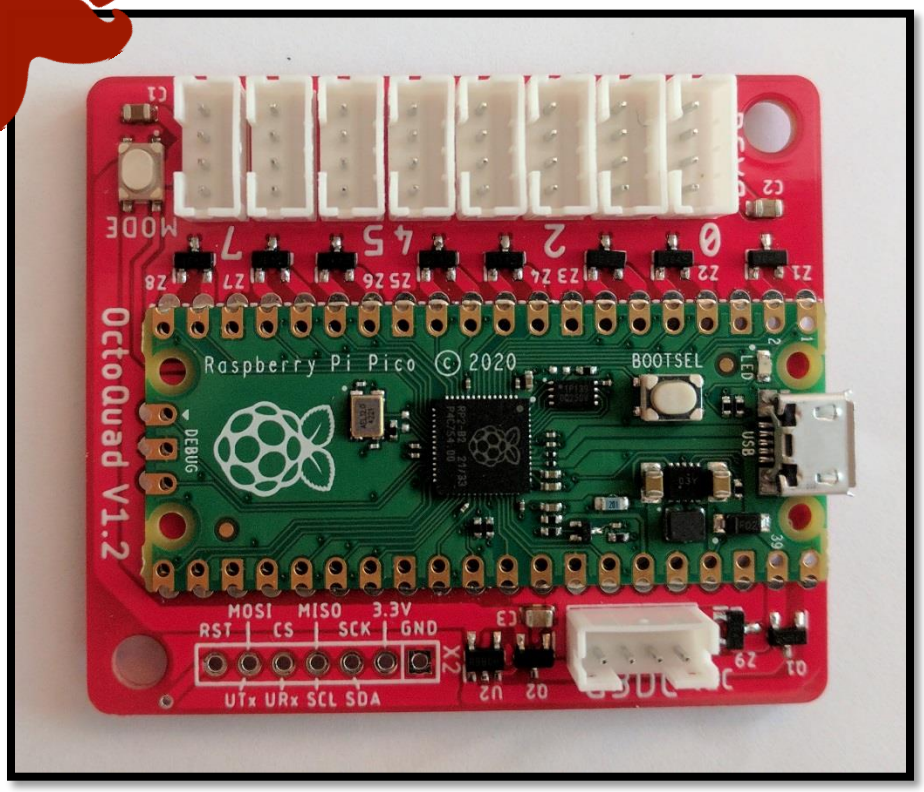
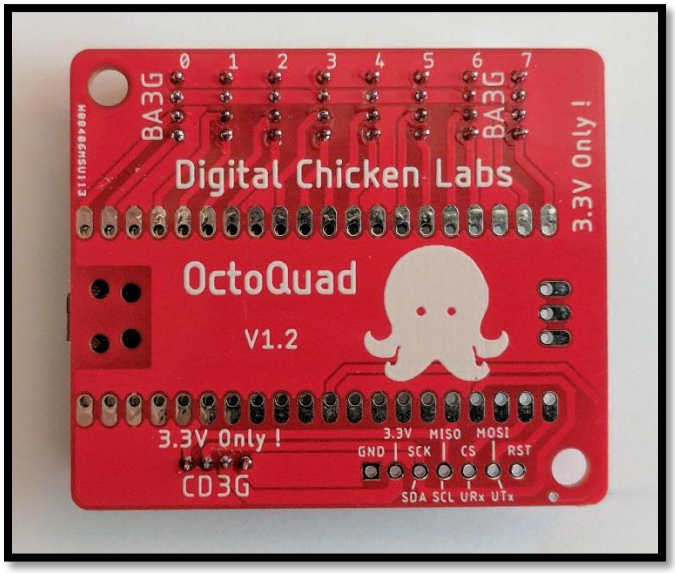
# OctoQuad Specification




