Unofficial Vex Motor Reference

Disclaimer: *This is an unofficial document*. I am just a computer science student and employee of Storming Robots. Due to the lack of formal cross platform documentation I have decided to create a comprehensive reference explaining the motors I2C commands, and how they can be used on other platforms. This project started as an introduction to logic analyzers and culminated with an in depth understanding of the Vex motor commands. Information sharing is important to me, so I have put everything I learned along the way into this document. For those of you who would like to use the Vex motors on a wider range of devices, such as Arduino, I hope this helps.

-Jeremy Desmond

AGAIN THIS IS NOT OFFICIAL. The Vex smart motors were developed and tested in a very particular closed environment. Deviating from what the motors expect will lead to errors. These motors were not designed to work in every condition imaginable so do not treat them that way. You must follow the commands very closely including all the necessary delays or this will not work for you. I created this document to share what I have learned and hopefully help others with their projects. This was not made by Vex and is in no way an official document. Either way I hope you enjoy it.

Resources

This Vex IQ Sensor Reference covers I2C communication between the Vex IQ brain and various sensors. It goes farther in depth about the I2C communication than I will. Reading through the original document will help you to understand the motor process explained here.

Table of Contents

Action	Address
Motor Movement	0x28
Motor Speed	0x2A
Motor Data	0x32
Encoder Target	0x2C
Encoder Reset	0x4F
Address Change	0x4D

Introduction

The VEX IQ brain has twelve ports that can be connected to sensors or smart motors. Each port is a member of an I2C bus within the brain. In total there are four I2C buses with three ports connected to each. Upon receiving power the brain will immediately initialize each individual sensor according to which bus it is on. The startup procedure is very specific, but replicable on other devices. Each motor must go through the same startup process in order to operate with different I2C addresses.

Each bus can support three sensors or motors at a time. Since the motors and sensors have the same default address, there has to be a way to quickly and easily initialize them with new addresses. **WARNING:** This startup process is very important to understand when using the motors on other platforms. The motors firmware is tailored to a very specific set of instructions. When powering the motors on for initialization, it is very important to send only the appropriate commands over the line.

EXAMPLE: leaving an Arduino on means data is always being sent over the line within the loop. If you power on the motors while the Arduino has power, other commands might be sent over the line to the motor. Because of this, the first data the motor receives is NOT the proper startup and initialization commands. Unfortunately, the motors expect to receive a very specific order of commands for initialization so this extra data causes unforeseen errors and the motor will not work properly. I cannot stress enough how important it is to act as a proper Vex brick as much as possible here. Power on the motors and then the Arduino to ensure that the startup process is the first information the motors receive.

Wire Configuration

Vex IQ cables can be stripped and used for communication among other devices or with a breadboard. Internally, there are 6 wires used to communicate over I2C and enable the motor.

Wiring:

Pin (Color)	Signal Name	Description
1 (Blue)	ANA	Analog pin for the output of information from sensors (not used for any motor functions)
2 (Yellow)	I2C_EN	Digital pin for sensor/motor initialization. Digital input, does not output anything.
3 (Green)	I2C_SDA	Serial Data (I2C Communication)
4 (Red)	GND	Ground (0V)
5 (Black)	PWR	Power (~7.2V)
6 (White)	I2C_CLK	Serial Clock (I2C Communication)

Initialization

When motors are first powered on, their I2C communication is disabled. Each motor is set to the default address 0x60. The initialization procedure begins with a broadcast resetting all motors to default values. The broadcast to address 0x00 contains three bytes: 0x4E, 0xCA, and 0x03.

1) Send a broadcast to reset the motors back to default settings:

0x00	0x4E	0xCA	0x03
Broadcast address	Commands		

All motors on the I2C bus are now at default values.

After a short (5 mS) delay the devices are ready for activation. All of the motors will have the same address, so we must **wake each motor and initialize them individually**. The first step is to set the **enable line *low* for 107microsecond**. Once the pulse is sent, request one byte from the default address 0x60. Until a byte is received, the motor is not yet active. The

byte does not need to contain any data (on Arduino, a simple requestFrom() command with a return value of one or more is sufficient). After the pulse is sent and a byte is successfully received from the default address, the motor is now receptive to an address change.

2) Pick which motor is to be initialized – send a pulse over the line.

