VEX IQ SENSOR REFERENCE

Version 1.00

Date: August 5, 2014

1st Edition Rev 1

Copyright Information

Revision History

Rev	Date	Description
Original	5 Aug 2014	First release

1 Introduction

The VEX IQ system was launched in April 2013 as an educational robotics platform aimed at younger students. The heart of the system is the VEX IQ brain, a microprocessor based hub with 12 ports able to control motors or receive information from sensors.

Five different sensors are currently available for the VEX IQ system, a bumper switch, touch led, gyro, distance sensor and color sensor. All except the bumper switch contain their own micro-controller and use serial communications to send or receive information to and from the brain.

Despite being designed for educational use, there has been considerable interest in the VEX IQ system by advanced users who would like to extend its capabilities. The first step towards achieving this is the addition of custom user created devices that can connect to the brain and communicate using the same mechanism as the official products.

This document describes how VEX IQ sensors connect to the VEX IQ brain, both the physical connection and the protocol used to communicate with them. The document then describes a reference sensor design that can be used as the basis for generic sensor development. The reference design also includes a schematic and open source firmware; it does not include PCB layout or packaging suggestions.

2 System Overview

The VEX IQ brain has 12 ports that can be connected to either smart motors or sensors. When the IQ brain is first turned on, it automatically detects which sensors are connected. A firmware update will allow, "hot swapping", that is the addition of sensors while the brain is powered, in the near future.

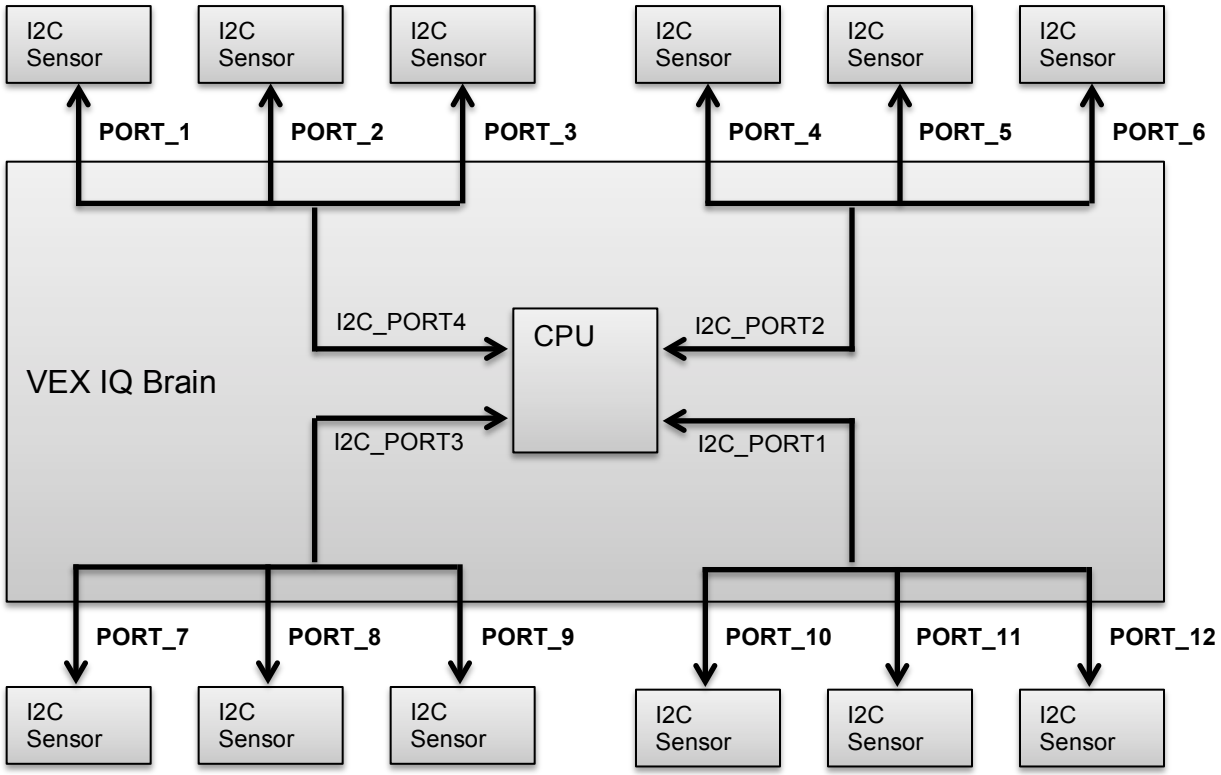


Figure 1 – VEX IQ I^2C bus layout

The VEX IQ brain divides its ports into four groups each of three ports as shown in figure 1. Each sensor or motor that is part of a group needs to be assigned a unique address so it can communicate with the brain. The brain automatically assigns this address when it is turned on.

2.1 Sensor connections

A VEX IQ sensor is connected to the brain using a smart cable. The cable uses an offset latch 6P6C connector at each end and is wired pin to pin, that is, pin 1 at the VEX IQ brain is connected to pin 1 at the sensor. The VEX smart cable is not compatible with those used for either telecommunications or other competitive robotic systems.

Pin	Signal Name	Signal Direction at sensor	Description
1	ANA	Output	Analog signal, output from sensor
2	I2C_EN	Input	Sensor enable, Digital input to sensor
3	I2C_SDA	Bi-directional	I2C Data signal - bidirectional
4	GND	Input	GND (0 Volts)
5	PWR	Input	POWER (~7.2 Volts)
6	I2C_CLK	Input	I2C Clock

Appendix A has a drawing of the VEX smart cable showing pin numbering and cable wire colors.

2.2 Sensor communications protocol

Sensors communicate with the VEX IQ Brain using the I^2C communications protocol. Two wires, serial data (SDA) and serial clock (SCL) carry information between the sensor and the VEX IQ brain. Each sensor is considered to be a slave device, the VEX IQ brain is the I^2C master supplying the clock and initiating all communication. The VEX IQ brain communicates with the sensors in standard mode, the I^2C clock is 100KHz. Information is placed on the I^2C bus 8 bits at a time followed by a single bit acknowledgement. It is beyond the scope of this document to describe the I^2C protocol in detail; however, an overview is given in Appendix B. For a full understanding of the protocol, including such things as detailed timing requirements, refer to the I^2C protocol specification and other references given in the Appendix.

As each sensor shares the I^2C bus with two other devices it needs a way to identify which messages it should use. To achieve this each sensor needs has a unique "slave address" that is assigned when the VEX IQ brain is first powered. Section 2.5 explains this procedure and the functionality that a sensor needs to implement to achieve this.

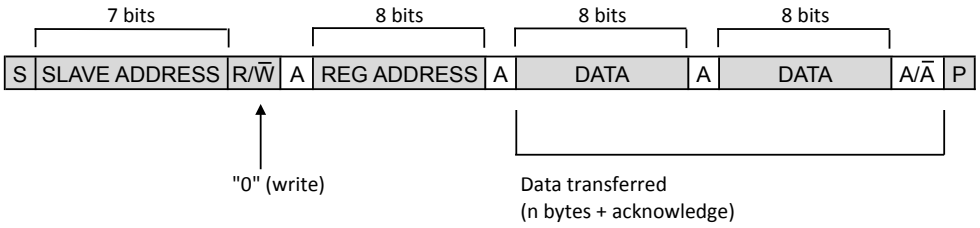
The I^2C slave address is a 7 bit number with an additional 8^{th} bit that designates a read or write operation. This document uses the convention of naming all slave addresses as their 8bit write address equivalent, for example, slave address 0110000 plus r/w bit as 0 will be 0x60 as an 8 bit address. The companion read address would be 0x61.

