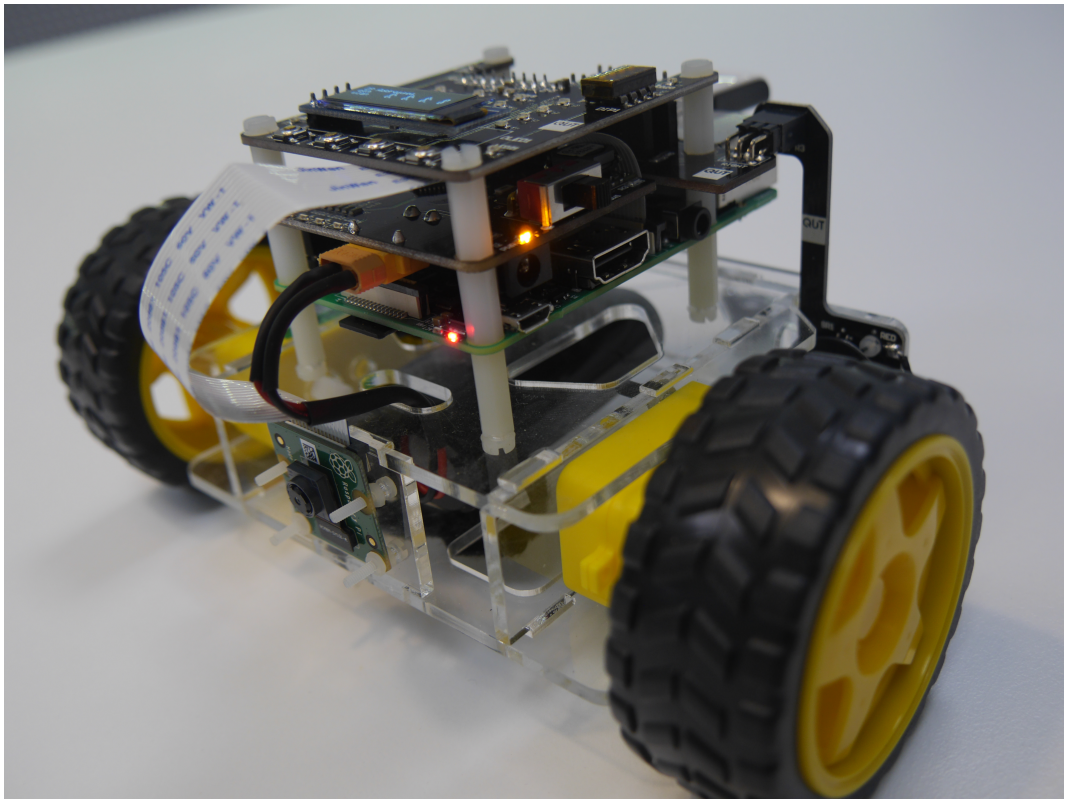
A mobile robot for education: the PenguinPi

Peter Corke

July 2018

This report describes a low-cost mobile robot designed for robotics education. The teaching objectives are described and this leads to a specification for the robot. The remainder of the report provides a detailed technical description of the Penguin Pi robot and its implementation.



Contents

1 Educational objectives

The mobile robotics learning outcomes we wish to achieve include:

1. Motion models: how configuration (pose) of the robot evolves with time for key types of platform such as Ackerman steering and differential steering.
2. Motion control: how to design a controller that will move the robot from an initial to a desired final configuration, or to follow a path. Initially we will assume that robot configuration is measured (externally) and provided as input to the robot controller.
3. Motion planning: how to find a path to move from A to B
   (a) For a grid-based map that denotes occupied and free space, how to plan an obstacle-free path using the distance transform (aka wavefront or grass fire) algorithm
   (b) For a graph-based map using breadth-first, uniform-cost, A* search, or D* algorithms to find an optimal path.
4. Robot localization: how to use observations of known landmarks to estimate robot configuration ("where am I?").
5. Map making: how to convert landmark observations into a consistent estimated map assuming that robot configuration is known.
6. Simultaneous localization and mapping: how to create a map and estimate robot configuration based on observations of initially unknown landmarks.

2 Robot specification

To explore these learning principles, we require a robot with the ability to move, to report its motion (odometry), to observe landmarks, and make use of an external sensor that reports the robot's configuration.
The robot requires the following features to support these learning outcomes:

1. Differential-drive platform which is very general. While quite simple and low-cost to fabricate, a software emulation can convert this platform to a virtual Ackerman-steered configuration (the reverse is not true).
2. A means to provide odometry which is required to estimate current configuration. Options include:
   (a) Wheel encoders to provide incremental wheel rotation. Ideally these would be quadrature but if motion is in only one direction it could be a simpler and cheaper non-quadrature encoder.
   (b) An optical mouse chip to measure x- and y-axis displacement of a point on the robot relative to the ground.

3. Camera to observe landmarks for localization or visual odometry. There are many options and the choice could be part of the student's exercise. Options include:
   (a) Front-mounted forward-looking perspective camera with a limited field of view.
   (b) A 360° panoramic camera, and there are some low-cost options such as the ArduCAM (based on 4 regular cameras.
   (c) An upward-facing catadioptric camera using a low-cost iPhone camera adaptor.
4. Battery and a means to charge it, ideally while the battery is connected to the robot and the robot is powered up. Also the ability to monitor battery voltage.
5. WiFi communications for tether-less operation. Ideally able to work in the university lab, a student's home or even act as a wireless access point to allow operations in areas with no WiFi infrastructure.
6. An external localization service to provide an estimate of the robot's pose as an input to a motion controller. This capability will be used initially until students have mastered the techniques to perform localization themselves. After that, the external localization can be used to provide a ground truth for comparison purposes. Some options for an external localizer include:
   (a) Dead reckoning based on wheel encoders, implemented onboard the robot at a relatively high sample rate.
   (b) An overhead camera, looking down at distinctive visual markers/beacons on the robot, possibly infra-red (IR) LEDs.
   (c) A camera on the robot looking up at specialized markers on the ceiling, for example the WhyCon system.
   (d) Two IR receivers working with one (maybe two) HTC/Valve Vive base stations which scan planes of IR light across the workspace.
7. Convenient programming environment for code running either onboard or offboard the robot. In either case it would be useful to have graphical display of the robot's state and its perceived environment. For code running offboard the robot, communications latency is likely to be an issue. Possible programming paradigms include:
   (a) Student MATLAB running offboard and communicating via a simple API with provided software running on the robot
   (b) Student Python code running onboard using a Python API
   (c) Student MATLAB code running in MATLAB Online (in the cloud so no need for a local MATLAB instance) communicating via a simple API with either the low-level robot hardware or provided software running on the robot
   (d) Student MATLAB code, using a simple API, compiled to C code and running onboard the robot.

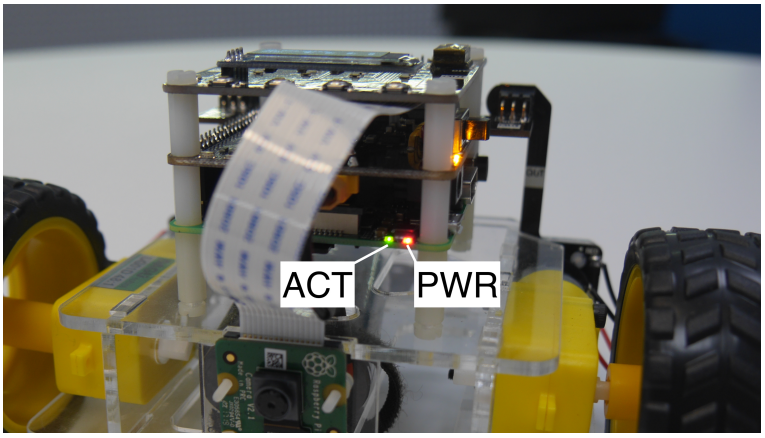


Figure 1: Raspberry Pi LED indicators

3 The PenguinPi robot

The robot is shown in Figure 2. It was developed as a final year student project by Jack Wright in 2016. Jack was also a student of the EGB439 Advanced Robotics unit, which at that time used Lego Mindstorm for the practical work. The robot was refined by Jack and Steven Bulmer over the summer of 2016/7 and the Mark I was used for the class in 2017 and for a workshop at the Robotic Vision Summer School in early 2018. A Mark II version was developed over the summer of 2017/8 and used for class in first semester of 2018. In the second half of 2018 the onboard software was substantially refactored and extended.

3.1 Computing systems

The main processor is a Raspberry Pi 3B with an attached camera (PiCam). It includes a microSD card slot, four USB ports, built in BlueTooth and WiFi and an HDMI port and a 40-pin expansion connector. The operating system is Rasberian and includes python and OpenCV. ROS is not pre-installed. By connecting an HDMI monitor plus USB keyboard and mouse the Raspberry Pi provides a capable standalone Linux system with windowed desktop environment. For mobile operation we use WiFi and connect to it using `ssh` or web services. We prefer a 5 GHz USB dongle over the onboard 2.4 GHz WiFi capability to avoid congestion in lab sessions with many robots.
The Raspberry Pi 3B has two LEDs to the right of the microSD card slot shown in Fig 1:

- ACT (green) indicates that the SD card is being read or written. A repetitive pattern of flashes at startup indicates issues with the SD card or boot process;
- PWR (red) indicates that the board is powered (it will flash for voltage below 4.63 V).

Alternative computer platforms would include:

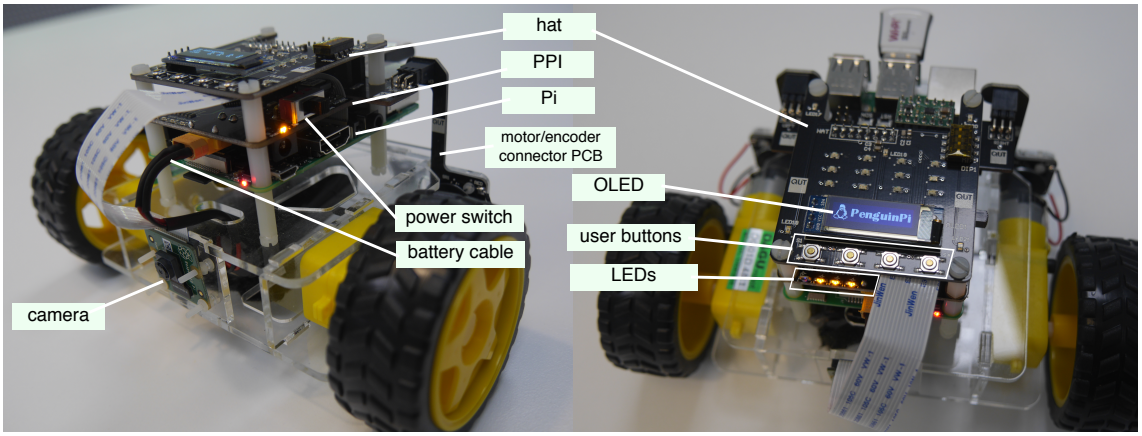


Figure 2: Annotated view of the Penguin Pi robot

- BeagleBone Blue, but by comparison the Raspberry Pi 3 has several key advantages: half the cost, has a low-cost camera and interface, and is easy to expand for i/o without needing specialised cables.
- Libre computer Le Potato or Renegade

The PenguinPi daughter board (PPI) is a Raspberry Pi expansion board that attaches to the Pi's 40-pin expansion connector. It contains an Atmel 644 8-bit processor with 64k program space and 4k of RAM. It performs motor control, power management and supports a simple user interface (UI), has a number of indicator LEDs. It can be expanded by an additional "hat" board. One such board – HAT1 – provides infra-red beacon LEDs and a simple user interface comprising push buttons and a small OLED display.
The RaspberryPi and the PPI processors communicate over a high-speed serial port for message exchange, error diagnostics (from the PPI) and display of text strings on the OLED display.