- Set the Enable line to low
- Wait for **107** microseconds
- Set the enable line high again

3) Read one byte from the default address to check the motors status. All motors have the same address, but only the enabled device is active at this time.

You are now ready to communicate with the specified device.

The address can now be changed using register **0x4D**.

4) Change the address from default (0x60) using the 0x4D register. The byte value sent after 0x4D will be the motors new address:

Write to address 0x60:	0x4D	0x24
		New Address

Now that the new address is set, there is no need for the enable line. Just use the motors new address from now on.

Simple Movement

After the motor is setup with a new I2C address, it is ready for movement commands. **Simple movements are handled with the registers 0x2A and 0x28**. First, set the speed using 0x2A. Two bytes will be sent over the line in this procedure. The first byte is the register 0x2A, and the second is the speed byte. Positive speeds (counterclockwise) are represented by byte values in the range from 1 to 126. Negative speeds (clockwise) are represented by the negative byte values, or the two's complement. A speed of -1% is 0xFF whereas a speed of -100% is 0x82 (see 0x2A for speed specification).

Next, command the motor to begin running with the register 0x28. The movement mode register (0x28) has five settings. For **simple forward movement**, use the setting **0x05**. Once the motor is moving, it will do so until set to a speed of zero.

Stop by setting the speed to zero with register 0x2A. The default brake mode is Coast Brake (0x00). If you wish to brake using either medium or hard brake, then use 0x28 to

change the motor brake mode prior to setting the speed to zero. Medium brake is 0x01 and the hard brake is 0x02.

Sample: Speed of -50% with medium brake (timing not included)

0x2A	0xC1		0x28	0x05		0x28	0x01		0x2A	0x00
Speed			Go			Set Brake (Medium)			Stop (with Medium Brake)	

Motor Encoder Target

The Vex Smart Motors use encoders to provide feedback on the number of rotations and distance traveled. There are 960 encoders per full rotation. The motors can be set to stop at any specific encoder count.

First, set the motor speed using register 0x2A. The motors stop target is set by using register 0x2C. The target consists of three bytes. The first byte sent over the line is the highest order byte, the second byte is the middle byte, and the last byte is the lowest order byte. Three rotations are equal to 2,880 encoders, which is equivalent to 0x0B40. In this case the highest order byte is 0x00, middle byte is 0x0B, and the lowest order byte is 0x40. Once the speed and target are set, the motor is told to go in a *specific motor movement mode*. **0x04** is the motor mode for stopping at the specified encoder target.

Sample: Speed of 100%, target at 3200 encoders (Encoder target mode utilizes hold brake automatically at the target distance)

0x2A	0x7E		0x2C	0x00	0x0C	0x80		0x28	0x04
Speed			Encoder Target (3200)					Go: Encoder Mode	

Servo Motor

Thanks to the hold brake mode, Vex IQ smart motors have the capability to operate as a servo motor. The motor has a special movement mode designated for this action.

First, the speed must be set using the register 0x2A. Then the servo position is set using the encoder target register, 0x2C (you can also check out Motor Encoder Target to see how 0x2C works).

Once the Encoder target has been sent, the motor must be set to the proper motor movement state. Using the motor movement register, 0x28, the servo mode can be activated with **0x03**.

Sample: Servo Motor set to a position of 90 degrees (90 encoders)

0x2A	0x3F		0x2C	0x00	0x00	0x5A		0x28	0x04
Speed: 50%			Position (Encoders)					Go: Servo Mode	

Checking Encoder Count

The best source of motor data is register 0x32. This register holds a total of 6 bytes. The first three bytes are the motors current encoder count. Each byte of the encoder value is sent over the line individually with the first byte being of the highest order and the last byte being the lowest. Appending those three bytes together will yield the motors current encoder count.

Sample (arduino): int encoderCount = (firstByte << 16) + (secondByte << 8) + thirdByte;

Resetting Motor Encoders

Command register 0x4F is used to reset the motor encoders back to zero. Resetting the encoders only requires writing **0x34** to the register 0x4F.

Sample: Encoder Reset

0x4F	0x34
Register	Command

This command will reset the encoders to zero. The encoders are also set to zero (default) when the motor is first initialized. However, this is just a soft reset of the encoders and will not affect the device address.

