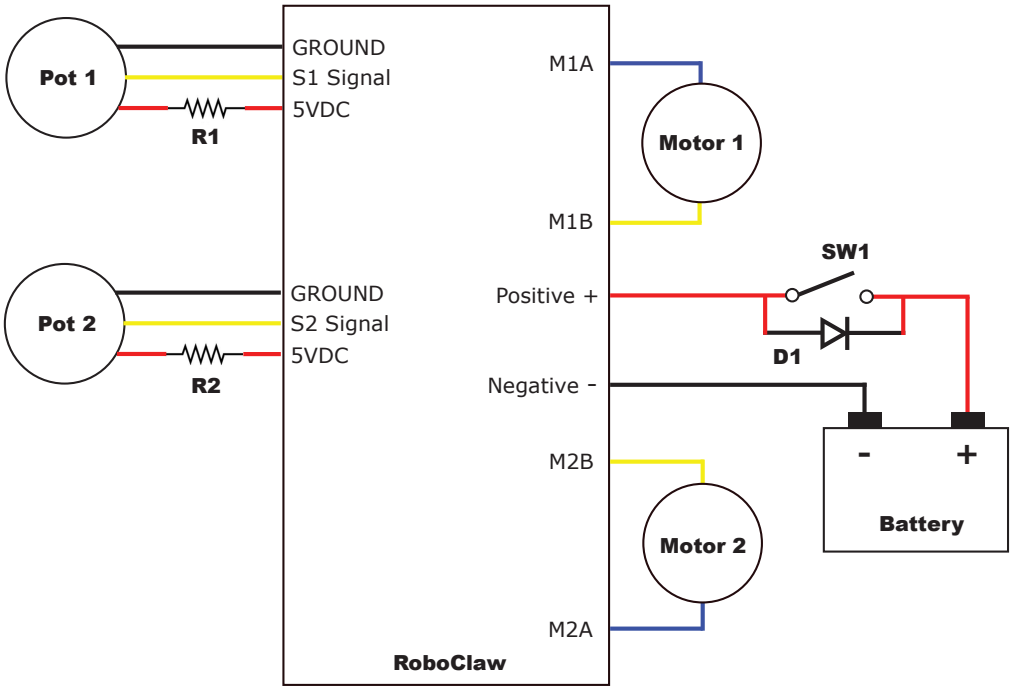
Option	Function	Description
1	TTL Flip Switch	Flip switch triggered by low signal.
2	TTL Flip and Exponential Enabled	Softens the center control position. This mode is ideal with tank style robots. Making it easier to control from an RC radio. Flip switch triggered by low signal.
3	TTL FLip and MCU Enabled	Continues to execute last pulse received until new pulse received. Disables Signal loss fail safe and auto calibration. Flip switch triggered by low signal.
4	TTL FLip and Exponential and MCU Enabled	Enables both options. Flip switch triggered by low signal.
5	RC Flip Switch Enabled	Same as mode 1 with flip switch triggered by RC signal.
6	RC Flip and Exponential Enabled	Same as mode 2 with flip switch triggered by RC signal.
7	RC Flip and MCU Enabled	Same as mode 3 with flip switch triggered by RC signal.
8	RC Flip and Exponential and MCU Enabled	Same as mode 4 with flip switch triggered by RC signal.

Analog Wiring Example

RoboClaw uses a high speed 12 bit analog converter. Its range is 0 to 2V. The analog pins are protected and 5V tolerant. The potentiometer range will be limited if 5V is utilized as the reference voltage. A simple resistor divider circuit can be used to reduce the on board 5V to 2V for use with a potentiometer(POT). See the below schematic. The POT acts as one half of the resistor divider. If using a 5k potentiometer R1 / R2 = 7.5k, If using a 10k potentiometer R1 / R2 = 15k and if using a 20k potentiometer R1 / R2 = 30k.

Set mode 3 with option 1 if using autocalibration. Center the potentiometers before applying power. The S1 potentiometer will control the motor 1 direction and speed. The S2 potentiometer will control the motor 2 direction and speed.

For model specific pinout information please refer to the data sheet for the model being used.



Standard Serial Control

Standard Serial Mode

In this mode S1 accepts TTL level byte commands. Standard serial mode is one way serial data. RoboClaw can receive only. A standard 8N1 format is used. Which is 8 bits, no parity bits and 1 stop bit. If you are using a microcontroller you can interface directly to RoboClaw. If you are using a PC a level shifting circuit (eg: Max232) is required. The baud rate can be changed using the SET button once a serial mode has been selected.

Standard Serial communications has no error correction. It is recommended to use Packet Serial mode instead for more reliable communications.

Serial Mode Baud Rates

Option	Description
1	2400
2	9600
3	19200
4	38400
5	57600
6	115200
7	230400
8	460800

Standard Serial Command Syntax

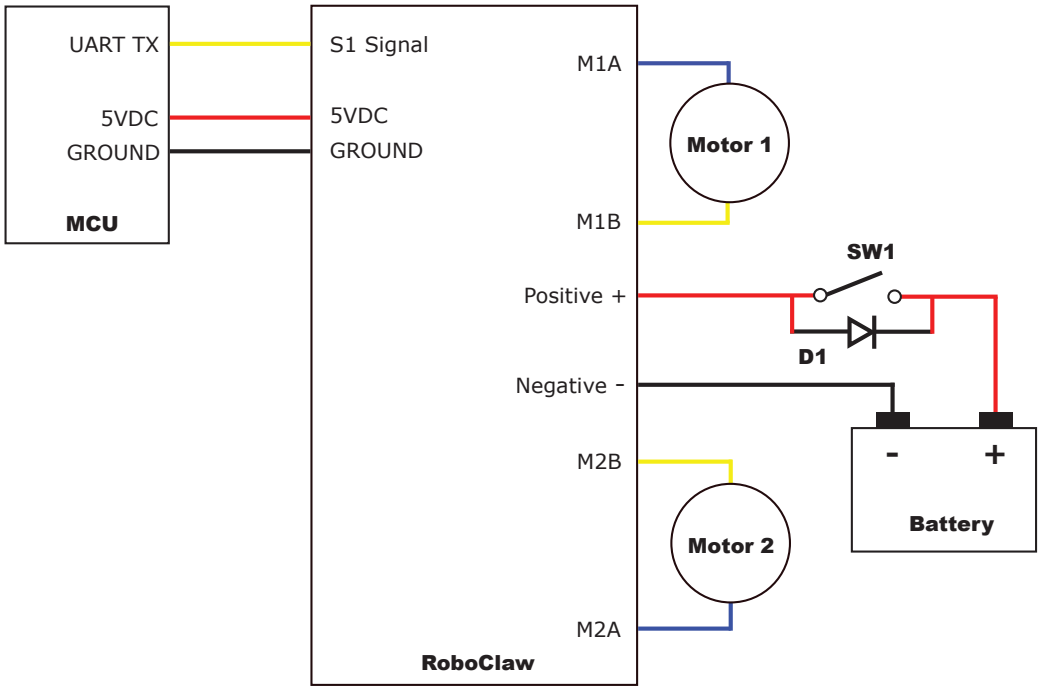
The RoboClaw standard serial is setup to control both motors with one byte sized command character. Since a byte can be any value from 0 to 255(or -128 to 127) the control of each motor is split. 1 to 127 controls channel 1 and 128 to 255(or -1 to -127) controls channel 2. Command value 0 will stop both channels. Any other values will control speed and direction of the specific channel.

Character	Function
0	Shuts Down Channel 1 and 2
1	Channel 1 - Full Reverse
64	Channel 1 - Stop
127	Channel 1 - Full Forward
128	Channel 2 - Full Reverse
192	Channel 2 - Stop
255	Channel 2 - Full Forward

Standard Serial Wiring Example

In standard serial mode the RoboClaw can only receive serial data. The below wiring diagram illustrates a basic setup of RoboClaw for use with standard serial. The diagram below shows the main battery as the only power source. Make sure the LB jumper is set correctly. The 5VDC shown connected is only required if your MCU needs a power source. This is the BEC feature of RoboClaw. If the MCU has its own power source do not connect the 5VDC line.

For model specific pinout information please refer to the data sheet for the model being used.



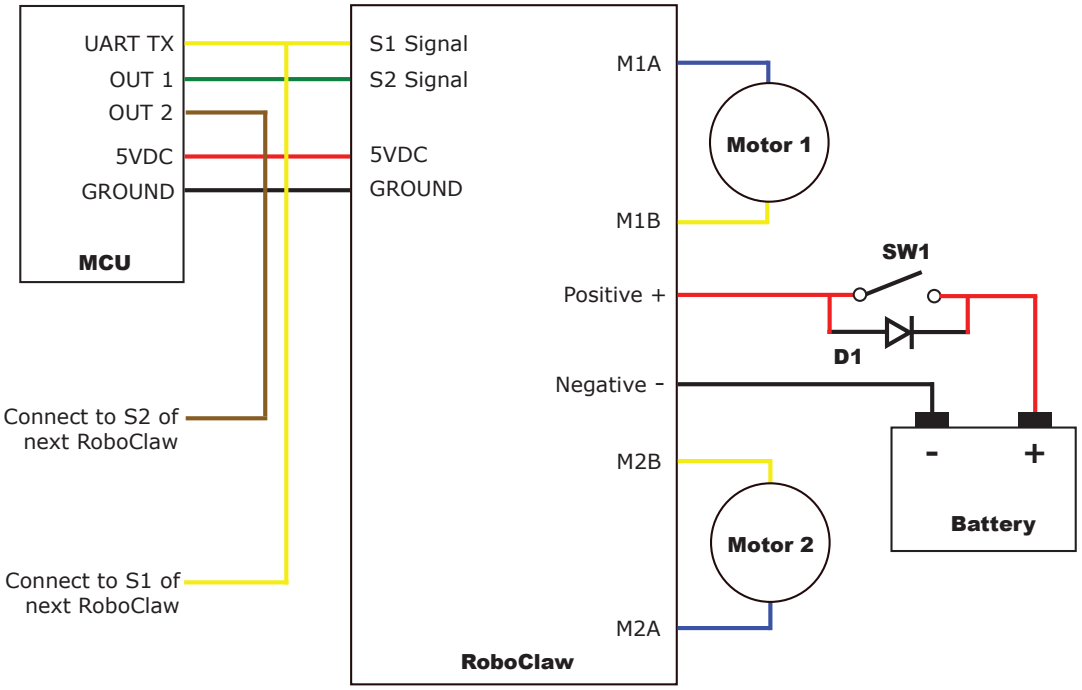
Standard Serial Mode With Slave Select

Slave select is used when more than one RoboClaw is on the same serial bus. When slave select is set to ON the S2 pin becomes the select pin. Set S2 high (5V) and RoboClaw will execute the next set of commands sent to S1 pin. Set S2 low (0V) and RoboClaw will ignore all received commands.

Any RoboClaw connected to a bus must share a common signal ground (GND) shown by the black wire. The S1 pin of RoboClaw is the serial receive pin and should be connected to the transmit pin of the MCU. All RoboClaw's S1 pins will be connected to the same MCU transmit pin. Each RoboClaw S2 pin should be connected to a unique I/O pin on the MCU. S2 is used as the control pin to activate the attached RoboClaw. To enable a RoboClaw hold its S2 pin high otherwise any commands sent are ignored.

The diagram below shows the main battery as the only power source. Make sure the LB jumper is set correctly. The 5VDC shown connected is only required if your MCU needs a power source. This is the BEC feature of RoboClaw. If the MCU has its own power source do not connect the 5VDC.

For model specific pinout information please refer to the data sheet for the model being used.



Packet Serial

Packet Serial Mode

Packet serial is a buffered bidirectional serial mode. More sophisticated instructions can be sent to RoboClaw. The basic command structures consist of an address byte, command byte, data bytes and a CRC16 16bit checksum. The amount of data each command will send or receive can vary.

