SAVONIA

OPINNÄYTETYÖ - **AMMATTIKORKEAKOULUTUTKINTO**
TEKNIIKAN JA LIIKENTEEN ALA

INTEGRATING INTELLIGENT SENSORS INTO THE EVO DISTRIBUTED INTELLIGENCE PLATFORM

T E K I J Ä : Jukka Arponen

SAVONIA-AMMATTIKORKEAKOULU OPINNÄYTETYÖ
Tiivistelmä

Koulutusala Tekniikan ja liikenteen ala	
Koulutusohjelma Sähkötekniikan koulutusohjelma	
Työn tekijä Jukka Arponen	
Työn nimi Integrating Intelligent Sensors into the EVO Distributed Intelligence Platform	
Päiväys 5.5.2014	Sivumäärä/Liitteet 48/9
Ohjaaja yliopettaja Väinö Maksimainen	
Toimeksiantaja/Yhteistyökumppani(t) Medikro Oy	

Tiivistelmä

Opinnäytetyö toteutettiin Medikro Oy:lle. Työn tavoitteena oli integroida älykkäitä sensoreita EVO-Distributed Intelligence -alustaan. Evo-alustan sensoreilla pystytään mittaamaan ympäröiviä oloja, mm. ilmanpainetta ja kosteutta. Ympäröivien olojen mittaaminen on tärkeää Medikron tuotteiden kannalta, sillä spirometrimittauksissa ympäröivät olot vaikuttavat varsinaisen mittaustuloksen matemaattiseen käsittelyyn. Työssä käytetty EVO-alusta ei itsessään ole myytävä tuote, mutta sillä suoritetaan kehitystyötä ja testaillaan uusia toiminnallisuuksia.

Työ toteutettiin Medikro Oy:n tarjoamilla laitteilla ja resursseilla. EVO-alustan prosessorin ja alustalla olevien sensoreiden välisen kommunikaation mahdollistamiseksi tuotettiin yleiset kommunikaatiofunktiot. Kommunikaatiofunktiot mahdollistivat sensoreiden luvun ja niille kirjoittamisen. Mittauksen ajoitus toteuttiin tavalla, joka mahdollisti sensoreiden kutsumisen käyttäjän asettamalla taajuudella. Lisäksi alustan virrankäyttöä suunniteltiin ohjelmallisesti. EVO-alustalla olevan litiumpatterin lataus ohjelmoitiin tapahtumaan hallitusti ja turvallisesti.

Työn tuloksena saatiin toimivat alimmat tason kommunikaatiofunktiot. I^{2}C-väylän kautta tapahtuva datansiirto toimi luotettavasti ja hallitusti. Isäntänä toimiva mikroprosessori kykenee lukemaan ja kirjoittamaan orjina toimiville sensoreille. Tuotetun ohjelmiston rakenne on selkeä ja toiminnot modulaarisia. Lisäksi virranhallinnan ohjaaminen ohjelmallisesti on mahdollista.

Avainsanat
EVO, I^{2}C, distributed, intelligence, ambient, measurement

SAVONIA UNIVERSITY OF APPLIED SCIENCES

THESIS
Abstract

Field of Study Technology, Communication and Transport			
Degree Programme Degree Programme in Electrical Engineering			
Author Jukka Arponen			
Title of Thesis Integrating Intelligent Sensors into the EVO Distributed Intelligence Platform			
Date	05 May 2014	Pages/Appendices	48/9
Supervisor(s) Mr. Väinö Maksimainen, Principal Lecturer			
Client Organisation /Partners Medikro Oy			

Abstract

The thesis was completed for a company called Medikro Oy. The goal of this thesis was to integrate intelligent sensors into the EVO distributed intelligence platform. The EVO platform contains sensors for ambient measurement. The ambient sensors can measure atmospheric pressure and temperature among other quantities. The measurement of these ambient conditions is important for the functionality of the Medikro spirometers. This is caused by the fact that ambient conditions have an effect on the spirometric measurement, and it has to be included in the mathematical handling of the spirometer data. The EVO platform in itself is not a retail product. It is used as a development platform for testing and planning new features.

The thesis was conducted with the resources and materials provided by Medikro Oy. The communication between the processor and the sensors was made possible by generating general communication functions. These functions could be used to initialize sensors as well as requesting them to measure their respective quantities. The frequency of these measurements was implemented in such a way that it allowed the user to set each sensor to measure at a specific frequency. Another important part of the thesis was power management. The recharging of the lithium battery was controlled programmatically. Safety and control were key features in recharging the battery, as overcharging and overvoltage can produce dangerous situations with lithium batteries.

As a result of this thesis, functioning communication between the master processor and slave devices was accomplished. I^2C communication functions were created, and the data transfer worked reliably and in a controlled fashion. The master processor is capable of writing and reading data from the slave devices. All of the produced code was structurally clear, and all of the functions modular. Programming the power management was also enabled.

Keywords
EVO, I^2C, distributed, intelligence, ambient, measurement

FOREWORD

This thesis was started in the early months of 2014. It was commissioned by Medikro Oy and my primary contact in Medikro was Olli Pohjolainen. The intent of this thesis was to integrate the use of intelligent sensors into an existing EVO platform.

I would like to thank my supervising teacher, Väinö Maksimainen, and Medikros Research and Development Director Olli Pohjolainen for their guidance during the creation of this thesis. I would also like to extend my gratitude to the people of Medikro Oy for this great opportunity. I also want to say a special thank you to my friends and family for their ongoing support during the creation of this thesis.

The creation of this thesis was a valuable learning experience and I truly believe it was an extremely beneficial experience for my future in this line of work.

Kuopio 05 May 2014
Jukka Arponen

CONTENTS

ABBREVIATIONS AND DEFINITIONS

EVO = A project in development by Medikro Oy. EVO is a programmable platform containing various sensors and devices.
PCB = Printed circuit board. Supports and connects electrical components by electricity conducting tracks.
MSP430 = A processor type manufactured by Texas Instruments.
MAX8934GETI+ = Power regulator unit.
Jumper = A connector for connecting nearby pins together to alter the functionality of the device.
SD-card = Secure Digital card. A non-volatile external memory card.
Master = A device that is in charge of making decisions and giving orders. Often it's the microprocessor.
Slave = A device that listens to and obeys the Masters command. In this application, the sensors are slaves.
NTC-thermistor = Negative temperature coefficient thermistor. A device used to measure temperature.
Peripheral = A supplementary device connected to a master device in order to achieve increased functionality.
USCI = Universal Serial Communication Interface. A device that enables the serial exchange of data between a microprocessor and peripheral devices.
DCO = Digitally-Controlled Oscillator.
RTC = Real-time clock.
PWM = Pulse-width modulation.
RAM = Random access memory.
FRAM = Ferroelectric RAM. Random access memory that is retained even if power is turned off.
SRAM = Static RAM. Data can be retained even if power is turned off, but it will be lost eventually if power is off for extended amount of time.
CPU = Central processing unit.
IDE = Integrated Development Environment. A platform that facilitates the development of software.
CCS = Code Composer Studio. A commercial IDE by Texas Instruments.
I^2C = Inter Integrated Circuit. A communication bus that is widely used in microprocessor applications.
SDA = Serial Data Line.
SCL = Serial Clock Line.
SPI = Serial Peripheral Interface. A communication bus that is widely used in microprocessor applications.
HAL = Hardware Abstraction Layer.
GND = Ground. Term used in electrical engineering to signify a zero reference point.
DNC = Do Not Care. An insignificant bit.
Input = A signal going into a device.
Output = A signal a device sends out.

Interrupt = Signal for an event that requires immediate attention.

Ack = Acknowledge. An affirmation in a communication.

Nack = Not-Acknowledge. A negative response in a communication.

TXIFG = Transmit interrupt flag. An interrupt for a sent message.

RXIFG = Receive interrupt flag. An interrupt for an incoming message.

Buffer = Temporary storage for data while it's being moved.

UCB1RXBUF = A buffer on the MSP430 used for sending and receiving data.

UCB1CTL1 = A register on the MSP430 that controls some functionalities in the MSP430.

Register = Small storage for holding a specific value.

UCB1I2CSA = Register that contains the address of the slave device the master is communicating with.

Queue[100] = Array of function pointers.

pAdd = Position in Queue where we want to add new data.

PCurrent = Position in Queue that stores the next command we need to execute.

I/O = Input/Output.

SSC = Standard Accuracy Silicon Ceramic.

RH = Relative Humidity.

INT1, INT2 = Output pins from the accelerometer. Interrupts are mapped to these.

GS_INT1, GS_INT2 = Pins on the processor that are directly connected to INT1 and INT2 respectively.

C = Programming language.

C++ = Programming language.

Port = Software construct used as a communications endpoint in host system.

A, mA = Ampere, a unit of current.

V = Volt, a unit of voltage.

FIFO = First In, First Out.

Callback function = a function that is called after completion of another function.

1 INTRODUCTION

Medikro Oy is a company that manufactures spirometers and various other instruments for respiratory measurements. A spirometer measurement can be used to determine the volume of a person's lungs as well as exhaling speed. This information can be used to diagnose various lung related diseases as well as measure the functionality of a patient's lungs. The spirometer measurement is susceptible to ambient conditions, so it is important to include ambient conditions when calculating the output data. These ambient conditions include the temperature and humidity of the surrounding air as well as atmospheric pressure.

The EVO Distributed Intelligence platform contains an MSP430 microprocessor that controls all of the platform's functions. Embedded on the device are various sensors that are used to measure the ambient conditions at controllable intervals. These sensors include a barometer, humidity sensor and an accelerometer. These sensors can measure atmospheric pressure, air humidity and temperature, as well as the orientation of the platform. The accelerometer also offers much more functionality, including the detection of various taps of the device. Background materials were provided by Medikro Oy, located in the internal network of Medikro Oy. Various datasheets were provided by equipment manufacturers. Datasheets can be found in references.

The development of the EVO platform is being done with the assistance of students. The project is ongoing and will remain in development after the completion of this thesis. The communication of the master processor and the slave devices is vitally important for the functionality of the platform. The main focus of this thesis is to facilitate I^{2}C communication between the master and the slaves fitted on the PCB board. Programming the power settings will also be important during the making of this thesis.

2 EVO DISTRIBUTED INTELLIGENCE PLATFORM

The EVO Distributed Intelligence platform is an ongoing project by Medikro Oy. The EVO platform is a PCB board that is controlled by a MSP430 processor. Various components and connections are embedded on the board. The microprocessor can programmatically control the functionality of the board. It is also possible to externally control some of the functions by various jumpers. For instance, the device can be set to reprogramming state by connecting two pins with a jumper. This function is especially useful in situations where the device is unresponsive to communication attempts and needs to be set into reprogramming state manually. In normal operation the device can be set into reprogramming state by sending a command to via the USB connection.

