Blodwen rover documentation

James Finnis, jcf1@aber.ac.uk

March 29, 2017

Contents

0.1 Checklists

Much of this data is repeated in more detail later in the document, which should be read before use (at least up to and including section 2.2.2).

0.2 Rover startup checklist

1. Ensure the battery inside the main compartment is charged.
2. Ensure main compartment lid is closed.
3. Switch on PCs using the PC power button.
4. If it's installed, you may wish to switch off the science PC (the right-hand PC). The button is on the front of the PC, near the floor of the compartment.
5. Check current is less than 2A.
6. Connect a laptop to the rover's WiFi network (the SSID contains "EnGenius").
7. SSH into the rover: host 192.168.0.10, username *blodwen*, password *robotics.*
8. Switch on the motors with the other power switch on the lid.
9. Check that the underside lights go steady on, then flash at about 1Hz.
10. Check current is about 1.8 to 2A.
11. (The list from here implies direct rover control via Angort)
12. In the terminal SSHing into the rover, issue the commands

    ```
    cd r
    ./rover
    ```

13. Ensure the connection is made — `READY` should appear and the Angort prompt `1|0>`.
14. Issue the commands

    ```
    reset calib
    ```

    to reset the initial boot exception state and send calibration data.
15. Now refer to section 2.2.2 for commands.
16. In the case of an emergency, hammer CTRL-D (or CTRL-C if that fails) to send an emergency stop. All gains will be zeroed, and `reset calib` must be reissued to restart the rover.

0.3 Rover shutdown checklist

1. Exit the Angort interpreter with CTRL-D or `quit`.
2. Switch off the motors.
3. Power down the locomotion PC with `sudo poweroff`.
4. Wait for the power consumption to drop to about 1A (indicating PC powerdown).
5. Switch off the PC power using the other power switch.

1 Introduction

The rover's control architecture consists of three layers, as shown in figure 1:

A PC running the main control software which is interfaced to the hardware via a register interface. It sends register read/write commands to the Arduino. In many situations, this control layer will itself be a server responding to commands sent by another system.

An Arduino accepting register read/write commands which passes them over I^2C to nine motor controllers. This is referred to in the text as the **master controller.** It also has a set of registers of its own for global system information.

Nine motor controller boards [1] each containing an ATMega328p processor (effectively an Arduino) to handle the PI control algorithms and respond to register read/write commands for two motors, using an L298P dual H-bridge to drive the motors. Each board may also have an analogue input for the chassis potentiometer.

These are referred to in the text as **slave** or **secondary controllers.**

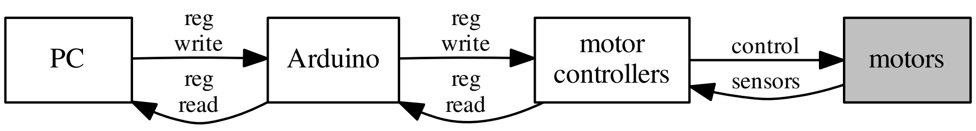


Figure 1: Overall architecture

All software is written in C++, with the both master and slave controller code being written in the Arduino variant, burned directly into the chip using ICSP.

1.1 Wheel pair architecture

The rover has six wheels, each pair of which is controlled by three slave controllers as shown in figure 2. This shows that there are two distinct kinds of slave controllers:

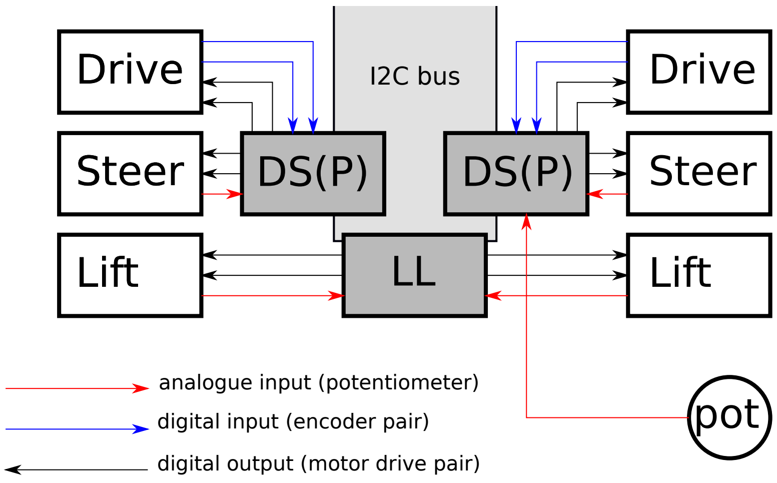
Two Drive/Steer/(Position) controllers per pair, which each control the driving and steer motor on one wheel. One of these also receives an input from one of the three chassis configuration potentiometers. On the other, this input is ignored.

One Lift/Lift controller per pair, which control the lift motors on both wheels.

The reason for this split is that each controller board can only handle two analogue inputs (in addition to the two internal current sensor channels.)

The controllers and the wheels they control are numbered as in figure 3. The wheel numbering scheme is that given in the rover documentation.

[1]Sparkfun "serial controlled motor driver" boards, ROB-09571



Drive
I2C bus
Drive
Steer
DS(P)
DS(P)
Steer
Lift
LL
Lift
analogue input (potentiometer)
digital input (encoder pair)
digital output (motor drive pair)
pot

Figure 2: Hardware overview of a single wheel pair

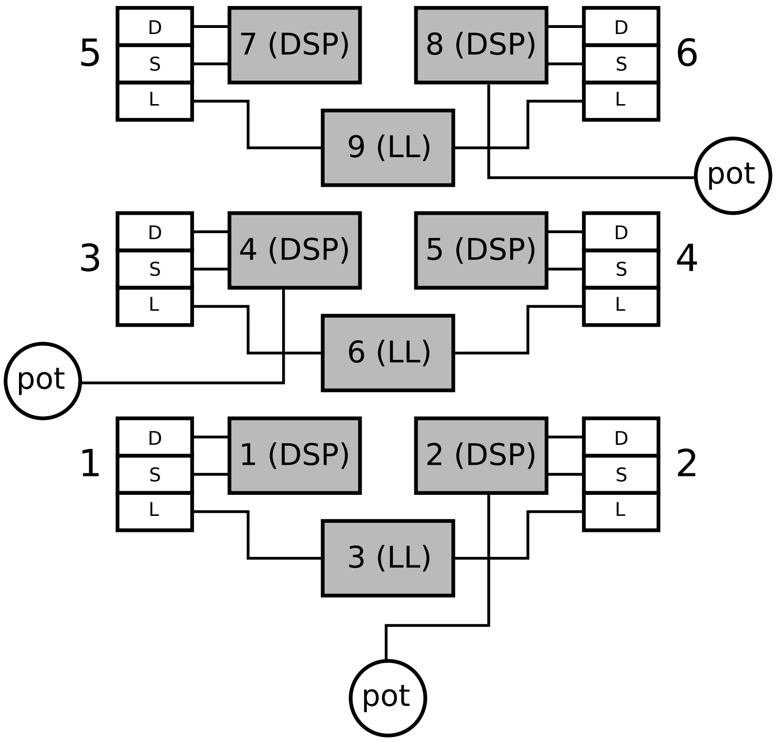


Figure 3: Numbering of controllers and wheels

2 Using the rover

The rover is shown in Figure 4. Note that the rover's "front" is the obvious, wedged end.