Address

Packet serial requires a unique address when used with TTL serial pins(S1 and S2). With up to 8 addresses available you can have up to 8 RoboClaws bussed on the same RS232 port when properly wired. There are 8 packet modes 7 to 14. Each mode has a unique address. The address is selected by setting the desired packet mode using the MODE button.

NOTE: When using packet serial commands via the USB connection the address byte can be any value from 0x80 to 0x87 since each USB connection is already unique.

Packet Modes

Mode	Description
7	Packet Serial Mode - Address 0x80 (128)
8	Packet Serial Mode - Address 0x81 (129)
9	Packet Serial Mode - Address 0x82 (130)
10	Packet Serial Mode - Address 0x83 (131)
11	Packet Serial Mode - Address 0x84 (132)
12	Packet Serial Mode - Address 0x85 (133)
13	Packet Serial Mode - Address 0x86 (134)
14	Packet Serial Mode - Address 0x87 (135)

Packet Serial Baud Rate

When in serial mode or packet serial mode the baud rate can be changed to one of four different settings in the table below. These are set using the SET button as covered in Mode Options.

Serial Mode Options

Option	Description
1	2400
2	9600
3	19200
4	38400
5	57600
6	115200
7	230400
8	460800

Packet Timeout

When sending a packet to RoboClaw, if there is a delay longer than 10ms between bytes being received in a packet, RoboClaw will discard the entire packet. This will allow the packet buffer to be cleared by simply adding a minimum 10ms delay before sending a new packet command in the case of a communications error. This can usually be accomodated by having a 10ms timeout when waiting for a reply from the RoboClaw. If the reply times out the packet buffer will have been cleared automatically.

Packet Acknowledgement

RoboClaw will send an acknowledgment byte on write only packet commands that are valid. The value sent back is 0xFF. If the packet was not valid for any reason no acknowledgement will be sent back.

CRC16 Checksum Calculation

Roboclaw uses a CRC(Cyclic Redundancy Check) to validate each packet it receives. This is more complex than a simple checksum but prevents errors that could otherwise cause unexpected actions to execute on the Roboclaw.

The CRC can be calculated using the following code(example in C):

```
//Calculates CRC16 of nBytes of data in byte array message
unsigned int crc16(unsigned char *packet, int nBytes) {
    for (int byte = 0; byte < nBytes; byte++) {
        crc = crc ^ ((unsigned int)packet[byte] << 8);
        for (unsigned char bit = 0; bit < 8; bit++) {
            if (crc & 0x8000) {
                crc = (crc << 1) ^ 0x1021;
            } else {
                crc = crc << 1;
            }
        }
    }
    return crc;
}
```

CRC16 Checksum Calculation for Received data

The CRC16 calculation can also be used to validate received data from the Roboclaw. The CRC16 value should be calculated using the sent Address and Command byte as well as all the data received back from the Roboclaw except the two CRC16 bytes. The value calculated will match the CRC16 sent by the Roboclaw if there are no errors in the data sent or received.

Easy to use Libraries

Source code and Libraries are available on the BasicMicro website that already handle the complexities of using packet serial with the Roboclaw. Libraries are available for Arduino(C++), C# on Windows(.NET) or Linux(Mono) and Python(Raspberry Pi, Linux, OSX, etc).

Handling values larger than a byte

Many Packet Serial commands require values larger than a byte can hold. In order to send or receive those values they need to be broken up into 2 or more bytes. There are two ways this can be done, high byte first or low byte first. Roboclaw expects the high byte first. All command arguments and values are either single bytes, words (2 bytes) or longs (4 bytes). All arguments and values are integers (signed or unsigned). No floating point values (numbers with decimal places) are used in Packet Serial commands.

To convert a 32bit value into 4 bytes you just need to shift the bits around:

```
unsigned char byte3 = MyLongValue>>24; //High byte
unsigned char byte2 = MyLongValue>>16;
unsigned char byte1 = MyLongValue>>8;
unsigned char byte0 = MyLongValue;            //Low byte
```

The same applies to 16bit values:

```
unsigned char byte1 = MyWordValue>>8;  //High byte
unsigned char byte0 = MyWordValue;            //Low byte
```

The oposite can also be done. Convert several bytes into a 16bit or 32bit value:

```
unsigned long MyLongValue = byte3<<24 | byte2<<16 | byte1<<8 | byte0;

unsigned int MyWordValue = byte1<<8 | byte0;
```

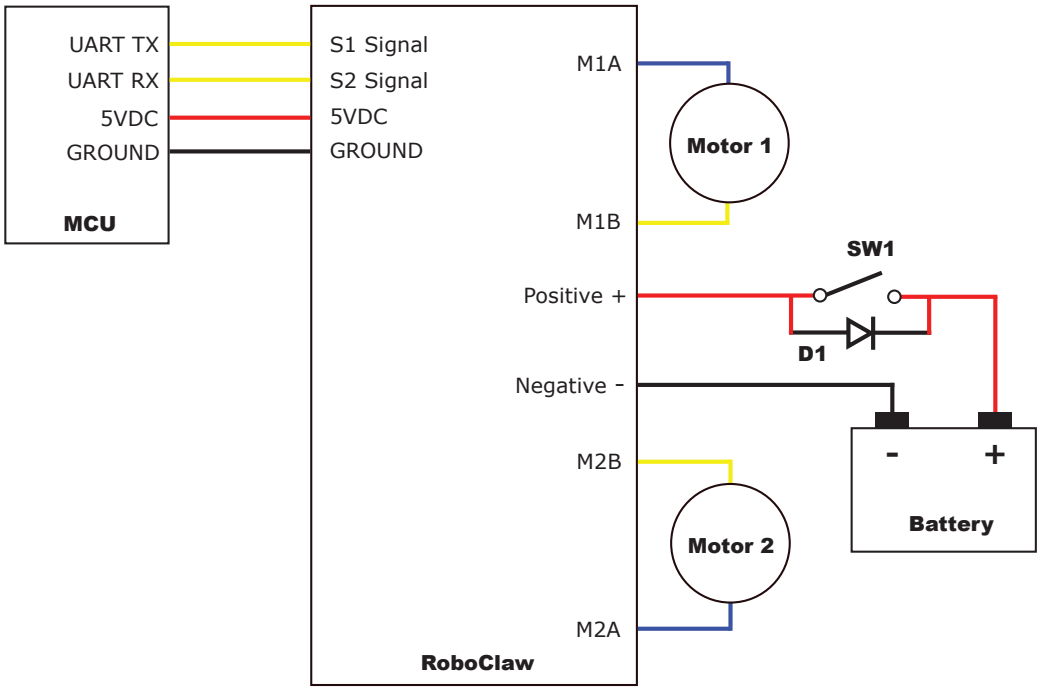
Packet Serial commands, when a value must be broken into multiple bytes or combined from multple bytes it will be indicated either by (2 bytes) or (4 bytes).

Packet Serial Wiring

In packet serial mode the RoboClaw can transmit and receive serial data. A microcontroller with a UART is recommended. The UART will buffer the data received from RoboClaw. When a request for data is made to RoboClaw the return data will have at least a 1ms delay after the command is received if the baud rate is set at or below 38400. This will allow slower processors and processors without UARTs to communicate with RoboClaw.

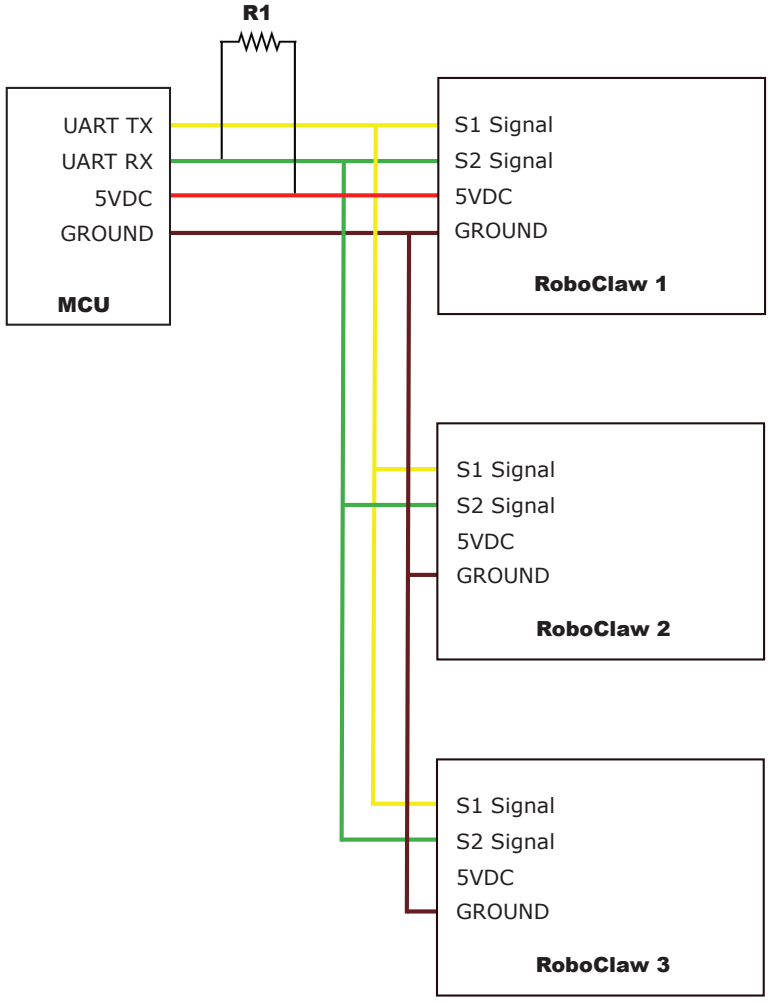
The diagram below shows the main battery as the only power source. The 5VDC shown connected is only required if your MCU needs a power source. This is the BEC feature of RoboClaw. If the MCU has its own power source do not connect the 5VDC.

For model specific pinout information please refer to the data sheet for the model being used.



Multi-Unit Packet Serial Wiring

In packet serial mode up to eight Roboclaw units can be controlled from a single serial port. The wiring diagram below illustrates how this is done. Each Roboclaw must have multi-unit mode enabled and have a unique packet serial address set. This can be configured using Motion Studio. Wire the S1 and S2 pins directly to the MCU TX and RX pins. Install a pull-up resistor (R1) on the MCU RX pin. A 1K to 4.7K resistor value is recommended. For model specific pinout information please refer to the data sheet for the model being used.