The EVO platform is equipped with a serial USB communication line. The device can use this connection to communicate with a PC, for example. If no other power source is available, the device can also get its operating voltage from this USB line. The platform has also other ways of communicating with external devices. A Bluetooth communication is available, as the platform has a Bluegiga BLE113 Bluetooth chip embedded in it. The Bluetooth connection requires additional work and will not be implemented during the making of this thesis. A micro SD card slot is also available, if the data needs to be stored on the platform as opposed to sending it to another device. (Medikro Oy, 2013)

The platform also contains various other functional parts, such as eight pushbuttons. The buttons can be used for external control of the device, but they are not as of yet implemented in any way. Three USB connections are also available for connecting external devices into the platform. Currently one of them is reserved for spirometer measurements. The platform contains several methods for powering the platform and this will be discussed in more detail in Chapter 2. The EVO platform is not a product in retail. It is used for developing new products. The Evo platform can be used for ambient measurements. Ambient measurement means the measurement of various surrounding conditions, such as the temperature and pressure of the surrounding air. Medikro produces spirometers that are used to measure patients' lung capacity among other quantities. The raw signal of the flow transducer is very noisy and it has to be mathematically filtered. The ambient conditions affect the result of the flow measurement, and this is the reason the ambient measurements are made. They are taken into account as the data is being calculated to improve the accuracy of the results. Different ambient measurements have different importance regarding the spirometry values. In addition some of the conditions are more susceptible to change. For example, air pressure is not changing as fast as temperature, as the pressure of the surrounding air will not fluctuate as much. The sensors are used to measure specific quantities and different sensors function with varying speeds. It is important to take this into account when designing a measurement system. After the prioritization of the measurements is carefully considered, each measurement is done with an appropriate frequency. (Medikro Oy, 2013)

The power usage of the EVO platform is quite versatile and user definable. The platform has three ways of getting power and even combinations of these methods are possible. The idea behind this is to ensure functionality as well as making the platform as versatile as possible. Having multiple ways of powering the device brings options and flexibility to the user as well as the platform. If more than one way of powering is present the system is capable of intelligently selecting which one to use. During the making of this thesis the primary means of powering the device was from the USB power source. The selection of the power source in the EVO platform is done by a jumper. Setting Jumper 5 selects between USB and DC power. Programming the power features will be further discussed in Chapter 5.6.1

The EVO platform is equipped with a MAX8934GETI+ power regulator that handles powering the entire board. If no power source is connected the device runs on battery, if possible. If more power is present than needed by the functions of the board, the regulator will charge the battery. The regulator is also capable of protecting the system from various over voltages and currents. The regulator is intelligent enough to be able to select the most useful power option if more than one is provided. Some of the inputs to the regulator are made with hardware, for example with resistors and capacitors. Other inputs can be set programmatically. This means that there exists a possibility to change some power options by coding different outputs into the power output pins. (Maxim Integrated, 2010).

The USB connection functions as a communication method between the EVO platform and an external device. Often the platform is connected to a PC in order to send the measured data to be stored. Having a PC connected to the platform is also vitally important for developing and uploading logic and programming into the platform. The USB connection is also capable of providing enough power to keep the platform running. Removing the USB power cords powers down the device immediately unless other way of acquiring power is present. The USB connection is capable of providing the system with 5 volts and a current of 500 mA. The voltage was measured to be 4.6 volts during the writing of Chapter 5.6. The 2.5 Watts of power that the USB is capable of providing is quite limited.

A direct connection to a power supply is also possible. A DC power connection is capable of providing more power than the USB power connection but it introduces new challenges. Overheating is a possibility and defending the delicate circuit from over voltages and currents becomes vitally important. The EVO system includes an NTC-temperature sensor that is designed to detect overheating issues and inform the regulator if such a threat is detected. A DC connection is more than enough to power the system and recharge the battery at the same time. When the battery is fully charged the regulator adjusts its functionality to prevent battery overheating. A DC connection provides up to 2 A of current and 5 volts to the regulator. This means a DC connection is capable of up to 10 Watts of power.

Wireless power transmission has been around for quite some time now. One early application was used in recharging of electric toothbrushes. The low amount of power that can be sent this way has proved to be problematic. In the EVO platform there exists a BQ510XXRHL wireless power receiver module. It can be used to power the EVO platform, but it is not in use at the moment. This is due to the fact that this device requires much more additional work before it can be implemented.

The battery used by the EVO platform is a Lithium-Ion battery. The battery can be recharged when other means of powering the device are present. If no other means of powering are present the regulator attempts to power the system from the battery. This of course happens if a battery is present at all. Lithium batteries can explode due to various faults and monitoring of the battery must be extremely strict. There are three special cases that need to be specially monitored. Overtemperature of the lithium-ion battery is the most important one. If an overtemperature of the battery is not detected, the battery will swell up and in the worst case scenario it will explode. Similar reactions can happen if the battery is overcharged. Overcharging happens if a battery keeps getting charged after it has reached a state of full charge. Last state that needs to be monitored happens when the regulator signals out a general fault. The battery used was Keeppower 14500. It has a charge rate of 800 mAh and its voltage is 3.7 V.

3 MSP430 PROCESSOR

The EVO platform uses an MSP430 microcontroller as its processor. The MSP430 is made by the US electronics company Texas Instruments. MSP430 functions with 16-bits and it is specifically designed for ultra-low power applications. The combination of the low cost and power consumption makes the MSP430 very appealing for embedded applications. The MSP430 can operate with frequencies up to 25 MHz. The specific processor used in the EVO platform is MSP430F5659. Key features of the MSP430 are highlighted in Table 1.

TABLE 1 MSP430 statistics

PROCESSOR	**MSP430F5695**
Program memory	512 kB
SRAM	64 kB
I/O pins	74
Timers	4
Watchdog timer	Included
Power management module	Included
Direct memory access	6 channels
ADC	16 channel ADC12 A
DAC	Included
Additional features	USB
USCI	Channel A: UART, LIN, IrDA, SPI Channel B: I^{2}C, SPI

The MSP430 is designed to be as efficient with its power usage as possible. The processor is designed with several low power states, which the processor enters whenever appropriate. The ability to enable and disable various clocks and oscillators is very important in achieving an efficient power usage. This means that only the required clocks are enabled, and once they have completed their functions they are once again disabled to conserve energy. (Texas Instruments, 2014)

The use of intelligent sensors is the main theme in this thesis. When an intelligent and autonomous device is not used or commanded by the processor, it automatically enters a low power state. The MSP430 is designed around the idea that the connected peripherals are efficient with their power usage. This plays a key role in achieving efficient power usage in embedded applications. (Texas Instruments, 2014)

MSP430 features a digitally-controlled oscillator or DCO. The DCO can be programmatically set by the programmer and it can reach up to 1 μs in start-up time. This means that the MSP430 can stay in low-power state as long as possible. Staying in low-power state as long as possible further improves power efficiency in embedded applications. (Texas Instruments, 2014)

The MSP430 has real-time clock or RTC modules in it. The RTC modules enable the processor to keep up real-time and RTC_A module even provides some basic calendar functions, such as keeping up seconds, minutes, hours, days, weeks, months and years in real time. The RTC modules can also be reconfigured to be used as a general-purpose counter. If used as a general purpose counter, the RTC modules can also be used as a timed interrupt. The MSP430 also contains other timers, such as Timer_A, that can be used for captures/compares, PWM outputs or interval timings. (Texas Instruments, 2014)

The newer models of the MSP430 family that contain the FR-code use FRAM type memory. This is a ferroelectric type of RAM that stores data, even if the device is powered off. MSP430F5695 contains regular SRAM that is used in applications. The MSP430 family uses a direct memory access controller that is capable of memory transfer independent from the CPU. This functionality is not only power-efficient but fast as well. (Texas Instruments, 2014)

Texas Instruments offers a variety of IDEs to develop code for the MSP430 family. Free versions of the commercial environments are available, but they are usually code-limited. It's also possible to develop software in a free and open sourced platform as well. The most common IDEs are Code Composer Studio, IAR Embedded Workbench, Energia and MSPGCC. MSPGCC and Energia are open sourced and free. CCS and IAR on the other hand are commercial IDEs. The IDE used in this thesis is Code Composer Studio and it will be further discussed in Chapter 7.1.

4 DEVICE COMMUNICATION

I^2C is an abbreviation for Inter Integrated Circuit. I^2C is a communication bus and it is used to connect peripheral devices to a master device. It can also be used in other applications, such as different control architectures. I^2C is designed to utilize the similarities between different device designs. Usually every system includes some kind of control device, a microcontroller for example. A system usually also includes some general purpose drivers, for example LCD drivers and I/O ports. Many systems also include application oriented circuits, such as sensors. The goal of the I^2C is to simplify the physical circuitry while maximizing efficiency. The goal in this thesis was to make the lowest level functions, such as read and write. These were then used in communication between a master and the slave devices. Functions for I^2C communications were produced and tested during the writing of this thesis. (NXP Semiconductors, 2012).

The I^2C bus only requires two lines for communication. These lines are called SDA and SCL. SDA is an abbreviation of Serial Data. SDA is used for the transfer of data. SCL is an abbreviation of Serial Clock. SCL provides a clock pulse to the bus. Each slave device, which is connected to the I^2C, has a unique address. All slave devices are connected to the same SDA and SCL lines. The slave devices can recognize their own address, if it is being transmitted through the data line. The Master-Slave relationship in the system is clear, and a slave device can only use the data line when a master allows it. Having only two lines is very beneficial for equipment manufacturers. For example, SPI communication uses four lines. The amount of connections required for connecting to peripheral devices complicates the designing of ICBs and increases the cost. The interface protocol of I^2C is integrated into the chips themselves which simplifies the design even more. Requiring only two lines for all connected devices makes the I^2C very easy to implement. (Philips Semiconductors, 2003).

5 MICROPROCESSOR PROGRAMMING

5.1 Structure

The structure of the application was split into three layers. This is done in order to simplify the understandability of the code as well as providing clear structure to the entirety of the application. The intention is that the layers do not overlap and changing one layer does not affect the others. Each layer is responsible for a specific area. The point of making a structure like this is to make it easier for programmers to build their applications. It is also easier to understand the code if it is structured well. For instance, once the lowest level functions such as the communication functions are made, it's possible to make use of them without having to know how they work in detail. When a new features is added into the system, only the layer that the changes occur in needs to be changed.