A sensor can be considered to be a memory device with 256 unique locations that the VEX IQ brain can read or write. The brain communicates with the sensor by first indicating which memory location to start from, followed by reading or writing one or more data bytes. After each byte is read or written, the sensor will automatically increment the memory location counter. Each of the 256 possible memory locations is called a register. Some are read-only and contain sensor identification information, others are read-write and many are sensor specific and have unique meanings. We cover the register map in detail in section 3.

2.3 Write transaction

A write transaction to a VEX IQ sensor has three parts.

- A slave address is sent with the r/w bit set to 0
- A register address is sent where the data that follows will be stored
- One or more bytes of data are sent. After each byte is received and stored by the sensor it increments the internal register address.



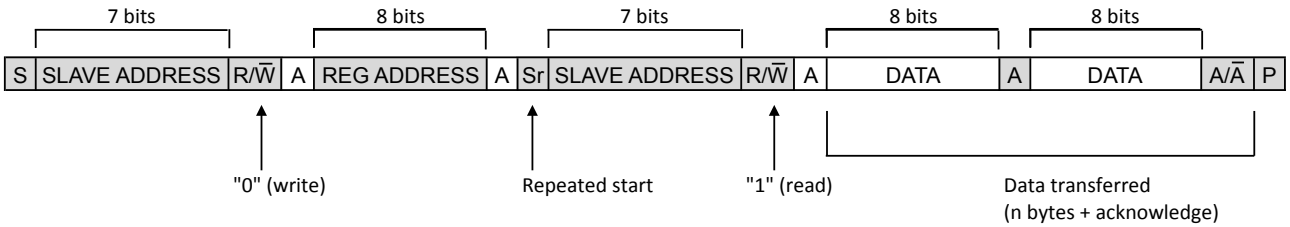
S Start bit
A Acknowledge bit
P Stop bit

Figure 1 - Write Transaction

2.4 Read transaction

A read transaction is slightly more complex than a write transaction.

- A slave address is sent and the r/w bit set to 0
- A register address is sent to where the data will be read from
- A new start bit is sent, this is called a repeated start bit and allows the brain to change the communications direction from write to read without releasing control of the I^2C bus
- A slave address is sent and the r/w bit set to 1
- One or more bytes of data are read. After each byte is sent by the sensor it increments the internal register address.



S Start bit
A Acknowledge bit
Sr Repeated start bit
P Stop bit

Figure 2 - Read Transaction

2.5 Sensor Initialization

When a sensor is first powered, it should disable its I²C communications or set the slave address to one that the VEX IQ brain does not use, for example, 0x62. The VEX IQ brain will attempt to detect which sensors are connected using the following sequence. (Refer to Figure 3 - Initialization sequence)

- A reset command is sent to the general call address (0x00)
- A wake up pulse is sent to the sensor on port 1 using a digital IO line. This instructs the sensor to enable its I²C bus and assign a default I²C address.
- The brain attempts to read one byte from the sensor. A register address is not sent as the read location is not important. If a sensor is present it will return an ACK signal after the slave address is sent followed by the one byte of data. If a sensor is not connected to this port then a NACK will be received by the brain (the I²C data line remains high) and it is assumed that a sensor is not connected to this port.
- The brain now communicates with this sensor and assigns it a new I²C address, usually the first port in a triplet will be assigned the address 0x20, the second 0x22 and the third 0x24, however, this behavior is not guaranteed.
- The brain repeats this process for each of the three ports in a triplet.
- After the VEX IQ brain has attempted initialization of the three ports, normal communications will start. The brain will typically request status from each sensor in turn to discover sensor type and firmware version.

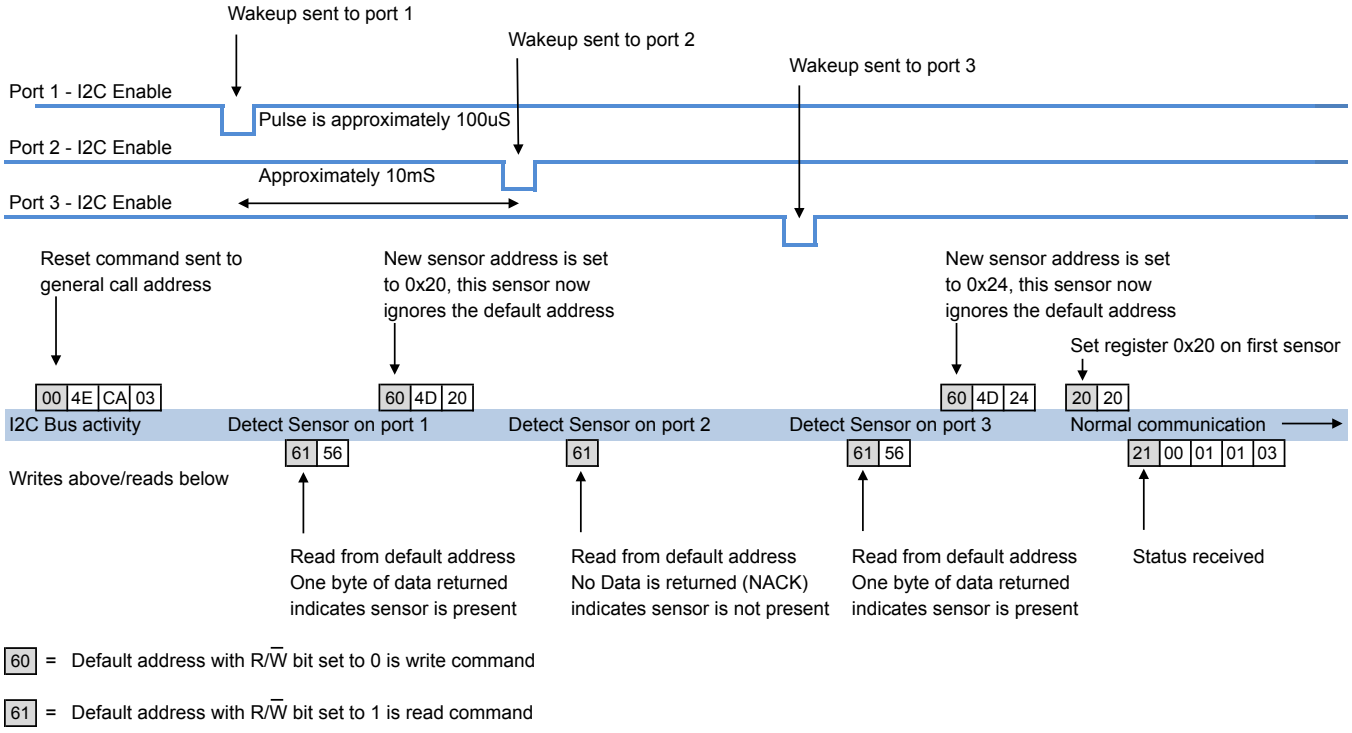


Figure 3 - Initialization sequence

3 Sensor Register Map

A VEX IQ sensor has 256 uniquely addressable locations available to store information; this is called the sensor register map. A sensor can implement this in several different ways depending on the resources it has available. The reference sensor I describe later in this document has a limited amount of SRAM and does not have room to store the whole register map. However, as some locations in the register map are not able to be written the sensor can divide the logical address space into two or more physical regions residing in different types of memory.