Command Registers

0x32: Motor Data – The motor constantly keeps track of encoders, speed, and current. This register contains six total bytes of data. The first three are the current encoder count, the fourth byte is the motors status, fifth byte is the speed, and the last byte is current (load). The speed is recorded in terms of the number of encoder ticks per 4 seconds (estimate). This is likely used in the process of syncing two motors to run at exactly the same speed. Based on the relative speeds, the power can be adjusted accordingly. The last byte is a measurement of current the motor needs to keep up speed.

Sample values: Write to 0x32

Read	0x00	0x00	0x08	0x00	0x21	0x02
	Current Encoder count (8)			Motor Status (inactive)		Current (no load)

0x28: Motor Movement Modes - commands written to this address are used to set motor modes, both for braking and movement.

Possible values: One byte is sent that determines the mode the motor is set to-

0x00: Coast Brake (default brake mode)

0x01: Medium Brake

0x02: Hold Brake

0x03: Servo Mode

0x04: Move to Encoder target - Must set target value first see 0x2C

0x05: Move at the set speed (just go)

0x28	0x02

Writing these two bytes would set the motors brake mode to *Hold*.

0x2A: Set Motor Speed - a byte value is written to this register specifying the amount of power to be used once the motor begins moving.

Possible values: byte values range from 0x00 - 0xFF

Counterclockwise: 0x01 - 0x7E (1-126)

Clockwise: 0xFF - 0x82 (255-130)

Stop: 0x00

0x2A	0x3F		0x2A	0xC0
Sending 0x2A and then 0x3F would set the speed to 50%			Sending 0x2A followed by 0xC0 (-0x3F) would set the speed to -50%	

0x2C: Encoder Target – Three bytes are sent to this address, which represent the encoder target.

Possible values: Encoder Target 960 -

0x2C	0x00	0x03	0xC0

Break Down of 960 into bytes (Bit shifting with Arduino)

Highest byte: 0x00 byte highest = (EncoderTarget >> 16) & 0xFF

Middle byte: 0x03 byte middle = (EncoderTarget >> 8) & 0xFF

Lowest byte: 0xC0 byte lowest = (EncoderTarget) & 0xFF

Once the encoder target and speed are set, the motor can be used in Encoder Target mode (see 0x04 motor mode) or Servo mode (see 0x03 motor mode). The motor will stop at the encoder amount sent to 0x2C by using the hold brake. **Negative encoder targets** work the same as with positive encoder targets. The only trick with **negative encoder targets** is that the speed must still be **positive**. First set the speed using the register 0x2A, then set the target using the register 0x2C and sending exactly three bytes, and finally set the motor mode at register 0x28 to 0x04 for regular motor movement or 0x03 for servo motor operations.

0x4D: Address change - Writing to this register assigns a new I2C address to the device.

Possible Values: Any address that is not already in use by another device

Example: 0x24 - writing 0x4D followed by 0x24 would change the device address to 0x24.

0x60	0x4D	0x24
Default Address	Register	New Address

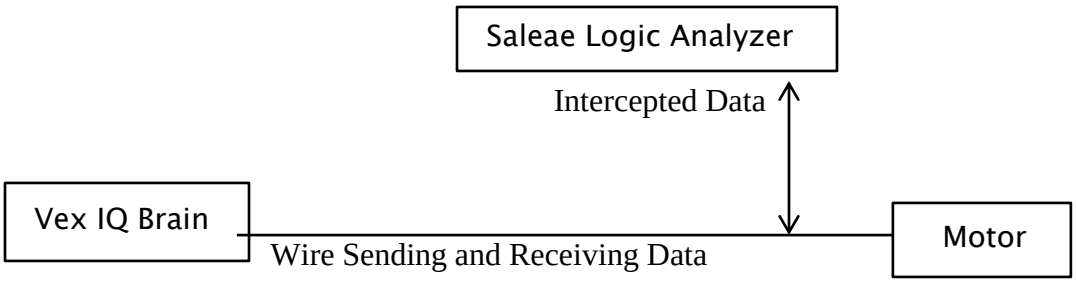
0x4F: Reset - Commands written to this register are used to reset the motor encoders to zero.

Possible Values: 0x34 is used to **reset the motor encoders** on the motor.