**Spec. Rev. 1.0.1 – 7/7/2022**

# Table of Contents

Table of Contents ..... 2
1 Introduction ..... 4
1.1 Overview ..... 4
1.2 Description ..... 4
2 Electrical Specifications ..... 4
2.1 Logic Level ..... 4
2.2 Maximum current draw ..... 4
2.3 Encoder step rate ..... 4
2.4 Quadrature signal ..... 4
2.5 ESD Protection ..... 4
3 Pinouts ..... 4
3.1 4-pin JST-PH Encoder Channel Connectors ..... 4
3.2 7-pin 0.1” header ..... 5
3.3 4-pin JST-PH connector ..... 5
4 Interface Selection ..... 5
4.1 Overview ..... 5
4.2 Changing the active interface ..... 6
5 LED Status indications ..... 6
5.1 Overview ..... 6
5.2 LED Patterns ..... 6
6 Field-Upgradable Firmware ..... 7
6.1 Obtaining firmware files ..... 7
6.2 Flashing firmware ..... 7
7 I2C Interface ..... 7
7.1 Overview ..... 7
7.2 I2C Wedged Bus Recovery ..... 7
7.3 Register access ..... 7
7.4 Register Map ..... 7
7.5 Register descriptions ..... 8
8 SPI Interface ..... 11
8.1 Bus Specifications ..... 11

8.2 Data Protocol ..... 11
8.3 Register Map ..... 12
9 USB Serial Interface ..... 13
9.1 Overview ..... 13
9.2 Protocol ..... 13
9.3 Command Table ..... 13
10 UART Interface ..... 14
10.1 Overview ..... 14

# 1 Introduction

## 1.1 Overview

This document describes the operation of the OctoQuad module and the programming interface.

## 1.2 Description

The OctoQuad can be interfaced to via I2C, SPI, UART, or USB and provides a means to read up to eight quadrature encoders on systems where monitoring the encoder pulses directly using GPIO pins is not feasible. Encoder counts are tracked using signed 32-bit integers, and the counts for each encoder are individually resettable. Additionally, the velocity of each encoder is tracked using a signed 16-bit integer which represents the delta counts during a user-configurable sampling interval.

# 2 Electrical Specifications

## 2.1 Logic Level

The OctoQuad module uses 3.3v logic and power for SPI/I2C/UART bus communications, as well as 3.3v power and logic for quadrature encoder signals. **The I2C/SPI/UART and encoder connections are *NOT* 5v tolerant!**

## 2.2 Maximum current draw

The OctoQuad module can be powered either via USB or via the 3v3 pin on the I2C/SPI/UART bus connection. If powered via USB, the combined current draw from all 8 encoder ports must not exceed 400ma.

## 2.3 Encoder step rate

The encoder step rate must not exceed 1 million steps/sec on any individual port.

## 2.4 Quadrature signal

Note that the OctoQuad does NOT provide pull-up resistors for the A/B quadrature channels. The encoder count will increment in the positive direction when applying positive power to the DC motor.

## 2.5 ESD Protection

The encoder channels and I2C lines are protected from ESD to +/- 15kV (air)

# 3 Pinouts

## 3.1 4-pin JST-PH Encoder Channel Connectors

Each encoder channel connector provides power and A/B quadrature signal connections.

| PCB Pin Label | Function |
|---|---|
| G | Ground |
| 3 | 3.3v power output |
| A | Quadrature channel A |
| B | Quadrature channel B |

### 3.2 7-pin 0.1" header

This connector provides connections to power the OctoQuad and exposes the I2C, SPI, and UART interface pins. Note that the pins are muxed and some pins serve multiple functions (however, only one interface can be active at a time; see section 4).

| PCB Pin # | PCB Pin Label | Function |
|---|---|---|
| 1 (square pad) | GND | Ground connection |
| 2 | 3.3v | 3.3v power input |
| 3 | SCK / SDA | SPI Clock or I2C bus data |
| 4 | MISO / SCL | SPI Transmit or I2C bus clock |
| 5 | CS / URx | SPI Chip Select or UART receive |
| 6 | MOSI / UTx | SPI receive or UART transmit |
| 7 | RST | Reset line for internal microcontroller (active low) |

