eml_uberdriver Documentation

Release 1.0

mincrmatt12

May 09, 2018

Contents

This is the documentation for project enamel, a system for driving Encoders, Motors and LimitSwitches with an arduino Due and C++. The documentation contains the spec for the protocol between the computer and the arduino, as well as how to use the provided C++ client library.

Warning: Implementation of limit switches is ongoing, and the documentation may not reflect its addition until it is complete.

CHAPTER 1

Introduction

`eml_uberdriver`, also known as project Enamel, is a way to simplify interfacing between motors/encoders/limit switches and ROS, without hassle and overhead.

The setup to use `eml_uberdriver` is fairly simple, but does have some hardware requirements. These are listed below.

1.1 Requirements

The hardware requirements for using `eml_uberdriver` are:

- An Arduino Due (although the firmware will work fine on other board if you change the software reset logic)
- I2C connection between ROS computer and Due

The supported hardware is as follows:

- Motors over PWM
- Two-wire encoders
- Limit switches (standard switches)

1.2 Setup

Setting up the Due for use with `eml_uberdriver` is very simple. Using platformio, the following command at the root of the `firmware/` directory will upload the code to any connected due:

```
$ pio run -t upload
```

This *should* download all the required Arduino libraries and tools and flash the Due.

Hardware setup is as simple as plugging in any motors to PWM pins, encoders to *any digital pins*, and limit switches again to *any digital pins*. The four pins labeled as `ADDRESS_PIN_N` in `firmware/src/constants.h`, where

`N` is from 1-4 are special and are used to set the I2C address of the Due. The four pins (in order) are converted from binary to decimal and added to `ADDRESS_BASE` to form the actual address of the Due.

CHAPTER 2

The Client Library

The Client Library in `eml_uberdriver` is the code on the ROS computer that gives commands to the Due. In this chapter we'll look at a few simple use cases for it.

2.1 Getting Started

To begin, you should add `eml_uberdriver` to your `package.xml` as both build and run dependencies. Also add it to the `CMakeLists.txt` file in the indicated locations. If creating a new package, add it to the end of `create_catkin_pkg` along with `roscpp`.

If all has gone well, you should be able to include `<eml_uberdriver/eml_uberdriver.h>` in your code (make sure to run `catkin_make` before complaining that it didn't work).

2.2 Overview

The Client Library is designed to be very simple, and only uses two classes. The main one you have to worry about is *`eml_uberdriver::ARDevice`*.

Note: For more information about all the methods in the *`eml_uberdriver::ARDevice`* class, see the full documentation in chapter 4.

Using it is very simple, simply construct one with two parameters: the i2c bus and address of the Due.

There are 3 main functions:

- *`eml_uberdriver::ARDevice::openPinAsMotor()`*
- *`eml_uberdriver::ARDevice::writeMicroseconds()`*
- *`eml_uberdriver::ARDevice::openPinAsEncoder()`*

> **Warning:** The last function is a bit of a misnomer, as encoders require two pins to work, and the function takes two pins.

`eml_uberdriver::ARDevice::openPinAsMotor()` opens a pin on the Due as a motor, `eml_uberdriver::ARDevice::writeMicroseconds()` sets a value in microseconds for that motor (by pin), and `eml_uberdriver::ARDevice::openPinAsEncoder()` opens *two* digital pins for use as an encoder, returning an `eml_uberdriver::Encoder` instance.

2.3 Examples

Here are some simple examples using the client library

2.3.1 Simple motors

```
// Open the arduino on address 0x30, bus 1.
// Using i2cdetect -y -q -a <busnum> can allow you to see what devices are on what␣
↪buses
eml_uberdriver::ARDevice device(1, 0x30);

device.openPinAsMotor(6); // open pin 6 as a motor
device.writeMicroseconds(6, 1000); // write 1000 ms pwm rate to pin 6

sleep(1); // wait

device.writeMicroseconds(6, 0); // write 0 ms pwm rate to pin 6
```

This example sends 1000 ms pwm to pin 6 for 1 second, then sets it to 0.

2.3.2 Simple encoders

The encoder API uses the `eml_uberdriver::Encoder` class. This class is documented in chapter 4, but its usage can be seen below

```
// Open the arduino on address 0x30, bus 1.
// Using i2cdetect -y -q -a <busnum> can allow you to see what devices are on what␣
↪buses
eml_uberdriver::ARDevice device(1, 0x30);

Encoder e = device.openPinAsEncoder(2, 3); // open pins 2 and 3 as pinA and pinB of␣
↪an encoder

e.resetEncoder(); // zero the encoder

std::cout << "Move the encoder around for 1 second" << std::endl;

sleep(1);

std::cout << "The encoder reads: " << e.encoderValue() << std::endl; //␣
↪encoderValue() gives the current position of the encoder
```

This example opens and reads an encoder on pins 2 and 3.

CHAPTER 3

I2C Protocol

The eml_uberdriver protocol is designed for Encoders, Motors and Limitswitches.

The device is *not* register based, and instead works more like a UART protocol than anything else.