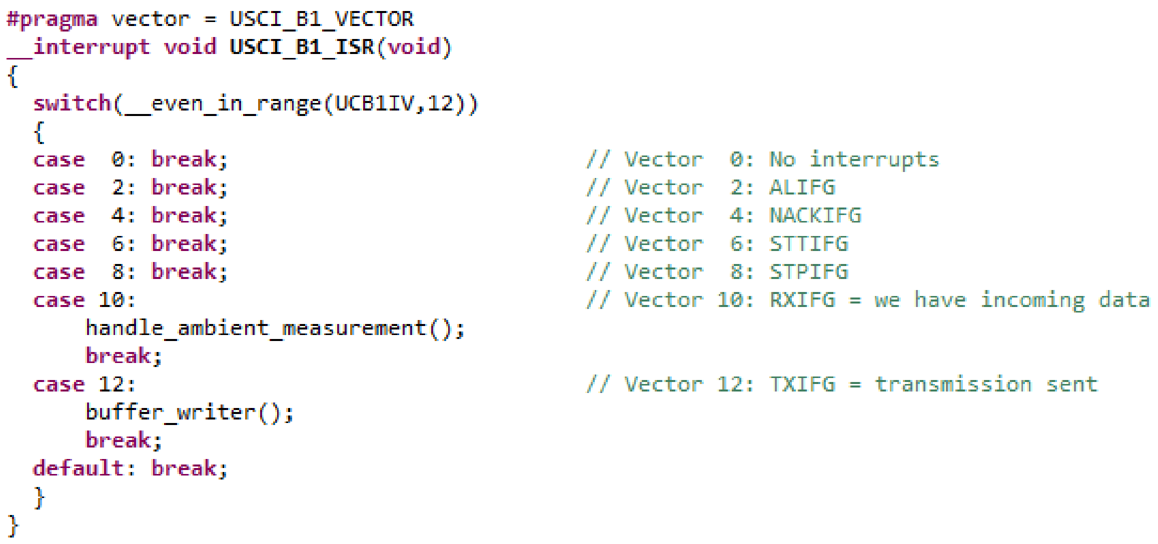
Application layer is the highest level of structure. It is the most visible to the users and all applications are located in this layer. This is the layer that the user sees.

The device layer contains specific information about the connected devices. It includes the addresses of the devices as well as some device specific details. For instance, an ambient measurement may be packed into 2-6 bytes depending on the sensor. It is important to know how many bytes need to be read from a sensor in order to receive all the data correctly. Naturally, the incoming data also needs to be handled in a device specific way. Different sensors have status bits and DNC bits that need to be taken into account. Each device was made to have its own file that contains all information needed to operate that device. This makes changing or adding devices easier and more modular. All device files are organized in a separate device folder for increased clarity.

Hardware abstraction layer is the part of the code that functions between the software and the hardware. It provides a device driver interface allowing a program to communicate with the hardware. In this thesis the hardware abstraction layer contains the functions for communicating between the master and a slave. These functions are made in such a way that their operation is not dependant on knowing specifics about the devices. The I^{2}C communication only sends and receives bytes of data, it does not need to know anything about the devices themselves.

5.2 Interrupts

An interrupt is a signal that requires immediate attention. An interrupt driven system means that the system reacts to interrupt events as they appear. This is extremely useful because continuous polling of pins or sensors can cause overheating and is not power efficient. Continuous polling would also take time from the processor, leaving fewer resources to other processes. The EVO platform is an interrupt driven system. There are several things that work based on different interrupts. The ambient measurements are made with regular intervals that are generated by timer interrupts. The I^{2}C communication utilises a variety of interrupts, including interrupts for sending and receiving messages. Interrupts in the communication are handled according to Figure 1. Incoming data is handled in a function called handle_ambient_measurement. This interrupt is triggered when data is coming from a sensor. Sending a transmission send triggers a buffer_writer function.



```
#pragma vector = USCI_B1_VECTOR
__interrupt void USCI_B1_ISR(void)
{
  switch(__even_in_range(UCB1IV,12))
  {
  case  0: break;                          // Vector  0: No interrupts
  case  2: break;                          // Vector  2: ALIFG
  case  4: break;                          // Vector  4: NACKIFG
  case  6: break;                          // Vector  6: STTIFG
  case  8: break;                          // Vector  8: STPIFG
  case 10:                                 // Vector 10: RXIFG = we have incoming data
      handle_ambient_measurement();
      break;
  case 12:                                 // Vector 12: TXIFG = transmission sent
      buffer_writer();
      break;
  default: break;
  }
}
```

FIGURE 1 I2C communication interrupts (Arponen 2014.04.04)

5.3 Basic low level functions

Basic low level functions are needed for communicating between a master and a slave. These functions are called read and write. The master processor needs to be able command the sensors as well as listen to their output. These functions generated during the making of this thesis can be used to communicate to any device connected into the I^{2}C. I2C_dataWrite can be used to write values to specific registers on a specific device. Some devices require most of their registers to be initialized before they function as intended. Other devices are simple and do not require additional initialization at all. More information about the requirements and specifics of the ambient measurement sensors can be found in Chapter 6. A structure is used to store valuable information while communicating. The structure is displayed in Figure 2. Address is used to store the address of the device we are communicating with. Data is the information that we are sending, and datasize tell how much data is to be sent. Completed_func is a callback function that needs to be called after all the bytes are read. Setting justwrite to 1 means we're writing to a register of a device, a value of 0 is used when requesting data.

```
struct {
    unsigned char address;
    unsigned char *data;
    char datasize;
    void (*completed_func)(unsigned char*);
    char justwrite;
} SendMem;
```

FIGURE 2 Store structure (Arponen 2014.04.04)

5.3.1 I2C_dataWrite

I2C_dataWrite is used by the master to write messages into the I^2C. All writes begin by sending the slave device address to the I^2C. All connected slave devices check if this address matches their own address, and if it does they respond. A functioning device responds by Ack. If a programmer wants to write a value into a specific register on a specific sensor, the communication follows the logic displayed in Figure 3**.** Figure 4 and Figure 5 show how the communication is handled in the code.

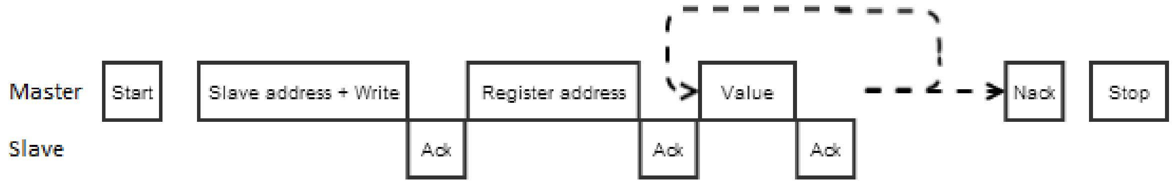
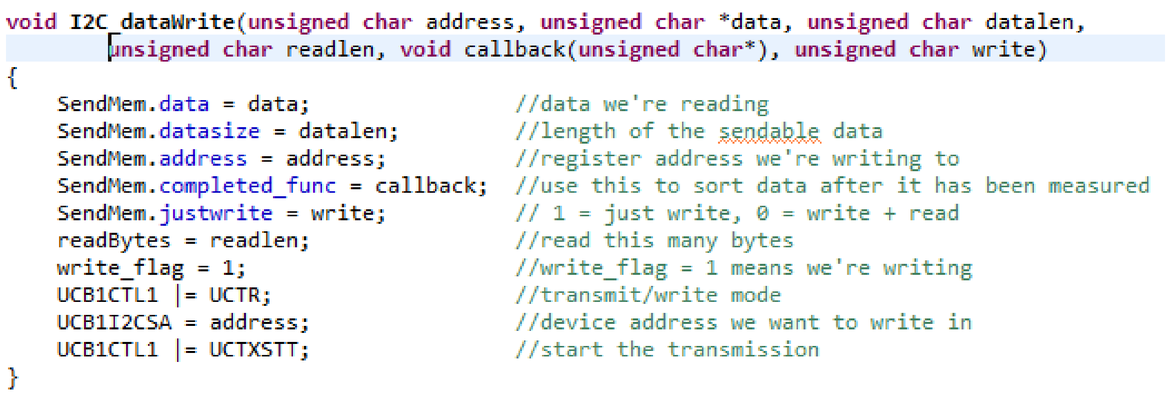


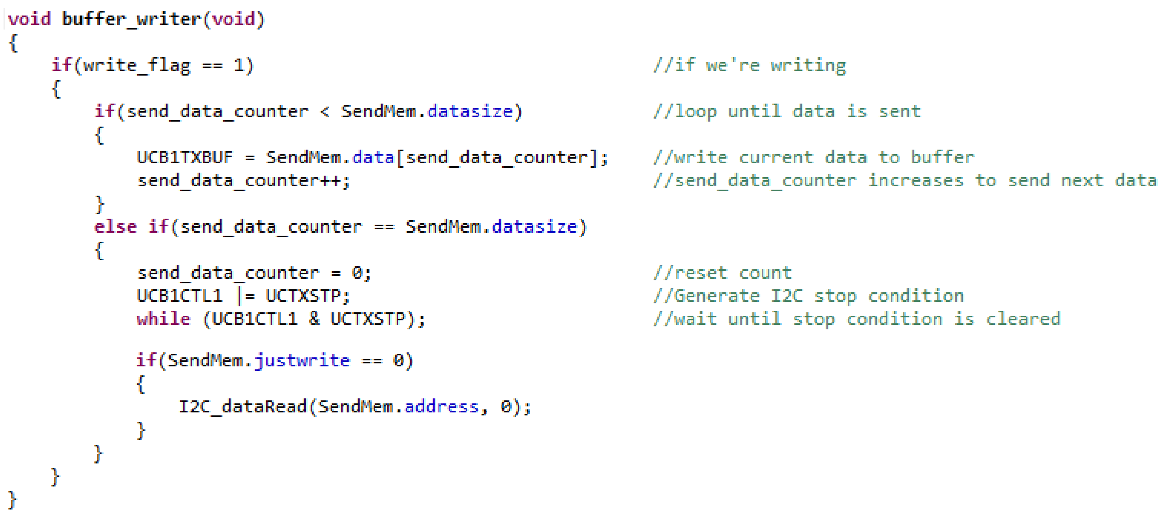
FIGURE 3 I2C_dataWrite logic (Arponen 2014.04.04)