3.2 Robot hardware

3.2.1 Motors

The robot is driven by a pair of ROB-13302 brushed permanent-magnet DC gearmotors with a right-angle gearbox and a reduction ratio of 48:1. Specifications are:

- voltage 4.5 V
- no load maximum speed of 140 RPM
- no load current 190 mA, this is effectively the Coulomb friction torque
- maximum load current 250 mA

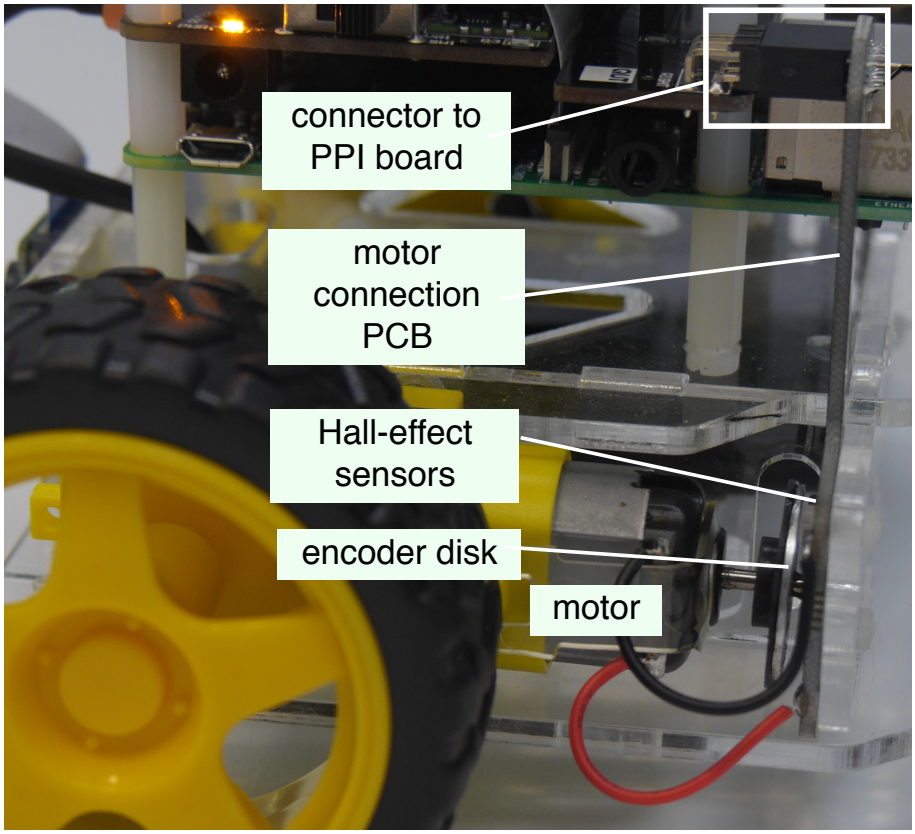


Figure 3: Closeup view the Penguin Pi robot showing the motor, the encoder disk and the PCB which holds the Hall-effect sensors, and which plugs into the PPI board.

- maximum torque 800 $\mathrm{gf} \cdot \mathrm{cm}$ or $0.8 \times 9.81 \times 0.01 = 78.4\,\mathrm{mN}$. If this is the nett torque *after* friction, then the torque constant (including gearbox) is $78.4/(250-190) = 1.31\,\mathrm{N} \cdot \mathrm{m/A}$. The back EMF constant would therefore be $1.31\,\mathrm{V} \cdot \mathrm{s/rad}$.

The wheels are ROB-13259 with a diameter of $D = 65\,\mathrm{mm}$.
The motor shaft, before the gearbox, is fitted with an ROB-12629 encoder disk which has 8 poles (4 north poles and 4 south poles) which can be seen in Figure 3.

3.2.2 Motor speed response

The response of the motor to applied voltage (percentage of maximum PWM) is shown in Figure 4. It fits well to a line through the origin with a slope of 11.5 but at command below 30 the effect of friction becomes evident, with no motion for a command of 9 or less. The maximum encoder rate is $1100\,\mathrm{enc/s}$ or $2.87\,\mathrm{rev/s}$ which is approximately 25% higher than the motor's rated speed of 140 RPM or $2.33\,\mathrm{rev/s}$ – perhaps indicating that the maximum applied voltage is higher than the motor's specified maximum.

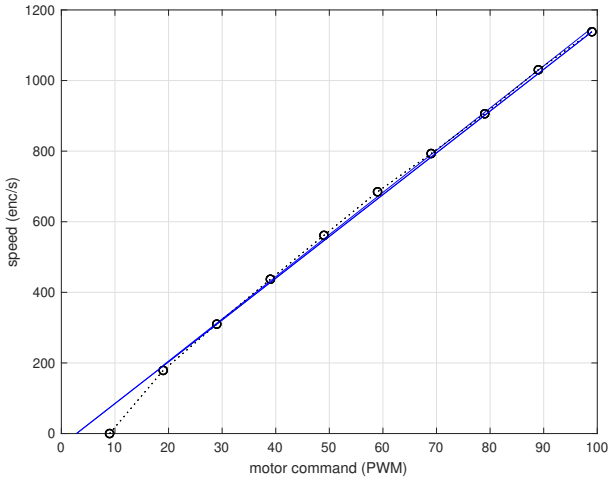


Figure 4: Open-loop motor response to applied PWM voltage (percentage) for motor A.

Ignoring motor inductance we can write motor current as

$$i = \frac{V - V_b}{R} \tag{1}$$

where R is motor armature resistance and

$$V_b = K_b \varpi \tag{2}$$

is the back EMF where K_b is the back EMF constant. Nett motor torque is

$$\tau = \begin{cases} 0 & |K_\tau i| \leq \tau_C \\ K_\tau i - B\varpi - \text{sgn}(\varpi)\tau_C & |K_\tau i| > \tau_C \end{cases} \tag{3}$$

where B is viscous friction, τ_C is Coulomb friction (assumed symmetric) and K_τ is the motor torque constant.
While moving at steady state, nett torque $\tau = 0$, we can substitute (1) and (2) into (3) to obtain

$$K_\tau \frac{V - K_b \varpi}{R} - B\varpi - \tau_C = 0$$

and simplifying with $K = K_b = K_\tau$ then

$$\varpi = \frac{KV - \tau_C R}{K^2 + BR} = V \frac{K}{K^2 + BR} - \frac{\tau_C R}{K^2 + BR} \tag{4}$$

which is similar in appearance to Figure 4. From the slope (11.95) and breakpoint (-33.74) , and assuming knowledge of K and τ_C from above, we could estimate R and B.
The resulting motor speed ignores disturbance torque from wheel-ground interaction which will be highly dependent on the ground material and be significant during skid steering. To overcome this a motor velocity controller can be enabled by setting switch 4 of the DIP switch to ON. It may be possible to estimate friction characteristics from the measured battery current.

3.3 Penguin Pi expansion board (PPI)

The Penguin Pi (PPI) is a small circuit board that connects to the Raspberry Pi GPIO connector. It manages a LiPo battery, drives the motors, and interfaces with motors, encoders and a number of LEDs.

3.3.1 Encoders

The sensor is an Allegro A1120 unipolar Hall-effect sensor which turns on when it sense a magnetic south pole. The motor shaft is fitted with an ROB-12629 encoder disk which has 8 poles (4 north poles and 4 south poles), so the sensor output will have four on-periods and four off-periods per shaft revolution. The processor is configured to interrupt on changes to the encoder signal (rising and falling edges) of which there are eight per revolution. Given the 48:1 gearbox ratio there will be $48 \times 8 = 384$ encoder interrupts per wheel revolution.
The encoder, and the encoder sensing board can be seen in Figure 3. This board, which connects to the main PPI board eliminates a lot of point-to-point wiring which led to high build time and poor reliability with the Mark 1 robot.

3.3.2 Motor driver

The motor voltages are controlled via 8-bit PWM motor drivers with values in the range 0 to 255. This is performed by Timer 0 and the output compare registers OCR0A and OCR0B. Motor direction, or applied polarity, is controlled by digital outputs PB0 and PB1 respectively.
The software considers motor commands in the range -100 to +100 are these are mapped to PWM settings in the range 0-255 plus motor direction.

3.3.3 Atmel clocks and timing

The processor has a 20 MHz external crystal connected between pins XTAL1 and XTAL2. The $\text{clk}_{\text{I/O}}$ signal (20 MHz) from the clock control logic is used for all timers. The $\text{clk}_{\text{I/O}}$ signal is controlled by the:

- Clock source multiplexer, which is set to external crystal oscillator (CKSEL bits in the `lfuse` fuse byte)
- System clock prescaler is bypassed, ie. no prescaler (CKKPS bits in the `lfuse` fuse byte)

This clock drives the following counters:

Timer	Width (bits)	Purpose
0	8	motor PWM
1	16	1 ms clock interrupt
2	8	additional PWM

All counters have the timer prescaler bypassed and clock at 20 MHz.

Pin	binary in	binary out	analog	PWM	other	RPi
PA0	ENCA1*					
PA1	ENCA2*					
PA2	ENCB1*					
PA3	ENCB2*					
PA4	HAT06		ADC4			
PA5	HAT01		ADC5			
PA6	-		VSENSE			
PA7	-		CSENSE			
PB0	-	DIRA				
PB1	-	DIRB				
PB2	HAT02					
PB3/OC0A				PWMA		
PB4/OC0B				PWMB		
PB5		shutdown request			MOSI†	GPIO10
PB6	-				MISO†	GPIO9
PB7	-	shutdown ack			SCLK†	GPIO11
PC0	-				SCL^{I2C}	$GPIO3^{I2C}$
PC1	-				SDA^{I2C}	$GPIO2^{I2C}$
PC2		LEDY0				
PC3		LEDY1				
PC4	pullup					
PC5		LEDY2				
PC6	HAT00					
PC7	HAT07*					
PD0	-				UART0 RX	UART TX
PD1	-				UART0 TX	UART RX
PD2	HAT05					
PD3	HAT03					
PD4	HAT04					
PD5/ICP3				LEDG		
PD6/OC3A				LEDB		
PD7/OC3B				LEDR		

Table 1: Allocation of Atmega i/o ports. * indicates pin-change interrupt is enabled. Ports A-D generate interrupts on PCINT0 to PCINT3 respectively. †indicates used for reprogramming the Atmega. ^{I2C} indicates used for I2C bus

3.3.4 Power

The PPI connects to a LiPo battery and has a power input socket and a slider "POWER" switch. When the switch is:

OFF the battery is connected to the charging socket, the battery charges but no power flows to the PPI.