### 3.3 4-pin JST-PH dedicated I2C Connector

This connector provides connections to power the OctoQuad and exposes the I2C interface data pins. **NOTE:** The I2C lines exposed on this connector are electrically connected to the corresponding pins on the 7-pin header.

| PCB Pin Label | Function |
|---|---|
| G | Ground |
| 3 | 3.3v power input |
| D | I2C bus data line |
| C | I2C bus clock line |

## 4 Interface Selection

### 4.1 Overview

Only one interface (I2C/SPI/UART/USB) can be active on the OctoQuad at a given time. Your choice of interface is saved to non-volatile storage and is automatically applied at power-up. **The default interface is I2C**. **WARNING: Operating the OctoQuad in a different interface mode than that which it has been electrically wired for in the connection to the host device may cause permanent damage to the OctoQuad or to the host device!**

### 4.2 Changing the active interface

To change the interface, follow the following steps:

1. Remove power from the OctoQuad
2. Hold the Mode Select button while applying power to the OctoQuad. Upon applying power, the LED should light and stay lit.
3. Release the Mode Select Button
4. The LED will now loop displaying a blink sequence followed by a pause to indicate a mode. Press and release the Mode Select button to cycle through the various modes. The table below lists how many blinks correspond to which interface mode.
5. Once the LED is indicating the desired mode, press and hold the Mode Select button until the LED stays on solid.
6. After a short time, the LED will then begin blinking the sequence for the interface just selected. Your interface choice is now stored in flash and will be applied at all future startups.

| Number of blinks | Interface |
|---|---|
| 1 | I2C |
| 2 | SPI |
| 3 | UART |
| 4 | USB |

## 5 LED Status indications

### 5.1 Overview

The status light on the OctoQuad is used to indicate various states of communication with a bus master.

### 5.2 LED Patterns

#### 5.2.1 Looping sequence of various number of blinks followed by a pause

Indicates that the OctoQuad is powered up and ready to accept communications over the interface corresponding to the number of blinks (see table in Interface Selection Mode section).

#### 5.2.2 Rapid flashing (7Hz)

Indicates that there is in-flight or recent communication on the bus

#### 5.2.3 Slow flashing (1Hz)

Indicates that bus communication has occurred since power-up, but no recent communication has occurred.

#### 5.2.4 Very rapid flashing (10Hz)

Internal error; please contact Digital Chicken Labs support (digitalchickenlabs@gmail.com)

# 6 Field-Upgradable Firmware

## 6.1 Obtaining firmware files

From time to time, official firmware updates for the OctoQuad may be released. Firmware binaries may be found at https://github.com/DigitalChickenLabs/OctoQuad. **WARNING:** Flashing unofficial firmware may cause permanent damage to the OctoQuad, or to devices to which it is connected.

## 6.2 Flashing firmware

To flash a firmware image onto the OctoQuad, follow these steps:

1. Remove all power and data connections from the OctoQuad.
2. Press and hold the BOOTSEL button
3. While holding BOOTSEL, connect the OctoQuad to a computer using the micro-USB port
4. Wait until the emulated USB drive appears on the computer. The LED will remain off.
5. Drag-n-drop the firmware image onto the emulated USB drive
6. The OctoQuad will automatically flash the firmware and reboot. Flashing is complete when the emulated USB drive disappears and the status LED begins blinking an interface code.

# 7 I2C Interface

## 7.1 Overview

The OctoQuad module conforms to the standard I2C specification, using the register model. Bus clock rates of up to 400KHz are supported. The default I2C address is 0x30.

## 7.2 I2C Wedged Bus Recovery

The OctoQuad can be configured to attempt recovery of a stuck I2C bus in certain scenarios. See the Set I2C Recovery Mode command for more details.

## 7.3 Register access

Some registers are read-only, some are write-only, and others are read/write, as indicated in the register map. Writing to a read-only register will have no effect. Data read from a write-only register is undefined.