The master begins by writing the slave address to the UCB1I2CSA register. The master also sets the write bit in UCB1CTL1 register to 1. After this, the master starts the transmission. A slave device in that address responds with an Ack. This triggers a TXIFG interrupt. The TXIFG interrupt is handled in an interrupt handler function. If the master wants to send additional info into to the slave, it can write it to the UCB1RXBUF buffer during the TXIFG interrupt. In this example, the master writes the address of a register on the device into the buffer. The buffer is then read by the slave device. The slave now knows that the following value is to be written into that register. This is acknowledged by the device with an Ack and a new TXIFG interrupt is generated. The value can now be written into the UCB1RXBUF buffer. The value is transmitted on to the slave, and the slave stores this value in the register that was designated before. Once again, the device responds with an Ack and another TXIFG interrupt is generated. Writing a new value is also possible, and it is stored in the subsequent register following the register originally specified. This functionality makes it possible to write a varying number of values into the registers of the device. The communication is stopped when the master sends out a Nack followed by a Stop. A device specific example of initializing a slave device with register values can be found in Chapter 6. In the code I2C_dataWrite is made in accordance with Figure 4. Figure 5 displays how the TXIFG interrupt is handled. If a device in the designated address is not found, Ack is never received. This means that the communication will not progress any further. This could potentially happen in a situation in which the sensor is damaged or disconnected.



```c
void I2C_dataWrite(unsigned char address, unsigned char *data, unsigned char datalen,
        unsigned char readlen, void callback(unsigned char*), unsigned char write)
{
    SendMem.data = data;                    //data we're reading
    SendMem.datasize = datalen;             //length of the sendable data
    SendMem.address = address;              //register address we're writing to
    SendMem.completed_func = callback;      //use this to sort data after it has been measured
    SendMem.justwrite = write;              // 1 = just write, 0 = write + read
    readBytes = readlen;                    //read this many bytes
    write_flag = 1;                         //write_flag = 1 means we're writing
    UCB1CTL1 |= UCTR;                       //transmit/write mode
    UCB1I2CSA = address;                    //device address we want to write in
    UCB1CTL1 |= UCTXSTT;                    //start the transmission
}
```

FIGURE 4 I2C_dataWrite code (Arponen 2014.04.04)



```c
void buffer_writer(void)
{
    if(write_flag == 1)                                     //if we're writing
    {
        if(send_data_counter < SendMem.datasize)            //loop until data is sent
        {
            UCB1TXBUF = SendMem.data[send_data_counter];    //write current data to buffer
            send_data_counter++;                            //send_data_counter increases to send next data
        }
        else if(send_data_counter == SendMem.datasize)
        {
            send_data_counter = 0;                          //reset count
            UCB1CTL1 |= UCTXSTP;                            //Generate I2C stop condition
            while (UCB1CTL1 & UCTXSTP);                     //wait until stop condition is cleared

            if(SendMem.justwrite == 0)
            {
                I2C_dataRead(SendMem.address, 0);
            }
        }
    }
}
```

FIGURE 5 TXIFG interrupt handler (Arponen 2014.04.04)

TABLE 2 I2C_dataWrite parameters

Parameter	Description
Address	This is the address we want to write to. Mandatory.
Data	The data we want to write. Optional.
Datalen	The length of the data we are writing. Optional.
Readlen	Amount of bytes we want to read. Only needed when requesting data.
Callback function	Function used to handle data in a device specific way. Implemented by the use of a function pointer.
Justwrite	Parameter that signifies whether dataWrite is used just for writing. Added to facilitate combined datafetch and request. 1 = Write, 0 = request

5.3.2 I2C_dataRead

I2C_dataRead is used by the master to read messages from the I^{2}C. All reads begin by sending the slave device address to the I^{2}C. All slave devices check if this address matches their own address, and if it does they respond. The read communication follows the logic displayed in Figure 6.

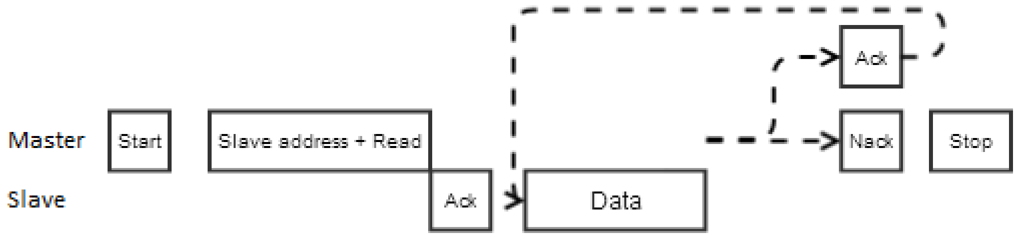
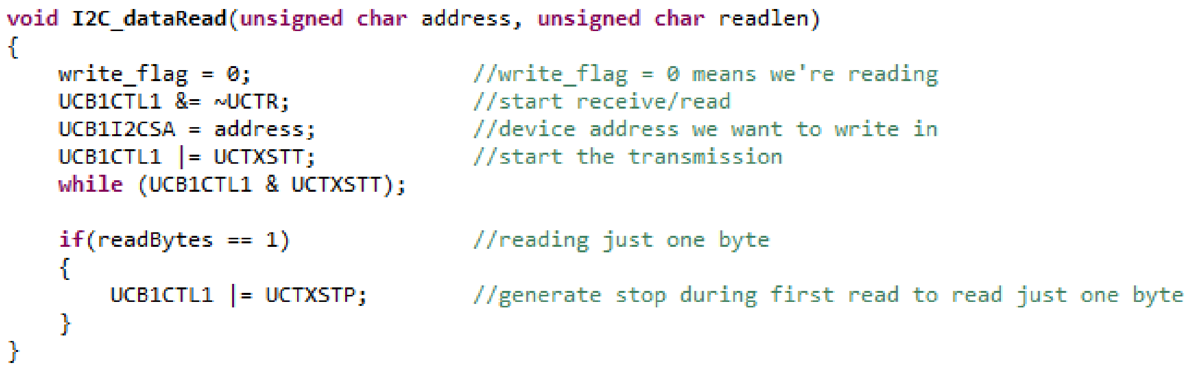


FIGURE 6 I2C_dataRead logic (Arponen 2014.04.04)

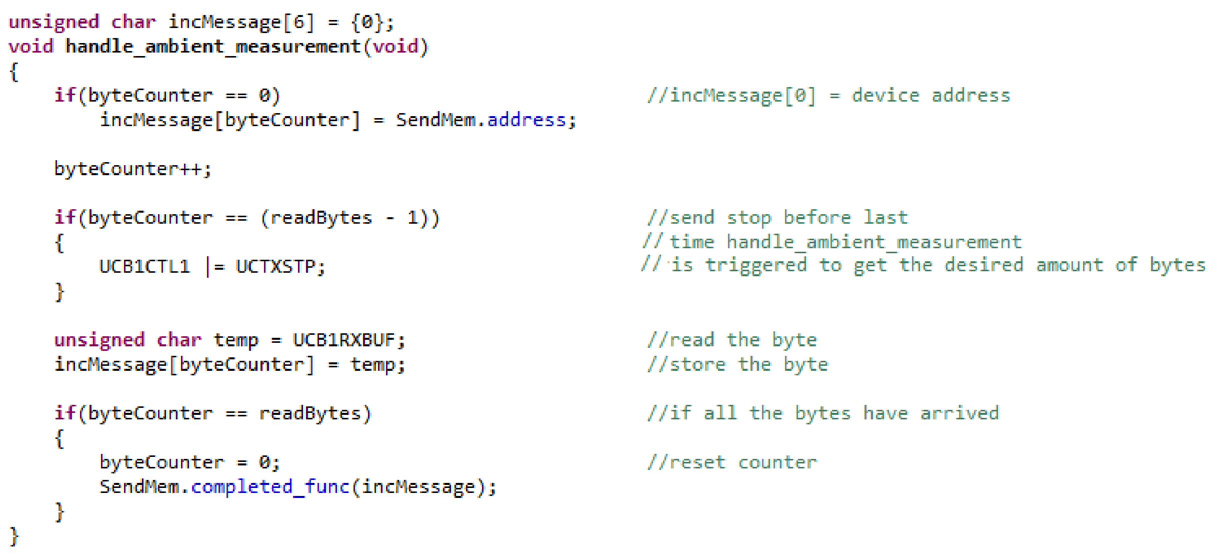
The master begins by writing the slave address to the UCB1I2CSA register. The master also sets the write bit in UCB1CTL1 register to 0. This means that the master enters a read mode. After this, the master starts the transmission. A slave device in that address responds with an Ack. Now that the master is in read mode, the slave device is allowed to speak. The slave begins to send data to the master and this triggers an RXIFG interrupt. The incoming byte is written in to the UCB1RXBUF buffer. The master can now read this byte during the RXIFG event handler. The byte is stored into an array called incMessage. If the master sends an Ack, the slave continues to send data. The communication continues until the master has read the designated amount of bytes from the slave. The communication is stopped when the master sends out a Nack followed by a Stop. Stop condition is generated after receiving data from the slave. This means that in order to receive the correct amount of bytes, the Stop condition must be sent after reading the second to last byte of data. It is vitally important to note that using just I2C_dataRead is not enough to get data from a sensor. The handling of the incoming data is further described in Chapter 5.4. In the code I2C_dataRead is made in accordance with Figure 7. Figure 8 displays how RXIFG interrupt is handled. Table 3 explains the parameters for I2C_dataRead function.



```
void I2C_dataRead(unsigned char address, unsigned char readlen)
{
    write_flag = 0;                //write_flag = 0 means we're reading
    UCB1CTL1 &= ~UCTR;             //start receive/read
    UCB1I2CSA = address;           //device address we want to write in
    UCB1CTL1 |= UCTXSTT;           //start the transmission
    while (UCB1CTL1 & UCTXSTT);

    if(readBytes == 1)             //reading just one byte
    {
        UCB1CTL1 |= UCTXSTP;       //generate stop during first read to read just one byte
    }
}
```

FIGURE 7 I2C_dataRead code (Arponen 2014.04.04)



```
unsigned char incMessage[6] = {0};
void handle_ambient_measurement(void)
{
    if(byteCounter == 0)                                    //incMessage[0] = device address
        incMessage[byteCounter] = SendMem.address;

    byteCounter++;

    if(byteCounter == (readBytes - 1))                      //send stop before last
    {                                                       // time handle_ambient_measurement
        UCB1CTL1 |= UCTXSTP;                                // is triggered to get the desired amount of bytes
    }

    unsigned char temp = UCB1RXBUF;                         //read the byte
    incMessage[byteCounter] = temp;                         //store the byte

    if(byteCounter == readBytes)                            //if all the bytes have arrived
    {
        byteCounter = 0;                                    //reset counter
        SendMem.completed_func(incMessage);
    }
}
```

FIGURE 8 RXIFG interrupt handler (Arponen 2014.04.04)

TABLE 3 I2C_dataRead parameters

Parameter	Description
Address	This is the address we want to read from. Mandatory for writing.
Readlen	Amount of bytes we want to read. Obsolete since this is set when initiating request.