ON the battery is connected to the PPI, the PPI is powered but the battery cannot charge. Power flows to the Raspberry Pi via the expansion connector, pins 2 and 4.

If the Raspberry Pi is powered via its micro-USB connector then the PPI will be energized by power flowing in from the expansion connector (reverse current flow to normal operation). This is not a recommended way of operation but if attempted, the PPI power switch *must be OFF* and the current available is limited so the motors can provide only limited torque.

3.3.5 Analog channels

Voltages are measured by a 10-bit analog to digital converter (ADC) with an 8-channel multiplexer and a reference voltage of $V_{\text{CC}} = 3.3\,\text{V}$. The PPI uses the ADC to measure:

- Battery voltage (ADC 6). A voltage divider across the battery rail has a scale factor of $0.20\,\text{V/V}$ giving an ADC scale factor of $3.3/1024/0.2 = 16.11\,\text{mV/count}$.
- Battery current (ADC 7). A $0.05\,\Omega$ shunt resistor and a TSC888B current sense amplifier (gain of 20) gives an ADC scale factor of $3.3/1024/0.05/20 = 3.223\,\text{mA/count}$.

The ADC clock uses a pre-scaler divisor of 64 giving a clock of 312 kHz. The first conversion takes 25 cycles (80 μs) and subsequent conversions take 13 cycles (42 μs).

3.3.6 I2C interface

The I2C (or TWI) interface is a high-speed two-wire interface for the Atmega processor. The I2C bus is clocked at 400 kHz. Each frame has a start condition, 9-bit address, and then N data bytes, each taking 9 bit times (including the acknowledge bit). Communications is performed using polled i/o so an N-byte packet requires 9N+20 bit times to transmit or receive, and one bit time is 2.5 μs.

3.3.7 LEDs

There are four physical LEDs on the PPI base board: one RGB LED (LED1), and three yellow LEDs (LEDS 2 to 4). They are located to the left of the battery connector and numbered increasing from left to right. All LEDs are connected between Vcc and the Atmel i/o pin and are lit when the pin is at a low voltage and sourcing current. The RGB LED is considered by the software as 3 separate LEDs (red, green and blue).

Pin	Purpose	GPIO	40-pin connector
1	Ground		
2	5 V		
3	HAT0	GPIO26	37
4	3.3 V		
5	HAT1	GPIO20	38
6	HAT7	GPIO19	35
7	HAT2	GPIO16	36
8	HAT6	GPIO13	33
9	HAT3	GPIO7	26
10	HAT5	GPIO5	29
11	SCLK†(SPI clock)	GPIO2†	3
12	HAT4	GPIO8	24
13	PB5/MOSI (SPI data in)	GPIO10†	19
14	PB6/MISO (SPI data out)	GPIO9†	21
15	PB7/SCL^{I2C} (I2C clock)	GPIO3	5
16	SDA^{I2C}(I2C data)	GPIO3^{I2C}	5

Table 2: Signals on the PPI expansion connector. †indicates used for Atmega reprogramming, ^{I2C} indicates used for I2C bus, GPIO refers to the pin on the BroadCom processor chip, also known as BCM. The fourth column is the pin number on the Raspberry Pi standard 40-pin expansion connector. Raspberry Pi GPIO pins can be numbered using either convention.

3.3.8 Atmel I/O ports and interrupts

The Atmel has four 8-bit digital i/o ports: PA, PB, PC and PD. Bits can be configured individually as inputs, outputs or open circuit. Most Atmel pins are multi-purpose and the digital i/o capability might be unavailable if an alternative function is used on that pin. The usage of the i/o signals is described in Table 1.
Each of the 32 digital i/o pins can generate a pin-change interrupt (PCINT0 to PCINT31) configured on either a rising or falling edge. There are only 4 interrupt vectors (PCINT0-7, 8-15,16-23, 24-31) and the associated interrupt handler must determine which of the 8 pins changed.

3.3.9 Expansion connector

The PenguinPi can be expanded with a hat board via a 16-pin connector, Table 2, on top of the PPI board. Some signals connect to the Atmel processor on the PPI board while others connect to the Raspberry Pi via the 40-pin expansion connector. Different hat boards could be used to support different teaching projects.
HAT1 supports:

- An OLED screen is an array of 128×32 pixels. With a 6×8 font this becomes a 21×4 array of characters.
- An array of 4 pushbuttons under the OLED that provide a simple user interface.

Device	Address	Data	
PCA6416A/0	0x40	LEDS 8-15	LEDS 0-7
PCA6416A/1	0x42	buttons	DIP switch
OLED/SSD1306	0x78	command/data	

Table 3: I2C peripherals on PPI HAT1

- Four IR LEDs at its corners which are always on. This requires viewing with an IR sensitive camera, possibly with a a visible light blocking filter (optical low-pass filter).
- A DIP switch provides 4 bits of configuration information. SW4 is used to enable PID motor control, the other bits are available to the user.
- A grid of 16 LEDs on the top board can be addressed as a single 16-bit word. The bits are numbered bottom to top, then left to right. The bottom right LED is bit 0 and top left LED is bit 15.

Communications with the hat is via I2C messaging and a digital i/o line. There are two PCA6416A, 16-bit general purpose I/O expanders, that provides remote I/O expansion via the I2C-bus interface. One provides 16 outputs for the blue LED array, the other has inputs from the pushbuttons and DIP switch. The HAT07 line is used to interrupt the PPI when data is available from either PCA6416A, and the PPI will then use the I2C bus to read that data.

3.3.10 Raspberry Pi expansion connector

The PPI board is a Raspberry Pi expansion board and the boards are physically and electrically connected by the 40-pin connector A subset of signals on connector are routed to pins on the Atmel and also the PPI expansion coonnector. In summary:

- it transfers DC power between the PPI and the Raspberry Pi.
- some Raspberry Pi digital i/o signals (GPIO) are connected to digital i/o signals on the Atmel, see column 7 of Table 1.
- some Raspberry Pi i/o signals (serial, SPI) are connected to pins of the Atmel, see column 7 of Table 1.
- some Raspberry Pi i/o signals are passed through to the PPI expansion connector (GPIO, SPI), see Table 2.
- the Atmel can be reflashed by the Raspberry Pi using using the SPI bus while asserting reset.

Expansion connector pins are referred to either by pin number or by GPIO number. The latter refers to the pin on the BroadCom processor chip, also known as BCM. Raspberry Pi GPIO pins can be numbered using either convention. Check online for the pin assignments for a particular type of Raspberry Pi at `https://pinout.xyz/pinout`.

3.3.11 Buttons

The Atmel can be reset by pushing the RST button or by the Raspberry Pi expansion connector pin 18 (GPIO24). The Raspberry Pi asserts reset while the Atmel is being reflashed via the SPI bus.
The SDN button drops Raspberry Pi expansion connector pin 16 (GPIO22), see Table 2, to signal that a shutdown is requested.
In `/boot/config.txt` it is possible to add device tree overlays

```
dtoverlay=gpio-shutdown,gpio_pin=5
dtoverlay=gpio-poweroff,gpiopin=21,active_low="y"
```

The first enables shutdown when GPIO 5 is set low – no other software is needed. This works as advertised but does not provide any feedback to the PPI that a shutdown is underway. The second line is supposed to assert GPIO 21 when the Raspberry Pi is booting but this in practice this does not happen.
Currently these options are not active.

4 PPI robot software

4.1 Code structure

The code is organized as one large polling loop with several interrupt handlers.

4.1.1 Interrupt handlers

The interrupt sources, and their actions, are:

- Timer 1 (16-bit clocked at 20 MHz) interrupt every 1 ms:
  - Every 20 ticks performs the velocity control (50 Hz)
  - Every 10 ticks (offset from above) triggers ADC conversion (100 Hz)
  - Every 50 ticks (offset from above) request the main loop to update the OLED display (20 Hz)
  - Decrement the LED counters
  - Updates the 32-bit system timer counters: `milliseconds_counter`, `seconds_counter`
- Encoder signal change (rising and falling edge) updates the motor-state structures `motorPosition`
- ADC conversion complete, changes the multiplexer settings and stashes data for action by the main loop
- HAT7 indicates data available from the hat board (push buttons or change to DIP switch), raise a flag for action in the main loop to read the new I2C data
- Serial port transmit and receive (defined in `lib/uart.c`) which implements a 64-byte receive data ring buffer.

The interval for velocity control, ADC conversion and OLED update are controlled by constants at the top of `main.c`.

4.1.2 Main polling loop

The tasks in the main loop include:

- Check for a received start byte, and if found read and parse the rest of the message. The datagram read is effectively blocking for up to the receiving time for 10 bytes, which is 0.87 ms, plus whatever time is required for processing the message.
- When requested by the timer interrupt handler, update the OLED display. This is a time consuming operation, the I2C communications will take at least 15 ms and is performed in 33 chunks spread across successive loops to minimize latency.
- Update the LED state machines, see Section 4.6.
- Digitally filter battery current and voltage if flags set by ADC interrupt handler indicate that new ADC data is available.
- Test for low battery voltage and take appropriate action.
- If signalled by the HAT7 interrupt handler, read data from the hat via the I2C port including the DIP switch and the button status.
- Measure loop time and update statistics

Time statistics are updated and available via the OLED "loop time" screen. The mean time is only 64 μs with a maximum of just under 2 ms.

4.1.3 Encoders

The PPI hardware supports one or two Hall-effect sensors for each motor encoder. They are connected to the PA digital input pins and generate "pin change" interrupts.
The encoder count is maintained as a signed 16-bit value. The counters increase when the robot moves forward. The counters will wrap (ignoring overflow) around above 32767 or below -32768. This is 85 wheel revolutions or 17 m of travel forwards or backwards.
The advantage of considering the encoders as signed values is that taking the difference on either side of the wrap will result in the correct value, ie. using signed 16-bit arithmetic -32768-(32767) = 1 (positive rotation) and 32767-(-32768) = -1 (negative rotation).
The encoder interrupt handler supports several encoder modes:

0. Single sensor. The Hall-effector sensor outputs one pulse per south-magnetic pole. Direction of rotation cannot be determined but can be inferred from the polarity of the applied motor voltage and the counter is incremented or decremented accordingly. If the motor PWM is zero (idle) the direction is indeterminate and the counters are not updated. Note that for very rapid changes in motor voltage polarity, the mechanical inertia of the system means that the motor direction may be opposite to the motor polarity for a short period which will cause odometry errors.

1. Dual quadrature sensors. At every edge (rising or falling) on the encoder A signal we determine direction of rotation from the value of the quadrature encoder B signal, and increment or decrement the counter accordingly.

2. XOR mode. At every edge (rising or falling) on the encoder A or B signal we determine direction of rotation from the value of the other signal, and increment or decrement the counter accordingly. This leads to a doubling of angular resolution and improved velocity estimation.