Commands 0 - 7 Compatibility Commands

The following commands are used in packet serial mode. The command syntax is the same for commands 0 thru 7:

```
Send: Address, Command, ByteValue, CRC16
Receive: [0xFF]
```

Command	Description
0	Drive Forward Motor 1
1	Drive Backwards Motor 1
2	Set Main Voltage Minimum
3	Set Main Voltage Maximum
4	Drive Forward Motor 2
5	Drive Backwards Motor 2
6	Drive Motor 1 (7 Bit)
7	Drive Motor 2 (7 Bit)
8	Drive Forward Mixed Mode
9	Drive Backwards Mixed Mode
10	Turn Right Mixed Mode
11	Turn Left Mixed Mode
12	Drive Forward or Backward (7 bit)
13	Turn Left or Right (7 Bit)

0 - Drive Forward M1

Drive motor 1 forward. Valid data range is 0 - 127. A value of 127 = full speed forward, 64 = about half speed forward and 0 = full stop.

```
Send: [Address, 0, Value, CRC(2 bytes)]
Receive: [0xFF]
```

1 - Drive Backwards M1

Drive motor 1 backwards. Valid data range is 0 - 127. A value of 127 full speed backwards, 64 = about half speed backward and 0 = full stop.

```
Send: [Address, 1, Value, CRC(2 bytes)]
Receive: [0xFF]
```

2 - Set Minimum Main Voltage (Command 57 Preferred)

Sets main battery (B- / B+) minimum voltage level. If the battery voltages drops below the set voltage level RoboClaw will stop driving the motors. The voltage is set in .2 volt increments. A value of 0 sets the minimum value allowed which is 6V. The valid data range is 0 - 140 (6V - 34V). The formula for calculating the voltage is: (Desired Volts - 6) x 5 = Value. Examples of valid values are 6V = 0, 8V = 10 and 11V = 25.

```
Send: [Address, 2, Value, CRC(2 bytes)]
Receive: [0xFF]
```

3 - Set Maximum Main Voltage (Command 57 Preferred)
Sets main battery (B- / B+) maximum voltage level. The valid data range is 30 - 175 (6V - 34V). During regenerative breaking a back voltage is applied to charge the battery. When using a power supply, by setting the maximum voltage level, RoboClaw will, before exceeding it, go into hard braking mode until the voltage drops below the maximum value set. This will prevent overvoltage conditions when using power supplies. The formula for calculating the voltage is: Desired Volts x 5.12 = Value. Examples of valid values are 12V = 62, 16V = 82 and 24V = 123.

```
Send: [Address, 3, Value, CRC(2 bytes)]
Receive: [0xFF]
```

4 - Drive Forward M2
Drive motor 2 forward. Valid data range is 0 - 127. A value of 127 full speed forward, 64 = about half speed forward and 0 = full stop.

```
Send: [Address, 4, Value, CRC(2 bytes)]
Receive: [0xFF]
```

5 - Drive Backwards M2
Drive motor 2 backwards. Valid data range is 0 - 127. A value of 127 full speed backwards, 64 = about half speed backward and 0 = full stop.

```
Send: [Address, 5, Value, CRC(2 bytes)]
Receive: [0xFF]
```

6 - Drive M1 (7 Bit)
Drive motor 1 forward or reverse. Valid data range is 0 - 127. A value of 0 = full speed reverse, 64 = stop and 127 = full speed forward.

```
Send: [Address, 6, Value, CRC(2 bytes)]
Receive: [0xFF]
```

7 - Drive M2 (7 Bit)
Drive motor 2 forward or reverse. Valid data range is 0 - 127. A value of 0 = full speed reverse, 64 = stop and 127 = full speed forward.

```
Send: [Address, 7, Value, CRC(2 bytes)]
Receive: [0xFF]
```

Commands 8 - 13 Mixed Mode Compatibility Commands

The following commands are mix mode commands used to control speed and turn for differential steering. Before a command is executed, both valid drive and turn data packets are required Once RoboClaw begins to operate the motors turn and speed can be updated independently.

8 - Drive Forward

Drive forward in mix mode. Valid data range is 0 - 127. A value of 0 = full stop and 127 = full forward.

```
Send: [Address, 8, Value, CRC(2 bytes)]
Receive: [0xFF]
```

9 - Drive Backwards

Drive backwards in mix mode. Valid data range is 0 - 127. A value of 0 = full stop and 127 = full reverse.

```
Send: [Address, 9, Value, CRC(2 bytes)]
Receive: [0xFF]
```

10 - Turn right

Turn right in mix mode. Valid data range is 0 - 127. A value of 0 = stop turn and 127 = full speed turn.

```
Send: [Address, 10, Value, CRC(2 bytes)]
Receive: [0xFF]
```

11 - Turn left

Turn left in mix mode. Valid data range is 0 - 127. A value of 0 = stop turn and 127 = full speed turn.

```
Send: [Address, 11, Value, CRC(2 bytes)]
Receive: [0xFF]
```

12 - Drive Forward or Backward (7 Bit)

Drive forward or backwards. Valid data range is 0 - 127. A value of 0 = full backward, 64 = stop and 127 = full forward.

```
Send: [Address, 12, Value, CRC(2 bytes)]
Receive: [0xFF]
```

13 - Turn Left or Right (7 Bit)

Turn left or right. Valid data range is 0 - 127. A value of 0 = full left, 0 = stop turn and 127 = full right.

```
Send: [Address, 13, Value, CRC(2 bytes)]
Receive: [0xFF]
```

Advanced Packet Serial

Commands

The following commands are used to read RoboClaw status information, version information and to set or read configuration values. All commands sent to RoboClaw need to be signed with a CRC16 (Cyclic Redundancy Check of 2 bytes) to validate each packet it received. This is more complex than a simple checksum but prevents errors that could otherwise cause unexpected actions to execute on the Roboclaw. See Packet Serial section of this manual for an explanation on how to create the CRC16 values.

Command	Description
14	Set Serial Timeout
15	Read Serial Timeout
21	Read Firmware Version
24	Read Main Battery Voltage
25	Read Logic Battery Voltage
26	Set Minimum Logic Voltage Level
27	Set Maximum Logic Voltage Level
48	Read Motor PWMs
49	Read Motor Currents
57	Set Main Battery Voltages
58	Set Logic Battery Voltages
59	Read Main Battery Voltage Settings
60	Read Logic Battery Voltage Settings
68	Set default duty cycle acceleration for M1
69	Set default duty cycle acceleration for M2
70	Set Default Speed for M1
71	Set Default Speed for M2
72	Read Default Speed Settings
74	Set S3,S4 and S5 Modes
75	Read S3,S4 and S5 Modes
76	Set DeadBand for RC/Analog controls
77	Read DeadBand for RC/Analog controls
80	Restore Defaults
81	Read Default Duty Cycle Accelerations
82	Read Temperature
83	Read Temperature 2
90	Read Status
91	Read Encoder Modes
92	Set Motor 1 Encoder Mode
93	Set Motor 2 Encoder Mode
94	Write Settings to EEPROM
95	Read Settings from EEPROM

Command	Description
98	Set Standard Config Settings
99	Read Standard Config Settings
100	Set CTRL Modes
101	Read CTRL Modes
102	Set CTRL1
103	Set CTRL2
104	Read CTRLs
105	Set Auto Home Duty/Speed and Timeout M1
106	Set Auto Home Duty/Speed and Timeout M2
107	Read Auto Home Settings
109	Set Speed Error Limits
110	Read Speed Error Limits
112	Set Position Error Limits
113	Read Position Error Limits
115	Set Battery Voltage Offsets
116	Read Battery Voltage Offsets
117	Set Current Blanking Percentages
118	Read Current Blanking Percentages
133	Set M1 Maximum Current
134	Set M2 Maximum Current
135	Read M1 Maximum Current
136	Read M2 Maximum Current
148	Set PWM Mode
149	Read PWM Mode
252	Read User EEPROM Memory Location
253	Write User EEPROM Memory Location

14 - Set Serial Timeout

Sets the serial communication timeout in 100ms increments. When serial bytes are received in the time specified both motors will stop automatically. Range is 0 to 25.5 seconds (0 to 255 in 100ms increments).

```
Send: [Address, 14, Value, CRC (2 byes)]
Receive: [0xFF]
```

15 - Read Serial Timeout

Read the current serial timeout setting. Range is 0 to 255.

```
Send: [Address, 15]
Receive: [Value, CRC (2 bytes)]
```

21 - Read Firmware Version

Read RoboClaw firmware version. Returns up to 48 bytes(depending on the Roboclaw model) and is terminated by a line feed character and a null character.

```
Send: [Address, 21]
Receive: ["RoboClaw 10.2A v4.1.11",10,0, CRC(2 bytes)]
```

The command will return up to 48 bytes. The return string includes the product name and firmware version. The return string is terminated with a line feed (10) and null (0) character.

24 - Read Main Battery Voltage Level

Read the main battery voltage level connected to B+ and B- terminals. The voltage is returned in 10ths of a volt(eg 300 = 30v).

```
Send: [Address, 24]
Receive: [Value(2 bytes), CRC(2 bytes)]
```

25 - Read Logic Battery Voltage Level

Read a logic battery voltage level connected to LB+ and LB- terminals. The voltage is returned in 10ths of a volt(eg 50 = 5v).

```
Send: [Address, 25]
Receive: [Value.Byte1, Value.Byte0, CRC(2 bytes)]
```

26 - Set Minimum Logic Voltage Level

Note: This command is included for backwards compatibility. We recommend you use command 58 instead.

Sets logic input (LB- / LB+) minimum voltage level. RoboClaw will shut down with an error if the voltage is below this level. The voltage is set in .2 volt increments. A value of 0 sets the minimum value allowed which is 6V. The valid data range is 0 - 140 (6V - 34V). The formula for calculating the voltage is: (Desired Volts - 6) x 5 = Value. Examples of valid values are 6V = 0, 8V = 10 and 11V = 25.

```
Send: [Address, 26, Value, CRC(2 bytes)]
Receive: [0xFF]
```

27 - Set Maximum Logic Voltage Level

Note: This command is included for backwards compatibility. We recommend you use command 58 instead.

Sets logic input (LB- / LB+) maximum voltage level. The valid data range is 30 - 175 (6V - 34V). RoboClaw will shutdown with an error if the voltage is above this level. The formula for calculating the voltage is: Desired Volts x 5.12 = Value. Examples of valid values are 12V = 62, 16V = 82 and 24V = 123.

```
Send: [Address, 27, Value, CRC(2 bytes)]
Receive: [0xFF]
```

48 - Read Motor PWM values

Read the current PWM output values for the motor channels. The values returned are +/-32767. The duty cycle percent is calculated by dividing the Value by 327.67.

```
Send: [Address, 48]
Receive: [M1 PWM(2 bytes), M2 PWM(2 bytes), CRC(2 bytes)]
```

49 - Read Motor Currents

Read the current draw from each motor in 10ma increments. The amps value is calculated by dividing the value by 100.

```
Send: [Address, 49]
Receive: [M1 Current(2 bytes), M2 Currrent(2 bytes), CRC(2 bytes)]
```

57 - Set Main Battery Voltages

Set the Main Battery Voltage cutoffs, Min and Max. Min and Max voltages are in 10th of a volt increments. Multiply the voltage to set by 10.

```
Send: [Address, 57, Min(2 bytes), Max(2bytes, CRC(2 bytes)]
Receive: [0xFF]
```

58 - Set Logic Battery Voltages

Set the Logic Battery Voltages cutoffs, Min and Max. Min and Max voltages are in 10th of a volt increments. Multiply the voltage to set by 10.

```
Send: [Address, 58, Min(2 bytes), Max(2bytes, CRC(2 bytes)]
Receive: [0xFF]
```

59 - Read Main Battery Voltage Settings

Read the Main Battery Voltage Settings. The voltage is calculated by dividing the value by 10

```
Send: [Address, 59]
Receive: [Min(2 bytes), Max(2 bytes), CRC(2 bytes)]
```

60 - Read Logic Battery Voltage Settings

Read the Logic Battery Voltage Settings. The voltage is calculated by dividing the value by 10

```
Send: [Address, 60]
Receive: [Min(2 bytes), Max(2 bytes), CRC(2 bytes)]
```

68 - Set M1 Default Duty Acceleration

Set the default acceleration for M1 when using duty cycle commands(Cmds 32,33 and 34) or when using Standard Serial, RC and Analog PWM modes.

```
      Send: [Address, 68, Accel(4 bytes), CRC(2 bytes)]
Receive: [0xFF]
```

69 - Set M2 Default Duty Acceleration

Set the default acceleration for M2 when using duty cycle commands(Cmds 32,33 and 34) or when using Standard Serial, RC and Analog PWM modes.

```
Send: [Address, 69, Accel(4 bytes), CRC(2 bytes)]
Receive: [0xFF]
```

70 - Set Motor1 Default Speed

Set M1 default speed for use with M1 position command and RC or analog modes when position control is enabled. This sets the percentage of the maximum speed set by QPSS as the default speed. The range is 0 to 32767.

```
Send: [Address, 70, Value(2 bytes), CRC(2 bytes)]
Receive: [0xFF]
```

71 - Set Motor 2 Default Speed

Set M2 default speed for use with M2 postion command and RC or analog modes when position control is enabled. This sets the percentage of the maximum speed set by QPSS as the default speed. The range is 0 to 32767.

```
Send: [Address, 71, Value(2 bytes), CRC(2 bytes)]
Receive: [0xFF]
```

72 - Read Default Speed Settings

Read current default speeds for M1 and M2.

```
Send: [Address, 72]
Receive: [M1Speed(2 bytes), M2Speed(2 bytes), CRC(2 bytes)]
```

74 - Set S3, S4 and S5 Modes
Set modes for S3,S4 and S5.

```
Send: [Address, 74, S3mode, S4mode, S5mode, CRC(2 bytes)]
Receive: [0xFF]
```

Mode	S3mode	S4mode	S5mode
0x00	Default	Disabled	Disabled
0x01	E-Stop	E-Stop	E-Stop
0x81	E-Stop(Latching)	E-Stop(Latching)	E-Stop(Latching)
0x14	Voltage Clamp	Voltage Clamp	Voltage Clamp
0x24	RS485 Direction		
0x84	Encoder toggle		
0x04		Brake	Brake
0xE2		Home(Auto)	
0x62		Home(User)	Home(User)
0xF2		Home(Auto)/ Limit(Fwd)	Home(Auto)/Limit(Fwd)
0x72		Home(User)/ Limit(Fwd)	Home(User)/Limit(Fwd)
0x12		Limit(Fwd)	Limit(Fwd)
0x22		Limit(Rev)	Limit(Rev)
0x32		Limit(Both)	Limit(Both)

Mode Description
Disabled: pin is inactive.
Default: Flip switch if in RC/Analog mode or E-Stop(latching) in Serial modes.
E-Stop(Latching): causes the Roboclaw to shutdown until the unit is power cycled.
E-Stop: Holds the Roboclaw in shutdown until the E-Stop signal is cleared.
Voltage Clamp: Sets the signal pin as an output to drive an external voltage clamp circuit
Home(M1 & M2): will trigger the specific motor to stop and the encoder count to reset to 0.