0x4F	0x34
Register	Command

Method

Finding out this information required a Vex IQ brain, Vex IQ motor, and a logic analyzer. The logic analyzer was used to see the data being sent over the line from brain to motor. A logic analyzer works as an intercept between devices and can relay back exactly what bits are being sent over the I2C lines.



The brain and motor use I2C communication for all procedures. This means that any device capable of being an I2C master can potentially be programmed to use these motors, we just need to know what commands to use. To determine what commands were controlling the motor I used a Saleae Logic Analyzer. This is a great device for troubleshooting, but here it was used predominately for the purpose of reverse engineering.

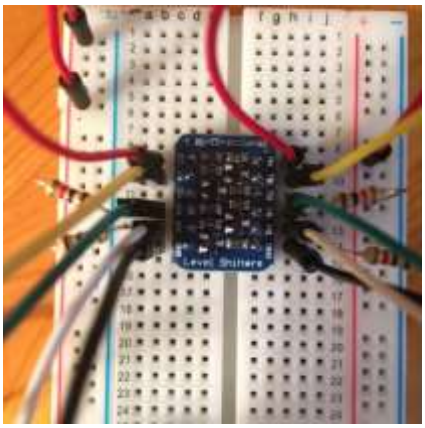
By attaching wires to the data, ground, and enable lines I was able to see all the communication going on between the motor and the brain. The goal here is to act *exactly* as the Vex does, so the first thing I looked for was the startup commands. To do this I simply setup the logic analyzer and powered the Vex on to see what commands were sent to the motor upon first powering on. Here I learned about the initial broadcast to all devices and how the enable line plays a crucial role in starting up the device. These commands were critical for the initialization process of the motor and needed to be matched exactly, including the timing.

Once I was sure how the initialization process worked I moved on to simple motor commands. This included setting the speed, using various brake modes, and operating in servo mode. I wrote up some simple code on RobotC and watched the commands being transmitted to the motor using the logic analyzer. The most complex process is initialization. All other commands are much simpler.

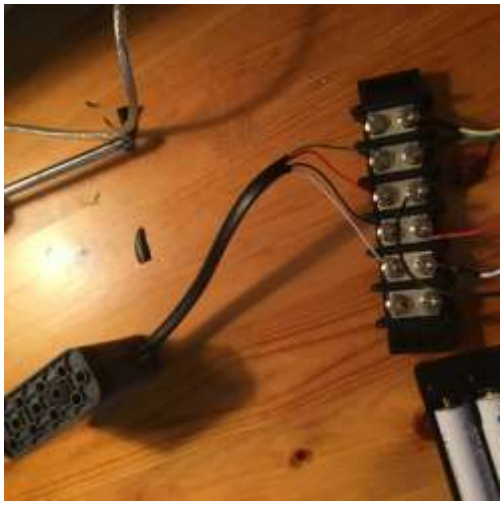
Arduino Setup

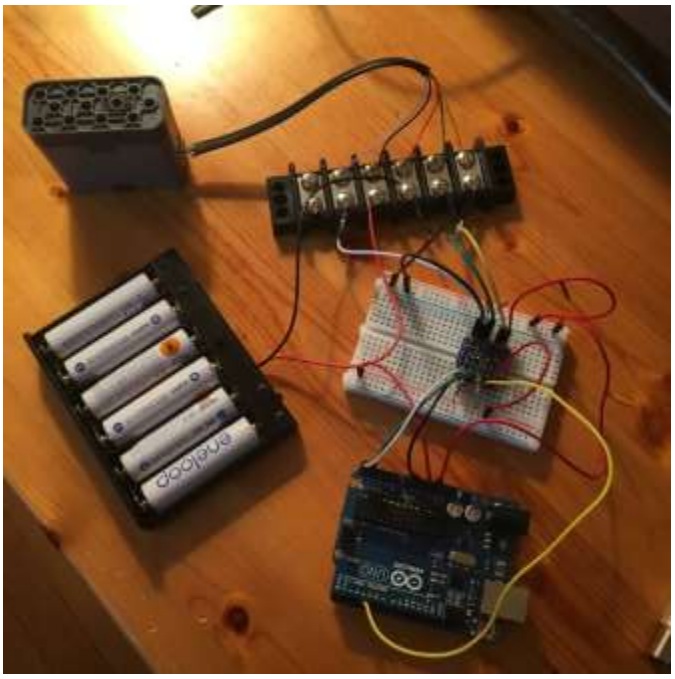
The Vex motors work best with the Vex rechargeable NiMH battery, as opposed to six alkaline 1.2V batteries, even though they reach the required 7.2V operating voltage. The motors can pull the proper amount of power from the official Vex battery pack, but not the alkaline batteries. This means that the motors will not be as powerful using an alkaline battery pack as opposed to a NiMH battery, but it will still work.