## 7.4 Register Map

| Address | Type | Access | Contents |
|---|---|---|---|
| 0x00 | uint8_t | Read-Only | Chip ID (will read 0x51) |
| 0x01 | uint8_t | Read-Only | Firmware version (major) |
| 0x02 | uint8_t | Read-Only | Firmware version (minor) |
| 0x03 | uint8_t | Read-Only | Firmware version (engineering) |
| 0x04 | uint8_t | Write-Only | Command Register |
| 0x05 | uint8_t | Write-Only | Command Data Register 1 |
| 0x06 | uint8_t | Write-Only | Command Data Register 2 |

| 0x07 | N/A | N/A | Reserved |
|---|---|---|---|
| 0x08 – 0x0B | int32_t | Read-Only | Encoder 0 count |
| 0x0C – 0x0F | int32_t | Read-Only | Encoder 1 count |
| 0x10 – 0x13 | int32_t | Read-Only | Encoder 2 count |
| 0x14 – 0x17 | int32_t | Read-Only | Encoder 3 count |
| 0x18 – 0x1B | int32_t | Read-Only | Encoder 4 count |
| 0x1C – 0x1F | int32_t | Read-Only | Encoder 5 count |
| 0x20 – 0x23 | int32_t | Read-Only | Encoder 6 count |
| 0x24 – 0x27 | int32_t | Read-Only | Encoder 7 count |
| 0x28 – 0x29 | int16_t | Read-Only | Encoder 0 velocity (counts / sampling interval) |
| 0x2A – 0x2B | int16_t | Read-Only | Encoder 1 velocity (counts / sampling interval) |
| 0x2C – 0x2D | int16_t | Read-Only | Encoder 2 velocity (counts / sampling interval) |
| 0x2E – 0x2F | int16_t | Read-Only | Encoder 3 velocity (counts / sampling interval) |
| 0x30 – 0x31 | int16_t | Read-Only | Encoder 4 velocity (counts / sampling interval) |
| 0x32 – 0x33 | int16_t | Read-Only | Encoder 5 velocity (counts / sampling interval) |
| 0x34 – 0x35 | int16_t | Read-Only | Encoder 6 velocity (counts / sampling interval) |
| 0x36 – 0x37 | int16_t | Read-Only | Encoder 7 velocity (counts / sampling interval) |
| 0x38 | uint8_t | Read / Write | Encoder 0 velocity sampling interval (milliseconds) |
| 0x39 | uint8_t | Read / Write | Encoder 1 velocity sampling interval (milliseconds) |
| 0x3A | uint8_t | Read / Write | Encoder 2 velocity sampling interval (milliseconds) |
| 0x3B | uint8_t | Read / Write | Encoder 3 velocity sampling interval (milliseconds) |
| 0x3C | uint8_t | Read / Write | Encoder 4 velocity sampling interval (milliseconds) |
| 0x3D | uint8_t | Read / Write | Encoder 5 velocity sampling interval (milliseconds) |
| 0x3E | uint8_t | Read / Write | Encoder 6 velocity sampling interval (milliseconds) |
| 0x3F | uint8_t | Read / Write | Encoder 7 velocity sampling interval (milliseconds) |

## 7.5 Register descriptions

### 7.5.1 Chip ID register

This register will always read 0x51 and may be used to confirm that the bus connection to the OctoQuad is operating correctly.

### 7.5.2 Firmware version registers

The firmware version follows the scheme *major.minor.engineering* where each of three numbers is obtained from the corresponding register. For instance, if the registers read (2, 3, 4) then the firmware version is 2.3.4.

### 7.5.3 Command Register & Command Data Registers 1-2

The Command Register can be used to issue various commands to the firmware, with up to 2 bytes of related data (to be written to the command operand registers). Not all commands require this extra data. For those that do, the operand register(s) *must be written in the same bus transaction in which the Command Register is written*.