4.2 Motor velocity control

The velocity control is invoked every $20\,\mathrm{ms}$ when DIP switch 4 is ON. It runs in the context of the timer interrupt handler, but with interrupts enabled. While it might be questionable practice to run a control algorithm within an interrupt handler it totally eliminates jitter compared to the alternative of settting a flag and having the main loop deal with it.
From Figure 4 we see the maximum encoder rate is $1100\,\mathrm{enc/s}$ which is 22 encoder counts within the $20\,\mathrm{ms}$ sample interval. The speed command V^* is in the range -100 to +100 so to interpret this as a speed command we first scale it by $1/5$ to obtain the desired number of encoder counts per control interval (alternately we could scale up the estimated velocity by 5). The downside of this simple strategy is a very quantized speed signal, with only 20 effective speed settings. This could be ameliorated by: using a longer servo interval, using an omnidirectional Hall-effect sensor ($\times 2$ improvement) or quadrature encoders ($\times 2$ improvement) or encoder mode 2 (see Section 4.1.3).
Velocity estimation is performed by differencing the current and previous encoder values (signed 16-bit). The controller is a PI controller with gains K_{vp} and K_{vi} and these parameters can be read or written using the commands MOTOR_SET_KVP and MOTOR_SET_KVI. The desired velocity is set with the command MOTOR_SET_VELOCITY and is in the interval [-100, 100].
All arithmetic is performed using 16-bit signed integers. The state and control parameters for each motor are kept in a `Motor` structure, one per motor.
The steady-state closed-loop speed is

$$\frac{V^*/5}{T}\,\mathrm{enc/s}$$

where V^* is the demanded speed (-100 to +100) and T is the sample interval. The closed loop gain is therefore $V^*/5/20{\cdot}10^{-3} = 10V^*\,\mathrm{enc/s}$.
In terms of rotational velocity the scaling is

$$10V^*\frac{1}{384} = 0.026V^*\,\mathrm{rev/s}$$

and the translational velocity scaling is

$$0.026V^*\pi D = 5.33V^*\,\mathrm{mm/s}$$

4.3 Analog channels

The first ADC reading (voltage) is triggered at 100 Hz by the clock interrupt handler and the ADC interrupt handler retrieves the result and initiates the second conversion (current).
Once the second conversion is complete the samples are smoothed by a simple first-order unity-gain digital filter executed in the main loop

$$y_{k+1} = by_k + (1 - b)u_k$$

which has a pole at $z = b$ which is related to the continuous time pole with frequency f by $z = e^{-2\pi f/T}$. For example, a pole at 5 Hz requires a discrete-time pole at $z = 0.9875$. The scale factor and digital filtering are implemented in floating point within the main polling loop. The raw value is obtained with the command ADC_GET_VALUE, the smoothed value with ADC_GET_SMOOTH, and the digital pole is set with ADC_SET_POLE.

4.4 Battery monitoring and low-voltage actions

If battery voltage falls below 7 V the screen colors are inverted: black text on white background. Once the voltage falls below 6.5 V for a specified number of cycles then a Raspberry Pi shutdown is initiated via GPIO22. The main loop is blocked by an infinite loop which disables all buttons and screen updates.

4.5 Measuring time

Functions exist to measure time to an accuracy of 0.05 μs. The `timer_t` object represents time and contains a snapshot of the seconds, milliseconds and 20 MHz counter. The function `timer_diff_us` computes the difference between two of these objects in units of microseconds.

```
#include "timer.h"

timer_t t0, t1;

timer_get(&t0);
 .
 .
timer_get(&t1);

uint32_t dt = timer_diff_us(&t1, &t0);
```

This module also supports simple statistics for `uint32_t` quantities

```
stats_t exec_time;

stats_init(&exec_time, dt);
stats_add(&exec_time, dt);
 .
printf("average time %lu\n", stats_mean(&exec_time) );
printf("standard deviation %lu\n", stats_std(&exec_time) );
printf("max time %lu\n", stats_max(&exec_time) );
```

4.6 LEDs

There are four physical LEDS on the PPI base board: one RGB LED (LED1), and three yellow LEDs (LEDS 2 to 4) which are considered as 6 separate LEDs designated: R, G, B, 2, 3, 4. The software interface allows any LED to be:

- turned on using the command LED_SET_STATE,1
- turned off using LED_SET_STATE,0
- pulsed for a period of time using LED_SET_COUNT,T. This is an unsigned 8-bit number of milliseconds, so the maximum on period is $0.255\,\mathrm{s}$

Each LED has some associated software state that is used by the timer interrupt handler and the main loop. It contains:

state either 0 (off) or 1(on). Set by a command or the PPI code itself. In every cycle of the main loop the LED is turned on or off according to this state.

count if non-zero it is decremented every millisecond and when it reaches zero the state is set to off.

If count is set to a non-zero value the state is set to on, and it will then count down and when it reaches zero the LED will be turned off.
The LEDs can be controlled via the API but they have hardcoded usage in the PPI code (enabled by the `DEBUG_LED` macro):

- Blue, pulse on every start byte, indicating that commands are being received from the Raspberry Pi.
- Green, pulse on every encoder interrupt. Note that in encoder mode 0 encoder pulses will not update the encoder counter if the motor is idle (zero applied voltage).
- Red, long pulse on every serial receive error (framing, overrun or buffer overflow) or bad datagram (overlength, CRC failure).

4.7 Error handling

Any errors generated by the PPI code are reported by the `errmessage()` function which has `printf()` like semantics. Messages are:

- sent to the OLED display "error" screen
- prefixed with `ERR` and transmitted on the serial connection to the Raspberry Pi where the Python library will send them to the logging channel (stderr).

4.8 Shutting down

A shutdown can be initiated by:

1. Pressing the SDN button on the rear end of the PPI board is pushed which directly sets GPIO22 to zero.
2. The Atmel can initiate a shutdown (in the event of low batter) by setting GPIO10 to zero.

A software daemon `python/GPIOSoftShutdown.py` monitors both GPIO lines and performs simple debouncing. At startup it sets GPIO11 to one. When a shutdown request is detected the daemon:

1. sets GPIO11 to zero to inform the Atmel that a shutdown is requested. The Atmel displays the message `"TURN RaspberryPI OFF SAFELY"`.
2. executes a `sudo shutdown now` command to shutdown the RaspberryPi. It will reboot almost straight away so wait until the disk activity has ceased (ACT LED, see Figure 1, has stopped blinking) and then set the power switch to OFF.

The Raspberry Pi cannot actually be shut down, the shutdown command is effectively a synonym for reboot so power must be turned off once the ACT LED has stopped blinking.

4.9 Hat

The PenguinPi can be expanded with a hat board that fits onto the 16-pin connector on top of the PPI board. The current hat supports a simple user interface as well as four infra-red LEDs for localization. Communications with the hat is via digital inputs and I2C messaging.

4.9.1 DIP switch

A DIP switch provides 4 bits of configuration information. It can be read using the HAT_DIP_GET command.

Switch	Bit value	Usage
1	8	
2	4	
3	2	beacon enable
4	1	PI velocity control enable

The top switch, SW4, is used to enable PID motor control. Unused switches might be useful to select a particular boot-time network configuration for the Raspberry Pi (eg. home or lab) via a GPIO line sensed by the Raspberry Pi initialization code.

4.9.2 More LEDs

The grid of 16 LEDs on the top board can be addressed as a single 16-bit word. The bits are numbered bottom to top, then left to right. The bottom right LED is bit 0 and top left LED is bit 15. They can be set by HAT_LEDARRAY_SET.

4.9.3 LED beacons

The hat board contains 4 IR LEDs at its corners which are always on. This requires viewing with an IR sensitive camera, possibly with a visible light blocking filter (optical low-pass filter). Alternatively we could view a beacon pattern on the 4×4 blue LED matrix. An L-shaped pattern can be enabled by setting switch 3 of the DIP switch or by using the command HAT_LEDARRAY_BEACON.

4.9.4 Buttons

An array of 4 pushbuttons on the OLED hat provide a simple but effective user interface. From left to right:

0. Select the next information screen on the OLED display.
1. Stop all motors.
2. Screen specific.
3. User button, HAT_BUTTON_GET returns the number of times the button has been pushed since the last invocation.

4.9.5 Screens

The OLED screen is an array of 128×32 pixels. With a 6×8 font this becomes a 21×4 array of characters. The software supports a number of "screens" which are selected by button 0 (mentioned above). The screens are:

1. Network IP address, shown at startup
2. User data, received text that is not part of a datagram is scrolled. Button 2 clears the screen.
3. Battery status: voltage and current
4. Motor command values
5. Encoder values. Button 2 resets the encoders. Note that in encoder mode 0 if the motor voltage is zero (motor's idle) these values will not change when the wheels are turned.
6. Motor controller data. Button 2 sets the right motor to speed of -20 and left motor to speed of +20.
7. Loop timing statistics
8. System statistics
9. Error messages. Button 2 clears the error message.
10. Last datagram received.

The error screen becomes the current screen whenever an error message is posted.
The screen background is normally black, but becomes white (inverted text) when the battery voltage falls below 7 V.

4.10 Code organization

File	Purpose
`main.c`	main entry point
`datagram.c`	all RPi communications
`motor.c`	timer measuring and stats
`timer.c`	timer measuring and stats
`io.c`	board hardware support
`hat.c`	hat specific code
`lib/uart.c`	serial interface
`lib/twi.c`	I2C interface

4.11 Programming the PPI

The CPU is an ATmega644PA. The C source code for the PPI is provided on the Raspberry Pi along with a cross-compiler toolchain. To build the code is simply

```
% make
```

The RaspberryPi can act like an ISP (`pi_isp`) to facilitate flashing the Atmel on the PPI board via certain lines in the GPIO connector. The code can be flashed on to the PPI

```
% sudo make load
```

which uses `avrdude`.
It is important to use `sudo` to achieve permission to access the Raspberry Pi i/o signals which are used for reflashing. In the event of error messages about the GPIO being busy (this occurs if the reflash is attempted without sudo) the condition can be cleared by

```
% sudo echo 24 > /sys/class/gpio/unexport
```

Note that sometimes the Raspberry Pi will reboot after the Atmel is reflashed.
The fuse values can be read by

```
% sudo  avrdude -c pi_isp -p atmega644p
```

and should be: E=0xff, L=0xff, H=0xd9.
For a brand new PPI board these will not be set correctly and the PPI board will appear to run very slowly, the LED sequence at boot time and the UI. To set the fuses correctly use

```
% make fuse
```

More information about fuse settings can be found at the online fuse calculator at `http://www.engbedded.com/fusecalc`.

4.12 The Git repository

The code currently resides in the Bitbucket repository:
`https://bitbucket.org/serenamou/egb439/src/master`.

5 Communications protocol

The Raspberry Pi and the PPI communicate using datagram messages over a high-speed serial port.

5.1 Serial communications channel

The devices communicate over a bidirectional asynchronous serial port at at 115,200 baud with one stop bit, no parity and no flow control – that is one byte every 87 μs.
The Raspberry Pi sends and receives data packets on `/dev/serial0`. The RaspberryPi 3 SoC has two UARTS: a high-performance UART connected to the BlueTooth device (by default) and a low-performance "mini UART" connected to the GPIO serial port pins and designated as `/dev/serial0`. The mini UART has an 8 symbol hardware FIFO receive buffer. We rely on the Linux kernel device driver to retrieve characters in a timely manner from the UART to prevent data overrun. The high-performance UART (PL011 with 32 byte hardware buffers) requires configuring a hardware overlay. The PySerial 3.4 package provides a convenient interface to the serial port.
The Atmel processor has two UARTS: UART0 and UART1. Each has a 3 byte hardware buffer: serial shift register and a 2-level FIFO buffer. Communications is handled by [1]. It implements interrupt driven circular buffers, with default size 64, to manage serial data in both directions. The buffer length is much greater than a serial packet length so if interrupt latency is kept below 3 character times (270 μs) it is possible to avoid missing any incoming characters. The function `uart_getc` returns a 16-bit value, with the top 8 bits being error flags. The flags are the UART hardware error flags as well as a software buffer overrun flag. These are all defined in `lib/uart.h`. Both UARTs are initialized but only UART0 (Atmel pins 9 and 10) is used. The pins of UART1 (pins 11 and 12) are used for other purposes.