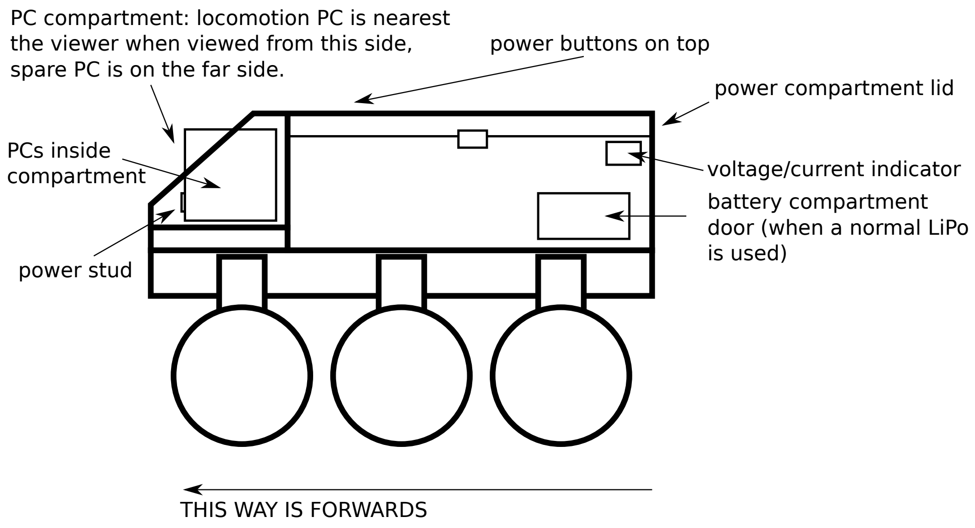

PC compartment: locomotion PC is nearest the viewer when viewed from this side, spare PC is on the far side.
power buttons on top
power compartment lid
PCs inside compartment
voltage/current indicator
battery compartment door (when a normal LiPo is used)
power stud
THIS WAY IS FORWARDS

Figure 4: The Blodwen rover

- The main compartment contains a LiPo battery suitable for most applications. Ensure this is charged prior to use.
- The battery compartment has space for an extra LiPo battery. If you are using this, the connector needs to be passed through from the main compartment. Ensure you connect with the correct polarity.
- Switch on the rover PCs by pressing the PC power button on the roof of the chassis. **Once this has been done, do not power off the rover without shutting the locomotion PC down from a terminal connection.**
- Check that the voltage and current are nominal (11.5-12V, < 3A) using the indicator. If they are not, shut down the rover and power off.
- The PCs are in the wedge-shaped compartment at the front of the rover. If another PC is in use, switch off the spare PC. This is the PC located on the right side of the rover. The power studs are located at the bottom of the PC front panels. Figure 5 shows the arrangement of the PCs more clearly. Power consumption should drop to ~ 1.5A.
- To power up the motor, drive and chassis electronics the motor power switch on the lid should be pressed.
- The lights under the chassis should come on briefly, and then begin to flash approximately once a second. Power consumption should rise to $\sim 1.8 - 2$ A. The hardware is now ready to receive commands. Proceed to section 2.2 to find out how to command it with the Angort language.

2.1 Detailed explanation of LED flashes

This guide may help diagnose wiring problems inside the rover chassis:

- Each controller has two LEDs. As the rover boots, each controller should show the lowest 2 bits of its I^2C address (1-9) on the LEDs for about half a second. This causes the initial lighting state.

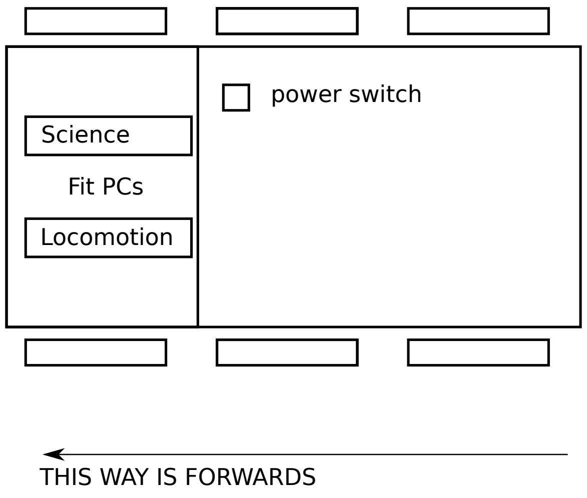


Figure 5: The Blodwen rover, from above

- The master and slaves will start in the boot exception state, so the LEDs will be steady on. Shortly after boot, the master will attempt to send resets to each slave.
- If the send fails, the I^2C bus will hang, so the master's watchdog timer will time out and the master will reboot and try again. A common failure mode is repeated reboot cycles, where the the good slaves are resetting successfully from the boot exception, shown by repeated off gaps, while the bad slave will just show a steady exception state (see below).
- If the slaves all reset successfully, the master will receive the acknowledgements and begin awaiting PC packets. It will also send a "ping" message to the slaves once per second. If a slave receives this, it will briefly flash an LED (LED1 on a drive/steer unit, LED2 on a lift/lift unit). Therefore, the LEDs should flicker once per second on all slaves once the rover is ready to receive commands.

In the exception state, the two LEDs will stay on showing:

- LED1: motor 1 is in exception
- LED2: motor 2 is in exception
- both LEDs: the exception is not a motor exception

2.1.1 Using the remote control

(Note: this hasn't been tested for a while). Turning on the remote control causes the master's main loop to ignore any instructions sent from the PC, and also recalibrates the rover (i.e. a new set of PID and extent parameters will be sent). Therefore, the rover will need to be recalibrated (by calling the **calibrate()** method) if PC control is desired again.

When the remote control is active, a blue LED should shine under the rover, clearly visible on the driving surface. The controls are as follows:

- Left stick forward/backward : drive speed
- Right stick left/right: turn steer wheels
- Right stick forward: alternate steer mode

By default, steering operates using Ackermann steering on the front and back wheel pairs in opposite directions, with the centre wheels not commanded. If the right stick is pushed forwards to its limit, the same steering angle will be applied to all wheels, permitting "crabbing".

2.2 Controlling the rover with Angort

The rover locomotion PC has the language Angort installed, with a set of scripts and native C++ word definitions to allow easy control. To run the language interpreter, one must connect to the locomotion PC via SSH. Both PCs are connected to a WiFi router installed inside the rover.

2.2.1 Connecting to the rover and starting Angort

- Power up the rover. Power down the science/spare PC immediately (see above).
- Connect to the rover on the EnGenius router via SSH (IP address 192.168.0.10, username **blodwen**, password **robotics**).
- `cd r` to get into the appropriate directory (this is a soft link to a subdirectory).
- Power on the electronics by closing the battery door.
- `./rover` to start a special version of the Angort interpreter with rover control words added. The script `script.ang` will automatically be loaded — it contains a useful set of word definitions to enhance those built in.
- A few lines should show the connection to the master Arduino being established, and initial rover library setup. The Angort prompt

  ```
  1|0 >
  ```

  should appear (the left number is the number of GC-objects in the system, the right number is the depth of the stack). If this initial connection phase fails, check that the safety switches are closed and that the microcontrollers are all flashing as they should.

2.2.2 Rover control commands

The rover boots in an exception state as a safety feature, and also has all PID gains in the motors set to zero. Before it can move, all the controllers need to have their exceptions reset, and correct PID gains uploaded. To do this, issue the following commands **but be prepared to hammer CTRL-D**:

```
reset calib
```

This may result in some small motion of the wheels if the rover has been moved by hand or is in an unusual configuration for some other reason. **If the motion is dangerous, hit CTRL-D** to reset the system.

Once this has been done, movement commands may be issued. If they appear to do nothing, check that `reset` and `calib` have been run. The following table shows examples of the most common commands for controlling the rover:

`1000 d`	**set all drive motors to speed 1000** forwards, towards the wedged end with the PCs.
`-2500 d`	**set all drive motors to speed -2500**, i.e. backwards — note that 2500 is about the highest safe speed.
`40 t`	**turn** 40°**,** done by setting the front steer wheels to the given angle and the rear wheels to the opposite.
`-40 t`	**turn the opposite way** to the previous command.
`40 a`	**set all steer motors to** 40°**,** useful for "crabbing".
`20 setliftall`	**set all lift motors** to 20°.
`20 1!lift`	**set lift lift motor 1** to 20°. The notation `<value> <motornumber>!<motortype>` can be used to set individual drive, steer and lift motor speeds and positions. **Be careful** — it's possible to make the rover perform potentially damaging actions by setting individual motors.
`1?lift.`	**get and print the requested position for lift motor 1.**

2.3 Monitoring the rover graphically

The Angort scripting system, once connected to the rover hardware, constantly sends a stream of key/value pairs over UDP to port 13231 on host 192.168.0.100 (the first router address available to WiFi hosts). The `monitor` program can read this data and display it graphically with an appropriate configuration file.

The program is available at `https://github.com/jimfinnis/monitor.git` and is a Qt4 application. It requires the `libmarblewidget` library to compile (adding support for mapping; the monitor is also used for robotic boat trials). A suitable configuration file is provided — once compiled, run the program with

```
monitor -f ./roverconfig
```

This configuration will display the desired and actual drive motor speeds, the actual lift and steer positions, the current readings for the 6 drive wheel controllers, the temperatures of all 9 controllers above ambient. Extra variables are shown for hormone settings and traversed distance, but these are provided by code built for particular experiments beyond the scope of this document.

3 Hardware details

The hardware is implemented as three motherboards, onto each of which are mounted three motor controller daughterboards. Each motherboard controls one pair of motors. Screw connectors on the board connect the motors, potentiometers and encoders; also the I^2C and power busses.