75 - Get S3, S4 and S5 Modes
Read mode settings for S3,S4 and S5. See command 74 for mode descriptions

```
Send: [Address, 75]
Receive: [S3mode, S4mode, S5mode, CRC(2 bytes)]
```

76 - Set DeadBand for RC/Analog controls
Set RC/Analog mode control deadband percentage in 10ths of a percent. Default value is 25(2.5%). Minimum value is 0(no DeadBand), Maximum value is 250(25%).

```
Send: [Address, 76, Reverse, Forward, CRC(2 bytes)]
Receive: [0xFF]
```

77 - Read DeadBand for RC/Analog controls
Read DeadBand settings in 10ths of a percent.

```
Send: [Address, 77]
Receive: [Reverse, SForward, CRC(2 bytes)]
```

80 - Restore Defaults
Reset Settings to factory defaults.

```
Send: [Address, 80]
Receive: [0xFF]
```

81 - Read Default Duty Acceleration Settings
Read M1 and M2 Duty Cycle Acceleration Settings.

```
Send: [Address, 81]
Receive: [M1Accel(4 bytes), M2Accel(4 bytes), CRC(2 bytes)]
```

82 - Read Temperature
Read the board temperature. Value returned is in 10ths of degrees.

```
Send: [Address, 82]
Receive: [Temperature(2 bytes), CRC(2 bytes)]
```

83 - Read Temperature 2
Read the second board temperature(only on supported units). Value returned is in 10ths of degrees.

```
Send: [Address, 83]
Receive: [Temperature(2 bytes), CRC(2 bytes)]
```

90 - Read Status

Read the current unit status.

```
Send: [Address, 90]
Receive: [Status, CRC(2 bytes)]
```

Function	Status Bit Mask
Normal	0x000000
E-Stop	0x000001
Temperature Error	0x000002
Temperature 2 Error	0x000004
Main Voltage High Error	0x000008
Logic Voltage High Error	0x000010
Logic Voltage Low Error	0x000020
M1 Driver Fault Error	0x000040
M2 Driver Fault Error	0x000080
M1 Speed Error	0x000100
M2 Speed Error	0x000200
M1 Position Error	0x000400
M2 Position Error	0x000800
M1 Current Error	0x001000
M2 Current Error	0x002000
M1 Over Current Warning	0x010000
M2 Over Current Warning	0x020000
Main Voltage High Warning	0x040000
Main Voltage Low Warning	0x080000
Temperature Warning	0x100000
Temperature 2 Warning	0x200000
S4 Signal Triggered	0x400000
S5 Signal Triggered	0x800000
Speed Error Limit Warning	0x01000000
Position Error Limit Warning	0x02000000

91 - Read Encoder Mode

Read the encoder mode for both motors.

```
Send: [Address, 91]
Receive: [Enc1Mode, Enc2Mode, CRC(2 bytes)]
```

Encoder Mode bits

Bit 7	Enable/Disable RC/Analog Encoder support
Bit 6	Reverse Encoder Relative Direction
Bit 5	Reverse Motor Relative Direction
Bit 4-1	N/A
Bit 0	Quadrature(0)/Absolute(1)

92 - Set Motor 1 Encoder Mode
Set the Encoder Mode for motor 1. See command 91.

```
Send: [Address, 92, Mode, CRC(2 bytes)]
Receive: [0xFF]
```

93 - Set Motor 2 Encoder Mode
Set the Encoder Mode for motor 2. See command 91.

```
Send: [Address, 93, Mode, CRC(2 bytes)]
Receive: [0xFF]
```

94 - Write Settings to EEPROM
Writes all settings to non-volatile memory. Values will be loaded after each power up.

```
Send: [Address, 94]
Receive: [0xFF]
```

95 - Read Settings from EEPROM
Read all settings from non-volatile memory.

```
Send: [Address, 95]
Receive: [Enc1Mode, Enc2Mode, CRC(2 bytes)]
```

98 - Set Standard Config Settings

Set config bits for standard settings.

```
Send: [Address, 98, Config(2 bytes), CRC(2 bytes)]
Receive: [0xFF]
```

Function	Config Bit Mask
RC Mode	0x0000
Analog Mode	0x0001
Simple Serial Mode	0x0002
Packet Serial Mode	0x0003
Battery Mode Off	0x0000
Battery Mode Auto	0x0004
Battery Mode 2 Cell	0x0008
Battery Mode 3 Cell	0x000C
Battery Mode 4 Cell	0x0010
Battery Mode 5 Cell	0x0014
Battery Mode 6 Cell	0x0018
Battery Mode 7 Cell	0x001C
Mixing	0x0020
Exponential	0x0040
MCU	0x0080
BaudRate 2400	0x0000
BaudRate 9600	0x0020
BaudRate 19200	0x0040
BaudRate 38400	0x0060
BaudRate 57600	0x0080
BaudRate 115200	0x00A0
BaudRate 230400	0x00C0
BaudRate 460800	0x00E0
FlipSwitch	0x0100
Packet Address 0x80	0x0000
Packet Address 0x81	0x0100
Packet Address 0x82	0x0200
Packet Address 0x83	0x0300
Packet Address 0x84	0x0400
Packet Address 0x85	0x0500
Packet Address 0x86	0x0600
Packet Address 0x87	0x0700
Slave Mode	0x0800
Relay Mode	0X1000
Swap Encoders	0x2000
Swap Buttons	0x4000
Multi-Unit Mode	0x8000

99 - Read Standard Config Settings
Read config bits for standard settings See Command 98.

```
Send: [Address, 99]
Receive: [Config(2 bytes), CRC(2 bytes)]
```

100 - Set CTRL Modes
Set CTRL modes of CTRL1 and CTRL2 output pins(available on select models).

```
Send: [Address, 20, CRC(2 bytes)]
Receive: [0xFF]
```

On select models of Roboclaw, two Open drain, high current output drivers are available, CTRL1 and CTRL2.

Mode	Function
0	Disable
1	User
2	Voltage Clamp
3	Brake

User Mode - The output level can be controlled by setting a value from 0(0%) to 65535(100%). A variable frequency PWM is generated at the specified percentage.

Voltage Clamp Mode - The CTRL output will activate when an over voltage is detected and released when the overvoltage disipates. Adding an external load dump resistor from the CTRL pin to B+ will allow the Roboclaw to disipate over voltage energy automatically(up to the 3amp limit of the CTRL pin).

Brake Mode - The CTRL pin can be used to activate an external brake(CTRL1 for Motor 1 brake and CTRL2 for Motor 2 brake). The signal will activate when the motor is stopped(eg 0 PWM). Note acceleration/default_acceleration settings should be set appropriately to allow the motor to slow down before the brake is activated.

101 - Read CTRL Modes
Read CTRL modes of CTRL1 and CTRL2 output pins(available on select models).

```
Send: [Address, 101]
Receive: [CTRL1Mode(1 bytes), CTRL2Mode(1 bytes), CRC(2 bytes)]
```

Reads CTRL1 and CTRL2 mode setting. See 100 - Set CTRL Modes for valid values.

102 - Set CTRL1
Set CTRL1 output value(available on select models)

```
Send: [Address, 102, Value(2 bytes), CRC(2 bytes)]
Receive: [0xFF]
```

Set the output state value of CTRL1. See 100 - Set CTRL Modes for valid values.

103 - Set CTRL2
Set CTRL2 output value(available on select models)

```
Send: [Address, 103, Value(2 bytes), CRC(2 bytes)]
Receive: [0xFF]
```

Set the output state value of CTRL2. See 100 - Set CTRL Modes for valid values.

104 - Read CTRL Settings
Read CTRL1 and CTRL2 output values(available on select models)

```
Send: [Address, 104]
Receive: [CTRL1(2 bytes), CTRL2(2 bytes), CRC(2 bytes)]
```

Reads currently set values for CTRL Settings. See 100 - Set CTRL Modes for valid values.

105 - Set Auto Homing Duty/Speed and Timeout for M1
Sets the percentage of duty or max speed and the timeout value for automatic homing of motor 1. If the motor is set up for velocity or position control the percentage of maximum speed is used, otherwise percent duty is used. The range for duty/speed is 0 to 32767. The timeout value is 32 bits and is set in increments of 1/300 of a second. As an example, a timeout value of 10 seconds would be set as 3000.

```
Send: [Address, 105, Percentage(2 bytes), Timeout(4 bytes), CRC(2 bytes)]
Receive: [0xFF]
```

106 - Set Auto Homing Duty/Speed and Timeout for M2
Sets the percentage of duty or max speed and the timeout value for automatic homing of motor 2. If the motor is set up for velocity or position control the percentage of maximum speed is used, otherwise percent duty is used. The range for duty/speed is 0 to 32767. The timeout value is 32 bits and is set in increments of 1/300 of a second. As an example, a timeout value of 10 seconds would be set as 3000.

```
Send: [Address, 106, Percentage(2 bytes), Timeout(4 bytes), CRC(2 bytes)]
Receive: [0xFF]
```

107 - Read Auto Homing Duty/Speed and Timeout Settings
Read the current auto homing duty/speed and timeout settings.

```
Send: [Address, 107]
Receiver: [Percentage(2 bytes), Timeout(4 bytes), CRC(2 bytes)]
```

109 - Set Speed Error Limits
Set motor speed error limits in encoder counts per second.

```
Send: [Address, 109, M1Limit(4 bytes), M2Limit(4 bytes), CRC(2 bytes)]
Receive: [0xFF]
```

110 - Read Speed Error Limits
Read the current speed error limit values.

```
Send: [Address, 110]
Receive: [M1Limit(4 bytes), M2Limit(4 bytes), CRC(2 bytes)]
```

112 - Set Position Error Limits
Set motor position error limits in encoder counts.

```
Send: [Address, 112, M1Limit(4 bytes), M2Limit(4 byes), CRC(2 bytes)]
Receive: [0xFF]
```

113 - Read Position Error Limits
Read the current motor position error limits.

```
Send: [Address, 113]
Receive: [M1Limit(4 bytes), M2Limit(4 bytes), CRC(2 bytes)]
```

115 - Set Battery Voltage Offsets
Set the main and logic battery offsets to correct for differences in voltage readings. Range of values is +/- 1V in .1V increments with a range of -10 to 10.

```
Send: [Address, 115, MainBatteryOffset, LogicBatteryOffset, CRC(2 bytes)]
Receive: [0xFF]
```