The register map of a typical sensor is as follows.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00	V	1	.	0	0	.	0	0	V	E	X		I	Q		
10	G	E	N	E	R	I	C		VI	PI						
20	VE	MD	ID	ST												
30																
40														ADR	DTA	CMD
50				FR	FW	FS	FE	FB	RST	MC						
60																
70																
80																
90																
A0																
B0																
C0																
D0																
E0																
F0																

System registers mapped to ROM

User registers specific to each VEX IQ sensor

System command and control registers

Mapped to NVRAM

Undefined area

3.1 System registers 0x00 – 0x23.

Implementation of the registers from address 0x00 through 0x23 is mandatory for a sensor to be recognized by the VEX IQ brain.

0x00 – 0x07: ASCII Version

The sensor firmware version in ASCII format. The value is "V1.XX.XX", where the X's are specific to the version number. These registers are read only. Writes are ignored.

0x08 – 0x0F: ASCII Vendor

The sensor vendor description in ASCII format. The value shown in the register map is "VEX IQ ". These registers are read only. Writes are ignored.

0x10 – 0x17: ASCII Device ID

The device identification for this sensor in ASCII format. The value is "GENERIC " for all user created sensors. These registers are read only. Writes are ignored.

0x18: VI (Vendor Id)

The vendor identification for this sensor, this only applies to user created sensors with a device ID (register 0x22) of 0xFF. This value will be 0x20 (a space) for all standard sensors supplied by VEX.

0x19: PI (Product Id)

The product identification for this sensor, this only applies to user created sensors with a device ID (register 0x22) of 0xFF. This value will be 0x20 (a space) for all standard sensors supplied by VEX.

0x1A – 0x1F: Reserved

The value is a series of 6 spaces (0x20). These registers are read only. Writes are ignored.

0x20: VE (Firmware Version)

The sensor firmware version number as a decimal. This is a read-only register. Writes are ignored.

0x21: MD (Device Mode)

The sensor operating mode. The value is 0x01 for all user created sensors. This is a read-only register. Writes are ignored.

0x22: ID (Device ID)

The sensor Device ID byte. The value is 0xFF = Generic sensor. This is a read-only register. Writes are ignored. VEX has defined the following sensor types. All values other than 0xFF are reserved for use by VEX.

0x02	Smart Motor.
0x03	Touch LED.
0x04	RGB Sensor.
0x05	Bumper switch.
0x06	Gyro.
0x07	Sonar (ultrasonic sensor).

0x23: Device Status

Device status. The default value is 0x00. This is a read-only register. Writes from the VEX IQ brain are ignored. However, this register can be written by the sensor firmware to reflect the sensor status. The meaning of this status byte is sensor specific.

3.2 User registers 0x24 – 0x43.

The registers in this region are sensor specific, that is, they have different functionality for each device. The sensor firmware will decide if these registers are read only or able to be written. The reference sensor implementation uses these as follows.

0x24: Sensor temperature

The value in this register gives the internal sensor temperature in degrees Celsius with 0.25 degree resolution, that is, this value is degrees C x 4. For example, the value 0x72 (144 in decimal) will be 28.5 degrees C.

0x25: Switch status

The lsb of this register will be the state of SW1 (S2 for the Launchpad) on the reference sensor board. A value of "1" indicates not pressed, a value of "0" indicates pressed.

0x28: LED brightness

The value of this register will determine the brightness of D1 (LED1 for the Launchpad). A value of 0 turns the led off, a value of 255 turns the led on, a value of 128 sets the led at half brightness.

3.3 Control Registers 0x44 – 0x5F

All registers in this region are reserved for use by VEX. A generic sensor must implement some of these registers and their associated functionality for a sensor to be compatible with the VEX IQ brain.

0x4D: Change I^2C Slave Address

Sets a new I^2C slave address. After this register is written, the slave will respond to the new slave address. Accesses to the previous slave address will no longer be serviced. This is a read-write register.

0x4E: Command Byte

Deprecated. The value written to this register has no effect. This is a read-write register.

0x4F: Command Register

Values passed to this register are interpreted as commands, the definitions of which are given below. This register is write-only.

Command = 0x03 : Reset

Reset the sensor to its power on state. This command should complete in less than 2ms.

Command = 0xC8 : Erase Non-Volatile Memory

If the sensor implements non-volatile memory, this command is a request to erase it.

Command = 0x34 : Initialize

Reinitialize the sensor and returns all registers to their default state.

0x58: Reset

Reset the sensor to its power on state. This command should complete in less than 2ms. This command has the same functionality as writing 0x03 to the command register 0x4F.

0x53: FR (flash read) – Boot loader only, included for reference

Read flash memory. This is used by the VEX IQ boot loader and does not need to be implemented by generic sensors.

0x54: FW (flash write) – Boot loader only, included for reference

Write flash memory. This is used by the VEX IQ boot loader and does not need to be implemented by generic sensors.

0x55: FS (flash status) – Boot loader only, included for reference

Get the write or erase status of the flash memory. This is used by the VEX IQ boot loader and does not need to be implemented by generic sensors.

0x56: FE (flash erase) – Boot loader only, included for reference

Erase flash memory. This is used by the VEX IQ boot loader and does not need to be implemented by generic sensors.

0x57: FB (flash write boot location) – Boot loader only, included for reference

Write the master start address. This is used by the VEX IQ boot loader and does not need to be implemented by generic sensors.

0x59: MC (boot master code) – Boot loader only, included for reference

This is used by the VEX IQ boot loader and does not need to be implemented by generic sensors.

3.4 Non-Volatile storage 0x60 – 0x9F.

This address range is reserved for non-volatile storage inside the sensor. The reference implementation maps this range to the INFOC memory bank inside the MSP430 micro-controller. When an erase command is received, this area will revert back to its un-programed condition with every location containing a value of 0xFF. Data may be written to this area that changes bits from a “1” to a “0” but not the other way around.

Implementation of this range of register addresses is not mandatory and is implementation specific.

3.5 Undefined 0xA0 – 0xFF.

This address range has not been defined and is available for the generic sensor to use.

4 Reference Sensor

To demonstrate the use of the protocol, a reference sensor was created. This is based on the design of the MSP430 launchpad evaluation board available from Texas Instruments, which can be used to run and debug the firmware.

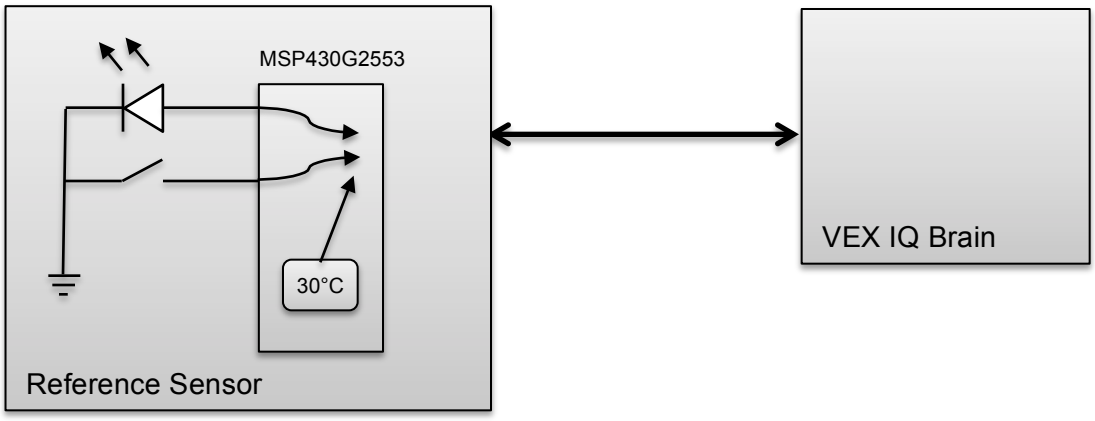


Figure 4 - Reference sensor block diagram

The schematic for the reference sensor is shown in Appendix C.