5.2 Datagram structure

The datagram message is organized as

0	1	2	3	4:3+N	4+N
0x11	N	Address	Command	Data	CRC

where N is the datagram length, and includes all those elements shown shaded. The minimum length is N=4 for a datagram with no data payload. The maximum length is 10 bytes but realistically the payload would be at most 4 bytes, that is, N=8. The address field indicates which software component the message is destined for, and the command is component specific.

The command has the top bit set if a response datagram is expected. In this case the return a packet has the same address and opcode and a finite number of payload bytes. The type of data, whether signed or unsigned 8-, 16- or 32-bit integer or float depends on the particular command and the sender and receiver code needs to agree on this.
The packets are read by the PPI main loop in a blocking read of the entire packet. The longest packet, including the start byte would take $11 \times 87 = 0.96\,\mathrm{ms}$ to receive. This induces a worst case delay of nearly $1\,\mathrm{ms}$. The last datagram received can be viewed on the OLED "last datagram" screen. Non-datagram characters are scrolled on the OLED "user" screen which works like a tiny terminal emulator. Lines are only 21 characters wide and the text is wrapped as required. Newline ('\n') starts a new line and formfeed ('\f') clears the screen.
Both the Atmel and Raspberry Pi ARM processor are little-endian (ie. least significant byte in low address). The data payload is organized in big-endian format which is consistent with standard network byte order. This introduces extra complexity from gratuitous byte reordering.
A CRC8 is computed with a generator polynomial of 10010111 (0x97). The algorithm, expressed in MATLAB code, is:

```
function crc = crc8(msg)
    crc = uint8(0);
    poly = uint8(hex2dec('97'));
    for i=1:length(msg)
        v = vals(i);
        crc = bitxor(crc, v);
        for j=1:8
            if bitand(crc, 1) > 0
                crc = bitxor(bitshift(crc, -1), poly);
            else
                crc = bitshift(crc, -1);
            end
        end
    end
end
```

5.3 Command dispatch

Once a datagram has been successfully received the appropriate handler function is invoked. This is executed in the context of the main loop.
The first step is to dispatch the datagram to the appropriate handler function for the particular address (eg. HAT_OLED) or for a class of addresses (eg. MOTOR_L and MOTOR_R share a handler). This is achieved by a large switch statement. An unknown address is passed to the hat's device handler `hat_datagram` and if it is not recognized there it is flagged as an error.
The device handler function uses another switch statement to dispatch the datagram to the appropriate handler function for the particular command. An invalid command is flagged as an error. The command handler function:

- checks that the payload length is correct
- takes appropriate action based on the payload

- if required, creates and queues a return datagram

6 APIs

The PPI hardware communicates with the Raspberry Pi over a high-speed serial link using datagram messages. A number of different APIs are defined: the serial port, Python interface library, RESTful web services and MATLAB.

6.1 PPI serial API

The PPI code maintains an abstraction of devices which are individually addressable:

Address	device
MOTOR_L	left wheel
MOTOR_R	right wheel
LED_R	LED1 RGB red channel
LED_G	LED1 RGB green channel
LED_B	LED1 RGB blue channel
LED_2	LED2 yellow
LED_3	LED3 yellow
LED_4	LED3 yellow
ADC_V	battery voltage
ADC_C	battery current
MULTI	virtual device representing both motors
HAT_OLED	OLED display on PPI hat board
HAT_LEDARRAY	array of LEDS on PPI hat board
HAT_DIPSW	DIP switch on PPI hat board

Each device responds to a device-specific set of commands.
While this structure is very modular, the most common operation of setting robot speed and reading the encoders requires 4 transactions: 4 messages transmitted and 2 messages received. For this reason the virtual "MULTI" device has been added which allows a single command to set both wheel velocities and return both encoder counts.
The most primitive method of interfacing with the PPI is by sending and receiving these datagrams over the `/dev/serial0` using the conventions described in Section 5. There is no C language API for this.

6.1.1 MATLAB and Raspberry Pi coder support

Using the MATLAB coder and Raspberry Pi support package it is possible to write code to communicate with the PPI directly via the serial port. Impressively this code could run in the cloud via MATLAB Online.

```
% PiBot using Raspberry Pi serial port access

MOTOR_SET_SPEED_DPS = uint8(1);
```

```
MOTOR_GET_DEGREES = uint8(hex2dec('82'));
STARTBYTE = uint8(hex2dec('11'));
AD_MOTOR_A                = uint8(4);
AD_MOTOR_B                = uint8(5);

orpi = raspi(ip, 'pi', 'PenguinPi')
serialdevice = serialdev(rpi, '/dev/serial0', 115200)

send_message(AD_MOTOR_A, MOTOR_SET_SPEED_DPS, int16(speedA));
send_message(AD_MOTOR_B, MOTOR_SET_SPEED_DPS, int16(speedB));

function send_message(address, command, value)
    % payload is uint16 big endian, MS byte first
    msg = [address command];
    if nargin > 3
        msg = [msg value2bytes(value)];
    end

    msg = [length(msg)+2 msg];
    crc = crc8(msg);
    msg = [STARTBYTE msg crc];
    write(serialdevice, msg);
end

function readencoder()
    send_message(AD_MOTOR_A, MOTOR_GET_ENCODER);
    m = get_message('int16');
end

function v = get_message(type)
    % look for the start byte and length
    while true
        c = read(serialdevice, 2);
        if c(1) == STARTBYTE
            len = c(2);
            break
        elseif c(2) == STARTBYTE
            len = read(serialdevice, 1);
        end
    end

    % read the payload
    payload = read(serialdevice, len-1);
    m = [len payload(1:end-1)];

    assert(payload(end) == crc8(m), 'Checksum failed')

    v = m(4:end);
    if endian == 'L'
        v = fliplr(v);
    end
    v = typecast(v, type);
end
```

```
function msg = value2bytes(obj, val)
    % convert input value of any type into a byte string in
    % big-endian order

    msg = typecast(val, 'uint8');

    if obj.endian == 'L'
        msg = fliplr(msg);
    end
end
```

and have it compiled and executed onboard the Raspberry Pi using MATLAB `codegen`.

6.2 Python API

The Python API `penguinPi.py` abstracts the PPI devices as Python objects and each object has a set of device-specific methods. These methods are generally light-weight wrappers around code that sends a datagram to, and optionally receives a datagram from, the PPI. The Python API can be used to support robot applications running onboard the robot. The Python API `penguinPi.py` allows parameters to set or retrieved from the PPI processor. The module `penguinPy.py` build upon the PySerial 3.4 package which provides a convenient interface to the serial port. The abstract devices implemented in the low-level API and supported by the Python API are listed in Table 4. For example to flash LED3 the following Python code is required

```
import penguinPi as ppi
ppi.init()
led3 = ppi.Led('AD_LED3')  # create an object representing LED3
led3.set_count(100)      # pulse the LED on for 100ms
```

To turn the left motor is

```
import penguinPi as ppi
ppi.init()
left = ppi.Motor('AD_MOTOR_L')
left.set_velocity(30)  # spin the wheel at 30 encs/sample
```

A key aspect of this API is that it is thread safe. A Python mutex locks the serial port resources and enforces serialization of requests from threads. A datagram reading thread is launched at initialisation which processes and checks incoming datagrams and places them in a queue for return to the thread that requested it. It is critical that only one thread reads from the PPI. The datagram reader thread receives datagrams and places them in a queue. Any other characters are send to the logging channel with time and data stamping for the first character after a newline. This allows error messages from the PPI to be monitored on the Raspberry Pi. A POSIX mutex serializes all datagram exchanges with the PPI. The mutex owning thread can transmit and optionally receive a response via the datagram queue.
The Python and PPI code need to agree on the numeric values of device address and command codes. These are defined by enums in the PPI code `datagram.h`. A python program

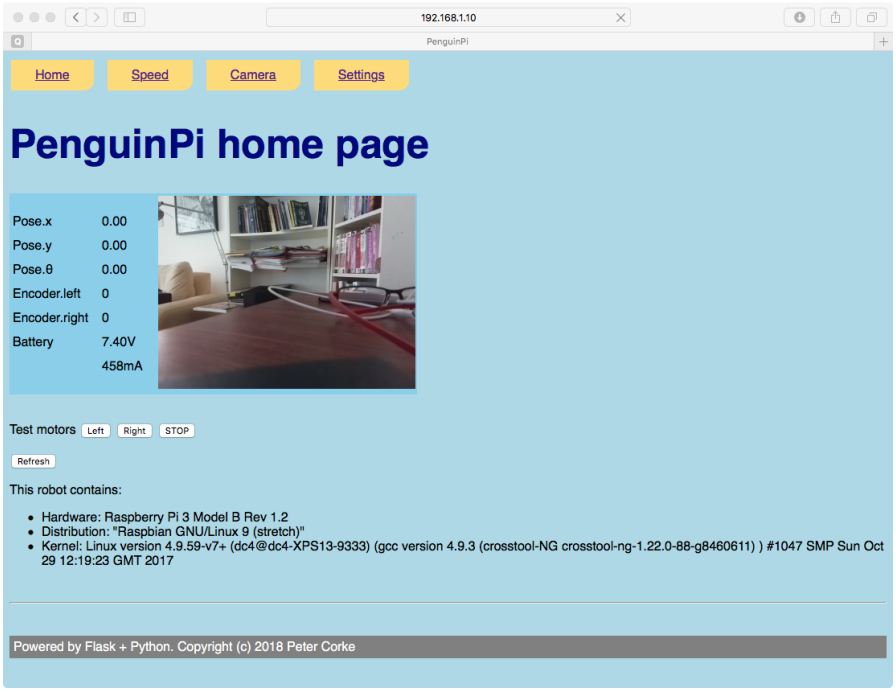


Figure 5: Home page

`defines.py` parses this and saves this as a pickled dictionary `atmel-symbols.pickle` for use by `penguinPi.py`.

6.2.1 Serial port exclusion on Raspberry Pi

The PPI code has a single thread reading datagrams and responding to them. The Raspberry Pi code is more complex, with several processes communicating with the PPI (robot application, IP address daemon). A process opens the serial port with POSIX file locking which is supported by PySerial 3.4. Once a process has the serial port open, others will fail to connect.

6.3 Boot process

At boot time a startup script is run via `crontab` which is edited by `sudo crontab -e`

```
@reboot bash /home/pi/.startup-servers.sh
#* * * * * bash /home/pi/check_network.sh
```

The first line invokes a hidden script in the user pi's home directory which invokes the servers to run at boot time. This used to be separate motor and camera servers, but is now just the `ppweb.py` server.
The second line, commented out, used to run the IP network update daemon every minute. That functionality is now subsumed by the `ppweb.py` server.