The following commands are supported:

| Command | Operand 1 | Operand 2 | Description |
|---|---|---|---|
| 0x00 | N/A | N/A | NO-OP (No command) |
| 0x01 | N/A | N/A | Power On Reset |
| 0x02 | Encoder index bitfield | N/A | Reset encoders |
| 0x03 | Encoder index bitfield | N/A | Set encoder directions |
| 0x04 | 8-bit integer | N/A | Set I2C Recovery Mode |

#### *7.5.3.1 Power On Reset Command*

This command resets encoder counts, directions, and velocity measurement intervals to their power-up defaults.

#### *7.5.3.2 Reset Encoders Command*

This command zeros encoder count(s) inside the firmware. The first operand is a bitfield mapping to encoder numbers. Each bit in the operand corresponds to an encoder, e.g., bit 3 corresponds to encoder 3. When issuing this command, for every bit that is set in the operand, the corresponding encoder's count will be reset.

Multiple encoders can be reset in one write to this register. For example, writing 01000001 to the operand will reset encoder 6 and encoder 0.

| Reset Encoder Command – Operand 1 | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Effect | E7 Reset | E6 Reset | E5 Reset | E4 Reset | E3 Reset | E2 Reset | E1 Reset | E0 Reset |

#### *7.5.3.3 Set Encoder Directions Command*

This command is used to set the encoder count direction on a per-port basis. The first operand is a bitfield mapping to encoder ports. Each bit in the operand corresponds to port, e.g., bit 3 corresponds to encoder 3. When issuing this command, for every bit that is set in the operand, the corresponding encoder channel will be negated.

Multiple encoder channels can be configured one write to this register. For example, writing 01000001 to the operand will set encoder 6 and encoder 0 to be negated.

| Set Encoder Directions Command – Operand 1 | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Effect | E7 DIR | E6 DIR | E5 DIR | E4 DIR | E3 DIR | E2 DIR | E1 DIR | E0 DIR |

#### *7.5.3.4 Set I2C Recovery Mode Command*

This command is used to set how aggressively the OctoQuad will attempt to unwedge a hung I2C bus. Note that issuing this command will save the specified mode to non-volatile storage and will persist across power cycles. Three modes are supported:

- 0: The OctoQuad will not attempt to perform any type of recovery for a stuck I2C bus
- 1: An inter-byte timeout is used for I2C transactions: successive byte transfers must occur within 50ms of each other in order to prevent the timeout from expiring. If the timeout expires, the firmware will assume that the bus has become wedged and will reset the I2C peripheral in an attempt to recover the bus.
- 2: Inter-byte timeout from Mode 1, plus pulling clock low for a small period of time if 1500ms elapses with no communications. May help to unwedge master-side I2C hardware on an incredibly glitch/noisy bus.

### 7.5.4 Encoder count registers

These registers contain the signed 32-bit counters for each encoder.

### 7.5.5 Encoder velocity registers

These registers contain signed 16-bit velocity measurements for each encoder. The velocity is defined as the net change in counts during the velocity sampling interval (see below). For example, if the sampling interval is 100ms and at the beginning of the interval the encoder count is 1234 and at the end of the interval the count is 1200, then the velocity value reported in the register will be -34. This would indicate a velocity of -34 counts/0.1s, or -340 counts/s. To determine the velocity in counts/s, user code must perform the appropriate multiplication factor based on the configured measurement interval. The velocity sampling interval can be reduced to prevent overflow of the 16-bit counters when using encoders that output a very large number of steps per second, or, it can be increased to provide greater velocity precision on low step-rate encoders.

### 7.5.6 Encoder velocity sampling interval registers

These registers are used to set the time interval at which the velocity is calculated for each encoder. The value in these registers is interpreted directly as milliseconds. For example, writing the decimal value “40” to one of these registers means that the velocity for the respective encoder will be measured at 40ms intervals. The default interval is 50ms. Setting the sampling interval to 0 will be disregarded.