A circuit diagram and PCB layout for a single board can be found in the *eagle* directory.

Each controller daughterboard has a unique I^2C address, which is set as a parameter of the *upload* script.

3.1 Procedure for updating the master firmware

Ino (`www.inotool.com`) must be installed to build the firmware, which is then uploaded over the USB connection:

```
cd rover/firmware/master
(edit code)
ino build
./upload
```

3.2 Procedure for updating the slave firmware

Ino (`www.inotool.com`) must be installed to build the firmware, which is then uploaded using a USBTiny ICSP programmer:

```
cd rover/firmware/slave
(edit code)
```

and then either

```
./blift
```

to build firmware for a lift/lift controller, or

```
./bdrive
```

to build firmware for a drive/drive controller. Then connect the ICSP to the controller and run

```
./upload N
```

where N is the controller number. It is possible to upload the software to several different controllers without rebuilding it:

```
./blift
./upload 3
./upload 6
./upload 9
./bdrive
./upload 1
./upload 2
./upload 4
./upload 5
./upload 7
./upload 8
```

is a typical scenario. Of course, you'll have to plug the programmer onto the ICSP pins for the each board for each command.

3.3 Pullup resistors

Note that the I^2C internal pullup resistors are disabled on all the microcontrollers. Therefore a pair of pullups of a suitable value are added, pulling up to 3V. These are on a shield board mounted onto the Arduino, along with an additional pullup for the OneWire bus managing the temperature sensors.

3.4 Controller assignments

Figure 3 shows the controller/wheel assignments. For example, the steer and drive wheel of controller 1 is wheel number 5, while controller 3 controls the lift motors on wheels 5 and 6.

3.5 Connections

The following connections are used on the master:

- **USB** connection for control
- **A4** for I^{2}C SDA line
- **A5** for I^{2}C SCL line

The following connections are used on the DS(P) slaves:

- **M1+/M1-** for drive motor
- **M2+/M2-** for steer motor
- **ADC6** for steer potentiometer
- **ADC7** for chassis potentiometer if present
- **TX** for drive motor encoder channel A
- **RX** for drive motor encoder channel B

The following connections are used on the LL slaves:

- **M1+/M1-** for lift motor 1
- **M2+/M2-** for lift motor 2
- **ADC6** for lift potentiometer 1
- **ADC7** for lift potentiometer 2

The layout of a single board is shown in figure 6, while the layout of all the boards inside the body of the rover is shown in figure 7.

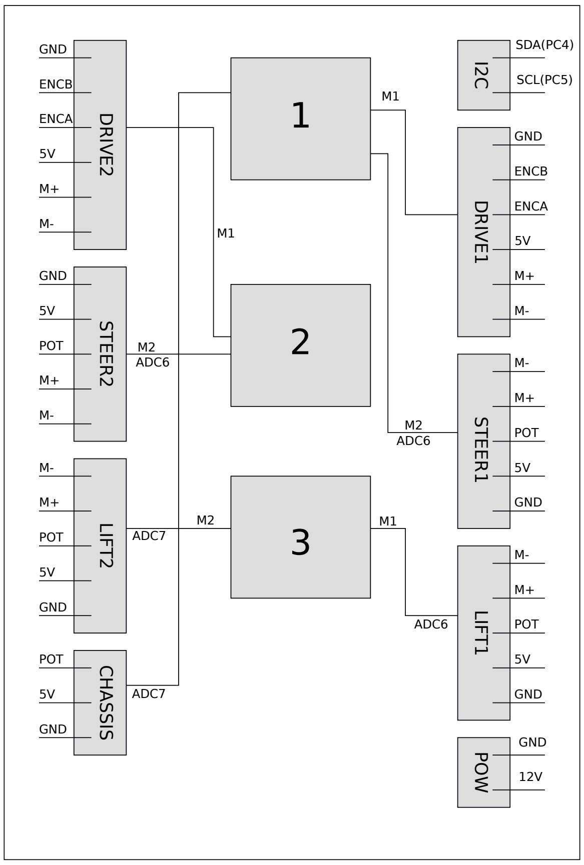


Figure 6: Board connections

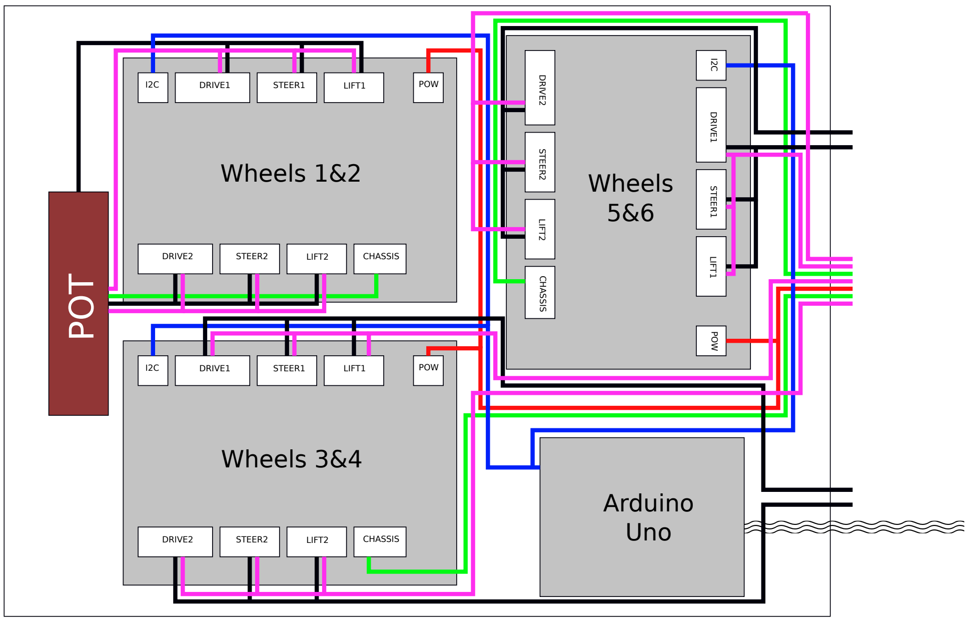


Figure 7: Internal layout of the main rover compartment

3.5.1 Connection labelling

Wires running to the motors — both sensor and motor control wires — are marked using coloured rings of insulation. Figures 8 and 9 show the schemata used. Note that in both diagrams, the proximal (motor controller board) end is shown to the right.