116 - Read Battery Voltage Offsets
Read current voltage offset values.

```
Send: [Address, 116]
Receive: [MainBatteryOffset, LogicBatteryOffset, CRC(2 bytes)]
```

117 - Set Current Blanking Percentage
Sets the percentage of PWM duty for which current readings will be blanked. This setting is used to prevent noise from low PWM duty from causing incorrect current readings. The range is 0 to 6554 (0 to 20%).

```
Send: [Address, 117, M1Blanking(2 bytes), M2Blanking(2 bytes), CRC(2 bytes)]
Receive: [0xFF]
```

118 - Read Current Blanking Percentage
Read the current blanking percentages.

```
Send: [Address, 118]
Receive: [M1Blanking(2 bytes), M2Blanking(2 bytes), CRC(2 bytes)]
```

133 - Set M1 Max Current Limit
Set Motor 1 Maximum Current Limit. Current value is in 10ma units. To calculate multiply current limit by 100.

```
Send: [Address, 134, MaxCurrent(4 bytes), 0, 0, 0, 0, CRC(2 bytes)]
Receive: [0xFF]
```

134 - Set M2 Max Current Limit
Set Motor 2 Maximum Current Limit. Current value is in 10ma units. To calculate multiply current limit by 100.

```
Send: [Address, 134, MaxCurrent(4 bytes), 0, 0, 0, 0, CRC(2 bytes)]
Receive: [0xFF]
```

135 - Read M1 Max Current Limit
Read Motor 1 Maximum Current Limit. Current value is in 10ma units. To calculate divide value by 100. MinCurrent is always 0.

```
Send: [Address, 135]
Receive: [MaxCurrent(4 bytes), MinCurrent(4 bytes), CRC(2 bytes)]
```

136 - Read M2 Max Current Limit
Read Motor 2 Maximum Current Limit. Current value is in 10ma units. To calculate divide value by 100. MinCurrent is always 0.

```
Send: [Address, 136]
Receive: [MaxCurrent(4 bytes), MinCurrent(4 bytes), CRC(2 bytes)]
```

148 - Set PWM Mode
Set PWM Drive mode. Locked Antiphase(0) or Sign Magnitude(1).

```
Send: [Address, 148, Mode, CRC(2 bytes)]
Receive: [0xFF]
```

149 - Read PWM Mode
Read PWM Drive mode. See Command 148.

```
Send: [Address, 149]
Receive: [PWMMode, CRC(2 bytes)]
```

252 - Write User EEPROM Memory Location
Write a 16 bit value to user EEPROM. The Address range is 0 to 255.

```
Send: [Address, 252, Memory Address, Value(2 bytes), CRC(2 bytes)]
Receive: [0xFF]
```

253 - Read User EEPROM Memory Location

Read a 16 bit value from user EEPROM address. Memory address range is 0 to 255.

```
Send: [Address, 253, Memory Address]
Receive: [Value(2 bytes), CRC(2 bytes)]
```

Encoders

Closed Loop Modes

RoboClaw supports a wide range of encoders for close loop modes. Encoders are used in velocity, position or a cascaded velocity with position control mode. This manual mainly deals with Quadrature and Absolute encoders. However additional types of encoders are supported.

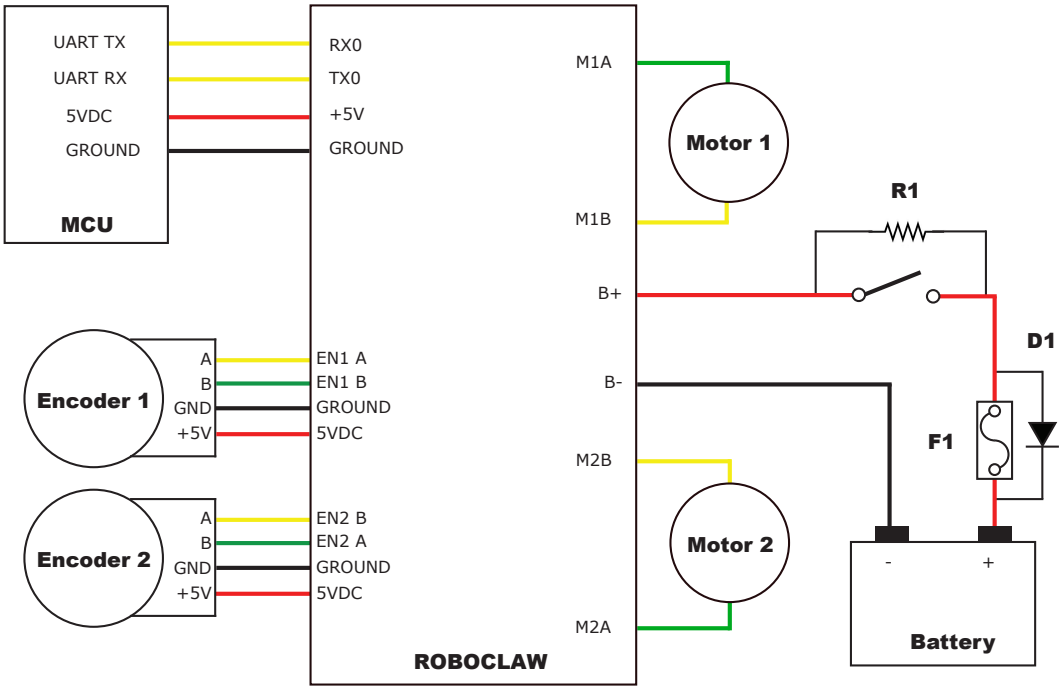
Encoder Tuning

All encoders will require tuning to properly function. Ion Studio incorperates an Auto Tune function which can automatically tune the PID and editable fields for manual tuning of the PID. Encoders can also be tunned using Advance Control Commands which can be sent by a MCU or other control devices.

Quadrature Encoders Wiring

RoboClaw is capable of reading two quadrature encoders, one for each motor channel. The main header provides two +5VDC connections with dual A and B input signals for each encoder.

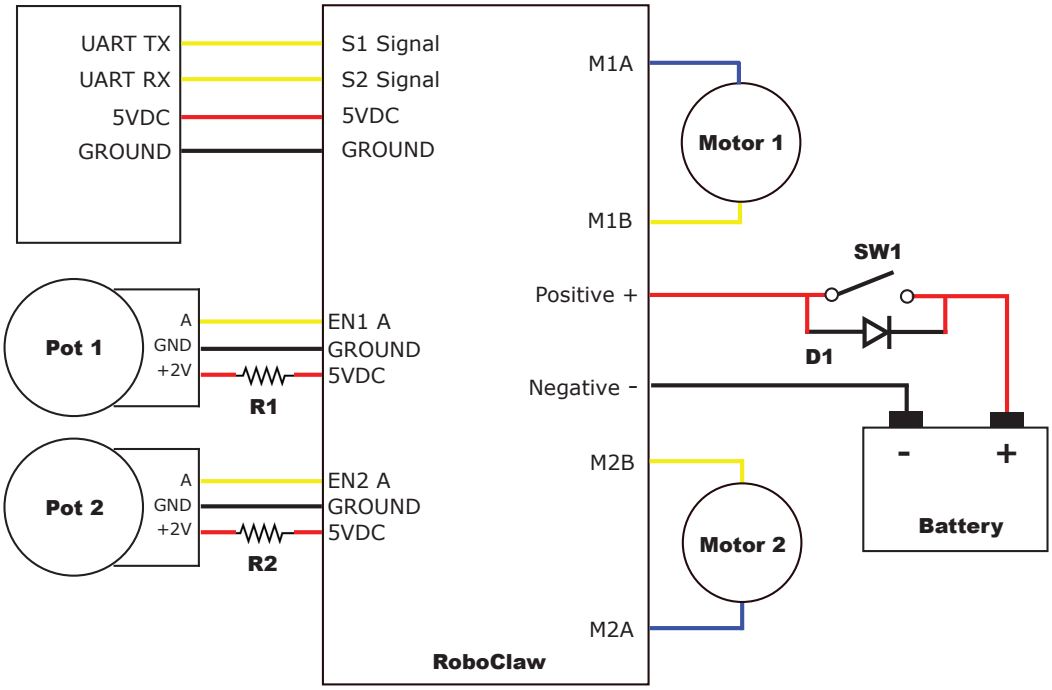
Quadrature encoders are directional. In a simple two motor robot, one motor will spin clock wise (CW) and the other motor will spin counter clock wise (CCW). The A and B inputs for one of the encoders must be reversed to allow both encoders to count up when the robot is moving forward. If both encoder are connected with leading edge pulse to channel A one will count up and the other down. This will cause commands like Mix Drive Forward to not work as expected. All motor and encoder combinations will need to be tuned. For model specific pinout information please refer to the data sheet for the model being used.



Absolute Encoder Wiring

RoboClaw is capable of reading absolute encoders that output an analog voltage. Like the Analog input modes for controlling the motors, the absolute encoder voltage must be between 0v and 2v. If using standard potentiometers as absolute encoders the 5v from the RoboClaw can be divided down to 2v at the potentiometer by adding a resistor from the 5v line on the RoboClaw to the potentiometer. For a 5k pot R1 / R2 = 7.5k, for a 10k pot R1 / R2 = 15k and for a 20k pot R1 / R2 = 30k.

The diagram below shows the main battery as the only power source. Make sure the LB jumper is installed. The 5VDC shown connected is only required if your MCU needs a power source. This is the BEC feature of RoboClaw. If the MCU has its own power source do not connect the 5VDC. For model specific pinout information please refer to the data sheet for the model being used.



Encoder Tuning

To control motor speed and or position with an encoder the PID must be calibrated for the specific motor and encoder being used. Using Motion Studio the PID can be tuned manually or by the auto tune fucntion. Once the encoders are tuned the settings can be saved to the onboard eeprom and will be loaded each time the unit powers up.

The Motion Studio window for Velocity Settings will auto tune for velocity. The window for Position Settings can tune a simple PD position controller, a PID position controller or a cascaded Position with Velocity controller(PIV). The cascaded tune will determine both the velocity and position values for the motor. Auto tune functions usually return reasonable values but manual adjustments may be required for optimum performance.

Auto Tuning

Motion Studio provides the option to auto tune velocity and position control. To use the auto tune option make sure the encoder and motor are running in the desired direction and the basic PWM control of the motor works as expected. It is recommend to ensure the motor and encoder combination are functioning properly before using the auto tune feature.

1. Go to the PWM Settings screen in Motion Studio.

2. Slide the motor slider up to start moving the motor forward. Check the encoder is increasing in value. If it is not either reverse the motor wires or the encoder wires. The recheck.

3. To start auto tune click the auto tune button for the motor channel that is will be tuned first. The auto tune function will try to determine the best settings for that motor channel.

If the motor or encoder are wired incorrectly, the auto tune function can lock up and the motor controller will become unresponsive. Correct the wiring problem and reset the motor controller to continue.

Manual Velocity Calibration Procedure

1. Determine the quadrature pulses per second(QPPS) value for your motor. The simplest method to do this is to run the Motor at 100% duty using Ion Studio and read back the speed value from the encoder attached to the motor. If you are unable to run the motor like this due to physical constraints you will need to estimate the maximum speed in encoder counts the motor can produce.

2. Set the initial P, I and D values in the Velocity control window to 1,0 and 0. Try moving the motor using the slider controls in IonMotion. If the motor does not move it may not be wired correctly or the P value needs to be increased. If the motor immediately runs at max speed when you change the slider position you probably have the motor or encoder wires reversed. The motor is trying to go at the speed specified but the encoder reading is coming back in the opposite direction so the motor increases power until it eventually hits 100% power. Reverse the encoder or motor wires(not both) and test again.

3. Once the motor has some semblance of control you can set a moderate speed. Then start increasing the P value until the speed reading is near the set value. If the motor feels like it is vibrating at higher P values you should reduce the P value to about 2/3rds that value. Move on to the I setting.