The Arduino will work off the 7.2V battery pack, but it is advised to use the suggested 9V external power supply. However, if you really don't want to use separate power supplies you can power an Arduino off of the same power supply being used for the motor, 7.2V. The problem here is that when you power the Arduino through the Vin pin, there is a voltage drop from the voltage regulator. The voltage regulator is not the best on the Arduino and there is always a drop of about 1.5V before the regulator lowers the voltage to the required 5V. So you can't use 6V to power the Arduino since the voltage drop from the regulator will go below the operating voltage right away. When using an alkaline battery pack the voltage slowly goes down throughout usage. Because of this, your Arduino and motor might not be getting enough power after only a short period of time.



In addition, since most Arduinos output information at 5 Volts you will need a logic level shifter to accommodate the motors 3.3V input. The path of the data, clock, and enable lines must all go through a bi-directional level shifter. Signals from the Arduino will be shifted to the appropriate 3.3 Volts required by the motor. This is just a strong suggestion. I did manage to connect the motor clock and data line directly to the Arduino and everything worked fine, but this was only for simple testing purposes. If you want a reliable setup that is sustainable I suggest using a level shifter or a 3.3V board.





The power and ground lines can come straight from the external power source. The analog line can be connected to any analog pin on the Arduino. This is what the final setup looks like. The Arduino is powering the 3.3V and 5V side of the level shifter. Also don't forget a common ground and the pullup resistors for clock and data lines!

This is a four-motor setup I did with a mux on the I2C bus to test reliability with multiple motors. I used the same idea as with a single motor, but this time needed an extra logic level shifter for all the clock, data, and enable lines. One level shifter has all the enable pins, while the other level shifter handles the clock and data lines. Since the Vex Motors have their own enable and initialization process, all devices can be on the same line despite having the same address in the beginning. As long as the addresses are properly changed before sending any other commands over the line there should be no problems. The mux is on the I2C bus to test for any interference it might cause to the motors. Fortunately there were no issues so many other I2C devices can be applied to this build as well.

Hardware

Wiring the Vex IQ motor to connect to a bread board is limited in option.

Screw Terminal – Once the wires are stripped, they can be attached to a screw terminal with jumper wires attached on the other side. From there the wires can be connected directly to a breadboard. This option is simple, but not very elegant and takes up a lot of room.

Solid Core Wire – Solid core wire is much stronger and more durable than the stripped Vex wires. In combination with heat shrink, or simply electrical tape, solid core wire can be an easy substitute for screw terminals. Since it can be plugged directly into the breadboard, there is no need for extra jumper wires.

Male headers – You can crimp male headers onto the stripped wires.

https://bigdanzblog.wordpress.com/2015/05/28/how-to-crimp-your-own-0-1-terminals-to-jumper-wires/ here's a tutorial for doing so.

Proto Board – The most permanent option is to solder the stripped wires directly to a proto board. This is the most difficult, but permanent method.

Arduino options:

3.3V board – This would be the easiest option. With a 3.3V board you don't have to worry about the logic level shifter since the Arduino will already be outputting signals at 3.3V, which is what the Vex IQ needs.

5V board – Any other Arduino that is 5V will work fine, but you have to remember to use the logic level shifter to drop the signals from 5V to the Vex IQ motor preferred 3.3V.

Logic level shifter – this is a necessity if you are not using a 3.3V board. The Arduino can power both sides of the device, but the I2C and enable lines from the Arduino should go on the 5V side and all the wires going to the Vex IQ motors should be on the 3.3V side.

Wiring – You need to make sure that there is a shared ground between all devices. This includes the level shifter, mux (if you have one), all the motors, and the Arduino. The Arduino can power both sides of the logic level shifter (if you have one). All the motors can be on the same I2C bus despite starting out with the same address. Make sure you have a pullup resistor on the I2C bus (pulled up to 3.3V not 5V).

Last update: 2/22/18