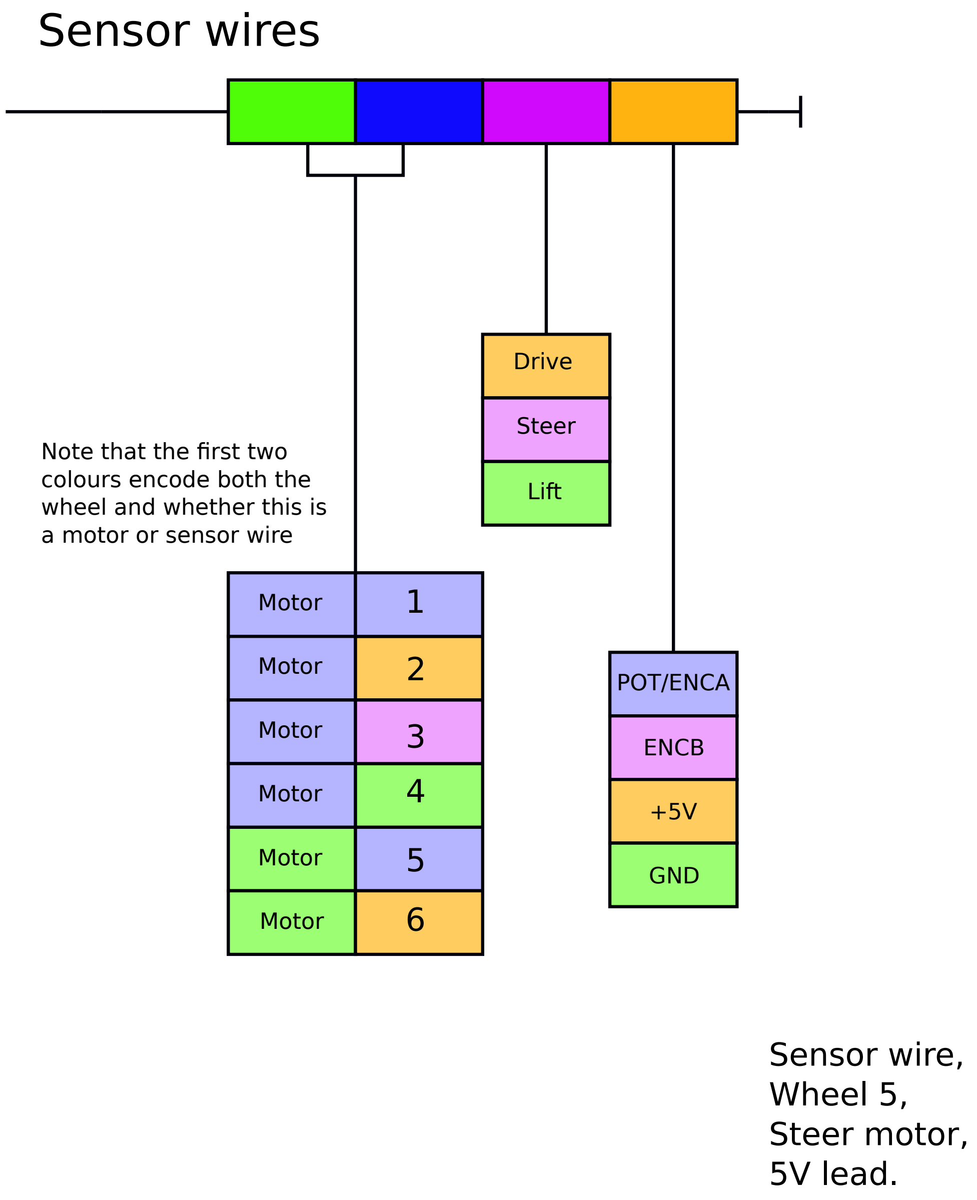

Sensor wires
Note that the first two colours encode both the wheel and whether this is a motor or sensor wire
Motor 1
Motor 2
Motor 3
Motor 4
Motor 5
Motor 6
Drive
Steer
Lift
POT/ENCA
ENCB
+5V
GND
Sensor wire, Wheel 5, Steer motor, 5V lead.

Figure 8: Sensor wire labelling scheme

Other colours:

- I^2C bus : blue and white twisted pair — the white line is the SDA, the blue line is SCL.
- Power bus : black and red twisted pair — red is +12V, black is ground
- Power input : brown is +12V, green/yellow is ground
- OneWire bus : orange and orange/white twisted pair — orange is ground, orange/white is data

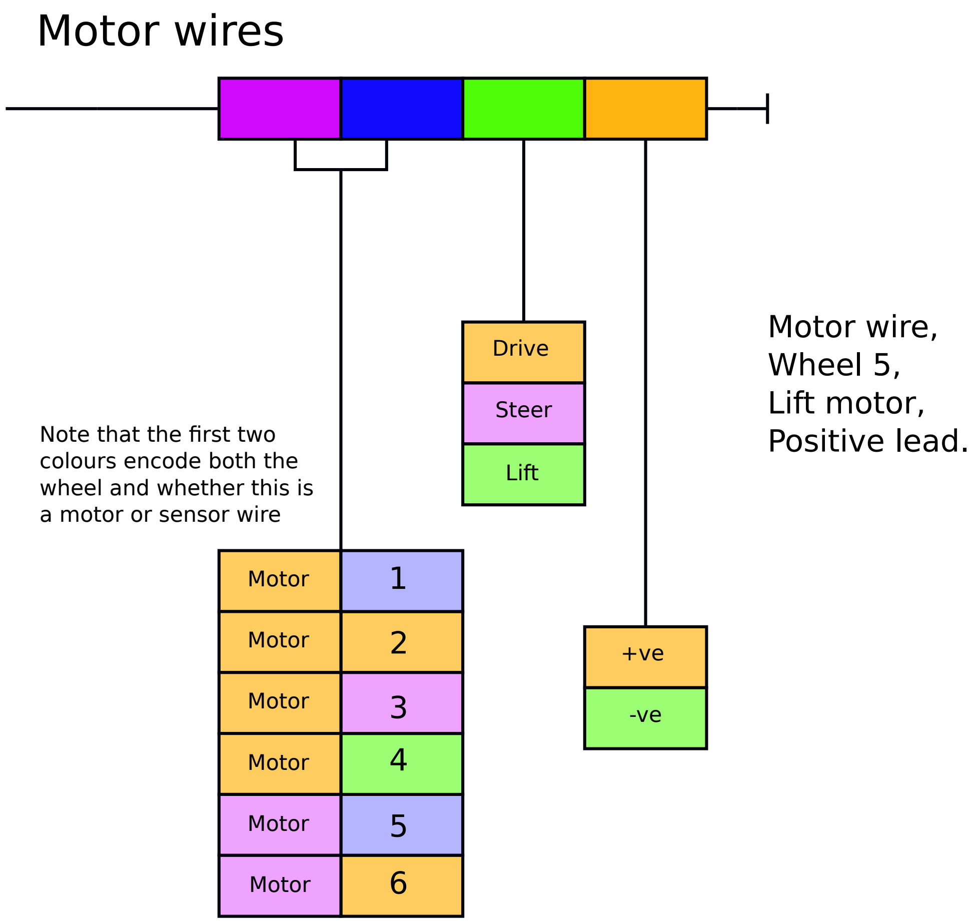

Motor wires
Note that the first two colours encode both the wheel and whether this is a motor or sensor wire
Motor
1
Motor
2
Motor
3
Motor
4
Motor
5
Motor
6
Drive
Steer
Lift
+ve
-ve
Motor wire, Wheel 5, Lift motor, Positive lead.

Figure 9: Motor wire labelling scheme

- Chassis lines: red is +5V, black is ground, yellow is data. There's no ID marking; the line can be clearly traced to its potentiometer by eye.

3.6 Remote control connections

The remote control is connected to the master Arduino Uno as shown in figure 10.

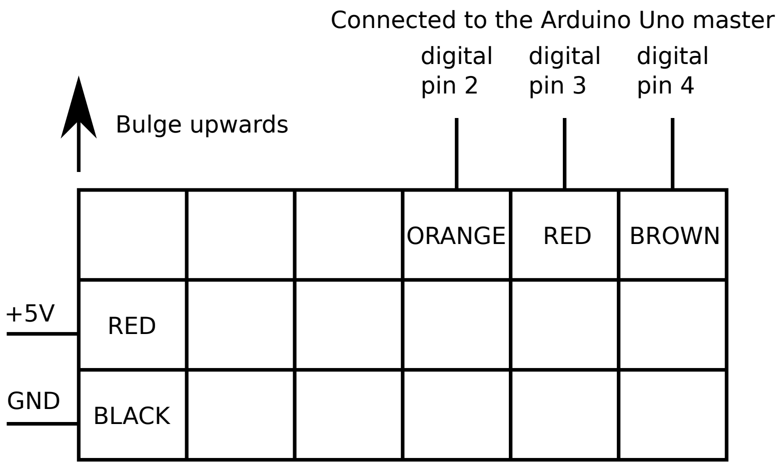


Figure 10: Servo receiver connections

4 Onboard PC library

The onboard PC currently uses a library of C++ classes to abstract the communications protocol, the details of I^2C, and the vagaries of the firmware register set (for the reasons given in section 6.3, working out which slave controller to send commands to in order to turn a particular wheel is not straightforward.)

This section contains a brief overview of the C++ classes, describing their functions but not going into any details of methods.

4.1 Rover

The *Rover* class is the top-level class. The utilising code is expected to instantiate a single object of this class. Calling the initialisation method *init* with a character device name and a baud rate will start a connection to the master controller (the Arduino) which will be reset. Once the connection is made, calling *update* periodically will request state information about all the devices.

Other methods allow the user access to motor objects, which will allow the user to change motor parameters and required speeds/positions; and to motor data blocks, allowing users to monitor the motors. The data blocks will contain information received at the last *update.*

Note that the communications over I^2C have been kept to a minimum — the rover uses the "set read set" firmware protocol command (see section 6) to send a set of registers. When a simple 3 byte "read set" command is set with a set number, the slave will send a packet of all the registers in that set.

Internally, the rover uses the *WheelPair* class to manage triplets of slave controller boards, which in turn use *MotorDriverData* objects to manage reading data from the two types of slave (the *Motor* class handles writing values.)