The I2C_dataWrite is also used to initiate a data request. A data request informs a slave device to make a measurement and instructs it to wait for a data fetch. Data requesting follows the logic displayed in Figure 9.

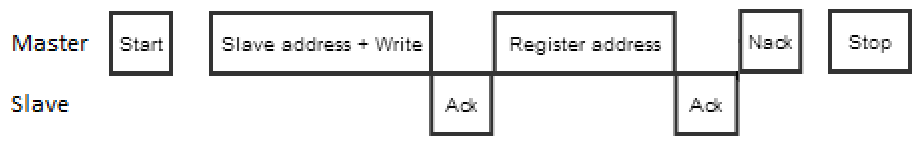


FIGURE 9 Data request logic (Arponen 2014.04.04)

The data request is almost the same as writing a value to a register. The master begins by writing the slave address to the UCB1I2CSA register. The master then sets the write bit in UCB1CTL1 register to 1. The transmission is then started. A slave device in that address responds with an Ack. This triggers a TXIFG interrupt. The master writes the address of a register on the device into the buffer. The buffer is then read by the device. A Nack followed by a Stop is then sent by the master. The master has now requested the value stored in this register. Following this request up with a data fetch reads the value stored in this register. Sensor specific data requesting will be discussed greater detail in Chapter 6.

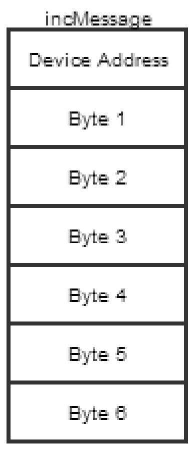
Data fetch follows a data request to complete the cycle of acquiring data. Data fetch is made by utilizing I2C_dataRead. It is important to understand that all reads must be preceded by a data request. The master begins by writing the slave address to the UCB1I2CSA register. The master also sets the write bit in UCB1CTL1 register to 0. This gives the sensor in question the permission to send its data. Some sensors require a certain time between requests and fetches. This is further discussed in Chapter 6. After all bytes have been received, the data fetch uses the function pointer provided by the data request to call a device specific function that handles the data.

Data request and fetch can be integrated together. This way the data fetch is sent immediately after the stop signal from data request. This produces a single function for reading a designated amount of bytes from a sensor. This single function can be seen in Figure 5, since dataread is called immediately. Some sensors have limits that need to be taken into consideration when combining these functions. Currently all ambient measurements are done by a combined data request and fetch.

5.4 Incoming data

Every incoming byte is stored into an array called incMessage. Bytes are written into the array when they are read from the UCB1RXBUF buffer. The size of the incMessage can be changed depending on the amount of bytes that are read from the sensors. The structure of the array is described in the Table 4. Once all the bytes have been read, a function pointer calls a device specific data handling function. A function pointer is a pointer that points to a function. By using a function pointer it is possible to do the data handling in a device specific file. This enables us to have a clear structure. (The Function Pointer Tutorials, 2011) As the data handling is very device specific, it will be further explained in Chapter 6. Measurements are updated and averaged so that they can be used in further calculations.

TABLE 4 Incoming data

Image	Indexing	Description
 incMessage Device Address Byte 1 Byte 2 Byte 3 Byte 4 Byte 5 Byte 6	incMessage[0]	The first index of the array always contains the address of the device that data is coming from. This is used to identify what device the data is coming from. Made obsolete by the use of function pointers.
	incMessage[1]	This index contains byte 1.
	incMessage[2]	This index contains byte 2.
	incMessage[3]	This index contains byte 3.
	incMessage[4]	This index contains byte 4.
	incMessage[5]	This index contains byte 5.
	incMessage[6]	This index contains byte 6.

5.5 Command queue

The ability to queue commands is desired in situations where commands need to be dealt with roughly at the same time or in quick succession. A command queue was implemented by adding an array of function pointers. When a measurement time for a sensor is reached, the measurement command is added to the queue. The index of the queue called pAdd will increase so that a new command may be added to the queue. When the master has time, a command is picked a command queue. After the command has been performed, it is removed from the queue and the queue is ready for the next command. The logic is displayed in Figure 10.

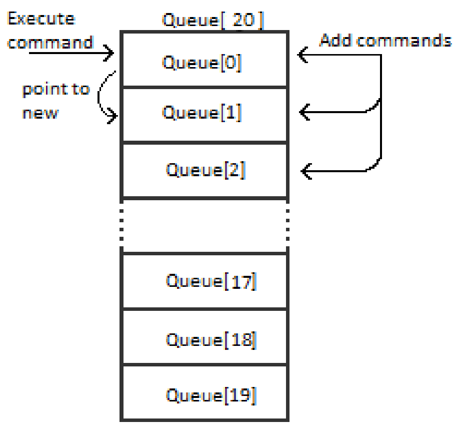


FIGURE 10 Command queue (Arponen 2014.04.04)

Adding functions to the function queue is implemented in the code according to Figure 12. The executeFunction is a defined type for a function pointer. The definition of executeFunction is displayed in Figure 13. The queue is handled in FIFO style. This means commands will be handled in the order they arrive in. FIFO comes from words "First In, First Out". Figure 11 displays the type defition that is used in the queue.

```
typedef void (*executeFunction)(void);
```

FIGURE 11 Type definition (Arponen 2014.04.04)

```
void Add_to_I2C_Command_Queue(executeFunction function)
{
    Queue[pAdd] = function;

    if(pAdd == 19)
        pAdd = 0;
    else pAdd++;
}
```

FIGURE 12 Adding commands to queue (Arponen 2014.04.04)

```
void Execute_I2C_Command_Queue(void)
{
    if(Queue[pCurrent] != 0)
    {
        //execute item
        Queue[pCurrent]();

        //remove current command from queue
        Queue[pCurrent] = 0;

        //increase pTask, if at limit start from zero
        if(pCurrent == 19)
            pCurrent = 0;
        else pCurrent++;
    }
}
```

FIGURE 13 Executing commands from queue (Arponen 2014.04.04)

Functions are added to the queue from a timer interrupt. This timer interrupt frequency was defined in Medikro to be 800 times each second because of the flow measurement sampling requirements. Each interrupt the system checks if it is time to measure ambient conditions. One command is executed for each timer interrupt. This means that the combined amount of measurements per second cannot exceed 800. If it does the queue will be filled faster than it’s emptied and commands will be lost. Timer logic is further explained for each sensor in Chapter 6. The logic of updating global battery variables is explained in Chapter 5.6.1. Commands in Figure 14 are executed each time the timer interrupt is triggered.

```
I2C_Barometer(1);               //parameter is the update frequency. The combined amount of
I2C_Humidity(1);                //measurements can't exceed 800 measurements/second or else
I2C_Accelerometer(1);           //the queue will be filled up and commands will be lost
Battery_Check_Timer(1);         //updates global variables every second
Execute_I2C_Command_Queue();    //completes one task per interrupt, max 800 tasks per second
```

FIGURE 14 Timer logic (Arponen 2014.04.04)

5.6 Power settings

MAX8934GETI+ controls the powering of the EVO platform. The power regulator unit has several input/output settings that are set with physical means, such as resistors or a capacitors. These are the features that are to remain constant throughout the use of the platform. For example, it is important to limit the maximum charge current for the battery. Such a limit is produced by connecting a resistor between the I_{SET} pin of the regulator and a GND connection. Charge limiting can also be done by setting register values in accordance with Table 7. Charging the lithium battery is a delicate operation and values of these resistors are to be carefully selected. The inputs/outputs of the MAX8934GETI+ regulator are documented in Table 5. An overline means that the pin is active-low.

TABLE 5 Power regulator pins of MAX8934. (MAXIM INTEGRATED, 2010).

PIN	NAME	FUNCTION
1	$\overline{\text{DONE}}$	Charge Complete Output. Used to signify that the battery is fully charged.
2,3	DC	DC Power Input. Current is limited by PEN1, PEN2 and R_{PSET}.
4	$\overline{\text{CEN}}$	Charger Enable Input. • 0 = Battery charging is enabled. • 1 = Battery charging is disabled.
5	PEN1	Input Limit Control 1.
6	PEN2	Input Limit Control 2.
7	PSET	DC Input Current Limit. Is connected to R_{PSET} = 3.3 kΩ. See Equation 1.
8	V_L	Internal Logic LDO Output Bypass Pin. Connected to 100 µF capacitor from V_L to GND. Powers the internal circuitry.
9,13	GND	Ground.
10	CT	Charge Timer Program Input. Connected to 68 nF capacitor.
11	ISET	Charge Current Limit. Is connected to R_{ISET} = 6.8 kΩ. See Equation 2.
12	USUS	USB Suspend Digital Input.
14	THM	Thermistor Input. Is connected to a NTC-thermistor to monitor battery temperature.
15	THMEN	Thermistor Enable Input.
16	THMSW	Thermistor Pull-up Supply Switch.
17	LDO	Always-On Linear Regulator Output. Always outputs 3.3 V and 30 mA. 1 µF capacitor is connected between LDO and GND.
18,19	USB	USB Power Input. USB current is limited by PEN2 and USUS.
20,21	BATT	Battery Connection. Pin is connected to the positive terminal of the Li+ battery.
22	$\overline{\text{CHG}}$	Charger Status Output. • 0 = fast charge or prequel • 1 = high impedance
23,24	SYS	System Supply Output. SYS is bypassed to GND with 10 µF capacitor.
25	$\overline{\text{OT}}$	Battery Overtemperature FLAG. • 0 = battery temperature ≥ 75 C. • 1 = no overtemperature detected
26	$\overline{\text{DOK}}$	DC Power-OK Output. • 0 = valid DC power connection • 1 = no DC power connected
27	$\overline{\text{UOK}}$	USB Power-OK Output. • 0 = valid USB power connection • 1 = no USB power connected
28	$\overline{\text{FLT}}$	Fault Output. • 0 = battery timer expired before prequel or fast charge complete • 1 = no fault
-	EP	Exposed Pad. Connected to GND.

Some of these inputs/outputs are connected directly to the pins of the MSP430 processor. These pins can be read or written to, in their designated port. All programmed power features are performed by manipulating these pins. The ports involved with the power control are P1, P3 and P5.