The features for the reference sensor are as follows.

- Variable brightness of the led connected to P1.0 of the MSP430 (port 1, bit 0) using pwm.
- The switch connected to P1.3 is read and its status made available.
- The internal temperature sensor is read and made available with 0.25ºC resolution.

When the MSP430 launchpad is used to simulate the reference sensor, some feature will not be available.

- The launchpad needs external power. It is not possible to power the board from the 7.2V available on the VEX IQ brain sensor port. The USB connector can be used to power the board connected to either a host computer or USB hub. The launchpad should be powered before turning on the VEX IQ brain.
- The I^2C bus will not have the pull-up or series resistors in the SDA and SCL wires. The I^2C bus and communication will still function due to pull-up resistors in the VEX IQ brain. However, signal rise and fall times will not be optimum.
- The jumper that connects LED2 to P1.6 must be removed on the launchpad, as P1.6 is the SCL line.

Connections from the Launchpad to the VEX IQ brain are as follows. The wire colors on this drawing match the colors used in the standard VEX smart cable. Care should be taken to follow the signal pin numbers rather than relying on wire colors as these may differ in third party cables.

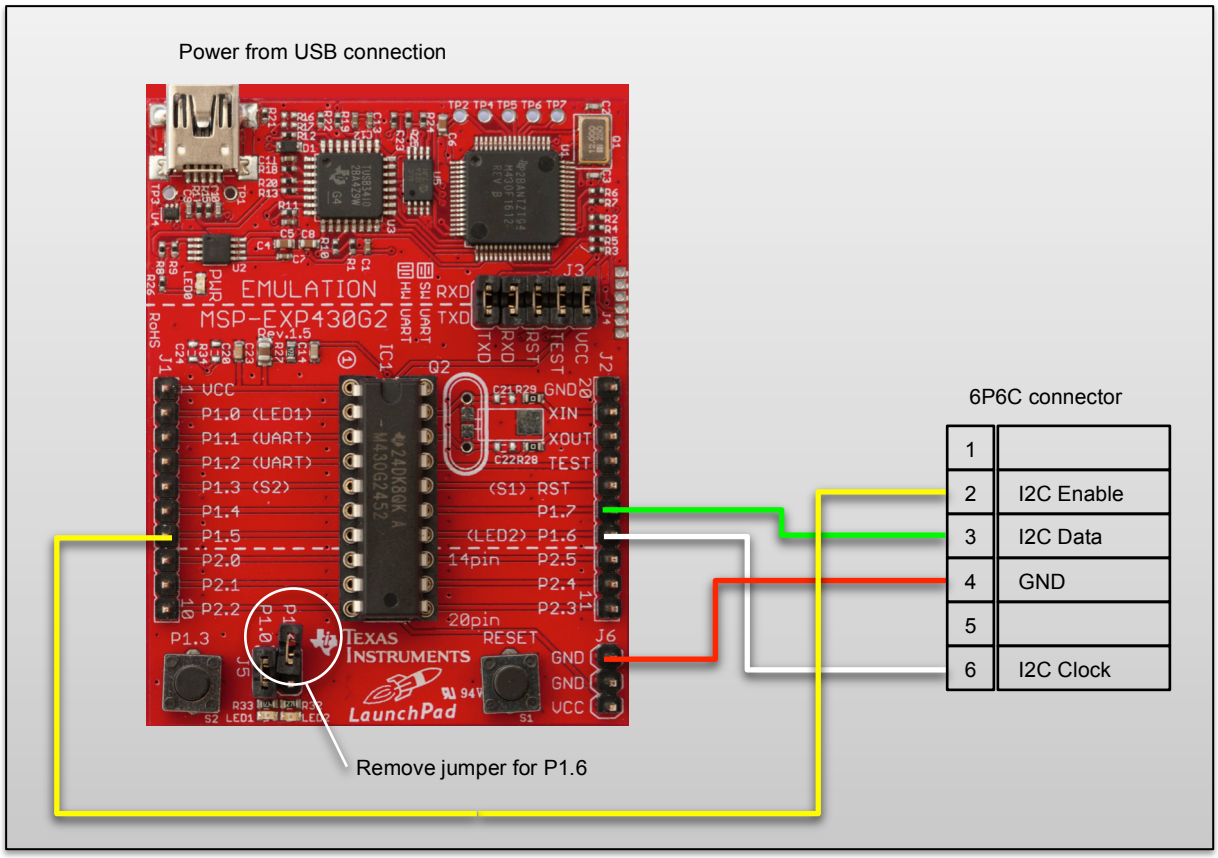


Figure 5 - Connections from the MSP430 launchpad to the VEX IQ Brain\

4.1 Sensor Firmware

The firmware for the reference sensor design is open source and released under the Apache license. It is intended to be built with the development tools available from Texas Instruments and has been tested with code composer studio V6.0.

The MSP430 launchpad is supplied with two different MSP devices, the MSP430G2553 and the MSP430G2452. The firmware can be built for either device; however, the MSP430G2553 is considered the more powerful of the two choices and contains double the amount of both flash and SRAM.

There are many different ways to structure embedded firmware to handle the multiple functions and IO operations that can be occurring. These range from a simple sequential loop, where inputs, outputs and other devices are polled in sequence, to a complex real time operating system (RTOS) with multiple threads of operation. The MSP430 is a resource-constrained processor; it has limited memory and hardware support and is not well suited to

running a RTOS. The demonstration firmware uses a small cooperative multi-tasking mechanism. A system timer generates a system tick that can be monitored by the main application loop. Tasks register themselves and request to be run after a specified number of system ticks. When the task runs it must perform any necessary processing and then return to the calling function; tasks must not block or other tasks may not be run.

Tasks may also be triggered by an event. An event is specified as the VEX IQ brain writing to a particular register, in the example firmware the brightness of the led connected to P1.0 will only be updated if the brain writes to the register controlling it.

The overall structure of the firmware and source files associated with it is as follow.