Communications are handled by *SlaveProtocol*, which in turn uses *SerialComms.*

4.2 Motor

This class contains methods to send parameters (such as PID gains) to a motor, set its required position or speed, and reset any exception condition[2]. There are three subclasses of *Motor*, one for each motor type: *DriveMotor*, *SteerMotor* and *LiftMotor.* They present similar interfaces to the user. Pointers to motor objects can be obtained by calling methods in *Rover.*

4.3 MotorData

This is a class containing general motor monitoring data, such as:

- error, error integral, error derivative
- control value being sent to the motor
- interval between control ticks
- current
- actual position or speed
- exception state

There are three subclasses, one for each motor type: the lift and steer subclasses are identical and the drive subclass also contains odometry data.

Pointers to motor data objects can be obtained by calling methods in *Rover.* Note that the data in these objects is **only updated when *update* is called.**

[2]for example, if a motor's current exceeds its overcurrent threshold, the slave will detect this and shut the motor down.

4.4 StatusListener

The communications system can inform third parties who register with it of changes to the comms status — for example, disconnection or protocol error. This is done by creating a *StatusListener* subclass and registering it with the *SerialComms* object, available through *Rover*.

4.5 Calibration settings

Default calibration settings for the motors can (and should) be sent to the controllers by calling `Rover::calibrate()` in the file `rover.cpp`.

4.6 Example code

```
int main(int argc,char *argv[]){
    Rover r;

    try {
        // set up the rover given the comms port and the
        // baud rate.
        r.init("/dev/ttyACM0",115200);

        // send default calibration
        r.calibrate();

        // some parameter data we're going to change;
        // you probably won't do this - it's just an illustration.

        MotorParams params = {
            0.004,0,0,   //PID
            0,0,         //integral cap and decay
            300,         //overcurrent threshold
        };

        // change parameters on the drive motors
        // and set a speed for them

        for(int i=1;i<=6;i++) { // motors are 1 to 6 as in the documentation
            Motor *m = r.getDrive(i); // get each drive motor

            // get a pointer to its parameters
            MotorParams *p = m->getParams();

            // copy some other data into them
            *p = params;

            // and send the changes
            m->sendParams();

            // and set a speed
            m->setRequired(1000);
        }

        for(;;){
            usleep(10000); // wait 1/100 s
            r.update(); // update the rover

            // get drive motor 1 data
            DriveMotorData *d = r.getDriveData(1);
            printf("%f\n",d->actual); // print actual speed
        }

    } catch(SlaveException e) {

        // slave exceptions are thrown by protocol and comms errors
        printf("Error in rover communication: %s\n",e.msg);
        return 0;
    }
}
```

5 Organisation of firmware source code

You probably don't need to read this section unless you're maintaining the code for the master controller.

The source code is kept in the `firmware` directory:

- **common** holds useful definitions for both slave and master, such as register structures and assignments.
- **master** holds code for the master (Arduino) side;
- **slave** holds code used by the slave (motor driver) side.

5.1 Common register table

The *common* directory contains files describing the registers used in the system:

- **regDefinitions** is a file with a special syntax from which `regsauto.h`, `regsauto.cpp` and `regsauto.ino` are built by the `build` script — see below for details.
- **regs.h** defines the *Register* structure, which encapsulates a one or two byte value which may or may not be mapped onto a arbitrary range of floating point values (see below).
- **regsauto.h**, **regsauto.cpp** and **regsauto.ino** are generated automatically by the *build* script. They contain register definitions — the include file contains the register names and numbers, while the identical `cpp` and `ino` files contain the *Register* structures describing each register: number of bits, writability, float range (or values indicating "unmapped"). These files are compiled into master, slave and PC code using symbolic links from their respective source directories.

5.2 Register definition file

The registers are defined in the register definition file `regDefinitions` which can be found in the `firmware/common` directory. It consists of a number of blocks, each of which specifies a different register table (the two types of slave controller have different register sets.) Named blocks are preceded by an unnamed block, which specifies a block of registers common to all tables.

A Haskell script, `regparse.hs`, is used to generate the `.h` and `.cpp/ino` files automatically, which is itself invoked by running `build`.

5.2.1 Register value mapping

There are two types of register: **unmapped**, which are straightforward 8 or 16 bit values (depending on the size of the register); and **mapped**, where a floating point range is mapped onto the underlying range.

These mappings are defined in the register definition file. For example:

```
RESET           writable  1 unmapped      "reset bits"
```

defines a register called RESET, which is a writable 1-byte unmapped value; whereas

```
DRIVE_REQSPEED writable  2 range -4000 4000  "required speed"
```

defines a register in which the floating point range [-4000,4000] is mapped onto the fixed point range [0,65535]. Mapping and unmapping are done using the `map()` and `unmap()` methods in Register; the PC and firmware deal with this automatically as required (see, for example, the `i2c_interface.ino` code in the slave, which catches register changes as they happen, unmaps them, and uses them to set values in the control system.)

If map or unmap are called on an unmapped register, the value is unchanged (apart from its type.)

6 Master firmware and PC-Master protocol

The master firmware takes commands from the USB serial port and communicates with the slaves over I^2C.

6.1 PC-Master protocol

This protocol is a binary serial protocol. Messages from the PC to the master have the following form:

- 1 byte: number of bytes in packet (including this one)
- 1 byte: command number and slave ID. The slave ID occupies the top 4 bits, the command number the bottom 4. If the slave ID is zero, the registers to be accessed are local registers on the master. The commands are:
    - 2 : write command
    - 3 : read command
    - 4 : set read set command
- Remaining bytes: payload, see below

The response depends on the command, as described in the following subsections which go into the details of the each command.

6.1.1 Write command, code 2

Payload:

- 1 byte: number of writes. **For each write:**
    - 1 byte: register number
    - 1-2 bytes (depending on size of register, obtained from register table) : value of register, LSB first

Response: 1 byte, value zero.

6.1.2 Read command, code 3

Payload is a single byte, the index of the read set to read. Response is, **for each register in the given read set:**

- 1-2 bytes (depending on size of register, obtained from register table) : value of register, LSB first

6.1.3 Set read set command, code 4

The first byte in the payload is index of the read set to change. Each subsequent byte in the payload is the index of a register which should be in the read set. The next read command for that read set will result in the value of these registers being sent, in the same order, to the PC. **Read sets are shared across all devices** so the slave number is irrelevant.

Response is 1 byte: the number of registers in the new read set.

6.2 Examples of PC-Master protocol commands

For the following examples, registers 0 and 1 are 1 byte registers, while 2 and 3 are 2 byte registers.

6.2.1 Example 1: write

PC to master: `0d 32 04 00 ff 01 ff 02 01 00 03 02 00`
Response: `00`

- `0d 32`: 13 bytes, command 2 (write) for slave 3
- `04`: contains 4 writes
- `00 ff` write `ff` to register 0
- `01 ff` write `ff` to register 1
- `02 01 00` write `0001` to register 2
- `03 02 00` write `0002` to register 3

Response is just zero to acknowledge.

6.2.2 Example 2: set read set

PC to master: `06 04 01 00 01 02 03`
Response: `04`

- `06 04`: 6 bytes, command 4 (set read set) for slave 0 (this value is ignored.)
- `01`: change read set 1
- `00 01 02 03`: registers 0, 1, 2 and 3 should be the new read set 1.

The response is `04`, the number of registers in the read set.

6.2.3 Example 3: read

PC to master: `03 12 01`
Response: `ff ff 01 00 02 00`

- `03 12`: 3 bytes, command 2 (read) for slave 1
- `01`: read set 1

The response is the values of the registers:

- `ff` : register 0 contains `ff`
- `ff` : register 1 contains `ff`
- `01 00` : register 2 contains `0001`
- `02 00` : register 3 contains `0002`

6.3 Firmware registers

In the following tables, the columns have the following meanings:

- **ID** : the ID number of the register
- **Symbol** : the C++ `#define` symbol given to that ID
- **n** : the number of bytes for this register
- **W** : whether the register is writable or not
- **Mapping** : if the register is mapped onto a floating point range, the interval is specified. "Unmapped" indicates an integer or bitfield register.
- **Description** : a brief description

6.3.1 Common block

These registers are common to all devices *except* the master device, which is the local master controller and has the special ID 0.