# 8 SPI Interface

## 8.1 Bus Specifications

The OctoQuad can be interfaced with via a SPI interface, at up to 1MHz clock rate. The SPI framing format used is Motorola format 3 (Clock high when idle, data latched on rising edge). The slave select line is high when idle. *The slave select line must remain asserted for the entirety of the transaction. The slave select line must be asserted for at least 50µs before the first clock cycle and asserted for at least 50µs after the last clock cycle.*

## 8.2 Data Protocol

The general format for SPI communication with the OctoQuad bears some degree of similarity to communicating with a register-based I2C device, but nonetheless is quite different.

### 8.2.1 Flag Byte

All data frames sent from the SPI bus master to the OctoQuad (including not only data frames used to write, but also those used to perform a read) must begin with a special flag byte. This flag byte is what distinguishes a read from a write. Since SPI is a full-duplex bus (that is, data is transferred from master to slave and from slave to master simultaneously on every clock) there would be no easy way to differentiate writes from reads otherwise.

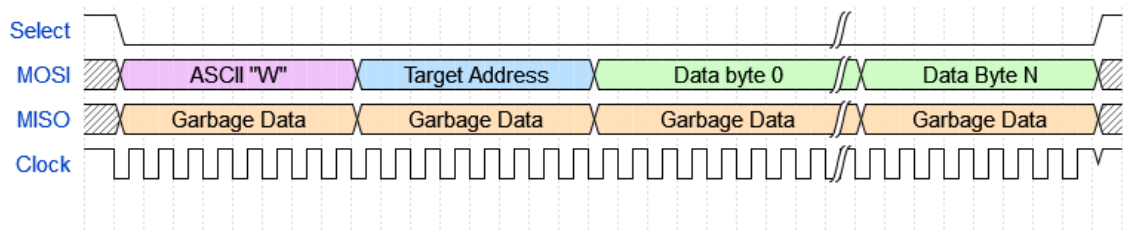
| Flag | Meaning |
|---|---|
| 0x57 (ASCII 'W') | Master is writing |
| 0x53 (ASCII 'S') | Master is writing "sticky" *Source Address* (see below) |
| 0x52 (ASCII 'R') | Master is reading |

### 8.2.2 Writing

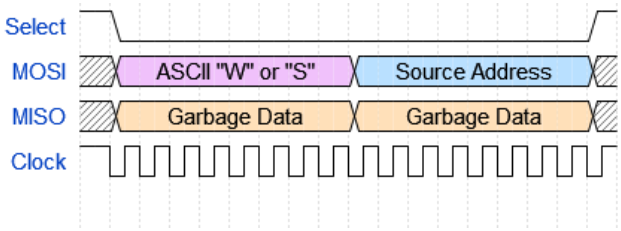
There are two different types of write operations that the master may perform, and these operations are implicitly distinguished by the length of the data frame: (a) Write Operation and (b) Write *Source Address* operation. Moreover, while the 'W' flag may be used for either operation, the 'S' flag may only be used with the Write *Source Address* operation.

In a Write Operation, the *Target Address* must immediately follow the 'W' flag byte. Data bytes to be written into memory starting at the *Target Address* directly follow the *Target Address* in the transaction. A Write Operation must provide at least one data byte following the *Target Address.* Otherwise, it will be interpreted as a Write *Source Address* operation.

The general format for a Write Operation is shown below:



In a Write *Source Address* operation, the new *Source Address* must immediately follow either the 'W' or 'S' flag bytes. The master must not send any more bytes following the *Source Address*; otherwise, the operation will be interpreted as a Write Operation. The *Source Address* is the address in memory from which all read operations will begin receiving data. The general format for a Write *Source Address* operation is shown below:



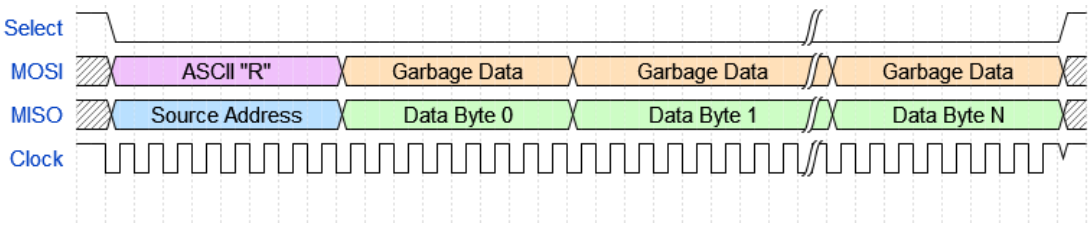

When performing either type of write operation, the master must ignore all received data from the OctoQuad.