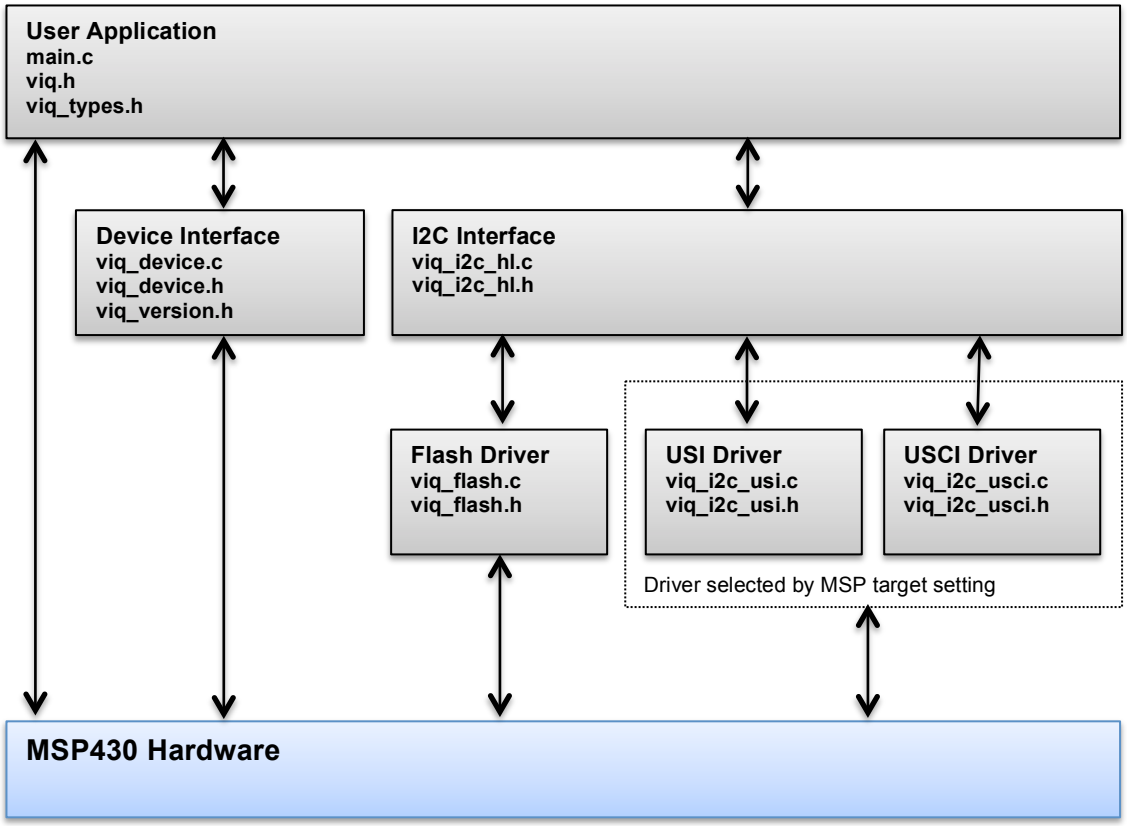


Figure 6 - firmware structure

A brief discussion of each function block follows, however, for detailed documentation refer to the source code.

User Application

main.c
viq.h
viq_types.h

This is the sensor application code; main.c contains the application entry point and the three cooperative tasks that control the three functional parts of the sensor, the led, the switch and the temperature sensor. To make the code more portable, basic types (e.g. uint16_t is a 16 bit unsigned integer) are defined in viq_types.h. viq.h allows inclusion of all necessary headers with one file.

Device Interface

viq_device.c
viq_device.h

The code in this file performs basic initialization actions such as setting processor clock speed. The watchdog timer is configured in interval mode and used as the system timer tick. Functions are available for adding the cooperative tasks and also checking to see if they should be called. The use of the system timer tick to run code allows the MSP430 to be put into a low power mode until a task has to run.

Flash Driver

viq_flash.c
viq_flash.h

The high-level interface code to the sensor NVRAM. In the case of the example code, this provides the functionality to access the INFOC area of flash memory in the MSP430 processor.

I^2C Interface

viq_i2c_hl.c
viq_i2c_hl.h

The high-level driver for the sensor I^2C interface. This insulates the main application code from the details of the low level I^2C drivers. As the sensor is a slave device and does not initiate I^2C communication, most of the I^2C code will operate without the assistance from the user application code.

The MSP430 will use one of two available serial interface blocks to implement the I^2C communications interface to the VEX IQ brain. The availability of these blocks is processor dependent, the smallest MSP processors usually have a single USI (universal serial interface) block. This interface has minimal support for I^2C beyond the ability to use a shift register to mode data onto the SDA wire using the master supplied clock. The more powerful USCI (universal serial communications interface) block has the ability to detect slave and general call addresses, detect start and stop bits, and generally remove some of the burden of implementing the I^2C protocol from the processor. Firmware to use either of these two interfaces is supplied with the reference design. The choice of which to use is controlled by the processor target setting in code composer studio.

USI Driver

viq_i2c_usi.c
viq_i2c_usi.h

The low-level I^2C driver for the USI block in the MSP430 processor.

USCI Driver

viq_i2c_usci.c
viq_i2c_usci.h

The low-level driver for the USCI block in the MSP430 processor.

4.1.1 Building the firmware

Compiling the firmware for the MSP430 launchpad evaluation board is very straightforward. Here are the necessary steps.

- Launch Code Composer Studio, create a new workspace or select an existing one.
- From the file menu select “new project..”
- Select “CCS project” under the code composer studio folder and click next.
- Give the project a name; select MSP430G2553 (or MSP430G2452) in the target pull down menu. Leave the compiler as the default (TI 4.3.1) and use an Empty Project (with main.c) as the project template. Click finish.
- A new folder will have been created in the workspace with the name of the project. Copy the source files for the reference sensor to this folder, main.c will be overwritten.
- In code composer, refresh the project (F5 or right click on the project name and select refresh).
- Build the project.

4.2 Sensor control from the robot application

All sensor control is achieved through the use of register reads and writes. ROBOTC provides four functions to assist with this.

TVexIqI2CResults vexIqGetI2CStatus(portName port)

This function returns the status of the I^2C bus. portName is an enumerated type with values ranging from **PORT1** (0) to **PORT12** (11). TVexIqI2CResults returns the enumerated type TVexIqI2CResults, this code snippet is taken from the ROBOTC header file.

```
typedef enum
{
       i2cRsltUndefined,                // Undefined
       i2cRsltNoDeviceOnPort,           // Device index not provisioned
       i2cRsltNotAvailable,             // Bus broken. Device in bootload mode. Etc
       i2cRsltBusy,                     // I2C currently busy with previous command
       i2cRsltTimedOut,                 // I2C message timed out before response.
       i2cRsltInvalidBufferStatus,      // Invalid data found in I2C record.
                                        // i.e. 'read' and 'write' counts both zero
       i2cRsltIdleAndOK,                // I2C idle. Last transaction
                                        // completed successfully
       i2cRsltIdleAndFailed             // I2c idle. Last transaction failed.
} TVexIqI2CResults;
```

This function should normally return i2cRsltIdleAndOK before a new I^2C transaction is started.

bool StartI2CDeviceBytesWrite(portName port, int nRegisterIndex, const char *pToBuffer, int nNumbOfBytes)

This function is used to write bytes to a sensor register, a simple use of this function is as follows.

```
task main()
{
    char    buf[4];

    // Write one byte to register 0x24 of the PORT1 sensor
    buf[0] = 0x55;
    StartI2CDeviceBytesWrite( PORT1, 0x24, buf, 1 );

    while(1)
        wait1Msec(10);
}
```

bool StartI2CDeviceBytesRead(portName port, int nRegisterIndex, int nNumbOfBytes)

This function is used to read bytes from a sensor register; the received data is stored internally until retrieved by the next function.

void StoreI2CDeviceBytesReadFromPortBuffer(portName port, char *pToBuffer, int nNumbOfBytes)

This function is used to retrieve the data requested by the StartI2CDeviceBytesRead function. When using these functions careful attention needs to be given to the state of the I^2C bus using vexIqGetI2CStatus. The register data that is requested will not be available instantaneously; the I^2C communications bus can transmit about 10 bytes every 1mS so long message requests may take significant time before the results are available. For this

reason I have wrapped both the read and write functionality into two high level functions that remove the complexity from the ROBOTC user code.