6.4 Web API

A light-weight web server written using the Flask Python package and is built on the Python API. It implements a human-friendly interface to the robot, see Figure 5 as well as a RESTful programmatic interface. This interface makes the PenguinPi robot into a network resource which can be controlled by any computer on the same network.
In order to use this we must first determine the robot's IP address which is usually assigned by a DHCP server (this depends on its network configuration defined by `/etc/network/interfaces`). The IP address is displayed on the OLED some time after the Raspberry Pi has booted. The web server can be accessed on port 8080 at the robot's IP address. The home page shows all relevant robot status including a snapshot from the robot's camera, encoder values, dead-reckoned pose, battery voltage, motor testing, Raspberry Pi hardware and software details. Additional pages, accessible via the navigation bar, allow motor speed setting and camera parameter adjustment. All pages can be automatically refreshed by adding the query `?refresh=N` where N is the refresh interval in seconds.
The robot's pose is estimated on board by a thread that performs dead reckoning. It reads the encoder values at 10 Hz (default value) and using knowledge of wheel diameter and wheel separation (distance between the middle of the tyres is used).
Commands are implemented by access to the URLs given in Table 5. In addition some higher level functions are provided

Operation	URL	Query	Return
Set velocity	`/robot/set`	`?vel=LEFT,RIGHT`	state
		`?time=DURATION`	state
		`?acc=DURATION`	state
Initialize state estimator	`/robot/pose/reset`		
Get image	`/robot/camera/get`		image
		`?resolution=(WIDTH,HEIGHT)`	image
		`?awb=AWB`	image

where state is a JSON message that contains odometry and onboard dead reckoned pose. The image is returned in PNG format.
If a time is specified, second argument, the motors are set to the desired velocities and after the specified interval the velocities are set to zeros. All timing done on the Raspberry Pi and this is a blocking command, it returns when the time period has elapsed. If acceleration is specified, third argument, the speed ramps up linearly over the acceleration period, and ramps down at the end. The total motion time is given by the time parameters which must be greater than twice the acceleration time. The displacement for each wheel is $V(T - T_a)$.
This web API allows the robot to be controlled by a program written in any language that has a web or HTTP interface library.
TODO: Commands to read the error log. Command to write string to the OLED.

6.5 MATLAB API

By comparison accessing the web API from MATLAB is very easy

```
webread('http://10.0.0.1/robot/set', 'speed', [20 -30], 'time', 5)
```

The `webread` function automatically converts numeric values to strings, turns vectors into comma separated lists and performs all neccessary character escaping.
A connection to the robot is established by creating an instance of a `PiBot` object

```
>> pibot = PiBot('131.72.14.7')
```

which is passed a string with the robot's IP address. This object has a number of methods:

MATLAB command	Purpose
`state = setMotorVelocity(vL, vR)`	Velocities given as 2 arguments
`state = setMotorVelocity([vL vR])`	Velocities given as 2-element vector
`state = setMotorVelocity([vL vR], time)`	Specify total motion time
`state = setMotorVelocity([vL vR], time, acc)`	Specify time and acceleration time
`stop()`	Stop all motors
`img = getCameraImage()`	Get image from camera
`setLED(led, state)`	Set LED (2-4) to logical value
`pulseLED(led, time)`	Pulse LED (2-4) for integer milliseconds
`n = getButtonCount()`	Get user button presses (and reset)
`printfOLED(str)`	Send text to OLED display

`state` is a structure that contains current encoder values and dead-reckoned robot configuration. For example:

```
>> pibot.setMotorVelocity(20, -20)
>> pause(2)
>> pibot.stop()
```

References

[1] P. Fleury, "AVR UART library." [Online]. Available: http://homepage.hispeed.ch/peterfleury/doxygen/avr-gcc-libraries/group__pfleury__uart.html

[2] P. Koopman and T. Chakravarty, "Cyclic redundancy code (crc) polynomial selection for embedded networks," in *International Conference on Dependable Systems and Networks, 2004*, June 2004, pp. 145–154.

[3] P. Koopman, "Best CRC polynomials." [Online]. Available: https://users.ece.cmu.edu/~koopman/crc/

A Complete command list

A complete list of all commands accessible through the low-level serial API, Python API and web API are given in Table 5. The implementation of each API call on the robot can be found by searching for the PPI opcode in the file `datagram.c`. The symbolic names of the virtual devices on the PPI are listed in Table 4.

Address	Purpose
`AD_ADC_C`	ADC current channel
`AD_ADC_V`	ADC current channel
`AD_HAT`	HAT board
`AD_LED_R`	LED1 red
`AD_LED_G`	LED1 green
`AD_LED_B`	LED1 blue
`AD_LED_2`	LED2
`AD_LED_3`	LED3
`AD_LED_4`	LED4
`AD_MOTOR_L`	left wheel motor
`AD_MOTOR_R`	right wheel motor
`AD_MULTI`	both motors
`AD_SERVO_A`	servo channel A
`AD_SERVO_B`	servo channel B

Table 4: Symbolic names for device addresses

PPI Opcode	Python method	Web call	Argument	Purpose
ADC_GET_POLE	get_pole	/adc/get/pole?chan=N	float	Get z-plane pole
ADC_GET_SCALE	get_scale	/adc/get/scale?chan=N	float	Get scale factor
ADC_GET_SMOOTH	get_smooth	/adc/get/smooth?chan=N	float	Get filtered value
		/battery/get/voltage	float	Get battery voltage (filtered)
		/battery/get/current	float	Get battery current (filtered)
ADC_GET_VALUE	get_value	/adc/get/value?chan=N	float	Get unfiltered value
ADC_SET_POLE	set_pole	/adc/set?chan=N&pole=P	float	Set z-plane pole
ADC_SET_SCALE	set_scale	/adc/set?chan=N&scale=S	float	Set scale factor
HAT_GET_BUTTON	get_button	/hat/button/get	uint8	Get number of button 4 presses since last read
HAT_GET_DIP	get_dip	/hat/dip/get	uint8	Get value of DIP switch
HAT_GET_LEDARRAY	get_ledarray	/hat/ledarray/get	uint16	Get value of LED array
HAT_SET_IP_ETH	set_ip_eth		uint32	Set eth IP address on OLED
HAT_SET_IP_WLAN	set_ip_wlan		uint32	Set wlan IP address on OLED
HAT_SET_LEDARRAY	set_ledarray	/hat/ledarray/set?value=L	uint16	Set value of LED array
HAT_SET_SCREEN	set_screen	/hat/screen/set?value=S	uint8	Select OLED information screen
LED_GET_STATE	get_state	/led/get?led=L	uint8	Get state of LED
LED_SET_COUNT	set_count	/led/set/count?id=L&value=N	uint8	Turn LED on for interval
LED_SET_STATE	set_state	/led/set/state?id=L&value=N	uint8	Set LED state (0 or 1)
MOTOR_GET_CONTROL_MODE				Get motor control mode
MOTOR_GET_ENC	get_encoder	/motor/get/encoder?id=N	uint16	Get motor encoder value
MOTOR_GET_ENC_MODE	get_encoder_mode	/motor/get/enc_mode?id=N	int8	Get motor encoder mode
MOTOR_GET_KVI	get_kvi	/motor/get/kvi?id=N	int16	Get velocity loop integral gain
MOTOR_GET_KVP	get_kvp	/motor/get/kvp?id=N	int16	Get velocity loop position gain
MOTOR_GET_VEL	get_velocity	/motor/get/vel/id=N	int8	Get velocity setpoint
MOTOR_SET_CONTROL_MODE				Set motor control mode
MOTOR_SET_ENC_MODE	set_encoder_mode		uint8	Set motor encoder mode
MOTOR_SET_ENC_ZERO				Set encoder to zero

MOTOR_SET_KVI	set_kvi	/motor/set/kvi?value=V&id=N	int16	Set velocity loop integral gain
MOTOR_SET_KVP	set_kvp	/motor/set/kvp?value=V&id=N	int16	Set velocity loop position gain
MOTOR_SET_VEL	set_velocity	/motor/set/velocity?value=V&id=N	int8	Set motor velocity setpoint
MULTI_ALL_STOP	set_allstop	/robot/stop		Stop all motors
MULTI_CLEAR_DATA	clear_data	/robot/hw/reset		Stop motors, zero encoders
MULTI_GET_ENC	get_encoders		uint16[2]	Get both encoders
MULTI_SET_VEL	set_velocity		int8[2]	Set both motor speeds
MULTI_SET_VEL_GET_ENC	setget_velocity_encoders	/robot/set/velocity?value=L,R&acc=A&time=T	int8[2], uint16[2]	Set both motor speeds, get both encoders
SERVO_GET_MAX_PWM	get_PWM_range		int16	
SERVO_GET_MAX_RANGE	get_range		int16	
SERVO_GET_MIN_PWM	get_PWM_range		int16	
SERVO_GET_MIN_RANGE	get_range		int16	
SERVO_GET_POSITION	set_position		int16	Get position of servo
SERVO_GET_STATE	get_state		uint8	
SERVO_SET_MAX_PWM				
SERVO_SET_MAX_RANGE	set_range		int16	Set maximum position
SERVO_SET_MIN_PWM				
SERVO_SET_MIN_RANGE	set_range		int16	Set minimum position
SERVO_SET_POSITION	set_position		int16	Set position of servo
SERVO_SET_STATE	set_state		uint8	

Table 5: Relationship between PenguinPi opcodes, Python API method name and web API URL.

B Recent changes to the code base

There were a number of limitations within the initial code base:

1. The TC2 overflow interrupt occurs every 12.8 μs. The high rate imposes a very significant computational load as well as being an inconvenient unit of time.
2. The motor speed is at least 20% less than commanded.
3. The motor control exhibits *grittyness* which was likely due to control jitter.
4. Random premature triggering of timers and bad encoder values.
5. Occasional and random data communications errors, particularly at high datagram rates.
6. An error occurred consistently every time the IP address update daemon ran.
7. The worst case loop time, well over 20 ms, introduces a signifcant latency on the response time for datagrams.
8. The datagram logic requires that every GET function has a corresponding SET function which is nonsensical in many cases. Maintaining consistency of opcodes between the PPI and Python code is problematic.
9. The original MATLAB-Python with a bespoke string-based protocol was difficult to extend, treated images inefficiently, and gave little information about the robot state to the student without having to write code.

Numerous changes were implemented to address these limitations:

1. Change the main clock interrupt period to 1 ms instead of 12.8 μs. This is achieved by using timer TC1 which was freed up by eliminating brightness control of the RGB LEDs which not a useful UI feature. This also frees up TC2 which could be used to control two servo channels. TC1, clocked at 20 MHz can also be used to make precision timing measurements, see `timer_get()`.
2. This was the result of the control interval (250 loops) being 20% longer than the 20 ms assumed in the code. This is largely due to interrupt burden, from the TC2 interrupts mentioned above. Additional instrumentation indicates the actual interval was 23.8 ms but it increases with motor speed due to the increased CPU load from encoder interrupts. With one wheel turning at speed 100 the interval increases to 27 ms. The motor control loop now occurs in the millisecond interrupt handler but with interrupts enabled, ie. it executes on the interrupt stack. This ensures accuracy of the control interval.
3. Analysis shows that the control loop interval is quite variable, in particular the OLED refresh (every 1000 loops) takes around 15 ms compared to the normal 0.1 ms loop time. To a lesser extent it is also impacted by datagrams received, and button presses. The estimated velocity is simply computed between calls which assumes that this time is constant.

