Miniature Unicycle Firmware Documentation

Release

Eric Wieser

Jul 12, 2017

Contents:

CHAPTER 1

Toolchain

PlatformIO

To build the code, you'll need to install PlatformIO. This is used as a a replacement for the Arduino IDE, which has a number of shortcomings:

- Poor dependency management - using other people's code is tricky, and specifying versions of their code is even harder
- No command-line interface - downloading code requires clumsy clicking.

The "Core" version of PlatformIO is sufficient, as that gives you the command line tools.

Once installed, open a terminal and type `pio run`. This should install all the libraries we depend on, and compile the code.

To upload the program, run `pio run -t upload`. See the notes about *Flashing a program* for hardware-related issues with this opperation.

Protobuf

To send messages from the robot to the PC, we use protobuf.

When building the code with `pio run`, the command `protoc --nanopb_out ...` will be executed. This requires that `protoc`, the protobuf compiler, is on the path, and that the nanopb plugin is also installed.

Documentation

The HTML version of this documentation is generated with Sphinx. To generate the help from C++ comments, both Breathe and Doxygen are required.

Once everything is installed, typing *make html* in the documentation directory should rebuild the documentation.

However, this is not necessary - the documentation will be automatically built by Read the Docs whenever this repository is pushed to.

CHAPTER 2

Hardware

The controller for the robot is a chipKIT Max32, which is an arduino-compatible board with a PIC32 microcontroller on board. The specific microcontroller on the board is a PIC32MX795F512, which is relevant when looking for specific hardware features such as timers and PWM.

Power comes from a 7.4V Li-ion battery.

Pin assignments

The following is autogenerated from the source code. Pin numbers correspond to those printed on the board. Within the code, these can be accessed with `pins::TT_DIR`. These can be passed to functions like *`io::timer_for()`* and *`io::oc_for()`* in order to convert them into peripheral objects.

namespace `pins`

Encoders

const uint8_t **`TT_DIR`** = 39

const uint8_t **`W_DIR`** = 47

const uint8_t **`TT_CLK`** = 22

const uint8_t **`W_CLK`** = 23

I2C Sensors

const uint8_t **`IMU_SCL`** = 21

const uint8_t **`IMU_SDA`** = 20

Motors

const uint8_t **TT_FWD** = 3

const uint8_t **TT_REV** = 5

const uint8_t **W_FWD** = 6

const uint8_t **W_REV** = 9

Human interaction

const uint8_t **SWITCH** = 43
Input for the limit switch.

const uint8_t **LED** = PIN_LED1

const uint8_t **USB_RX** = 0

const uint8_t **USB_TX** = 1

Compiler-verification of pin assignments

This project uses a novel approach to selecting microprocessor peripherals from their pin numbers.

A typical microprocessor has multiple different features available on each pin. However, very rarely is every feature available on every pin, and the mapping between these features and their pins is only discoverable through a very long table in the data sheet. For instance, we might find that `OC1` is available on pin `3`.

The result is typically that the source code does not contain any reference to pin `3` at all, and only refers to the underlying hardare, `OC1`. This is bad for maintainability, as it tells the programmer nothing about how to check the wiring, or what to change should the wiring change.

Obviously then, we want a function to convert pin numbers into their corresponding hardware names, and this is not a hard task. But this creates a new danger - it encourages the programmer to change pin a pin to one that no longer supports the features needed! This kind of mistake can be caught at run-time, but when programming takes a few minutes, or getting feedback is difficult, this is not acceptable.

C++11 introduces a new keyword called *constexpr*. When attached to a variable, this tells the compiler that its value must be calculated at compile-time. The trick then, is to produce a function that for a valid pin, *is* calculable at compilation-time, and for an invalid pin, is *not* calculable at compilation-time. We capitalize on the fact that the compiler only cares about the compile-time-calculability of the code-path it takes:

```
// note: no constexpr
int failure(const char* msg) { report_error(msg); while(1); }

constexpr int must_be_even(int i) {
    return i % 2 == 0
        ? i                     // calculable at compile-time
        : failure("Not even");  // not calculable at compile-time
}
```

This almost works as intended:

```
constexpr int ok            = must_be_even(2);
constexpr int compile_error = must_be_even(3);
```

```
int           ok            = must_be_even(2);
int           runtime_error = must_be_even(3);
```

There is a isocpp paper *[N3583]* and a follow-up paper that details possible language changes and workaround to the problem of this third line. The solution opted for was to encourage writing these as:

```
constexpr int ok            = must_be_even<2>();
constexpr int compile_error = must_be_even<3>();
int           compile_error = must_be_even<2>();
int           compile_error = must_be_even<3>();
```