4.2.1 Generic Read

```
/*---------------------------------------------------------------------------*/
/** @brief Read registers from the I2C sensor                                */
/** @param[in] port the I2C port                                             */
/** @param[in] addr the sensor register to start reading from               */
/** @param[in] buf pointer to storage to save the register data             */
/** @param[in] len the number of bytes to read from the sensor              */
/*---------------------------------------------------------------------------*/

void
genericI2cRead( portName port, int addr, char *buf, int len )
{
    TVexIqI2CResults     status;
    unsigned long        timeout = nSysTime + I2C_STATUS_TIMEOUT;

    // bounds check address and length
    if( addr < 0 || addr+len > 255 )
        return;

    // make sure bus is free
    status = vexIqGetI2CStatus( port );
    while(( status == i2cRsltBusy ) && (nSysTime < timeout) )
        {
        abortTimeslice();
        status = vexIqGetI2CStatus( port );
        }

    // Under these conditions send the message
    if( (status == i2cRsltIdleAndOK) || (status == i2cRsltIdleAndFailed) ||
        (status == i2cRsltInvalidBufferStatus) || (status == i2cRsltTimedOut) )
        {
        // Send read register message
        StartI2CDeviceBytesRead( port, addr, len );

        // wait for bus to be free, message is about 10 bytes/mS
        timeout = nSysTime + (len/10) + I2C_STATUS_TIMEOUT;
        status = vexIqGetI2CStatus( port );
        while(( status == i2cRsltBusy ) && (nSysTime < timeout) )
            {
            abortTimeslice();
            status = vexIqGetI2CStatus( port );
            }

        // check status
        status = vexIqGetI2CStatus( port );
        if(status == i2cRsltIdleAndOK)
            // Get the data returned from the sensor
            StoreI2CDeviceBytesReadFromPortBuffer( port, buf, len );
        }
}
```

4.2.2 Generic write

```
/*---------------------------------------------------------------------------*/
/** @brief Write registers in the I2C sensor                                 */
/** @param[in] port the I2C port                                             */
/** @param[in] addr the sensor register to start writing to                  */
/** @param[in] buf pointer to buffer with the register data                  */
/** @param[in] len the number of bytes to write to the sensor                */
/*---------------------------------------------------------------------------*/

void
genericI2cWrite( portName port, int addr, char *buf, int len )
{
    TVexIqI2CResults     status;
    unsigned long        timeout = nSysTime + I2C_STATUS_TIMEOUT;

    // bounds check address and length
    if( addr < 0 || addr+len > 255 )
        return;

    // make sure bus is free
    status = vexIqGetI2CStatus( port );
    while(( status == i2cRsltBusy ) && (nSysTime < timeout) )
        {
        abortTimeslice();
        status = vexIqGetI2CStatus( port );
        }

    // Under these conditions send the message
    if( (status == i2cRsltIdleAndOK) || (status == i2cRsltIdleAndFailed) ||
        (status == i2cRsltInvalidBufferStatus) || (status == i2cRsltTimedOut) )
        {
        StartI2CDeviceBytesWrite( port, addr, buf, len );
        }
}
```

4.2.3 Example ROBOTC program using read and write wrappers

A simple ROBOTC program can then be written as follows (this example changes brightness of the led on reference design.)

```
#include "generic_i2c.c"

task main()
{
    char buffer[4];
    int  i = 0;

    while(1)
        {
        // Set led brightness
        buffer[0] = i;
        genericI2cWrite( PORT1, 0x28, buffer, 1 );

        // Increase brightness
        i += 4;
        wait1Msec(5);
        }
}
```

4.2.4 Detecting user sensors

ROBOTC provides a function to retrieve information about each port of the VEX IQ brain. Using this functionality we can write code to find a particular sensor type and automatically detect which port a generic sensor is connected to.

void getVexIqDeviceInfo(short nDeviceIndex, TVexIQDeviceTypes &nDeviceType, TDeviceStatus &nDeviceStatus, short &nDeviceVersion)

This function can be used in the following way to detect a generic sensor connected to the brain.

```
/*---------------------------------------------------------------------------*/
/** @brief  Find the first generic sensor installed                          */
/** @returns the port number if found else (-1)                              */
/*---------------------------------------------------------------------------*/
/**
 * @details
 *  scan all ports looking for an installed generic sensor
 */

portName
genericI2cFindFirst()
{
    TVexIQDeviceTypes   type;
    TDeviceStatus       status;
    short               ver;
    portName            index;

    // Get all device info
    for(index=PORT1;index<=PORT12;index++)
        {
        getVexIqDeviceInfo( index, type, status, ver );
        if( (char)type == vexIQ_SensorUSER )
            return((portName)index);
        }
    return((portName)-1);
}
```

Appendix A. I2C Communication

A brief overview.

The I^2C bus protocol is a serial protocol. Two wires, serial data (SDA) and serial clock (SCL), carry information between devices on the bus. A device that sends data onto the bus is defined as a transmitter and a device receiving data as a receiver. The device that controls the message is called a master. The devices that are controlled by the master are referred to as slaves. A master device that generates the serial clock (SCL), controls the bus access, and generates the START and STOP conditions that control the bus. The VEX IQ sensor operates as a slave on the I^2C bus. Within the bus specifications, a standard mode (100kHz maximum clock rate) and a fast mode (400kHz maximum clock rate) are defined. The VEX IQ sensors only work in the standard mode. Connections to the bus are made through the open-drain I/O lines.

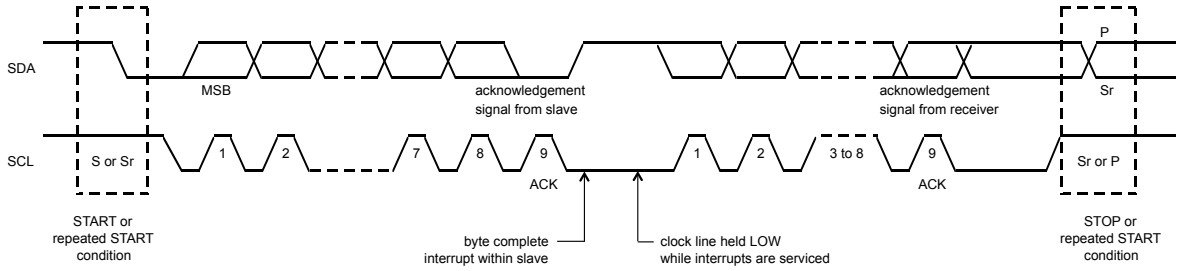


Figure 7 – I^2C Bus Protocol

The following bus protocol has been defined (Figure 7):

- Data transfer may be initiated only when the bus is not busy.
- During data transfer, the data line must remain stable whenever the clock line is HIGH. Changes in the data line while the clock line is HIGH are interpreted as control signals.

Accordingly, the following bus conditions have been defined:

Bus not busy: Both data and clock lines remain HIGH.

Start data transfer: A change in the state of the data line, from HIGH to LOW, while the clock is HIGH, defines a START condition.

Stop data transfer: A change in the state of the data line, from LOW to HIGH, while the clock line is HIGH, defines the STOP condition.

Data valid: The state of the data line represents valid data when, after a START condition, the data line is stable for the duration of the HIGH period of the clock signal. The data on the line must be changed during the LOW period of the clock signal. There is one clock pulse per bit of data.

Each data transfer is initiated with a START condition and terminated with a STOP condition. The number of data bytes transferred between START and STOP conditions are not limited, and are determined by the master device. The information is transferred byte-wise and each receiver acknowledges with a ninth bit.