ID	Symbol	n	W	Mapping	Description
0	REG_RESET	1	Y	unmapped	reset bits - beware race conditions
1	REG_TIMER	2		unmapped	millis since start
2	REG_INTERVALI2C	2		unmapped	interval between I2C ticks
3	REG_STATUS	2		unmapped	see status flags in regs.h
4	REG_DEBUGLED	1	Y	unmapped	debugging LEDs, turns on for some time
5	REG_EXCEPTIONDATA	2	Y	unmapped	LSB: type, MSB: id. Write causes REMOTE exception
6	REG_DISABLEDEXCEPTIONS	2	Y	unmapped	bitfield of disabled exceptions
7	REG_PING	1	Y	unmapped	debugging
8	REG_DEBUG	2	Y	unmapped	debugging

6.3.2 Drive/steer registers

These registers appear on drive/steer controllers, and control a drive and steer motor. There may also be a chassis potentiometer, although it is only present on one of the two D/S slaves in each wheel pair.

ID	Symbol	n	W	Mapping	Description
9	REGDS_DRIVE_REQSPEED	2	Y	[-4000.0,4000.0]	required speed
10	REGDS_DRIVE_PGAIN	2	Y	[0.0,10.0]	P-gain
11	REGDS_DRIVE_IGAIN	2	Y	[0.0,10.0]	I-gain
12	REGDS_DRIVE_DGAIN	2	Y	[-10.0,10.0]	D-gain
13	REGDS_DRIVE_INTEGRALCAP	2	Y	[0.0,1000.0]	integral error cap
14	REGDS_DRIVE_INTEGRALDECAY	2	Y	[0.0,1.0]	integral decay
15	REGDS_DRIVE_OVERCURRENTTHRESH	2	Y	[0.0,1000.0]	overcurrent threshold
16	REGDS_DRIVE_ACTUALSPEED	2		[-4000.0,4000.0]	actual speed from encoder
17	REGDS_DRIVE_ERROR	2		[-1000.0,1000.0]	required minus actual speed
18	REGDS_DRIVE_ERRORINTEGRAL	2		[-1000.0,1000.0]	error integral magnitude
19	REGDS_DRIVE_ERRORDERIV	2		[-200.0,200.0]	error derivative
20	REGDS_DRIVE_CONTROL	2		[-255.0,255.0]	value being sent to motor
21	REGDS_DRIVE_INTERVALCTRL	2		[0.0,1000.0]	time between control runs (ms)
22	REGDS_DRIVE_CURRENT	2		unmapped	raw current reading
23	REGDS_DRIVE_ODO	2		unmapped	encoder ticks
24	REGDS_DRIVE_STALLCHECK	1	Y	[0.0,255.0]	stall check control signal level
25	REGDS_DRIVE_DEADZONE	1	Y	[0.0,50.0]	if below this value, error is set to zero
26	REGDS_STEER_REQPOS	2	Y	[-200.0,200.0]	required position
27	REGDS_STEER_PGAIN	2	Y	[0.0,100.0]	P-gain
28	REGDS_STEER_IGAIN	2	Y	[0.0,10.0]	I-gain
29	REGDS_STEER_DGAIN	2	Y	[-10.0,10.0]	D-gain
30	REGDS_STEER_INTEGRALCAP	2	Y	[0.0,1000.0]	integral error cap
31	REGDS_STEER_INTEGRALDECAY	2	Y	[0.0,1.0]	integral decay
32	REGDS_STEER_OVERCURRENTTHRESH	2	Y	[0.0,1000.0]	overcurrent threshold
33	REGDS_STEER_ACTUALPOS	2		[-200.0,200.0]	actual position from pot
34	REGDS_STEER_ERROR	2		[-200.0,200.0]	required minus actual position
35	REGDS_STEER_ERRORINTEGRAL	2		[-1000.0,1000.0]	error integral magnitude
36	REGDS_STEER_ERRORDERIV	2		[-200.0,200.0]	error derivative
37	REGDS_STEER_CONTROL	2		[-255.0,255.0]	value being sent to motor
38	REGDS_STEER_INTERVALCTRL	2		[0.0,1000.0]	time between control runs (ms)
39	REGDS_STEER_CURRENT	2		unmapped	raw current reading
40	REGDS_STEER_STALLCHECK	1	Y	[0.0,255.0]	stall check control signal level
41	REGDS_STEER_DEADZONE	1	Y	[0.0,50.0]	if below this value, error is set to zero
42	REGDS_STEER_CALIBMIN	1	Y	[-120.0,120.0]	minimum angle, mapped onto pot value 0
43	REGDS_STEER_CALIBMAX	1	Y	[-120.0,120.0]	maximum angle, mapped onto pot value 1024
44	REGDS_CHASSIS	2		[0.0,1024.0]	chassis pot reading

6.3.3 Lift/lift registers

These registers appear on lift/lift controllers, and control two lift motors.

ID	Symbol	n	W	Mapping	Description
9	REGLL_ONE_REQPOS	2	Y	[-200.0,200.0]	required position
10	REGLL_ONE_PGAIN	2	Y	[0.0,100.0]	P-gain
11	REGLL_ONE_IGAIN	2	Y	[0.0,10.0]	I-gain
12	REGLL_ONE_DGAIN	2	Y	[-10.0,10.0]	D-gain
13	REGLL_ONE_INTEGRALCAP	2	Y	[0.0,1000.0]	integral error cap
14	REGLL_ONE_INTEGRALDECAY	2	Y	[0.0,1.0]	integral decay
15	REGLL_ONE_OVERCURRENTTHRESH	2	Y	[0.0,1000.0]	overcurrent threshold
16	REGLL_ONE_ACTUALPOS	2		[-200.0,200.0]	actual position from pot
17	REGLL_ONE_ERROR	2		[-200.0,200.0]	required minus actual position
18	REGLL_ONE_ERRORINTEGRAL	2		[-1000.0,1000.0]	error integral magnitude
19	REGLL_ONE_ERRORDERIV	2		[-200.0,200.0]	error derivative
20	REGLL_ONE_CONTROL	2		[-255.0,255.0]	value being sent to motor
21	REGLL_ONE_INTERVALCTRL	2		[0.0,1000.0]	time between control runs (ms)
22	REGLL_ONE_CURRENT	2		unmapped	raw current reading
23	REGLL_ONE_CALIBMIN	1	Y	[-120.0,120.0]	minimum angle, mapped onto pot value 0
24	REGLL_ONE_CALIBMAX	1	Y	[-120.0,120.0]	maximum angle, mapped onto pot value 1024
25	REGLL_ONE_STALLCHECK	1	Y	[0.0,255.0]	stall check control signal level
26	REGLL_ONE_DEADZONE	1	Y	[0.0,50.0]	if below this value, error is set to zero
27	REGLL_TWO_REQPOS	2	Y	[-200.0,200.0]	required position
28	REGLL_TWO_PGAIN	2	Y	[0.0,100.0]	P-gain
29	REGLL_TWO_IGAIN	2	Y	[0.0,10.0]	I-gain
30	REGLL_TWO_DGAIN	2	Y	[-10.0,10.0]	D-gain
31	REGLL_TWO_INTEGRALCAP	2	Y	[0.0,1000.0]	integral error cap
32	REGLL_TWO_INTEGRALDECAY	2	Y	[0.0,1.0]	integral decay
33	REGLL_TWO_OVERCURRENTTHRESH	2	Y	[0.0,1000.0]	overcurrent threshold
34	REGLL_TWO_ACTUALPOS	2		[-200.0,200.0]	actual position from pot
35	REGLL_TWO_ERROR	2		[-200.0,200.0]	required minus actual position
36	REGLL_TWO_ERRORINTEGRAL	2		[-1000.0,1000.0]	error integral magnitude
37	REGLL_TWO_ERRORDERIV	2		[-200.0,200.0]	error derivative
38	REGLL_TWO_CONTROL	2		[-255.0,255.0]	value being sent to motor
39	REGLL_TWO_INTERVALCTRL	2		[0.0,1000.0]	time between control runs (ms)
40	REGLL_TWO_CURRENT	2		unmapped	raw current reading
41	REGLL_TWO_CALIBMIN	1	Y	[-120.0,120.0]	minimum angle, mapped onto pot value 0
42	REGLL_TWO_CALIBMAX	1	Y	[-120.0,120.0]	maximum angle, mapped onto pot value 1024
43	REGLL_TWO_STALLCHECK	1	Y	[0.0,255.0]	stall check control signal level
44	REGLL_TWO_DEADZONE	1	Y	[0.0,50.0]	if below this value, error is set to zero

6.3.4 Master registers

These registers are local to the master controller — they are currently used for temperature and exception monitoring.