API documentation

namespace io

Devices

These variables just turn the long #defined names into C++ references of the approapriate types

constexpr p32_oc &**oc1** = *reinterpret_cast<p32_oc*>(_OCMP1_BASE_ADDRESS)

constexpr p32_oc &**oc2** = *reinterpret_cast<p32_oc*>(_OCMP2_BASE_ADDRESS)

constexpr p32_oc &**oc3** = *reinterpret_cast<p32_oc*>(_OCMP3_BASE_ADDRESS)

constexpr p32_oc &**oc4** = *reinterpret_cast<p32_oc*>(_OCMP4_BASE_ADDRESS)

constexpr p32_timer &**tmr1** = *reinterpret_cast<p32_timer*>(_TMR1_BASE_ADDRESS)

constexpr p32_timer &**tmr2** = *reinterpret_cast<p32_timer*>(_TMR2_BASE_ADDRESS)

constexpr p32_timer &**tmr3** = *reinterpret_cast<p32_timer*>(_TMR3_BASE_ADDRESS)

constexpr p32_timer &**tmr4** = *reinterpret_cast<p32_timer*>(_TMR4_BASE_ADDRESS)

constexpr p32_timer &**tmr5** = *reinterpret_cast<p32_timer*>(_TMR5_BASE_ADDRESS)

constexpr p32_i2c &**i2c1** = *reinterpret_cast<p32_i2c*>(_I2C1_BASE_ADDRESS)

constexpr p32_i2c &**i2c2** = *reinterpret_cast<p32_i2c*>(_I2C2_BASE_ADDRESS)

constexpr p32_cn &**cn** = *reinterpret_cast<p32_cn*>(_CN_BASE_ADDRESS)

Helpers

These functions allow conversion between devices and relevant configuation needed to use them elsewhere. These prevent a device having to be referred to by name in more than one place.

These should all be evaluated at compile time!

constexpr int **irq_for**(**const** p32_timer &*tmr*)
Get the interrupt bit number for a given timer.

constexpr int **irq_for**(**const** p32_cn &*cn*)
Get the interrupt bit number for the change notice hardware.

constexpr int **vector_for**(**const** p32_timer &*tmr*)
Get the interrupt vector number for a given timer.

constexpr int **vector_for**(**const** p32_cn &*cn*)
Get the interrupt vector number for the change notice hardare.

constexpr p32_timer &**timer_for**(uint8_t *pin*)
Get the timer connected to a given CK (clock) pin.

constexpr p32_oc &**oc_for**(uint8_t *pin*)
Get the output compare (PWM) connected to a given pin.

constexpr p32_i2c &**i2c_for**(uint8_t *scl*, uint8_t *sda*)
Get the I2C connected to a given pair of pins.

Compile-time Helpers

Like the above functions, but with arguments re-expresed as template parameters to force errors at compile time. These make it impossible to choose an invalid pin.

template <uint8_t *pin*>
p32_timer &**timer_for**()

template <uint8_t *pin*>
p32_oc &**oc_for**()

template <uint8_t *scl*, uint8_t *sda*>
p32_i2c &**i2c_for**()

Functions

template <**typename** T>
T **failed**(**const** char *)
Helper function for when an invalid argument is supplied.

Flashing a program

The trace under `JP5` has been deliberately cut on the board. This jumper connects the `DTR` line of the RS232 interface (from the FTDI232RQ chip that tunnels RS232 over USB), to the reset pin of the microcontroller. The flash programmer normally uses this line to send the chip into the bootloader, so its code can be changed.

Unfortunately, this line is also unconditionally asserted when plugged into a computer, making it impossible to attach to a long-running program. The fact that this is a problem at all is a design flaw in the Max32 - were the `CTS` line used instead, there would be no problem.

To reprogram the board then, `JP5` must be temporarily closed. This could be done with a jumper, but this requires that the top layer of circuit board be removed. In practive, this is best done using a screwdriver to short the two pins.

Danger: Do not reprogram the board while the battery is attached! Shorting `JP5` is clumsy, and has a risk of shorting other parts of the board. Your USB port is probably protected against this, but the battery may do unspeakably bad things.

Motors

There are a pair of motors on the unicycle, driver with some custom and unfortunately poorly-documented hardware. The interface to the microcontroller is a pair of pwm lines, one for the forward direction, and one for the reverse direction.

Each motor is a Maxon 110134 with a Maxon 134158 gearbox attached. These motors also have attached *Encoders*.

Functions

void **`setMotorTurntable`**(float *cmd*)
Set the speed of the turntable.

Parameters

- `cmd`: The fraction of maximum speed, in [-1 1]. Positive is counter-clockwise around the positive Z axis

void **`setMotorWheel`**(float *cmd*)
Set the speed of the wheel

Parameters

- `cmd`: The fraction of maximum speed, in [-1 1].

void **`setupMotors`**()
Initialize the timers and PWM needed for the motors.

void **`beep`**(float *freq*, int *duration*)
Play a tone, using the turntable motor.

Not all frequencies resonate well. The built in `<ToneNotes.h>` is handy for turning note names to frequencies. Octaves 6 and 7 are loud and audible.

Parameters

- `freq`: The frequency in Hz
- `duration`: The duration in milliseconds

Sensors

There is a limit switch attached to the top of the robot, for use as basic human input. This switch must be connected to a `CN` pin, as only those pins support pull-up resistors.