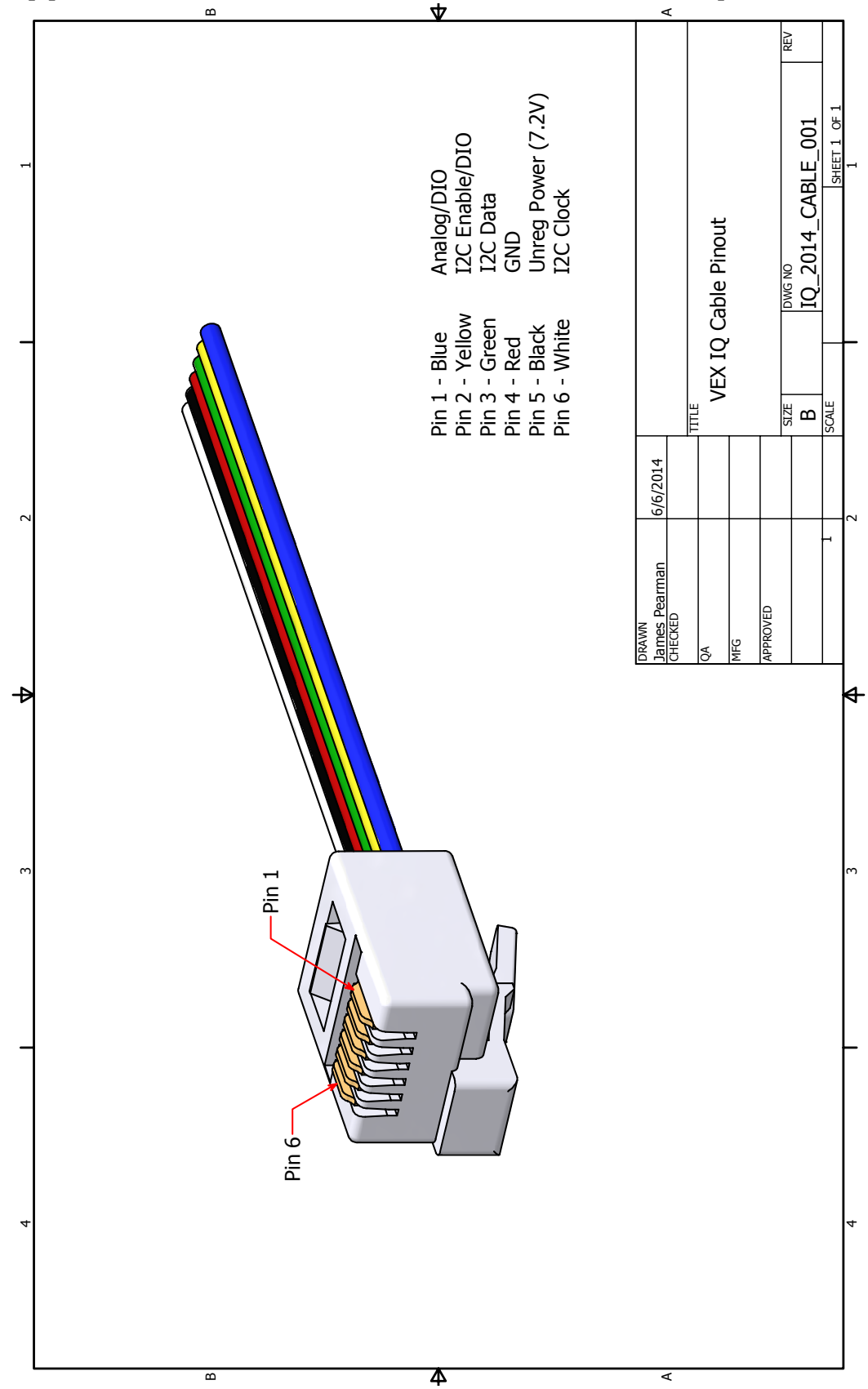
Acknowledge: Each receiving device, when addressed, is obliged to generate an acknowledge after the reception of each byte. The master device must generate an extra clock pulse that is associated with this acknowledge bit.

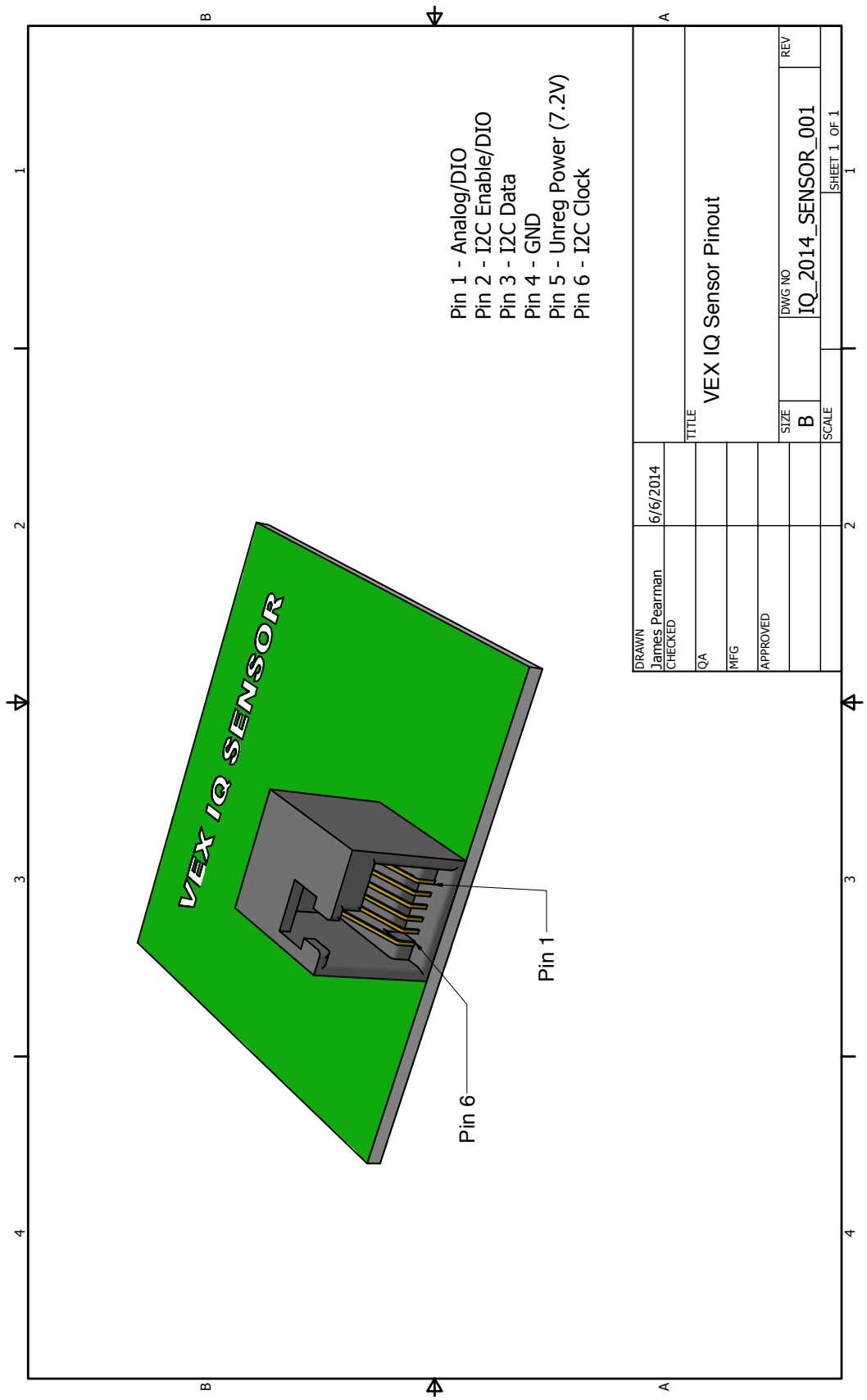
A device that acknowledges must pull down the SDA line during the acknowledge clock pulse in such a way that the SDA line is stable LOW during the HIGH period of the acknowledge-related clock pulse. Of course, setup and hold times must be taken into account. A master must signal an end of data to the slave by not generating an acknowledge bit on the last byte that has been clocked out of the slave. In this case, the slave must leave the data line HIGH to enable the master to generate the STOP condition.