ID	Symbol	n	W	Mapping	Description
0	REGMASTER_RESET	1	Y	unmapped	set to clear exception state
1	REGMASTER_TEMPAMBIENT	2		[-20.0,100.0]	temperature sensor
2	REGMASTER_TEMP1	2		[-20.0,100.0]	temperature sensor
3	REGMASTER_TEMP2	2		[-20.0,100.0]	temperature sensor
4	REGMASTER_TEMP3	2		[-20.0,100.0]	temperature sensor
5	REGMASTER_TEMP4	2		[-20.0,100.0]	temperature sensor
6	REGMASTER_TEMP5	2		[-20.0,100.0]	temperature sensor
7	REGMASTER_TEMP6	2		[-20.0,100.0]	temperature sensor
8	REGMASTER_TEMP7	2		[-20.0,100.0]	temperature sensor
9	REGMASTER_TEMP8	2		[-20.0,100.0]	temperature sensor
10	REGMASTER_TEMP9	2		[-20.0,100.0]	temperature sensor
11	REGMASTER_EXCEPTIONDATA	2		unmapped	LSB: type, MSB: motor—slave

6.4 Main code (sketch.ino)

The main code defines and uses a *BinarySerialReader* abstract class, extended as *MySerialReader,* which processes packets from the PC in three methods:

- **dowrite** handles writing a set of registers for a given slave device on the I^2C bus;
- **doread** handles a request to read the registers in the current readset of a device;
- **doreadset** handles a request to change the current readset for a device.

6.5 I2CDevice (i2c.h)

This class encapsulates a link to an I^2C slave device, abstracted as a register interface. Devices may each have their own set of register definitions, which can include registers of different byte widths and access controls (i.e. read/write or read only.) Currently, two device types are supported, as shown in section 1.1: drive/steer and lift/lift. While very similar, the registers are slightly different.

The bus address of the device is specified by passing the address into the constructor. The registers are also specified in the constructor, by passing a table of *Register* structures. This table comes from the *regs.ino* file in the common directory.

- **writeRegister** takes the register number and the new value as an integer — size conversion is done automatically according to the size in the register table.
- **readRegister** reads data from a register, writing it to an integer (again, it's auto-resized.) A return value is provided giving status — generally a failure means that a bad register or device number has been passed in.

Note that mapping/unmapping are done at a higher level than this — these two methods deal with the integer values. Writing a register does the following:

- check register exists and is writable
- begin transmission (i.e. start writing on the bus and send the device number)
- write the register number and the value
- end transmission

Reading a register does the following:

- check register exists
- begin transmission (i.e. start writing on the bus and send the device number)
- write the register number
- end transmission
- request a read from the device
- wait for data to become available
- while data is available, read it and add it to the data (which arrives LSB first)

It follows, therefore that on both master and slave *reading is more expensive than writing.*

Note that the constructor for this object will not initialise the Wire library.

6.6 Remote control

Code for the remote is in the following files:

- `rc.ino` and `rc.h` contain the **RCReceiver** class, which handles reading servo pulses from the receiver and decoding them. This is done by using a pin change interrupt handler on the three relevant pins: we count the number of rising edges since the last update call. There is also a **ready()** method which returns true if data was recently received.
- `rcRover.ino` and `rcRover.h` extend the above class with methods specific to this application. In particular, the **update()** method updates the underlying receiver and sends register writes to the relevant slaves if there is servo data present.

The main loop in `sketch.ino` (in **BinarySerialReader::poll()** checks the remote for data, and does not perform the usual processing — reading register writes from the PC and sending them on to the relevant slave — if there is data. Instead, the remote code will start to send register writes.

Note that initiating remote control (by turning on the remote) will reset any calibration data to defaults and also reset any exception states!

7 Slave firmware

You probably don't need to read this section unless you're maintaining the code for the slave controllers.

The slave firmware processes incoming I^2C messages, reads encoders and potentiometers, and handles control of the motors. It's the most complex part of the code.

7.1 I2C slave code (slave_i2c.h)

This code implements the communications as an abstract class and a namespace.

7.1.1 I2CSlave namespace

This sets up I^2C and handles communications from the master using listener classes (actual communications are handled by the Wire library and interrupt callbacks.) The methods are:

- **init** initialises comms with a given device address, which must be unique for each device. It also takes a pointer to the register table, which is in the same format as that passed to the master firmware's *I2CSlave* class constructor, and is stored in the **regs.ino** file in the common directory.
- **setRegister** changes the value of a register, so that it can then be read by the master.
- **addListener** adds an event listener (see below) to the system, which will be informed on register writes.
- **poll** should be called periodically, and checks whether any registers have been changed since it was last called. If so, it will call the listener with the register index and new value.

7.1.2 I2CSlaveListener

This is an abstract class, implementations of which can be added to the list of listeners (see above.) When a register has changed, the *changeEvent()* method is called with the register index and new value — but only when *poll()* is called.

There is also a *changeEventImmediate()* callback, which will be called from inside the interrupt as soon as the new data arrives. Because it requires a little more processing, this mechanism is disabled by default.

Therefore, when a register is changed, the hardware will only take action when the listener's *changeEvent()* is called at the next poll. **This may lead to a very slight delay.**

7.1.3 I2CLoop() in i2c.ino

This is called repeatedly during the run, as fast as possible. It does the following:

- Calls *poll()* so that received messages will cause registers to change. These changes will be handled by the *MyI2CListener* object — for example, writing a required speed to the REG_SPEED register for a speed motor will cause *setRequiredSpeed()* to be called with that value on that motor.
- Sets monitoring registers with values from the motors.

7.1.4 Low-level I2C code

Callbacks are registered with the Wire library, which are called from the I2C interrupt inside the library. They work thus:

- **receiveEvent** is used when data is written to the slave. The current register is set to the first byte of the data, and if there are any more bytes, these are written to the register using the methods given above. This causes the register's changed bit to be set, so that *poll()* will call a listener when next called.

- **requestEvent** is used when a request is made for data. It assumes that the *receiveEvent()* has been called, and has set the current register number, and writes the value of the register to the bus. If the register doesn't exist a dummy value (`0xcd`) is written.

It follows therefore that on both master and slave *reading is more expensive than writing,* because two things happen: a message is received which sets the register number, and then data is requested.

7.2 Motors (motor.h)

A *MotorController* class is used as the basic core of motor functionality. It has:

- the enable, positive and negative pin assignments;
- methods to set both pins high or low for braking;
- methods to set the speed;
- methods for current monitoring including hysteresis using a geometric decay;
- a pointer to a chain of *MotorListeners* for handling overcurrent events, with a programmable overcurrent threshold.

7.3 PIDController (pid.h)

This class wraps the *MotorController* class with PID calculations, used by both speed and position motors. The PID results are used slightly differently by the two subclasses. It provides:

- methods for setting the desired value (speed or position);
- methods for setting the gains;
- monitoring methods (e.g. *getError()*;
- an *update()* method which must be called every tick;

7.4 Speed motors (speedmotor.h)

SpeedMotorController is an extension of *PIDController* which uses a PID controller for speed, using a quadrature encoder as the sensor. As such, it provides:

- a *tickEncoder()* method which must be called whenever the encoder's A channel gets a rising edge;
- a method to be called when ADC on the current monitors is completed, to update that value.

7.4.1 Encoders (quadencoder.h) and the encoder interrupt

The *QuadEncoder* class is normally used only by *SpeedMotorController*. It assumes that the encoder is wired to port D, pins 0 and 1 for channels A and B respectively (usually these are the RX and TX pins, or digital pins 0 and 1.) This means that each controller slave can only handle a single encoder motor.

An interrupt is required to use the encoder — on port D pin 0 rising edge, the *tickEncoder()* method must be called on the *SpeedMotorController* instance. This is set up thus in **sketch.ino**:

```
void setupEncoderISR(){
    cli();
    PCMSK2 |= 1<<PCINT16; // PD-0.
    PCICR |= 1<<PCIE2; // turn on ints for port D
    sei();
}
ISR(PCINT2_vect){
    if(PIND & 1) // rising edge on pin 0
        driveMotor.tickEncoder();
}
```

7.5 Position motors (posmotor.h)

PositionMotorController is an extension of *PIDController* which uses the PID controller for positional control, using a potentiometer.

7.5.1 Calibration

Each positional motor's potentiometer is connected to an ADC which produces a value in the range 0-1023. This value is mapped onto a user-specified range, given by the calibration registers which are initialised to the range -512 to 512.