TABLE 6 MSP430 port values with USB power, 95mA limit.

	NAME	I/O	MSP430 Port.Pin	Default Value	Description
P1	$\overline{\text{DOK}}$	Input	1.7	1	NO DC power connected.
	$\overline{\text{UOK}}$	Input	1.6	0	USB Power connected.
	$\overline{\text{FLT}}$	Input	1.5	1	No fault.
	$\overline{\text{DONE}}$	Input	1.4	1	Charging is not done.
	$\overline{\text{CEN}}$	Output	1.3	0	Charging is enabled.
	PEN1	Output	1.2	1	See TABLE 7.
	PEN2	Output	1.1	0	See TABLE 7.
	USUS	Output	1.0	0	Digital input not suspended
P3	$\overline{\text{CHG}}$	Input	3.1	1	Battery not charging
	$\overline{\text{OT}}$	Input	3.0	1	No overtemperature
P5	THMEN	Output	5.5	1	Thermistor enabled

5.6.1 Programmed power features

Table 7 displays the logic behind controlling the current limits for the power regulator. These limits need to be taken into account when the device is connected to a new power source.

TABLE 7 Input limit control. (MAXIM INTEGRATED, 2010).

Power	$\overline{\text{DOK}}$	$\overline{\text{UOK}}$	PEN1	PEN2	USUS	DC input current limit	USB input current limit	Maximum charge current
AC adapter at DC input	0	X	1	X	X	Equation 1.	USB input off, DC has priority	Equation 2.
USB power at DC input	0	X	0	1	0	475 mA		475 mA
	0	X	0	0	0	95 mA		95 mA
	0	X	0	X	1	USB suspend		0
USB power at USB input	1	0	X	1	0	No DC input	475 mA	Equation 2.
	1	0	X	0	0		95 mA	
	1	0	X	X	1		USB suspend	0
No power	1	1	X	X	X		No USB input	0

Calculating the maximum DC input current limit follows the Equation 1**.** This equation describes the limit of the maximum input current into the system.

$$\mathrm{I_{DCIMAX}} = \frac{3000\ \mathrm{V}}{\mathrm{R_{PSET}}} = \frac{3000\ \mathrm{V}}{3300\ \Omega} \approx 0{,}91\ \mathrm{A} \qquad (1)$$

Calculating the maximum charge current is defined by Equation 2. This equation describes the limit of the maximum charge current that will be used to charge the battery.

$$\mathrm{I_{CHGMAX}} = \frac{3000\ \mathrm{V}}{\mathrm{R_{ISET}}} = \frac{3000\ \mathrm{V}}{6800\ \Omega} \approx 441.17\ \mathrm{mA} \approx 441\ \mathrm{mA} \qquad (2)$$

During the initialization of the EVO platform, the system reads port P1. From this port the system can see the connected power source. It is then possible to set the limits properly. The DC current limit is set with PEN1, PEN2 and $\mathrm{R_{PSET}}$. The USB current limit is set with PEN2 and USUS.

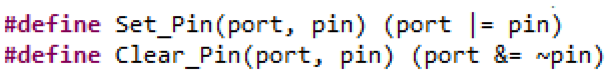
The default values given in Table 6 are used with the USB power connection. From Table 7 it is possible to see that those values limit the USB input current to 95 mA. The maximum charge current in this case follows Equation 2. However, the maximum charge current cannot exceed the input current limit. This means that the USB connection will charge the battery with a maximum charge current of 95 mA. A battery charge current measurement setup can be seen in Appendix 3.

If the USB power source was replaced with a DC power source, the DC input current limit would be set in accordance with Equation 1. From Equation 1 it is possible to see that the maximum DC input current is 1 A. The maximum charge current follows Equation 2. The DC connection will therefore have a maximum charge current of ~441 mA and a DC input current of 1 A.

Initialization functions were created to enable the user to set charge current values. Two versions were created, a 95 mA version and a 475 mA version. When the charge limit was set to 95 mA, the system charged the battery with an average of 65 mA. When the charge limit was set to 475 mA, the battery was charged with about 440 mA. This was because the system load was drawing roughly 30 mA to power the functions of the board. The initialization of the ports was done in accordance with Table 7.

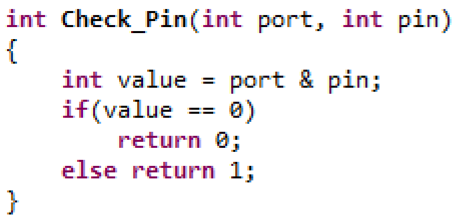
Battery charging can be manually disabled. This is done by setting CEN to 1. If the lithium battery has been fully charged, the regulator will set DONE to 0. Overcharging lithium batteries can cause them to swell up and eventually catch on fire. If the thermistor detects temperatures that exceed 75 C in battery discharge mode, OT will be set to 0. When the battery timer expires before prequel or fast charge is complete, FLT will be set to 0. In this case as well, CEN will be set to 1 and the charging will be disabled. The regulator is capable of doing this independently. Monitoring the regulator outputs is important because the processor needs to be aware of possibly dangerous events.

Special functions were created to control the settings to the regulator. Functions were created to set and clear the bits of a port. A function for checking a value of a specific bit was also created. These functions can be seen in Figure 15 and Figure 16.



```
#define Set_Pin(port, pin) (port |= pin)
#define Clear_Pin(port, pin) (port &= ~pin)
```

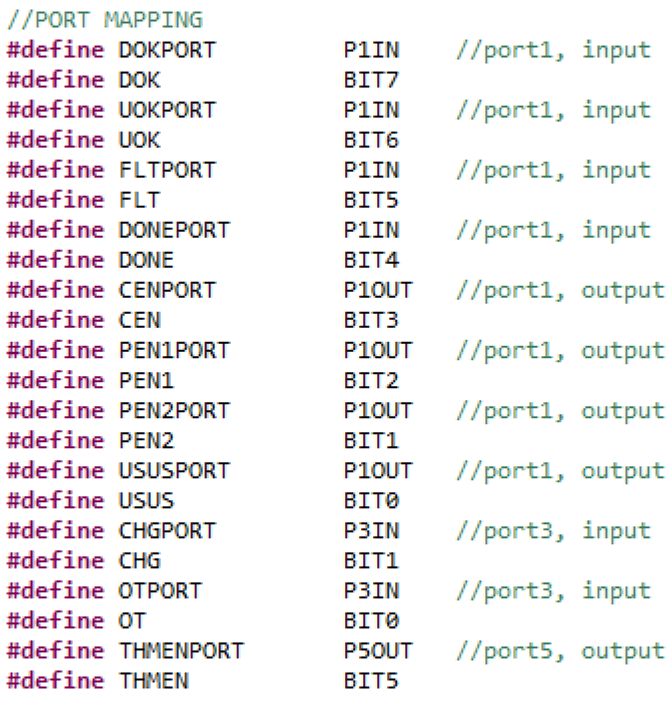
FIGURE 15 Bit operators



```
int Check_Pin(int port, int pin)
{
    int value = port & pin;
    if(value == 0)
        return 0;
    else return 1;
}
```

FIGURE 16 Check pin function (Arponen 2014.04.04)

Initialization of the current limits was made with the bit operator functions. The periodic checking of the power regulator outputs are made with the help of Check_Pin. In order to make the system modular, some port mapping is required. This is due to the fact that the ports and pins of the system may change in future versions. Having a port mapping done in a separate file enables the system to be changed easily in the future. The port mapping of the current system can be seen in Figure 17. After the functions and ports mappings are defined, the desired settings can be easily handled. The initialization of the charge currents is displayed in Figure 18, and it is done in accordance with Table 7.



```
//PORT MAPPING
#define DOKPORT         P1IN    //port1, input
#define DOK             BIT7
#define UOKPORT         P1IN    //port1, input
#define UOK             BIT6
#define FLTPORT         P1IN    //port1, input
#define FLT             BIT5
#define DONEPORT        P1IN    //port1, input
#define DONE            BIT4
#define CENPORT         P1OUT   //port1, output
#define CEN             BIT3
#define PEN1PORT        P1OUT   //port1, output
#define PEN1            BIT2
#define PEN2PORT        P1OUT   //port1, output
#define PEN2            BIT1
#define USUSPORT        P1OUT   //port1, output
#define USUS            BIT0
#define CHGPORT         P3IN    //port3, input
#define CHG             BIT1
#define OTPORT          P3IN    //port3, input
#define OT              BIT0
#define THMENPORT       P5OUT   //port5, output
#define THMEN           BIT5
```

FIGURE 17 Port mapping (Arponen 2014.04.04)

```
void Initialize_Battery_Charge_Limits(void)
{
    //USB, 95 mA, PEN1=X, PEN2=0, USUS=0
    Clear_Pin(PEN2PORT, PEN2);
    Clear_Pin(USUSPORT, USUS);

    //USB, 475 mA, PEN1=X, PEN2=1, USUS=0
//  Set_Pin(PEN2PORT,PEN2);
//  Clear_Pin(USUSPORT,USUS);

    //charge enable
    //Clear_Pin(CENPORT,CEN);

    //charge disable
    //Set_Pin(CENPORT,CEN);
}
```

FIGURE 18 Charge limits (Arponen 2014.04.04)

The regulator can intelligently control itself based on the limits it is given. The regulator can turn off the battery charging, when appropriate, but the application needs to be aware of the state the regulator is in. This is achieved by making global variables that are periodically updated. The global variables contain all the I/O values of the regulator. The system can then react to changes in the power settings. For instance, the system can send out a message, or toggle LEDs. If the system detects that the battery is in discharge mode, it will set THMEN high to enable battery temperature monitoring. While charging the battery, the power regulator is monitoring the temperature automatically. Figure 19 displays the updating of the global battery variables.

```
void Battery_Settings_Update(void)
{
    //this updates the current I/O statuses of the power settings
    DOK_INPUT = Check_Pin(DOKPORT, DOK);
    UOK_INPUT = Check_Pin(UOKPORT, UOK);
    FLT_INPUT = Check_Pin(FLTPORT, FLT);
    DONE_INPUT = Check_Pin(DONEPORT, DONE);
    CEN_OUTPUT = Check_Pin(CENPORT, CEN);
    PEN1_OUTPUT = Check_Pin(PEN1PORT, PEN1);
    PEN2_OUTPUT = Check_Pin(PEN2PORT, PEN2);
    USUS_OUTPUT = Check_Pin(USUSPORT, USUS);
    CHG_INPUT = Check_Pin(CHGPORT, CHG);
    OT_INPUT = Check_Pin(OTPORT, OT);
    THMEN_OUTPUT = Check_Pin(THMENPORT, THMEN);

    if(FLT_INPUT == 1)
    {
        //inform user that fault occurred
    }

    if(DONE_INPUT == 1)
    {
        //inform user that charging is done
    }

    if(OT_INPUT == 1)
    {
        //inform user that overtemp is detected
    }
}
```

FIGURE 19 Global regulator I/O (Arponen 2014.04.04)

6 AMBIENT MEASUREMENT SENSORS

6.1 Barometer

A barometer is a device that is used to measure atmospheric pressure. The barometer is made by Honeywell and it's a part of their TruStability sensor line. The specific brand of the barometer on the EVO platform is SSCMNNN015PA3A3. The barometer is a part of the SSC series and its functionality is based on the piezoresistive phenomenon. Piezoresistive material is a material that changes its resistivity when a pressure is applied to it. The barometer does not require initialization or register specific instructions. (Honeywell, 2010).