4. Start increasing the I setting. You will usually want to increase this value by .1 increments. The I value helps the motor reach the exact speed specified. Too high an I value will also cause the motor to feel rough/vibrate. This is because the motor will over shoot the set speed and then the controller will reduce power to get the speed back down which will also under shoot and this will continue oscillating back and forth form too fast to too slow, causing a vibration in the motor.

5. Once P and I are set reasonably well usually you will leave D = 0. D is only required if you are unable to get reasonable speed control out of the motor using just P and I. D will help dampen P and I over shoot allowing higher P and I values, but D also increases noise in the calculation which can cause oscillations in the speed as well.

Manual Position Calibration Procedure

1. Position mode requires the Velocity mode QPPS value be set as described above. For simple Position control you can set Velocity P, I and D all to 0.

2. Set the Position I and D settings to 0. Set the P setting to 2000 as a reasonable starting point. To test the motor you must also set the Speed argument to some value. We recommend setting it to the same value as the QPPS setting(eg maximum motor speed). Set the minimum and maximum position values to safe numbers. If your motor has no dead stops this can be +-2 billion. If your motor has specific dead stops(like on a linear actuator) you will need to manually move the motor to its dead stops to determine these numbers. Leave some margin infront of each deadstop. Note that when using quadrature encoders you will need to home your motor on every power up since the quadrature readings are all relative to the starting position unless you set/reset the encoder values.

3. At this point the motor should move in the appropriate direction and stop, not necessarily close to the set position when you move the slider. Increase the P setting until the position is over shooting some each time you change the position slider. Now start increasing the D setting(leave I at 0). Increasing D will add dampening to the movement when getting close to the set position. This will help prevent the over shoot. D will usually be anywhere from 5 to 20 times larger than P but not always. Continue increasing P and D until the motor is working reasonably well. Once it is you have tuned a simple PD system.

4. Once your position control is acting relatively smoothly and coming close to the set position you can think about adjusting the I setting. Adding I will help reach the exact set point specified but in most motor systems there is enough slop in the gears that instead you will end up causing an oscillation around the specified position. This is called hunting. The I setting causes this when there is any slop in the motor/encoder/gear train. You can compensate some for this by adding deadzone. Deadzone is the area around the specified position the controller will consider to be equal to the position specified.

5. One more setting must be adjusted in order to use the I setting. The Imax value sets the maximum wind up allowed for the I setting calculation. Increasing Imax will allow I to affect a larger amount of the movement of the motor but will also allow the system to oscillate if used with a badly tuned I and/or set too high.

Encoder Commands

The following commands are used with the encoders both quadrature and absolute. The Encoder Commands are used to read the register values for both encoder channels.

Command	Description
16	Read Encoder Count/Value for M1.
17	Read Encoder Count/Value for M2.
18	Read M1 Speed in Encoder Counts Per Second.
19	Read M2 Speed in Encoder Counts Per Second.
20	Resets Encoder Registers for M1 and M2(Quadrature only).
22	Set Encoder 1 Register(Quadrature only).
23	Set Encoder 2 Register(Quadrature only).
30	Read Current M1 Raw Speed
31	Read Current M2 Raw Speed
78	Read Encoders Counts
79	Read Raw Motor Speeds
108	Read Motor Average Speeds
111	Read Speed Errors
114	Read Position Errors

16 - Read Encoder Count/Value M1

Read M1 encoder count/position.

```
Send: [Address, 16]
Receive: [Enc1(4 bytes), Status, CRC(2 bytes)]
```

Quadrature encoders have a range of 0 to 4,294,967,295. Absolute encoder values are converted from an analog voltage into a value from 0 to 2047 for the full 2v range.

The status byte tracks counter underflow, direction and overflow. The byte value represents:

Bit0 - Counter Underflow (1= Underflow Occurred, Clear After Reading)
Bit1 - Direction (0 = Forward, 1 = Backwards)
Bit2 - Counter Overflow (1= Underflow Occurred, Clear After Reading)
Bit3 - Reserved
Bit4 - Reserved
Bit5 - Reserved
Bit6 - Reserved
Bit7 - Reserved

17 - Read Quadrature Encoder Count/Value M2
Read M2 encoder count/position.

```
Send: [Address, 17]
Receive: [EncCnt(4 bytes), Status, CRC(2 bytes)]
```

Quadrature encoders have a range of 0 to 4,294,967,295. Absolute encoder values are converted from an analog voltage into a value from 0 to 2047 for the full 2v range.

The Status byte tracks counter underflow, direction and overflow. The byte value represents:

Bit0 - Counter Underflow (1= Underflow Occurred, Cleared After Reading)
Bit1 - Direction (0 = Forward, 1 = Backwards)
Bit2 - Counter Overflow (1= Underflow Occurred, Cleared After Reading)
Bit3 - Reserved
Bit4 - Reserved
Bit5 - Reserved
Bit6 - Reserved
Bit7 - Reserved

18 - Read Encoder Speed M1
Read M1 counter speed. Returned value is in pulses per second. RoboClaw keeps track of how many pulses received per second for both encoder channels.

```
Send: [Address, 18]
Receive: [Speed(4 bytes), Status, CRC(2 bytes)]
```

Status indicates the direction (0 – forward, 1 - backward).

19 - Read Encoder Speed M2
Read M2 counter speed. Returned value is in pulses per second. RoboClaw keeps track of how many pulses received per second for both encoder channels.

```
Send: [Address, 19]
Receive: [Speed(4 bytes), Status, CRC(2 bytes)]
```

Status indicates the direction (0 – forward, 1 - backward).

20 - Reset Quadrature Encoder Counters
Will reset both quadrature decoder counters to zero. This command applies to quadrature encoders only.

```
Send: [Address, 20, CRC(2 bytes)]
Receive: [0xFF]
```

22 - Set Quadrature Encoder 1 Value

Set the value of the Encoder 1 register. Useful when homing motor 1. This command applies to quadrature encoders only.

```
Send: [Address, 22, Value(4 bytes), CRC(2 bytes)]
Receive: [0xFF]
```

23 - Set Quadrature Encoder 2 Value

Set the value of the Encoder 2 register. Useful when homing motor 2. This command applies to quadrature encoders only.

```
Send: [Address, 23, Value(4 bytes), CRC(2 bytes)]
Receive: [0xFF]
```

30 - Read Raw Speed M1

Read the pulses counted in that last 300th of a second. This is an unfiltered version of command 18. Command 30 can be used to make a independent PID routine. Value returned is in encoder counts per second.

```
Send: [Address, 30]
Receive: [Speed(4 bytes), Status, CRC(2 bytes)]
```

The Status byte is direction (0 – forward, 1 - backward).

31 - Read Raw Speed M2

Read the pulses counted in that last 300th of a second. This is an unfiltered version of command 19. Command 31 can be used to make a independent PID routine. Value returned is in encoder counts per second.

```
Send: [Address, 31]
Receive: [Speed(4 bytes), Status, CRC(2 bytes)]
```

The Status byte is direction (0 – forward, 1 - backward).

78 - Read Encoder Counters

Read M1 and M2 encoder counters. Quadrature encoders have a range of 0 to 4,294,967,295. Absolute encoder values are converted from an analog voltage into a value from 0 to 2047 for the full 2V analog range.

```
Send: [Address, 78]
Receive: [Enc1(4 bytes), Enc2(4 bytes), CRC(2 bytes)]
```

79 - Read ISpeeds Counters

Read M1 and M2 instantaneous speeds. Returns the speed in encoder counts per second for the last 300th of a second for both encoder channels.

```
Send: [Address, 79]
Receive: [ISpeed1(4 bytes), ISpeed2(4 bytes), CRC(2 bytes)]
```

108 - Read Motor Average Speeds
Read M1 and M2 average speeds. Return the speed in encoder counts per second for the last second for both encoder channels.

```
Send: [Address, 108]
Receive: [Speed1(4 bytes), Speed2(4 bytes), CRC(2 bytes)]
```

111 - Read Speed Errors
Read current calculated speed error in encoder counts per second.

```
Send: [Address, 111]
Receive; [M1Error(4 bytes), M2Error(4 bytes), CRC(2 bytes)]
```

114 - Read Position Errors
Read current calculated position error in encoder counts.

```
Send: [Address, 114]
Receive: [M1Error(4 bytes), M2Error(4 bytes], CRC(2 bytes)]
```

Advanced Motor Control

Advanced Motor Control Commands

The following commands are used to control motor speeds, acceleration distance and position using encoders. The PID can also be manually adjusted using Advanced Motor Control Commands.

Command	Description
28	Set Velocity PID Constants for M1.
29	Set Velocity PID Constants for M2.
32	Drive M1 With Signed Duty Cycle. (Encoders not required)
33	Drive M2 With Signed Duty Cycle. (Encoders not required)
34	Drive M1 / M2 With Signed Duty Cycle. (Encoders not required)
35	Drive M1 With Signed Speed.
36	Drive M2 With Signed Speed.
37	Drive M1 / M2 With Signed Speed.
38	Drive M1 With Signed Speed And Acceleration.
39	Drive M2 With Signed Speed And Acceleration.
40	Drive M1 / M2 With Signed Speed And Acceleration.
41	Drive M1 With Signed Speed And Distance. Buffered.
42	Drive M2 With Signed Speed And Distance. Buffered.
43	Drive M1 / M2 With Signed Speed And Distance. Buffered.
44	Drive M1 With Signed Speed, Acceleration and Distance. Buffered.
45	Drive M2 With Signed Speed, Acceleration and Distance. Buffered.
46	Drive M1 / M2 With Signed Speed, Acceleration And Distance. Buffered.
47	Read Buffer Length.
50	Drive M1 / M2 With Individual Signed Speed and Acceleration
51	Drive M1 / M2 With Individual Signed Speed, Accel and Distance
52	Drive M1 With Signed Duty and Accel. (Encoders not required)
53	Drive M2 With Signed Duty and Accel. (Encoders not required)
54	Drive M1 / M2 With Signed Duty and Accel. (Encoders not required)
55	Read Motor 1 Velocity PID Constants
56	Read Motor 2 Velocity PID Constants
61	Set Position PID Constants for M1.
62	Set Position PID Constants for M2
63	Read Motor 1 Position PID Constants
64	Read Motor 2 Position PID Constants
65	Drive M1 with Speed, Accel, Deccel and Position
66	Drive M2 with Speed, Accel, Deccel and Position
67	Drive M1 / M2 with Speed, Accel, Deccel and Position
119	Drive M1 with Position.
120	Drive M2 with Position.
121	Drive M1/M2 with Position.
122	Drive M1 with Speed and Position.
123	Drive M2 with Speed and Position.
124	Drive M1/M2 with Speed and Postion.

28 - Set Velocity PID Constants M1

Several motor and quadrature combinations can be used with RoboClaw. In some cases the default PID values will need to be tuned for the systems being driven. This gives greater flexibility in what motor and encoder combinations can be used. The RoboClaw PID system consist of four constants starting with QPPS, P = Proportional, I= Integral and D= Derivative. The defaults values are:

```
QPPS = 44000
P = 0x00010000
I = 0x00008000
D = 0x00004000
```

QPPS is the speed of the encoder when the motor is at 100% power. P, I, D are the default values used after a reset. Command syntax:

```
Send: [Address, 28, D(4 bytes), P(4 bytes), I(4 bytes), QPPS(4 byte), CRC(2 bytes)]
Receive: [0xFF]
```

29 - Set Velocity PID Constants M2

Several motor and quadrature combinations can be used with RoboClaw. In some cases the default PID values will need to be tuned for the systems being driven. This gives greater flexibility in what motor and encoder combinations can be used. The RoboClaw PID system consist of four constants starting with QPPS, P = Proportional, I= Integral and D= Derivative. The defaults values are:

```
QPPS = 44000
P = 0x00010000
I = 0x00008000
D = 0x00004000
```

QPPS is the speed of the encoder when the motor is at 100% power. P, I, D are the default values used after a reset. Command syntax:

```
Send: [Address, 29, D(4 bytes), P(4 bytes), I(4 bytes), QPPS(4 byte), CRC(2 bytes)]Re-
ceive: [0xFF]
```

32 - Drive M1 With Signed Duty Cycle

Drive M1 using a duty cycle value. The duty cycle is used to control the speed of the motor without a quadrature encoder.

```
Send: [Address, 32, Duty(2 Bytes), CRC(2 bytes)]
Receive: [0xFF]
```

The duty value is signed and the range is -32767 to +32767 (eg. +-100% duty).