7.6 ADC (adc.h/adc.ino)

The *ADCReader* class simplifies the process of running a series of free-running ADC reads. Only one read can be running at a time, so the ADC reader cycles through multiple reads which the developer can register with it. When a read completes, it calls a method in a *ADCListener* class giving the value, and starts the next read running. Methods are:

- **init()** : initialise (for which interrupts must be disabled)
- **poll()** which must be called regularly
- **addRead(channel,type,listener)** : register a new read, giving the ADC channel (i.e. the pin number), a type (which is an integer whose meaning is determined by the developer) and a pointer to the listener object. The listener is some implementation of the **ADCListener** interface, which will have a **onADC(type, value)** call.

ADC listeners are registered for all motors for current monitoring, the steer and lift motors for position monitoring, and the chassis system for inclinometer readings.

7.7 State

The `State` class handles exception states on both the slave and master, and is declared in `common.h` because it's used by both (with slight modifications.) Exceptions, when they occur, cause both a change in the state object and any exception listener registered with it to be notified.

Exceptions are discussed further in section 8.

7.8 Main code (sketch.ino)

The main code consists of:

- A listener which sets LEDs when an exception occurs;
- Two motor instances (speed and position for drive/steer controllers, or two position motors for lift/lift controllers;)
- An array of pointers to those motor instances, for convenience;
- various routines for handling PWM rate changing, and ADC work;
- Setup code;
- The main loop.

7.8.1 Setup

- Disable pullups for I2C and encoder
- Enable watchdog timer
- Set pin modes
- Read the I2C address from EEProm
- Flash some lights
- Change the PWM frequency for the motors
- Setup the two motors with their pin assignments
- Register the ADC listeners
- Initialise ADC
- Set up the interrupt for the encoder if this is a drive/steer slave
- Initialise I2C comms

7.8.2 Main loop

- Poll the I2C system, setting monitoring registers to the values and calling update on the motors.
- Poll the ADC system, updating data on completion of an ADC and moving on to reading another value.
- Handle debugging LEDs

8 Safety

Safety is achieved through two mechanisms:

- **Constraints** can be set on certain quantities which are checked at all possible levels;
- **Exceptions** are raised when certain conditions are met or constraints are violated. It may be possible to stop certain contraints from throwing exceptions.

8.1 Input constraints

The simplest constraints are the limits placed on input values, typically the required positions and speeds. These are already adequately handled by the register mapping system — if an attempt is made to set a register to a value which is outside its mapped range, a *SlaveException* is thrown by the PC code.

8.1.1 Lift constraints

Lift constraints are more complex, as they involve detecting combinations of required inputs which would cause a clash of legs. This is done at the PC API level, in the `isAdjacencyViolated()` method of *LiftMotor.*

This looks at adjacent pairs of legs when `setRequired()` is called, and if one leg is requested negative while the leg on its negative side is requested positive (or *vice versa*) a *ConstraintException* is thrown.

8.2 Rover exceptions

Another set of constraints operate at the slave level to detect stalls, encoder malfunctions and other problems. They work by examining the speed at which the motor is moving, the control signal being sent to the motor, and the current flowing. The errors produced, if any, in all combinations of these values are are shown in figure 11.

Currently this code is **disabled** because it leads to too many false positives. This has been done by setting the `STALLCHECK` value to 255 — the (unsigned 8 bit) "stall state counter," which is incremented each tick the system is in such a state, must exceed this value before a stall/fault is detected.

Speed	Control value	Current	Result
Stop	Low	Zero	
Stop	Low	Low	
Stop	Low	Nominal/High	SHORT
Stop	High	Zero	FAULT (drive)
Stop	High	Low	FAULT (encoder)
Stop	High	Nominal	STALL
Run	Low	Zero	?
Run	Low	Low	?
Run	Low	Nominal	?
Run	High	Zero	?
Run	High	Low	
Run	High	Nominal	
-	High	High	OVERCURRENT

Figure 11: Error table

- **Short** — there is a short in the drive wires, so a high current is being passed without the motor turning;
- **Fault** — there is either a break in the drive wires; (in which case the current will be zero) or a fault in the encoder or encoder wires (in which case the current will be non-zero);
- **Stall** — the motor is unable to move;
- **Overcurrent** — too high a current through the motor;
- **?** — probably inertial effects, ignore.

When such a condition occurs, the slave goes into the exception state and informs the master. The master then transmits a message to all the slaves, so they also go into an exception state. Exception data includes:

- The motor ID on the controller (0 or 1)
- The type of the exception

After an exception, therefore, the slaves will all be in an exception state. Most will be showing the REMOTE exception, but one will show the original problem and the motor for which it occurred.

Exception states and transitions are shown in Figure 12. Dotted lines indicate causation, so a dotted line from a transitions to another transition indicates that the first transition causes the second.

8.2.1 Exception transitions

Firstly, both the slaves and master **start in the exception state**. This is because, in the case of a communications error, the I^2C routines will stall, causing the processor to reboot when the watchdog times out.

It is the responsibility of the PC library to reset the all slaves, and then reset the master, when the system starts up. If an exception occurs later, the master and slaves will once again be in the exception state and the PC code will again have to reset them.

If a "normal" exception (i.e. not a communications error) occurs on a slave, the slave goes into the exception state and reports the error to the master, which also goes into the exception state and tells all other slaves to do the same. The exception registers on the master will hold information about the original slave's exception.

In addition, the failure of a slave to respond to a message will cause it to reboot, which will cause the master to reboot (this is the only failure mode possible for I^2C) which will send RESET messages to all slaves. The result is all devices in exception.

8.2.2 Exception actions

On entry to the exception state, the master signals all the slaves to also go into exception. The slaves, in an exception state, do the following:

- Set both negative and positive pins of the motors to LOW
- Set the duty cycle on the enable pins to zero

This causes the motors to perform an unbraked stop.

In the exception state, register values can be changed but will have no effect until the exception is reset, because the control loop does not run in exception[3]. It's important to note that the **boot exception will have set all values to their defaults.** This is not the case for other exceptions.

[3]In an earlier version, register changes were ignored. This led to the problem of being unable to recover from an error resulting from a bad register setting (such as overcurrent threshold to zero.)

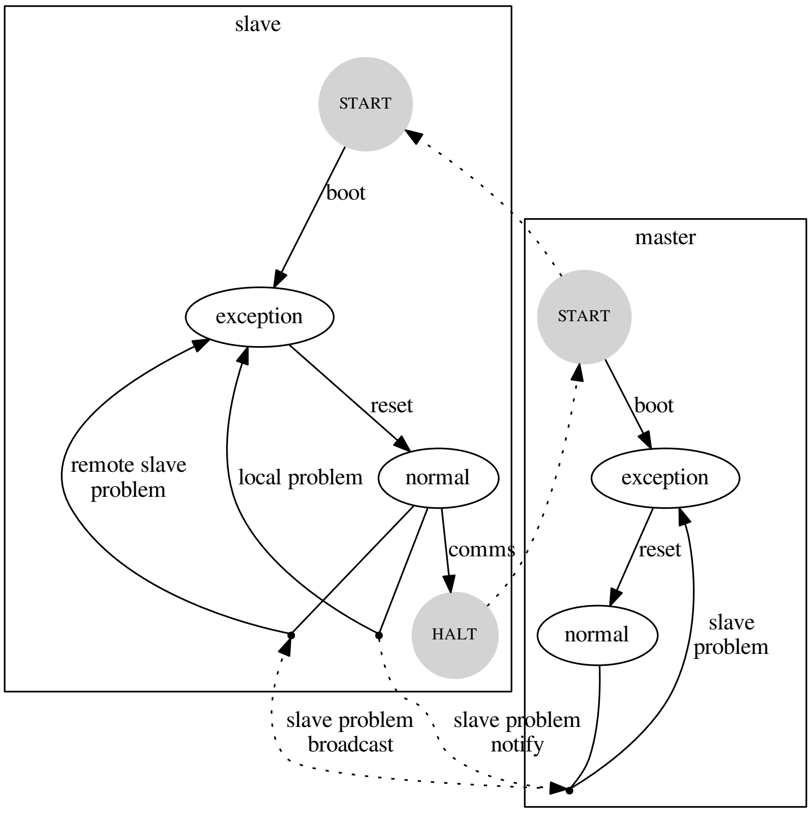


Figure 12: Exception states

8.2.3 State flags

The exception substates for the master are:

- Boot state (could be caused by I^2C comms failure)
- Problem reported by slave (ID and problem specified)

The exception substates for the slaves are:

- Boot state (powerup, or signalled by master using RESET command)
- Local problem as shown in figure 11
- Problem with another slave, reported by master
- Halt state (used when a comms error occurs, forces the master to reboot)