This implies that the assumed sampling time is incorrect. A longer control interval makes the estimated velocity higher (more encoder ticks accumulate over the longer interval) so the controller will reduce motor speed.

Change 2 also eliminates the control jitter and the motors run much more quietly. The OLED is also updated more periodically in the polling loop, triggered by a counter updated by the millisecond interrupt handler.

4. A number of counters (for time and encoders) are implemented as 16-bit integers that are modified in the timer overflow interrupt handler and tested or reset by non-interrupt code. This is unsafe and declaring the variables as volatile is not sufficient. Operations in the non-interrupt code must be performed with interrupts disabled using the `ATOMIC` macro. With slower timer interrupts, many counters are now 8 bit instead of 16 bit which means that updates are atomic. All counters > 8 bit are now protected by `ATOMIC` macros.

5. Several design issues were addressed:

   - Datagram receive code did not test the UART for data availability and relied on a fixed delay which assumed no inter-character gap. Datagram serial receive now polls the UART for data every 20 μs upto a maximum timeout period.
   - Mutex to prevent multiple Python threads simultaneously communicating with the PPI, has resolved long-standing subtle data communications bugs and performance is now error free, even with very high levels of traffic.

6. Several design issues were addressed:

   - A missing switch break meant that OLED update "fell through" into the ADC code generating an error.
   - The IP address update daemon opens the serial port and its instance of `penguinPi.py` steals incoming bytes from the control process, leading to a reasonable number of failed read requests. An updated version of PySerial has been installed which supports exclusive access to the serial port – preventing the IP address update daemon running when a control process is running.

7. The latency has been reduced by:

   (a) modifying the OLED screen update logic to one memory block per polling loop, instead of all 16 in one polling loop.

   (b) removing the high rate clock overflow interrupt.

8. The datagram logic has been rewritten so that a GET command returns the GET opcode, not the corresponding SET opcode. This allows nonsensical SET codes to be eliminated. All GET codes have their top-bit set. The datagram defines were rewritten as `enums`, rather than `#defines` to ensure that there can be no value clash, and `defines.py` parses the datagram definitions in `datagram.h` and creates a Python pickle file `atmel-symbols.pickle` imported by `penguinPi.py`.

9. A webserver on the robot provides a unified user and programmatic interface to the robot, and is easily extensible.

Other significant changes compared to PenguinPi 2018 include:

- Refactoring of code, greater use of `varargs`, attributes and `setjmp/longjmp` for error handling. All code relating to degrees and angle PID control removed.
- Code is now split across 6 files written in pseudo-OO format. Each `module.c` has a corresponding `module.h` and any exported functions or variables are prefixed with `module_`. Each `module.h` uses preprocessor conditionals to prevent multiple inclusion.
- ADC conversions are now initiated by the millisecond interrupt handler, eliminating jitter and allowing a simple digital filter to be implemented.
- IP addresses were set using one opcode per digit with a 16-bit payload, yet an IP address is simply a 32-bit number. IP addresses are set using a single opcode with a 32-bit payload.
- The CRC polynomial 0xAE is not optimal for small packet sizes. [2] and [3] suggests that for messages in the range 32 to 64 bits the optimal polynomial is 0x97, and this change has been made to the PPI and Python code.
- The start byte is not unique and byte stuffing should be implemented, or introduce a 2-byte start sequence. The start byte runs the risk of collision with data, and this could be reduced by making a 2-byte sequence or byte stuffing, the latter is more complex. With judicious choice of address and commands, the risk of start byte collision is just with the payload so in the worst case their would be four bytes stuffed into the packet. The start byte issue does not appear to be a problem in practice.
- A builtin web server using Flask, see Section **??**, which allows user and programmatic control of the robot
- On board pose estimation by dead reckoning from encoder velocity
- OLED interface improvements:
  - The ability for a user program to display text on the OLED display
  - Additional information screens on the OLED: timing details, last datagram, user text etc.
  - Use of inverse text (black on white) to make the OLED display more readable
  - Some screens use button 3 to perform an action, ie. zero encoders from the encoder screen
  - The OLED screen color is inverted when the battery voltage is low.
- Motor state and control structures have been merged.
- Consistently change motor A and B notation to R and L respectively.

See the git log for more details.

C Desirable hardware changes

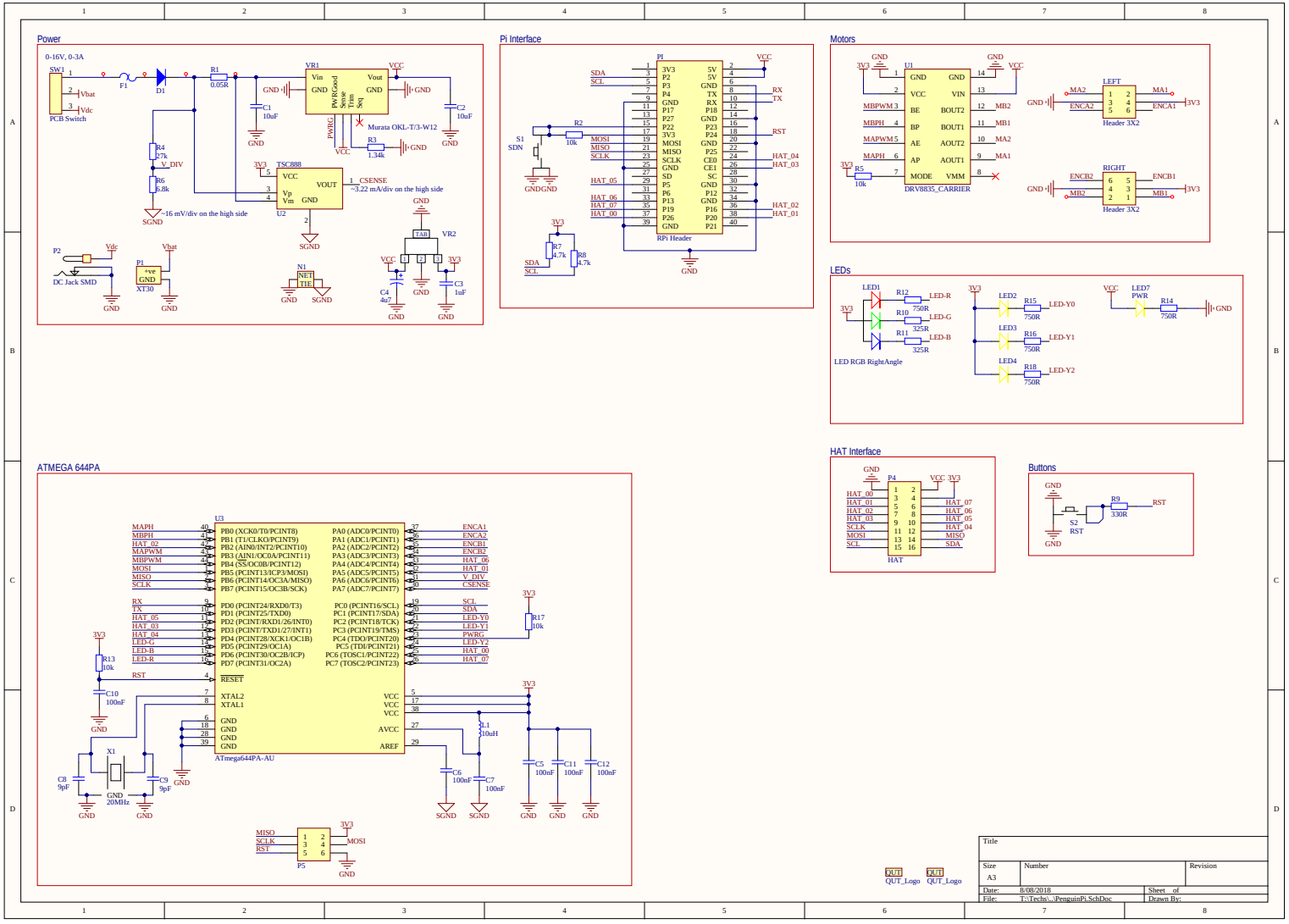
- Power related:
  - Fix the charging connector, it detaches from the board too easily
  - Recharge battery and operate the robot at the same time
  - Can we charge the battery from a standard USB port, rather than special charger?
- Hat related:
  - Add a header on the hat that allows easy prototyping using hat i/o signals (and Vcc), eg. ribbon connection to a prototyping board (and somewhere to put it).
  - Add standard 3 pin headers for servo motors connected to timer 1/2 OCP
  - Add a header for switched DC supply driven by a GPIO from Raspberry Pi or Atmel.
- Fix the placement of the second Hall-effect sensor to enable quadrature operation
- Use an omni-polar Hall-effect sensor to double the encoder resolution