### 8.2.3 Reading

All Read Operations begin with the master sending the 'R' flag, after which it may continue to perform N more byte transactions on the bus. All data sent by the master after the 'R' flag will be ignored by the OctoQuad. All Read Operations will begin sending data from the current *Source Address*. The first received byte from the OctoQuad (that is, the byte received while the master is transmitting the 'R' flag) will be the *Source Address* from which the data came. This means if the master wishes to read N bytes from the OctoQuad memory, it must actually perform N+1 byte operations on the bus.

What happens after the master has finished performing a Read Operation is determined by whether the *Source Address* was set in sticky mode or not. If the *Source Address* was set in "sticky" mode, then the *Source Address* will remain **unchanged.** If the *Source Address* was not set in "sticky" mode, then the *Source Address* will be incremented by the number of bytes read during the Read Operation. Using "sticky" mode is helpful should the master wish to repeatedly read the same block of registers without reading from other locations in-between those reads. By using "sticky" mode, the master may simply perform the same read sequence repeatedly without performing a Write *Source Address* operation.

The general format for a read sequence is shown below:




## 8.3 Register Map

In SPI mode, the OctoQuad uses the same register map as in I2C mode. Please refer to the register map in the I2C interface section.

# 9 USB Serial Interface

## 9.1 Overview

The OctoQuad module can be interfaced to as a USB virtual serial port, supporting the USB CDC ACM protocol. No baud rate configuration is necessary, because the USB interface is not bridging to a physical UART.

## 9.2 Protocol

The USB serial protocol is a simple ASCII text-based format. On powerup, the OctoQuad will begin streaming a CSV string of all encoder counts to the host at 10Hz. The host can issue various one-character commands to the OctoQuad when in this mode, including enabling velocity reporting.

### 9.2.1 Data Format

For count-only readings, the string will take the format

`“enc0,enc1,enc2,enc3,enc4,enc5,enc6,enc7\r\n”`

For count & velocity readings, the string will take the format

`“enc0,enc1,enc2,enc3,enc4,enc5,enc6,enc7,vel0,vel1,vel2,vel3,vel4,vel5,vel6,vel7\r\n”`

The values reported are in base 10 (decimal). An example string with velocity reporting might look like:

`“5897,0,0,3974,0,0,0,0,121,0,0,-230,0,0,0,0\r\n”`

In this case, encoder 0 count is 5897, encoder 3 count is 3974, encoder 0 velocity is 121, and encoder 3 velocity is -230

## 9.3 Command Table

| ASCII Char | Effect |
|---|---|
| ‘R’ | Reset all encoder counts |
| ‘0’ | Reset encoder 0 count |
| ‘1’ | Reset encoder 1 count |
| ‘2’ | Reset encoder 2 count |
| ‘3’ | Reset encoder 3 count |
| ‘4’ | Reset encoder 4 count |
| ‘5’ | Reset encoder 5 count |
| ‘6’ | Reset encoder 6 count |
| ‘7’ | Reset encoder 7 count |
| ‘V’ | Enable velocity reporting (following count reporting) |
| ‘v’ | Disable velocity reporting |
| ‘F’ | Set streaming rate to “fast” (60Hz) |
| ‘M’ | Set streaming rate to “medium” (30Hz) |

| | |
|---|---|
| ‘S’ | Set streaming rate to “slow” (10Hz) |

# 10 UART Interface

## 10.1 Overview

The UART interface runs at 115200 baud and mirrors the USB serial interface protocol.

# Special Thanks To


- Chris Johannesen: hardware testing
- Laina Galayde: OctoQuad logo artwork
- Uday Vidyadharan: help with Python sample code