In order to send commands to the device, one byte indicating the command type is sent, followed by the command payload (arguments)

The commands are as follows:

3.1 Commands

3.1.1 `0x01` - open pin as motor

This command will open a servo on the Due. It will return one byte indicating the name of this servo. The format looks like this:

```
0x01 0xPP
```

Where `PP` is this pin number on the Due.

Important: Servo names are numbers which refer to a servo. In other commands, use this instead of the pin

3.1.2 `0x02` - set servo value

This command sets the value of a servo in microseconds. The input is the servo name to change, and the microseconds to set it to. Its format looks like this:

```
0x02 0xPP 0xVV 0xVV
```

`PP` is the name of the servo to set, and `VVVV` is an unsigned little-endian short denoting the value in microseconds to set it to.

Danger: Sending a servo name not first acquired with command `0x01` will cause **undefined and potentially dangerous** behavior.

3.1.3 `0x03` - reset

Sending this command on its own will reset the state of the Due. All servo names and values are lost.

Note: This is usually called when the program starts and when it quits, to make sure the Due is in a blank state when it is running.

3.1.4 `0x04` - open pins as encoder

Encoders take two pins, let's call them *A* and *B*. These pins are sent in this format:

```
0x04 0xAA 0xBB
```

`AA` is the pin for *A* as an unsigned byte, and `BB` is the same but for pin *B*. This command will (like the servo create command) return one unsigned byte known as the encoder name.

Important: Encoder names are what refer to the encoder.

3.1.5 `0x05` - zero encoder

This command will zero an encoder. The format is this:

```
0x05 0xEE
```

`EE` is the name of the encoder to zero as an unsigned byte.

Danger: Sending an encoder name not first acquired with command `0x04` will cause **undefined and potentially dangerous** behavior.

3.1.6 `0x06` - read encoder

To read an encoder, use this command. The format is this:

```
0x05 0xEE
```

`EE` is the name of the encoder to read.

This command will return a *signed* 4-byte (32-bit) integer denoting the current relative position of the encoder since last zero.

Note: Positive values mean clockwise, negative means anticlockwise.

Danger: Sending an encoder name not first acquired with command `0x04` will cause **undefined and potentially dangerous** behavior.

3.2 Usage Examples

At this time, there are no usage examples for how to use the I2C protocol. A great way to learn is to look at how the client library manages it.

CHAPTER 4

API Documentation

class eml_uberdriver::**ARDevice**

The ARDevice class is the base class used to interface with a connected `eml_uberdriver` Due.

ARDevice(int *busnum*, int *address*)

Constructs a new ARDevice connected to a Due.

Parameters

- **busnum** – The I2C bus this Due is connected to
- **address** – The address the Due has on the I2C bus.

void **openPinAsMotor**(uint8_t *pin*)

Opens a pin on the Due as a PWM-controlled motor

Parameters **pin** – The pin to open

void **writeMicroseconds**(uint8_t *pin*, uint16_t *microSeconds*)

Sets the PWM rate on an open motor pin.

Parameters

- **pin** – The motor *pin* to change PWM for
- **microSeconds** – The duty cycle in microseconds to change to

eml_uberdriver::encoder_id_t **openPinAsEncoderId**(uint8_t *pin1*, uint8_t *pin2*)

Opens *two* pins on the Due as an encoder, returning the **raw id** (see :doc:‘the protocol documentation <protocol.rst> for more information) of this new encoder.

Parameters

- **pin1** – The first pin to open
- **pin2** – The second pin to open

Returns The **raw** encoder id

eml_uberdriver::*Encoder* **`openPinAsEncoder`**(uint8_t *pin1*, uint8_t *pin2*)

Opens *two* pins on the due as an encoder, returning an `eml_uberdriver::Encoder` instance representing this new encoder.

Parameters

- **`pin1`** – The first pin to open
- **`pin2`** – The second pin to open

Returns The `eml_uberdriver::Encoder` instance representing this new encoder.

void **`resetEncoder`**(eml_uberdriver::encoder_id_t *encoder*)

Resets an encoder by its **raw id**.

Parameters **`encoder`** – The **raw id** of the encoder to reset (also known as zeroing the encoder)

int32_t **`readEncoder`**(eml_uberdriver::encoder_id_t *encoder*)

Reads the value contained in an encoder by its **raw id**

Note: Encoder values are positive for clockwise, and negative for anticlockwise

Parameters **`encoder`** – The **raw id** of the encoder to read

Returns The number of ticks since the last reset of this encoder

`class` eml_uberdriver::**`Encoder`**

`Encoder`()

Constructs an *unassigned* `eml_uberdriver::Encoder` instance.

Danger: Using this constructor is only provided so global variables can initialize without needing a constructed `eml_uberdriver::ARDevice` instance available. Using *any other method* on an instance created with this constructor will cause an exception.

void **`resetEncoder`**()

Resets this encoder

int32_t **`encoderValue`**()

Gets number of ticks since last reset of this encoder.

Note: Encoder values are positive for clockwise, and negative for anticlockwise

Returns The number of ticks since last reset of this encoder.

Index