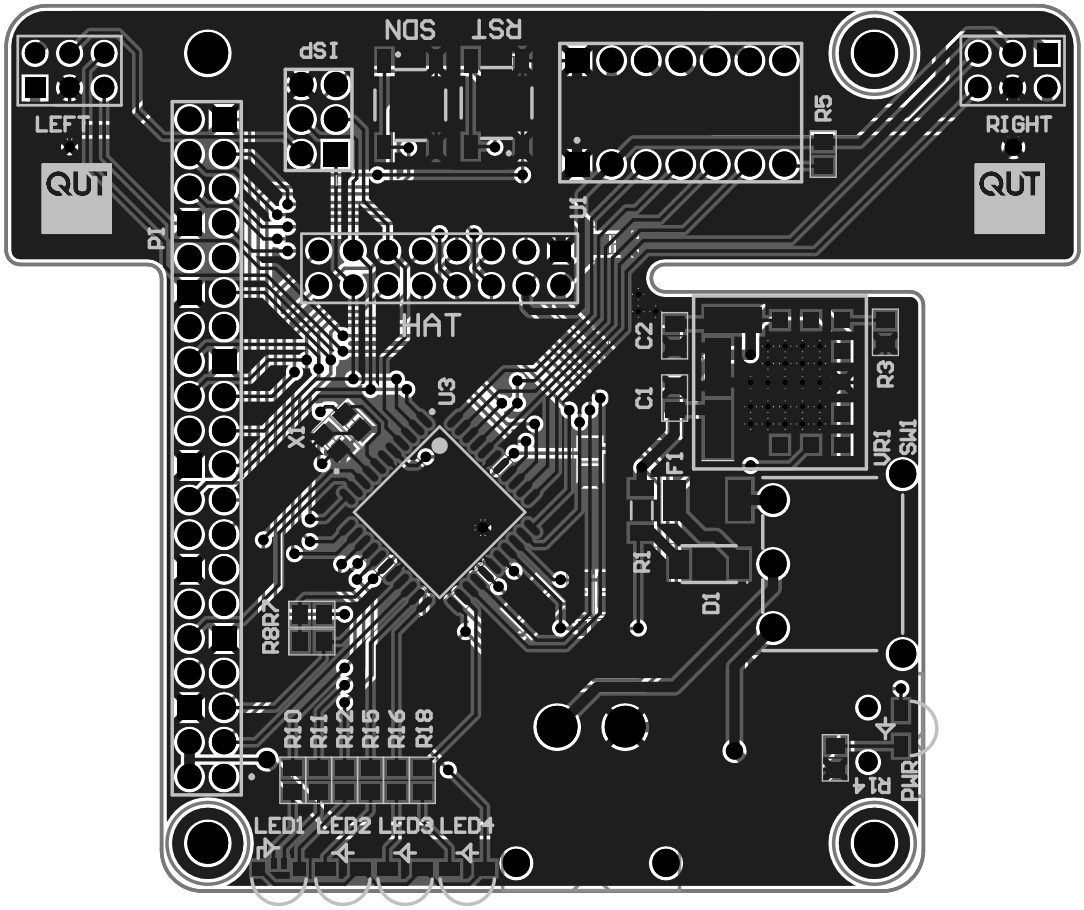
D Random notes

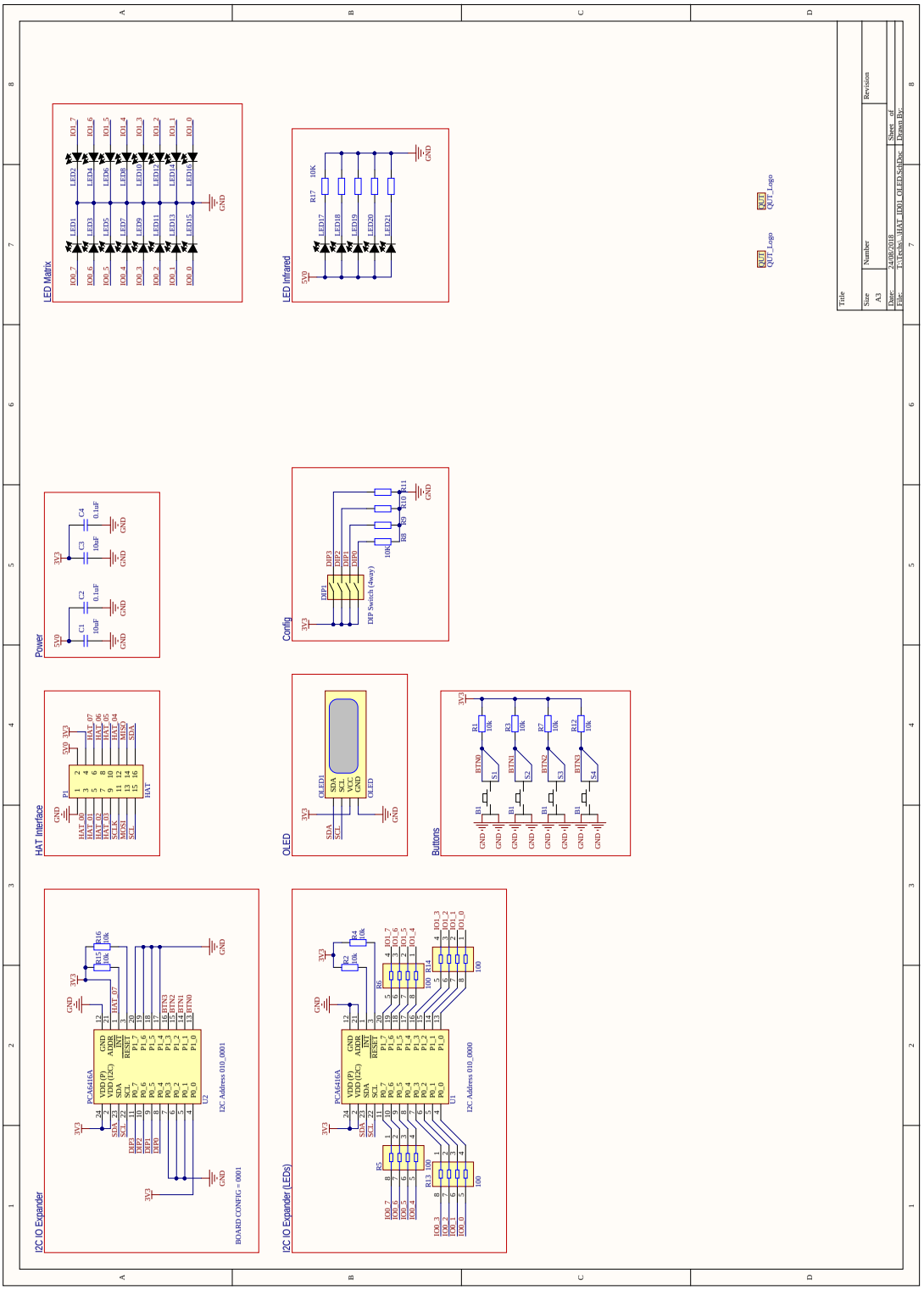
- Login credentials are pi/PenguinPi.
- Network settings: `sudo vi /etc/network/interfaces`
- Check interfaces: `ip address show`
- Automatic process startup: `sudo crontab -e`
- Boot-time process start is handled by `/home/pi/.startup-servers.sh` which is invoked by root's crontab @reboot event.
- We disable the RPi builtin WiFi by a line in `/boot/config.txt`

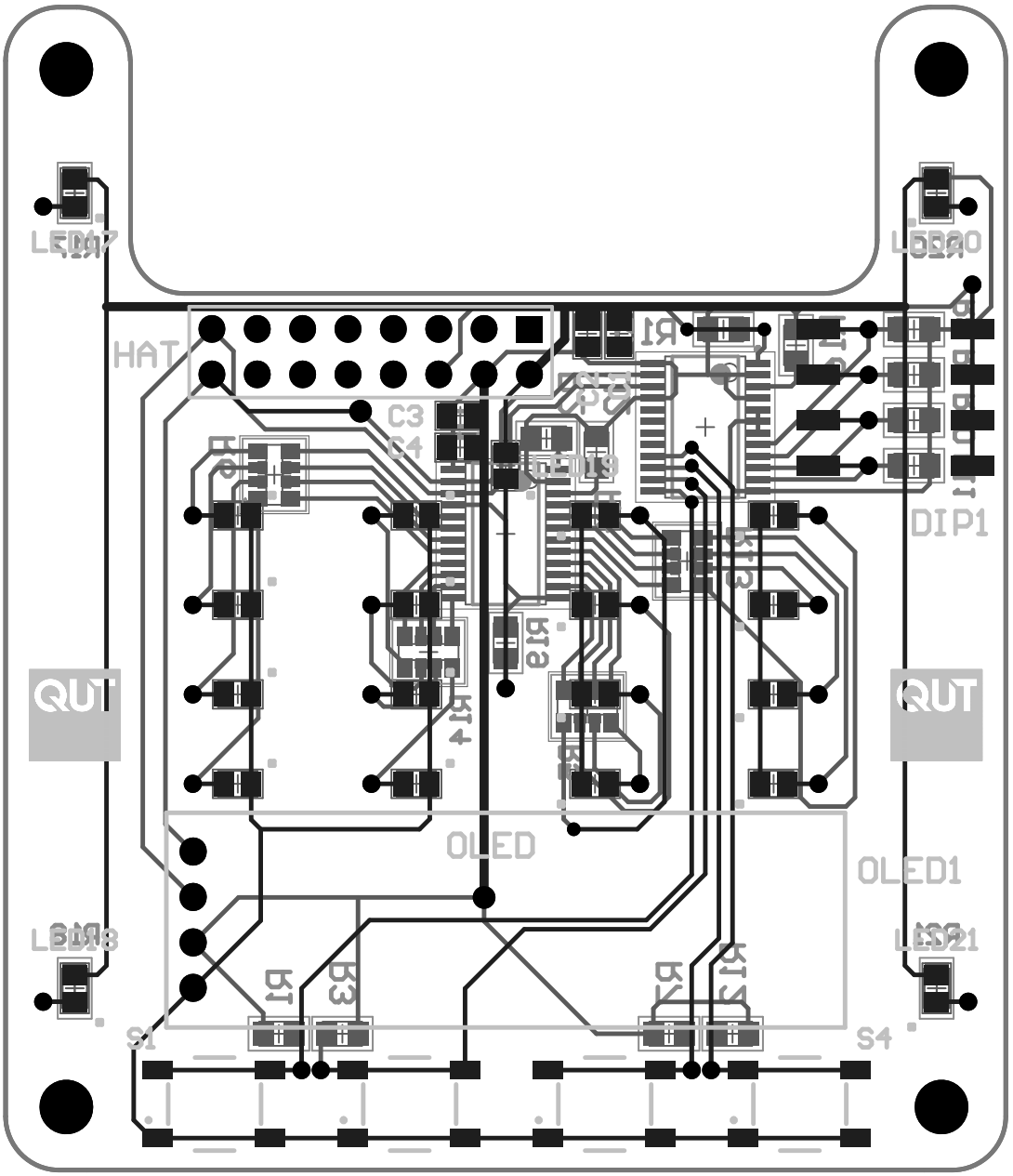
```
dtoverlay=pi3-enable-wifi
```

E Schematics


Power
0-16V, 0-3A
SW1
PCB Switch
F1
D1
R1
0.05R
VR1
Murata OKL-T/3-W12
C1
10uF
C2
10uF
R4
27k
V_DIV
R6
6.8k
-16 mV/div on the high side
TSC888
U2
1_CSENSE
~3.22 mA/div on the high side
R3
1.34k
P2
DC Jack SMD
P1
XT30
N1
VR2
C3
1uF
C4
4u7
Pi Interface
PI
RPi Header
R2
10k
S1
SDN
R7
4.7k
R8
4.7k
Motors
U1
DRV8835_CARRIER
LEFT
RIGHT
Header 3X2
R5
10k
LEDs
LED RGB RightAngle
R12
R10
R11
325R
LED2
LED3
LED4
R15
R16
R18
750R
LED7
PWR
R14
HAT Interface
P4
HAT
Buttons
S2
RST
R9
330R
ATMEGA 644PA
U3
ATmega644PA-AU
X1
20MHz
C8
9pF
C9
9pF
C10
100nF
R13
10k
R17
10k
L1
10uH
C5
100nF
C11
100nF
C12
100nF
C6
100nF
C7
100nF
P5
QUT_Logo
Title
Size
A3
Number
Revision
Date:
8/08/2018
Sheet of
File:
T:\Techs\..\PenguinPi.SchDoc
Drawn By:


RST
SDN
ISP
LEFT
QUT
RIGHT
QUT
R5
J1
P1
HAT
U3
C2
C1
R3
VR1
SW1
X1
F1
R1
D1
R8R7
R10
R11
R12
R15
R16
R18
LED1
LED2
LED3
LED4
R14
PWR


LED Matrix
LED Infraed
Power
Config
HAT Interface
OLED
Buttons
I2C IO Expander
I2C IO Expander (LEDs)
Title
Size: A3
Number
Revision
Date: 24/08/2018
File: F:\Techn...\HAT_D01_OLED.SchDoc
Sheet of
Drawn By:


HAT
C3
C4
DIP1
QUT
QUT
OLED
OLED1
S1
S4