33 - Drive M2 With Signed Duty Cycle

Drive M2 using a duty cycle value. The duty cycle is used to control the speed of the motor without a quadrature encoder. The command syntax:

```
Send: [Address, 33, Duty(2 Bytes), CRC(2 bytes)]
Receive: [0xFF]
```

The duty value is signed and the range is -32768 to +32767 (eg. +-100% duty).

34 - Drive M1 / M2 With Signed Duty Cycle

Drive both M1 and M2 using a duty cycle value. The duty cycle is used to control the speed of the motor without a quadrature encoder. The command syntax:

```
Send: [Address, 34, DutyM1(2 Bytes), DutyM2(2 Bytes), CRC(2 bytes)]
Receive: [0xFF]
```

The duty value is signed and the range is -32768 to +32767 (eg. +-100% duty).

35 - Drive M1 With Signed Speed

Drive M1 using a speed value. The sign indicates which direction the motor will turn. This command is used to drive the motor by quad pulses per second. Different quadrature encoders will have different rates at which they generate the incoming pulses. The values used will differ from one encoder to another. Once a value is sent the motor will begin to accelerate as fast as possible until the defined rate is reached.

```
Send: [Address, 35, Speed(4 Bytes), CRC(2 bytes)]
Receive: [0xFF]
```

36 - Drive M2 With Signed Speed

Drive M2 with a speed value. The sign indicates which direction the motor will turn. This command is used to drive the motor by quad pulses per second. Different quadrature encoders will have different rates at which they generate the incoming pulses. The values used will differ from one encoder to another. Once a value is sent, the motor will begin to accelerate as fast as possible until the rate defined is reached.

```
Send: [Address, 36, Speed(4 Bytes), CRC(2 bytes)]
Receive: [0xFF]
```

37 - Drive M1 / M2 With Signed Speed

Drive M1 and M2 in the same command using a signed speed value. The sign indicates which direction the motor will turn. This command is used to drive both motors by quad pulses per second. Different quadrature encoders will have different rates at which they generate the incoming pulses. The values used will differ from one encoder to another. Once a value is sent the motor will begin to accelerate as fast as possible until the rate defined is reached.

```
Send: [Address, 37, SpeedM1(4 Bytes), SpeedM2(4 Bytes), CRC(2 bytes)]
Receive: [0xFF]
```

38 - Drive M1 With Signed Speed And Acceleration

Drive M1 with a signed speed and acceleration value. The sign indicates which direction the motor will run. The acceleration values are not signed. This command is used to drive the motor by quad pulses per second and using an acceleration value for ramping. Different quadrature encoders will have different rates at which they generate the incoming pulses. The values used will differ from one encoder to another. Once a value is sent the motor will begin to accelerate incrementally until the rate defined is reached.

```
Send: [Address, 38, Accel(4 Bytes), Speed(4 Bytes), CRC(2 bytes)]
Receive: [0xFF]
```

The acceleration is measured in speed increase per second. An acceleration value of 12,000 QPPS with a speed of 12,000 QPPS would accelerate a motor from 0 to 12,000 QPPS in 1 second. Another example would be an acceleration value of 24,000 QPPS and a speed value of 12,000 QPPS would accelerate the motor to 12,000 QPPS in 0.5 seconds.

39 - Drive M2 With Signed Speed And Acceleration

Drive M2 with a signed speed and acceleration value. The sign indicates which direction the motor will run. The acceleration value is not signed. This command is used to drive the motor by quad pulses per second and using an acceleration value for ramping. Different quadrature encoders will have different rates at which they generate the incoming pulses. The values used will differ from one encoder to another. Once a value is sent the motor will begin to accelerate incrementally until the rate defined is reached.

```
Send: [Address, 39, Accel(4 Bytes), Speed(4 Bytes), CRC(2 bytes)]
Receive: [0xFF]
```

The acceleration is measured in speed increase per second. An acceleration value of 12,000 QPPS with a speed of 12,000 QPPS would accelerate a motor from 0 to 12,000 QPPS in 1 second. Another example would be an acceleration value of 24,000 QPPS and a speed value of 12,000 QPPS would accelerate the motor to 12,000 QPPS in 0.5 seconds.

40 - Drive M1 / M2 With Signed Speed And Acceleration

Drive M1 and M2 in the same command using one value for acceleration and two signed speed values for each motor. The sign indicates which direction the motor will run. The acceleration value is not signed. The motors are sync during acceleration. This command is used to drive the motor by quad pulses per second and using an acceleration value for ramping. Different quadrature encoders will have different rates at which they generate the incoming pulses. The values used will differ from one encoder to another. Once a value is sent the motor will begin to accelerate incrementally until the rate defined is reached.

```
Send: [Address, 40, Accel(4 Bytes), SpeedM1(4 Bytes), SpeedM2(4 Bytes), CRC(2 bytes)]
Receive: [0xFF]
```

The acceleration is measured in speed increase per second. An acceleration value of 12,000 QPPS with a speed of 12,000 QPPS would accelerate a motor from 0 to 12,000 QPPS in 1 second. Another example would be an acceleration value of 24,000 QPPS and a speed value of 12,000 QPPS would accelerate the motor to 12,000 QPPS in 0.5 seconds.

41 - Buffered M1 Drive With Signed Speed And Distance

Drive M1 with a signed speed and distance value. The sign indicates which direction the motor will run. The distance value is not signed. This command is buffered. This command is used to control the top speed and total distance traveled by the motor. Each motor channel M1 and M2 have separate buffers. This command will execute immediately if no other command for that channel is executing, otherwise the command will be buffered in the order it was sent. Any buffered or executing command can be stopped when a new command is issued by setting the Buffer argument. All values used are in quad pulses per second.

```
Send: [Address, 41, Speed(4 Bytes), Distance(4 Bytes), Buffer, CRC(2 bytes)]
Receive: [0xFF]
```

The Buffer argument can be set to a 1 or 0. If a value of 0 is used the command will be buffered and executed in the order sent. If a value of 1 is used the current running command is stopped, any other commands in the buffer are deleted and the new command is executed.

42 - Buffered M2 Drive With Signed Speed And Distance

Drive M2 with a speed and distance value. The sign indicates which direction the motor will run. The distance value is not signed. This command is buffered. Each motor channel M1 and M2 have separate buffers. This command will execute immediately if no other command for that channel is executing, otherwise the command will be buffered in the order it was sent. Any buffered or executing command can be stopped when a new command is issued by setting the Buffer argument. All values used are in quad pulses per second.

```
Send: [Address, 42, Speed(4 Bytes), Distance(4 Bytes), Buffer, CRC(2 bytes)]
Receive: [0xFF]
```

The Buffer argument can be set to a 1 or 0. If a value of 0 is used the command will be buffered and executed in the order sent. If a value of 1 is used the current running command is stopped, any other commands in the buffer are deleted and the new command is executed.

43 - Buffered Drive M1 / M2 With Signed Speed And Distance

Drive M1 and M2 with a speed and distance value. The sign indicates which direction the motor will run. The distance value is not signed. This command is buffered. Each motor channel M1 and M2 have separate buffers. This command will execute immediately if no other command for that channel is executing, otherwise the command will be buffered in the order it was sent. Any buffered or executing command can be stopped when a new command is issued by setting the Buffer argument. All values used are in quad pulses per second.

```
Send: [Address, 43, SpeedM1(4 Bytes), DistanceM1(4 Bytes),
            SpeedM2(4 Bytes), DistanceM2(4 Bytes), Buffer, CRC(2 bytes)]
Receive: [0xFF]
```

The Buffer argument can be set to a 1 or 0. If a value of 0 is used the command will be buffered and executed in the order sent. If a value of 1 is used the current running command is stopped, any other commands in the buffer are deleted and the new command is executed.

44 - Buffered M1 Drive With Signed Speed, Accel And Distance

Drive M1 with a speed, acceleration and distance value. The sign indicates which direction the motor will run. The acceleration and distance values are not signed. This command is used to control the motors top speed, total distanced traveled and at what incremental acceleration value to use until the top speed is reached. Each motor channel M1 and M2 have separate buffers. This command will execute immediately if no other command for that channel is executing, otherwise the command will be buffered in the order it was sent. Any buffered or executing command can be stopped when a new command is issued by setting the Buffer argument. All values used are in quad pulses per second.

```
Send: [Address, 44, Accel(4 bytes), Speed(4 Bytes), Distance(4 Bytes),
            Buffer, CRC(2 bytes)]
Receive: [0xFF]
```

The Buffer argument can be set to a 1 or 0. If a value of 0 is used the command will be buffered and executed in the order sent. If a value of 1 is used the current running command is stopped, any other commands in the buffer are deleted and the new command is executed.

45 - Buffered M2 Drive With Signed Speed, Accel And Distance

Drive M2 with a speed, acceleration and distance value. The sign indicates which direction the motor will run. The acceleration and distance values are not signed. This command is used to control the motors top speed, total distanced traveled and at what incremental acceleration value to use until the top speed is reached. Each motor channel M1 and M2 have separate buffers. This command will execute immediately if no other command for that channel is executing, otherwise the command will be buffered in the order it was sent. Any buffered or executing command can be stopped when a new command is issued by setting the Buffer argument. All values used are in quad pulses per second.

```
Send: [Address, 45, Accel(4 bytes), Speed(4 Bytes), Distance(4 Bytes),
            Buffer, CRC(2 bytes)]
Receive: [0xFF]
```

The Buffer argument can be set to a 1 or 0. If a value of 0 is used the command will be buffered and executed in the order sent. If a value of 1 is used the current running command is stopped, any other commands in the buffer are deleted and the new command is executed.

46 - Buffered Drive M1 / M2 With Signed Speed, Accel And Distance

Drive M1 and M2 with a speed, acceleration and distance value. The sign indicates which direction the motor will run. The acceleration and distance values are not signed. This command is used to control both motors top speed, total distanced traveled and at what incremental acceleration value to use until the top speed is reached. Each motor channel M1 and M2 have separate buffers. This command will execute immediately if no other command for that channel is executing, otherwise the command will be buffered in the order it was sent. Any buffered or executing command can be stopped when a new command is issued by setting the Buffer argument. All values used are in quad pulses per second.

```
Send: [Address, 46, Accel(4 Bytes), SpeedM1(4 Bytes), DistanceM1(4 Bytes),
      SpeedM2(4 bytes), DistanceM2(4 Bytes), Buffer, CRC(2 bytes)]
Receive: [0xFF]
```

The Buffer argument can be set to a 1 or 0. If a value of 0 is used the command will be buffered and executed in the order sent. If a value of 1 is used the current running command is stopped, any other commands in the buffer are deleted and the new command is executed.

47 - Read Buffer Length

Read both motor M1 and M2 buffer lengths. This command can be used to determine how many commands are waiting to execute.

```
Send: [Address, 47]
Receive: [BufferM1, BufferM2, CRC(2 bytes)]
```

The return values represent how many commands per buffer are waiting to be executed. The maximum buffer size per motor is 64 commands(0x3F). A return value of 0x80(128) indicates the buffer is empty. A return value of 0 indiciates the last command sent is executing. A value of 0x80 indicates the last command buffered has finished.