Inertial

The robot has a combined accelerometer and gyro board that is sold by Sparkfun. The accelerometer is an ADXL345, and the gyroscope is an ITG-3200.

Both of these sensors use the I2C protocol - in fact, they share a bus. Unfortunately, the builtin arduino Wire interface that implements this protocol does not appear to work on our microcontroller board.

Functions

void **gyroAccelSetup**()
Initialize the connection to the accelerometer and gyro.

template <typename T>
Vector3<T> **chipToRobotFrame**(Vector3<T> *v_chip*)
Convert from the chip coordinate frame to the robot frame.

The robot frame has x pointing forwards (the side with the microcontroller) z pointing left y pointing up

template <typename T>
Vector3<float> **gyroRawToSI**(Vector3<T> *raw*)
Convert from the raw chip reading to radians per second, in the robot frame.

template <typename T>
Vector3<float> **accRawToSI**(Vector3<T> *raw*)
Convert from the raw chip reading to meters per second squared, in the robot frame.

Vector3<int16_t> **accelReadRaw**()
Read the raw values of the accelerometer, in internal frame and units.

Vector3<float> **accelRead**()
Get the acceleration in m s^-2, in the robot frame.

Vector3<int16_t> **gyroReadRaw**()
Read the raw values of the gyroscope, without subtracting initial values.

Vector3<float> **gyroCalibrate**(int *N*)
calibrate the offset for the gyro. Returns the stdev of each component

Vector3<float> **gyroRead**()
Get the angular velocity in the robot frame.

quat **accelOrient**(Vector3<float> *acc*)
Get the robot orientation based on the accelerometer reading. Only accurate when static

quat **accelOrient**()

Encoders

The robot has an encoder on each motor. These go via some circuitry on the board that convert them into two lines - a tick, and a direction.

We count the ticks using the builtin hardware timers, but in order to deal with the direction reversing, we have to monitor the direction pin. We use the "change notifier" hardware to fire an interrupt whenever these pins change, and correct the sign accordingly.

Each encoder is a Maxon 201937, "Encoder MR, Type M, 512 CPT, 2 Channels, with Line Driver".

Functions

void **setupEncoders**()
Initialize the hardware required by the encoders.

void **resetEncoders**()
Reset the counts of the encoders.

wrapping<uint16_t> **getTTangle**()
Get the angle of the turntable, in encoder ticks. Increases with counter-clockwise in the vertical axis

wrapping<uint16_t> **getWangle**()
Get the angle of the wheel, in encoder ticks. Increases with forwards motion

CHAPTER 3

Communication protocol

CHAPTER 4

Terminal

To open the terminal to connect to the robot, run:

```
cd tools
python3 terminal.py
```

You'll be greeted with a `yauc>` prompt, into which you can type `help` for more information.

A typical session looks as follows:

```
...\tools> .\terminal.py
Yaw Actuated UniCycle command line interface
Connected!
Yaw Actuated UniCycle startup
Starting PWM setup
Starting I2C setup
Starting encoder setup
All done
yauc> policy ../ctrl.mat
controller {
  wheel {
    k_pitch: 1.2732395447351628
  }
  turntable {
    k_dyaw: -0.6366197723675814
  }
}

yauc> go 5
Starting bulk mode
Asking for logs
Waiting for reply
No data yet
Asking for logs
Waiting for reply
No data yet
```

```
Asking for logs
Waiting for reply
Test completed
Sending test data
Success
Saved rollout of 5 steps to ..\logs\2017-04-04 102550\0.mat
yauc> disconnect
Connection lost
yauc> ^D
Exiting cli and stopping motors
Already disconnected
yauc>
```

After using the `go` command, the USB cable can be disconnected. The terminal will automatically reconnect to and recieve data from the robot when the USB cable is reattached.

CHAPTER 5

Matlab interface

Unfortunately, there is no stable implementation of protobuf for matlab. Instead, the python terminal outputs .mat files. These contains arrays of structs in the `msg` variable, with the fields in the structs matching the names of the protobuf fields.

Rather than reading these files directly, we expose the same interface as **PILCO_**:

`rollout`(*start*, *ctrl*, *H*, *plant*, *cost*, *verb*)

Shadows the PILCO method. Writes *ctrl.mat* with the necessary information to load in the terminal, and then brings up a file/open dialog to load the logs from the robot.

Parameters

- **`start`** – Ignored
- **`ctrl`** – Contains the policy parameters. Currently assumed to be a linear controller or a random one. When passed a random one, a hand-crafted deterministic controller is used instead.
- **`H`** – The number of steps to run for. Assumed to be 50.
- **`plant`** – Information about the plant. Provides `.dt`, `.in_frame`, and `.out_frame`, which are useful for converting to and from protobuf field names
- **`cost`** – As in the normal rollout, produces costs from the states
- **`verb`** – Ignored

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Bibliography

[N3583] Scott Schurr, Exploring constexpr at Runtime, 2013. https://isocpp.org/files/papers/n3583.pdf

Index