For detailed information about the I^2C specification refer to the official I^2C-bus specification and user manual, which can be found on the NXP Semiconductor web site.

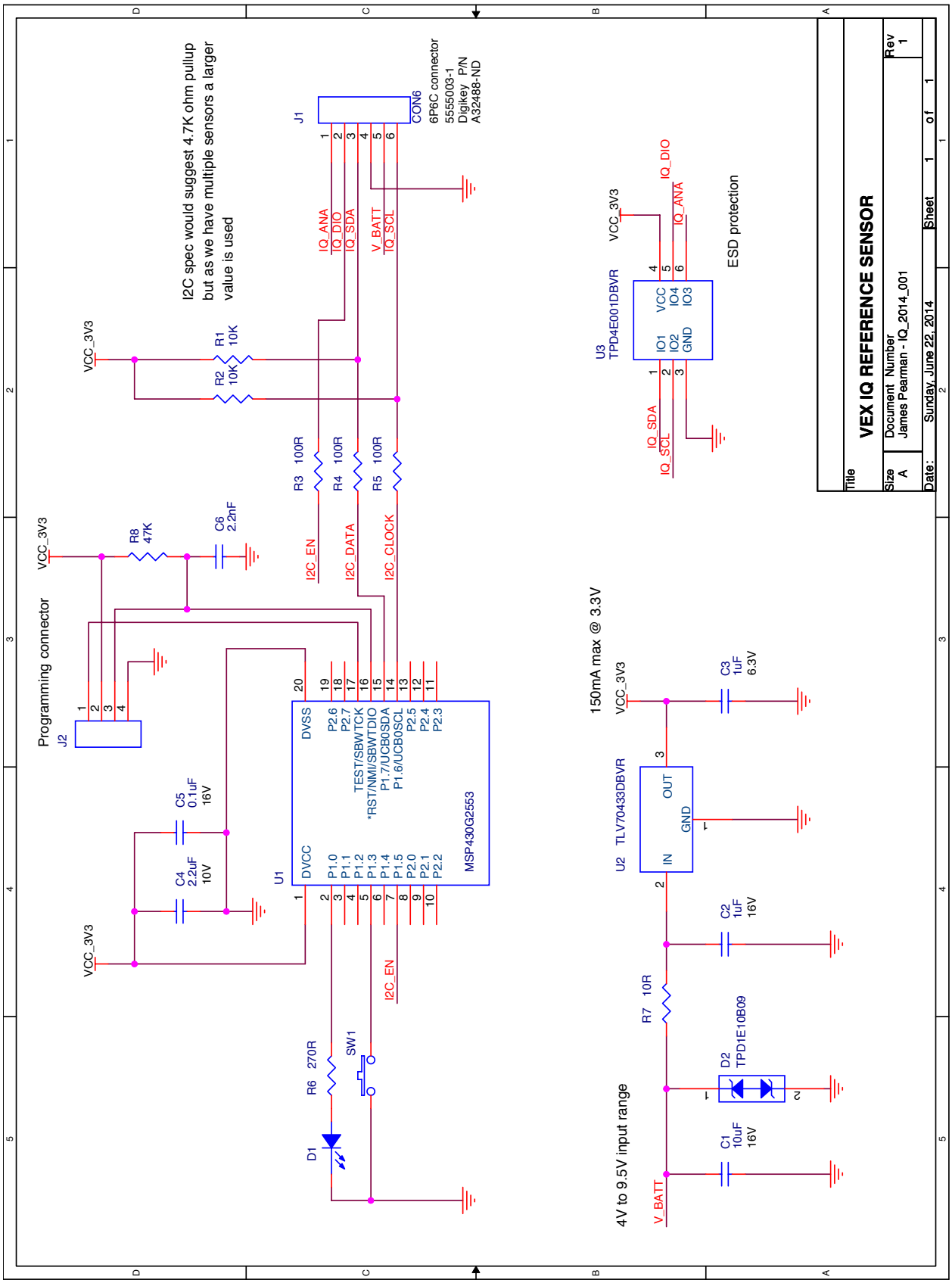
http://www.nxp.com/documents/user_manual/UM10204.pdf

Appendix B. Cable and sensor connector pin numbers




Pin 1 - Analog/DIO
Pin 2 - I2C Enable/DIO
Pin 3 - I2C Data
Pin 4 - GND
Pin 5 - Unreg Power (7.2V)
Pin 6 - I2C Clock
VEX IQ SENSOR
Pin 1
Pin 6
DRAWN James Pearman 6/6/2014
CHECKED
QA
MFG
APPROVED
TITLE
VEX IQ Sensor Pinout
SIZE B
DWG NO IQ_2014_SENSOR_001
REV
SCALE
SHEET 1 OF 1

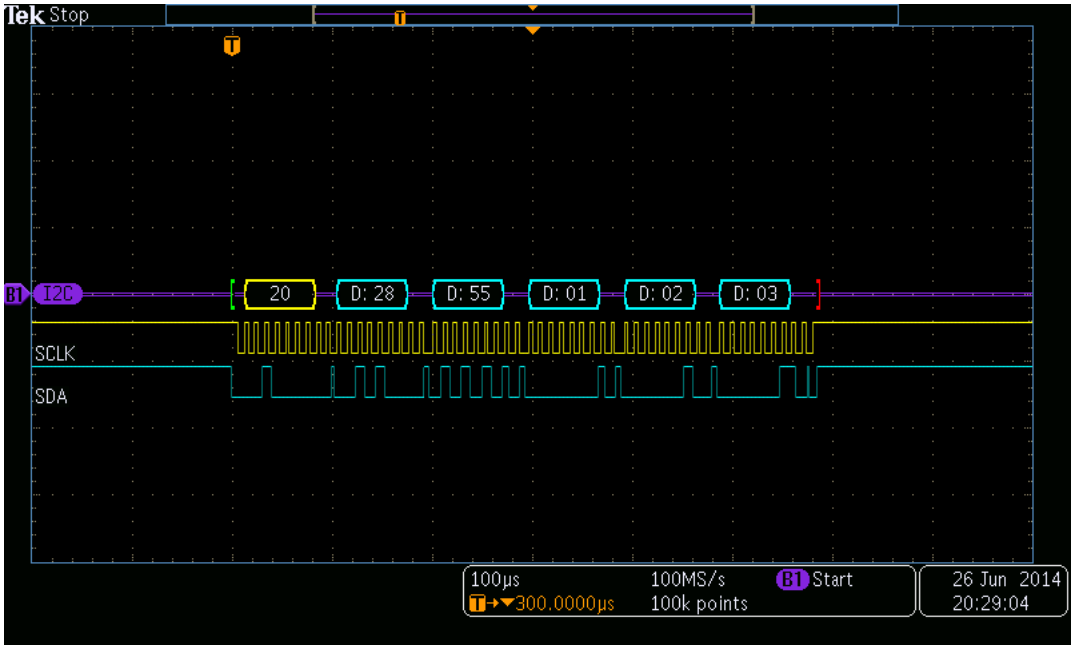
Appendix C. Reference sensor schematic



Appendix D. I2C Bus transactions

An oscilloscope trace showing a typical write transaction, in this case 4 bytes starting at register 0x28. The following ROBOTC code generated this waveform.

```
char buffer[4];
buffer[0] = 0x55; buffer[1] = 1; buffer[2] = 2; buffer[3] = 3;
genericI2cWrite( userPort, 0x28, buffer, 4 );
```



An oscilloscope trace showing a typical read transaction, in this case 4 bytes starting at register 0x28. The following ROBOTC code generated this waveform.

```
genericI2cRead( userPort, 0x28, buffer, 4 );
```