The barometer does not require any initialization from the master. It also does not have any registers that need to be polled to acquire its measurement data. When the barometer gets a data request, it makes a new measurement. The barometer only needs to see its address, 0x38, to be written into the I^{2}C. Requesting data from the barometer follows the logic displayed in Figure 20**.** The I2C_dataWrite is made in such a way that it can be used to write just the slave address on to the I^{2}C. (Honeywell, 2010).

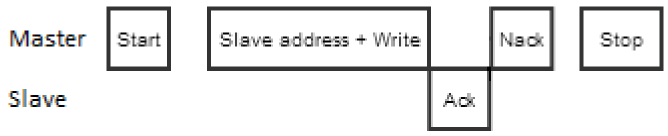


FIGURE 20 Barometer data request (Arponen 2014.04.04)

Requesting data causes the barometer to power up, make the measurement and start the conversion of the measurement into its output. The conversion typically lasts 36.65 milliseconds and afterwards new data is ready to be fetched. The measurement data is fetched by I2C_dataRead and the output is two bytes long. Fetching data from the barometer follows the logic displayed in Figure 21.

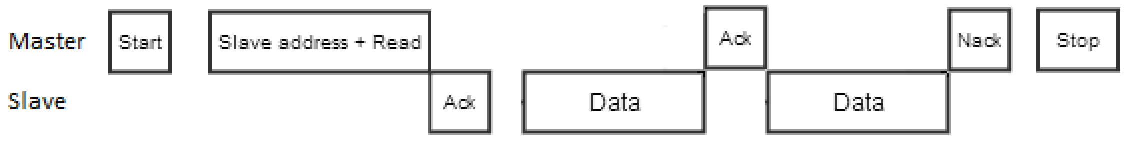
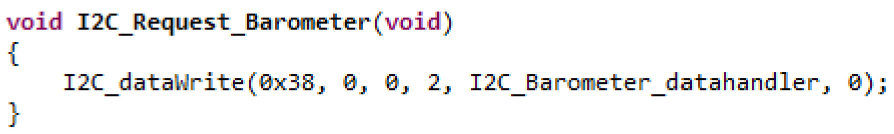


FIGURE 21 Barometer data fetch (Arponen 2014.04.04)

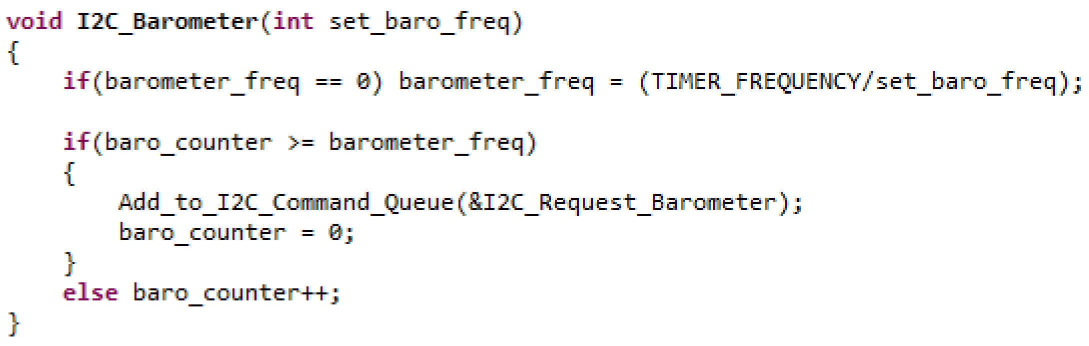
Figure 22 displays how two bytes of barometer data are requested. The address of the barometer is 0x38, and there is no need to write additional data. After the two bytes are read, I2C_Barometer_datahandler is called by a callback function pointer. Justwrite is set to 0 as we are requesting data and not writing to the registers.



```
void I2C_Request_Barometer(void)
{
    I2C_dataWrite(0x38, 0, 0, 2, I2C_Barometer_datahandler, 0);
}
```

FIGURE 22 I2C barometer data request (Arponen 2014.04.04)

Timing of barometer measurements is done in accordance with Figure 23. If the counter exceeds the value calculated from the desired frequency, a barometer measurement request is added to the function queue.



```
void I2C_Barometer(int set_baro_freq)
{
    if(barometer_freq == 0) barometer_freq = (TIMER_FREQUENCY/set_baro_freq);

    if(baro_counter >= barometer_freq)
    {
        Add_to_I2C_Command_Queue(&I2C_Request_Barometer);
        baro_counter = 0;
    }
    else baro_counter++;
}
```

FIGURE 23 Barometer timer (Arponen 2014.04.04)

The barometer outputs two bytes of data. The data is packed according to Table 8.

TABLE 8 Barometer output

Byte 1								**Byte 2**							
Status		Barometer data						Barometer data							
S1	**S0**	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1	B0

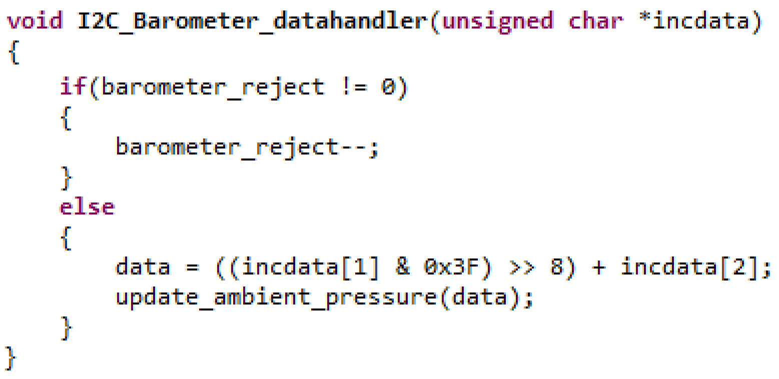
The first two bits are status bits to indicate various faults. These bits are taken into account in the handling of the data. When both status bits are zero, the device is in normal operation. They can show if a conversion is not yet ready, or if the device is not working. If the device is working as intended, the status bits are bit masked off of the measurement. Both of the measured bytes need to be combined so that the measurement can be used. The first byte is bit shifted to the left eight times and added to the second byte. Then a device specific calculation is performed to transform the two bytes into an actual barometric value. Figure 24 displays how incoming data is handled in the barometer specific data handler.

The two status bits are masked off and the data is combined in accordance to Equation 3. These equations describe how the measured bytes from the sensor are handled.

$$\text{Barometer output} = ((\text{Byte 1} \,\&\, 0\text{x3F}) \gg 8) + \text{Byte 2} \qquad (3)$$

The PSI value of the barometric measurement is calculated in accordance with Equation 4.

$$\mathrm{P_{applied}} = \frac{\left(\frac{\text{Barometer output}*100\%}{2^{14}} - 10\%\right)*15\ \text{PSI}}{80\ \%} \qquad (4)$$



```
void I2C_Barometer_datahandler(unsigned char *incdata)
{
    if(barometer_reject != 0)
    {
        barometer_reject--;
    }
    else
    {
        data = ((incdata[1] & 0x3F) >> 8) + incdata[2];
        update_ambient_pressure(data);
    }
}
```

Figure 24 Barometer data handling (Arponen 2014.04.04)

TABLE 9 contains actual measurement data from the barometer. The measurement value for byte 1 is 0x37 and the value for byte 2 is 0x83.

TABLE 9 Example calculation

Byte 1 = 0x37							
Status		Barometer data					
S1	**S0**	B13	B12	B11	B10	B9	B8
0	0	1	1	0	1	1	1

Byte 2 = 0x83							
Barometer data							
B7	B6	B5	B4	B3	B2	B1	B0
1	0	0	0	0	0	1	1

$$\text{Barometer output} = \big((0\text{x37}\& \ 0\text{x3F}) \gg 8\big) + 0\text{x83} = 0\text{x3783} = 14211$$

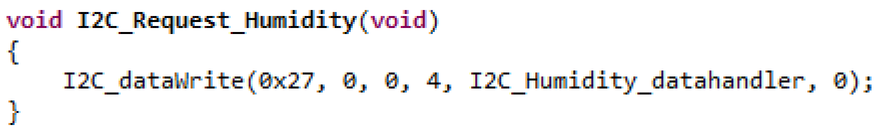
$$\mathrm{P_{applied}} = \frac{\left(\frac{14211 * 100\%}{2^{14}} - 10\%\right) * 15\ \text{PSI}}{80\ \%} \approx 14{,}388\ \text{PSI} \approx 14{,}4\ \text{PSI} \approx 0{,}99\ \text{bar}$$

These calculations are done in accordance with EQUATION 3 and 4. This measurement roughly equals the pressure of the atmosphere, which means the device is working as intended.