50 - Drive M1 / M2 With Signed Speed And Individual Acceleration

Drive M1 and M2 in the same command using one value for acceleration and two signed speed values for each motor. The sign indicates which direction the motor will run. The acceleration value is not signed. The motors are sync during acceleration. This command is used to drive the motor by quad pulses per second and using an acceleration value for ramping. Different quadrature encoders will have different rates at which they generate the incoming pulses. The values used will differ from one encoder to another. Once a value is sent the motor will begin to accelerate incrementally until the rate defined is reached.

```
Send: [Address, 50, AccelM1(4 Bytes), SpeedM1(4 Bytes), AccelM2(4 Bytes),
            SpeedM2(4 Bytes), CRC(2 bytes)]
Receive: [0xFF]
```

The acceleration is measured in speed increase per second. An acceleration value of 12,000 QPPS with a speed of 12,000 QPPS would accelerate a motor from 0 to 12,000 QPPS in 1 second. Another example would be an acceleration value of 24,000 QPPS and a speed value of 12,000 QPPS would accelerate the motor to 12,000 QPPS in 0.5 seconds.

51 - Buffered Drive M1 / M2 With Signed Speed, Individual Accel And Distance

Drive M1 and M2 with a speed, acceleration and distance value. The sign indicates which direction the motor will run. The acceleration and distance values are not signed. This command is used to control both motors top speed, total distanced traveled and at what incremental acceleration value to use until the top speed is reached. Each motor channel M1 and M2 have separate buffers. This command will execute immediately if no other command for that channel is executing, otherwise the command will be buffered in the order it was sent. Any buffered or executing command can be stopped when a new command is issued by setting the Buffer argument. All values used are in quad pulses per second.

```
Send: [Address, 51, AccelM1(4 Bytes), SpeedM1(4 Bytes), DistanceM1(4 Bytes),
AccelM2(4 Bytes), SpeedM2(4 bytes), DistanceM2(4 Bytes), Buffer, CRC(2 bytes)]
Receive: [0xFF]
```

The Buffer argument can be set to a 1 or 0. If a value of 0 is used the command will be buffered and executed in the order sent. If a value of 1 is used the current running command is stopped, any other commands in the buffer are deleted and the new command is executed.

52 - Drive M1 With Signed Duty And Acceleration

Drive M1 with a signed duty and acceleration value. The sign indicates which direction the motor will run. The acceleration values are not signed. This command is used to drive the motor by PWM and using an acceleration value for ramping. Accel is the rate per second at which the duty changes from the current duty to the specified duty.

```
Send: [Address, 52, Duty(2 bytes), Accel(2 Bytes), CRC(2 bytes)]
Receive: [0xFF]
```

The duty value is signed and the range is -32768 to +32767(eg. +-100% duty). The accel value range is 0 to 655359(eg maximum acceleration rate is -100% to 100% in 100ms).

53 - Drive M2 With Signed Duty And Acceleration

Drive M1 with a signed duty and acceleration value. The sign indicates which direction the motor will run. The acceleration values are not signed. This command is used to drive the motor by PWM and using an acceleration value for ramping. Accel is the rate at which the duty changes from the current duty to the specified dury.

```
Send: [Address, 53, Duty(2 bytes), Accel(2 Bytes), CRC(2 bytes)]
Receive: [0xFF]
```

The duty value is signed and the range is -32768 to +32767 (eg. +-100% duty). The accel value range is 0 to 655359 (eg maximum acceleration rate is -100% to 100% in 100ms).

54 - Drive M1 / M2 With Signed Duty And Acceleration

Drive M1 and M2 in the same command using acceleration and duty values for each motor. The sign indicates which direction the motor will run. The acceleration value is not signed. This command is used to drive the motor by PWM using an acceleration value for ramping. The command syntax:

```
Send: [Address, CMD, DutyM1(2 bytes), AccelM1(4 Bytes), DutyM2(2 bytes),
      AccelM1(4 bytes), CRC(2 bytes)]
Receive: [0xFF]
```

The duty value is signed and the range is -32768 to +32767 (eg. +-100% duty). The accel value range is 0 to 655359 (eg maximum acceleration rate is -100% to 100% in 100ms).

55 - Read Motor 1 Velocity PID and QPPS Settings

Read the PID and QPPS Settings.

```
Send: [Address, 55]
Receive: [P(4 bytes), I(4 bytes), D(4 bytes), QPPS(4 byte), CRC(2 bytes)]
```

56 - Read Motor 2 Velocity PID and QPPS Settings

Read the PID and QPPS Settings.

```
Send: [Address, 56]
Receive: [P(4 bytes), I(4 bytes), D(4 bytes), QPPS(4 byte), CRC(2 bytes)]
```

61 - Set Motor 1 Position PID Constants

The RoboClaw Position PID system consist of seven constants starting with P = Proportional, I= Integral and D= Derivative, MaxI = Maximum Integral windup, Deadzone in encoder counts, MinPos = Minimum Position and MaxPos = Maximum Position. The defaults values are all zero.

```
Send: [Address, 61, D(4 bytes), P(4 bytes), I(4 bytes), MaxI(4 bytes),
      Deadzone(4 bytes), MinPos(4 bytes), MaxPos(4 bytes), CRC(2 bytes)]
Receive: [0xFF]
```

Position constants are used only with the Position commands, 65,66 and 67 or when encoders are enabled in RC/Analog modes.

62 - Set Motor 2 Position PID Constants

The RoboClaw Position PID system consist of seven constants starting with P = Proportional, I= Integral and D= Derivative, MaxI = Maximum Integral windup, Deadzone in encoder counts, MinPos = Minimum Position and MaxPos = Maximum Position. The defaults values are all zero.

```
Send: [Address, 62, D(4 bytes), P(4 bytes), I(4 bytes), MaxI(4 bytes),
      Deadzone(4 bytes), MinPos(4 bytes), MaxPos(4 bytes), CRC(2 bytes)]
Receive: [0xFF]
```

Position constants are used only with the Position commands, 65,66 and 67 or when encoders are enabled in RC/Analog modes.

63 - Read Motor 1 Position PID Constants

Read the Position PID Settings.

```
Send: [Address, 63]
Receive: [P(4 bytes), I(4 bytes), D(4 bytes), MaxI(4 byte), Deadzone(4 byte),
            MinPos(4 byte), MaxPos(4 byte), CRC(2 bytes)]
```

64 - Read Motor 2 Position PID Constants

Read the Position PID Settings.

```
Send: [Address, 64]
Receive: [P(4 bytes), I(4 bytes), D(4 bytes), MaxI(4 byte), Deadzone(4 byte),
            MinPos(4 byte), MaxPos(4 byte), CRC(2 bytes)]
```

65 - Buffered Drive M1 with signed Speed, Accel, Deccel and Position

Move M1 position from the current position to the specified new position and hold the new position. Accel sets the acceleration value and deccel the decceleration value. QSpeed sets the speed in quadrature pulses the motor will run at after acceleration and before decceleration.

```
Send: [Address, 65, Accel(4 bytes), Speed(4 Bytes), Deccel(4 bytes),
            Position(4 Bytes), Buffer, CRC(2 bytes)]
Receive: [0xFF]
```

66 - Buffered Drive M2 with signed Speed, Accel, Deccel and Position

Move M2 position from the current position to the specified new position and hold the new position. Accel sets the acceleration value and deccel the decceleration value. QSpeed sets the speed in quadrature pulses the motor will run at after acceleration and before decceleration.

```
Send: [Address, 66, Accel(4 bytes), Speed(4 Bytes), Deccel(4 bytes),
                Position(4 Bytes), Buffer, CRC(2 bytes)]
Receive: [0xFF]
```

67 - Buffered Drive M1 & M2 with signed Speed, Accel, Deccel and Position

Move M1 & M2 positions from their current positions to the specified new positions and hold the new positions. Accel sets the acceleration value and deccel the decceleration value. QSpeed sets the speed in quadrature pulses the motor will run at after acceleration and before decceleration.

```
Send: [Address, 67, AccelM1(4 bytes), SpeedM1(4 Bytes), DeccelM1(4 bytes),
            PositionM1(4 Bytes), AccelM2(4 bytes), SpeedM2(4 Bytes), DeccelM2(4 bytes),
                  PositionM2(4 Bytes), Buffer, CRC(2 bytes)]
Receive: [0xFF]
```

119 - Buffered Drive M1 with Position

Move M1 from the current postion to the specified new postion and hold the new postion. Default accel, decel and speeds are used.

```
Send: [Address, 119, Position (4 bytes), Buffer, CRC (2 bytes)]
Receivr: [0xFF]
```

120 - Buffered Drive M2 with Position

Move M2 from the current postion to the specified new position and hold the new position. Default accel, decel and speeds are used.

```
Send: [Address, 120, Position (4 bytes), Buffer, CRC (2 bytes)]
Receive: [0xFF]
```

121 - Buffered Drive M1/M2 with Position

Move M1 and M2 from the current position to the new specified position and hold the new position. Default accel, decel and speeds are used.

```
Send: [Address, 121, M1Position (4 bytes), M2Position, (4 bytes), Buffer, CRC (2 bytes)]
Receive; [0xFF]
```

122 - Buffered Drive M1 with Speed and Position

Move M1 form the current position to the specified new position and hold the new position. Default accel and decel are used.

```
Send; [Address, 122, Speed (4 bytes), Position (4 bytes), Buffer, CRC (2 bytes)]
Receiver: [0xFF]
```

123 - Buffered Drive M2 with Speed and Position

Move M2 from the current position to the specified new position and hold the new position. Default accel and decel are used.

```
Send: [Address, 123, Speed (4 bytes), Position (4 bytes), Buffer, CRC (2 bytes)]
Receiver: [0xFF]
```

124 - Buffered Drive M1/M2 with Speed and Position

Move M1 and M2 from the current postion to the new specified position and hold the new position. Default accel and decel are used.

```
Send: [Address, 124, SpeedM1 (4 bytes), PositionM1 (4 bytes), SpeedM2 (4 bytes),
PositionM2 (4 bytes), Buffer, CRC (2 bytes)]
Receive: [0xFF]
```

Warranty

Basicmicro warranties its products against defects in material and workmanship for a period of 1 year. If a defect is discovered, Basicmicro will, at our sole discretion, repair, replace, or refund the purchase price of the product in question. Contact us at sales@basicmicro.com. No returns will be accepted without proper authorization.

Copyrights and Trademarks

Disclaimer

Basicmicro cannot be held responsible for any incidental or consequential damages resulting from use of products manufactured or sold by Basicmicro or its distributors. No products from Basicmicro should be used in any medical devices and/or medical situations. No product should be used in any life support situations.

Contacts

Email: sales@basicmicro.com
Tech support: support@basicmicro.com
Web: http://www.basicmicro.com

Discussion List

A web based discussion board is maintained at http://www.basicmicro.com

Technical Support

Technical support is available by sending an email to support@basicmicro.com, by opening a support ticket on the Ion Motion Control website or by calling 800-535-9161 during normal operating hours. All email will be answered within 48 hours.

B Component List

Komponent	Antall		
PI4 MODEL B/4GB - Raspberry Pi 4 1.5GHz Quad-Core, 4GB RAM	1		
Powerbank for å drive RPI			
131:1 Metal Gearmotor 37Dx73L mm 12V with 64 CPR Encoder (Helical Pinion)	4		
RoboClaw 2x15A Motor Controller (V5E) x2	1		
Chassis, planlegger å lasekutte dette.			
Leverandør	**Artikkelnummer hos leverandør**	**Produsentens delenummer (valgfritt)**	
Elfa Distrelec	301-52-781	PI4 MODEL B/4GB	
Pololu	3285		
Link til komponenten			
https://www.elfadistrelec.no/no/raspberry-pi-5ghz-quad-core-4gb-ram-raspberry-pi-pi4-model-4gb/p/30152781?track=true		**Pris**	
https://www.elfadistrelec.no/no/raspberry-pi-adapter-mikro-hdmi-til-hdmi-1m-raspberry-pi-t7689ax/p/30152784		kr 540,00	
https://www.pololu.com/product/3285		kr 37,30	
		$89.95	

https://www.pololu.com/product/4756
https://www.basicmicro.com/RoboClaw-2x30A-Motor-Controller_p_9.html

C Source Code

Source code can be found on GitHub; https://github.com/krydderen/Mecanum-Robot