6.2 Humidity sensor

The humidity sensor measures the humidity of the surrounding air. The same sensor can measure the temperature of the surrounding air as well. The humidity sensor is a part of the HumidIcon product line and it is manufactured by Honeywell. The specific brand of the humidity sensor on the EVO platform is HIH-6130-021-001S. The humidity sensor does not require initialization or register specific instructions. (Honeywell a, 2012)

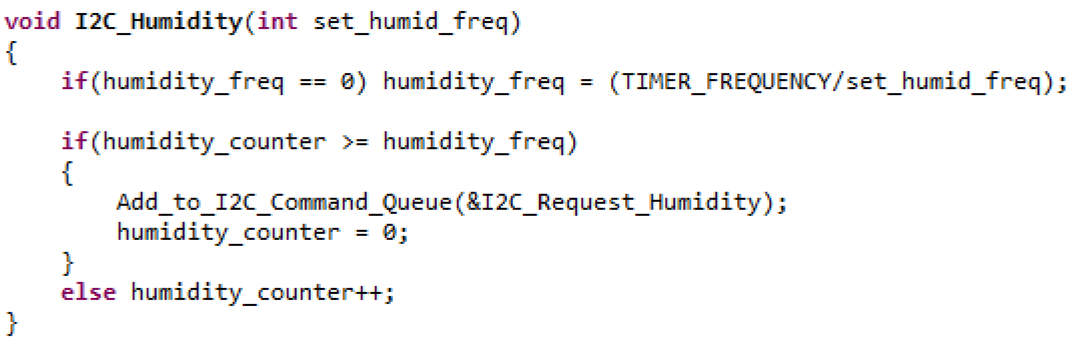
The humidity sensor is almost identical to the barometer in its operation. The humidity sensor does not require any initialization from the master. It also does not have any registers that need to be polled to acquire its measurement data. The humidity sensor is powered down unless it receives a measurement request. It only needs to see its address, 0x27, to be written into the I^{2}C. Requesting data from the humidity sensor follows the logic displayed in Figure 20**.** The conversion of the measured data typically lasts 36.65 milliseconds and afterwards new data is ready to be fetched. The measurement data is fetched by I2C_dataRead and the output is four bytes long. The address of the humidity sensor is 0x27, no additional data is needed and four bytes are being read. I2C_Humidity_datahandler designates the callback function for the humidity data. (Honeywell a, 2012)



```
void I2C_Request_Humidity(void)
{
    I2C_dataWrite(0x27, 0, 0, 4, I2C_Humidity_datahandler, 0);
}
```

FIGURE 25 Humidity data request (Arponen 2014.04.04)

Timing of humidity measurements is done in accordance with Figure 26. If the counter exceeds the value calculated from the desired frequency, a humidity measurement request is added to the function queue.



```
void I2C_Humidity(int set_humid_freq)
{
    if(humidity_freq == 0) humidity_freq = (TIMER_FREQUENCY/set_humid_freq);

    if(humidity_counter >= humidity_freq)
    {
        Add_to_I2C_Command_Queue(&I2C_Request_Humidity);
        humidity_counter = 0;
    }
    else humidity_counter++;
}
```

FIGURE 26 Humidity timer (Arponen 2014.04.04)

The humidity sensor outputs four bytes of data. The data is packed according to Table 10.

TABLE 10 Humidity output

Byte 1							
Status		Humidity data					
S1	**S0**	B13	B12	B11	B10	B9	B8

Byte 2							
Humidity data							
B7	B6	B5	B4	B3	B2	B1	B0

Byte 3							
Temperature data							
T13	T12	T11	T10	T9	T8	T7	T6

Byte 4							
Temperature data						DNC	
T5	T4	T3	T2	T1	T0	x	x

The status bits indicate various faults. These bits are taken into account in the handling of the data. When both status bits are zero, the device is in normal operation. They can show if a conversion is not yet ready, or if the device is not working. If the device is working as intended, the status bits are bit masked off of the measurement. Both humidity data bytes need to be combined to so that the measurement can be used. The last two bits of the fourth byte are DNC bits.

The two status bits are masked off and the data is combined in accordance to Equation 5.

$$\text{Humidity output} = ((\text{Byte 1} \ \& \ 0\text{x}3\text{F}) \gg 8) + \text{Byte 2} \qquad (5)$$

The relative humidity value of the measurement is calculated in accordance with Equation 6.

$$\text{Humidity (\%RH)} = \frac{\text{Humidity output}}{2^{14}-2} * 100\% \qquad (6)$$

Byte 3 needs to be shifted six times to the right because of the DNC bits in the fourth byte. Byte 4 needs to be shifted twice to the left to eliminate the DNC bits. This is done in accordance with Equation 7.

$$\text{Temperature output} = ((\text{Byte 3} \gg 6) + (\text{Byte 4} \gg 2) \qquad (7)$$

The temperature value of the measurement is calculated in accordance with Equation 8.

$$\text{Temperature (°C)} = \frac{\text{Temperature output}}{2^{14}-2} * 165 - 40 \qquad (8)$$

```
void I2C_Humidity_datahandler(unsigned char *incdata)
{
    if(humidity_reject != 0)
    {
        humidity_reject--;
    }
    else
    {
        data = ((incdata[1] & 0x3F) >> 8) + incdata[2];
        update_ambient_humidity(data);
        data = (incdata[3] >> 6) + (incdata[4] << 2);
        update_ambient_temperature2(data);
    }
}
```

FIGURE 27 Humidity data handling (Arponen 2014.04.04)

Table 11 contains actual measurement data from the humidity sensor. The measured value for byte 1 is 0x08, byte 2 is 0xE4, byte 3 is 0x60 and byte 4 is 0xD8.

TABLE 11 Example calculation

Byte 1 = 0x08							
Status		Humidity data					
S1	**S0**	B13	B12	B11	B10	B9	B8
0	0	0	0	1	0	0	0

Byte 2 = 0xE4							
Humidity data							
B7	B6	B5	B4	B3	B2	B1	B0
1	1	1	0	0	1	0	0

Byte 3 = 0x60							
Temperature data							
T13	T12	T11	T10	T9	T8	T7	T6
0	1	1	0	0	0	0	0

Byte 4 = 0xD8							
Temperature data						DNC	
T5	T4	T3	T2	T1	T0	x	x
1	1	0	1	1	0	0	0

These equations describe how the measured bytes from the sensor are handled. They are done in accordance with EQUATION 5-8.

$$\text{Humidity output} = \big((0\text{x}08 \mathbin{\&} 0\text{x}3\text{F}) \gg 8\big) + 0\text{xE}4 = 0\text{x}08\text{E}4 = 2276$$

$$\text{Humidity (\%RH)} = \frac{2276}{2^{14}-2} * 100\% \approx 13{,}893\ \% \approx 14\ \%$$

$$\text{Temperature output} = ((0\text{x}60 \gg 6) + (0\text{xD}8 \gg 2) = 0\text{x}1836 = 6198$$

$$\text{Temperature (°C)} = \frac{6198}{2^{14}-2} * 165 - 40 \approx 22{,}426\ \text{°C} \approx 22{,}4\ \text{°C}$$

These values were measured inside an office building and they are quite accurate.

6.3 Accelerometer

An accelerometer is a device that can measure acceleration. The accelerometer used in the EVO platform is capable of measuring acceleration in three axes. The accelerometer is made by Analog Devices and its specific brand is ADXL346. The accelerometer is very complex in comparison to the other sensors. There are many registers that contain different values and the accelerometer can be used as a control method as well. (Analog Devices, 2010.)

Some of the values in the registers of the accelerometer need to be initialized before the accelerometer begins to function. Initializing is handled by writing all the values with a single I2C_dataWrite. Initializing the registers follows the logic displayed in Figure 3. The registers of the accelerometer are described in Table 12. The Read/Write registers are used for settings that define functionalities. Generally speaking, all the READ ONLY registers are outputs that are generated from the measurements. In its current operation, the orientation of the platform is read from the X, Y and Z statuses. It is also possible to detect taps of the device. INT_MAP register maps single tap to INT1 and double tap to INT2. These are physical output pins that are connected directly to the GS_INT1 and GS_INT2 pins on the processor. This way we can directly detect taps on processor side. Once a tap interrupt has occurred, INT_SOURCE must be read to clear the interrupt. The settings for register values that limit tap detection can be found in APPENDIX 4. (Analog Devices, 2010.)

TABLE 12 Accelerometer initialization

Register address	NAME	DEFAULT	Initialized	Type	Note
0x1D	THRESH_TAP	0x00	0x30	R/W	Threshold for tap detection, 3g.
0x1E	OFSX	0x00	0x00	R/W	X-axis offset.
0x1F	OFSY	0x00	0x00	R/W	Y-axis offset.
0x20	OFSZ	0x00	0x00	R/W	Z-axis offset.
0x21	DUR	0x00	0x10	R/W	Max duration for tap, 10 ms.
0x22	Latent	0x00	0x10	R/W	Latency after tap, 20 ms.
0x23	Window	0x00	0x40	R/W	Time limit for double tap, 80 ms.
0x24	THRESH_ACT	0x00	0x00	R/W	Activity threshold.
0x25	THRESH_INACT	0x00	0x00	R/W	Inactivity threshold.
0x26	TIME_INACT	0x00	0x00	R/W	Inactivity time.
0x27	ACT_INACT_CTL	0x00	0x00	R/W	Axis enable for activity/inactivity.
0x28	THRESH_FF	0x00	0x00	R/W	Freefall threshold.
0x29	TIME_FF	0x00	0x00	R/W	Freefall time.
0x2A	TAP_AXES	0x00	0x07	R/W	Include all axes in tap detection.
0x2B	ACT_TAP_STATUS	0x00	-	R	Axis tap status, READ ONLY.
0x2C	BW_RATE	0x0B	0x0C	R/W	400 Hz mode.
0x2D	POWER_CTL	0x00	0x08	R/W	Measure mode on.
0x2E	INT_ENABLE	0x00	0x60	R/W	Enable single and double tap.
0x2F	INT_MAP	0x00	0x20	R/W	Single tap to INT1, double to INT2
0x30	INT_SOURCE	0x02	-	R	Interrupt source, READ ONLY.
0x31	DATA_FORMAT	0x00	0x0B	R/W	full resolution, +-16g.
0x32	DATAX0	varies	-	R	X-axis status x0, READ ONLY.
0x33	DATAX1	varies	-	R	X-axis status x1, READ ONLY.

0x34	DATAY0	varies	-	R	Y-axis status y0, READ ONLY.
0x35	DATAY1	varies	-	R	Y-axis status y1, READ ONLY.
0x36	DATAZ0	varies	-	R	Z-axis status z0, READ ONLY.
0x37	DATAZ1	varies	-	R	Z-axis status z1, READ ONLY.
0x38	FIFO_CTL	0x00	0x00	R/W	FIFO Control.
0x39	FIFO_STATUS	0x00	-	R	READ ONLY.
0x3A	TAP_SIGN	0x00	-	R	READ ONLY.
0x3B	ORIENT_CONF	0x25	0x25	R/W	Orientation configuration.
0x3C	ORIENT	0x00	-	R	Orientation, READ ONLY.

Tap detection values are explained in FIGURE 28. Tap threshold is the limit that must be exceeded before an event will be registered as a tap. Duration is the maximum amount of time an event can last to be registered as a tap. If acceleration exceeds this time it won't trigger a tap interrupt. Latency is an amount of time where no new taps can be detected. Window is a time limit in which another tap must occur to trigger a double-tap.

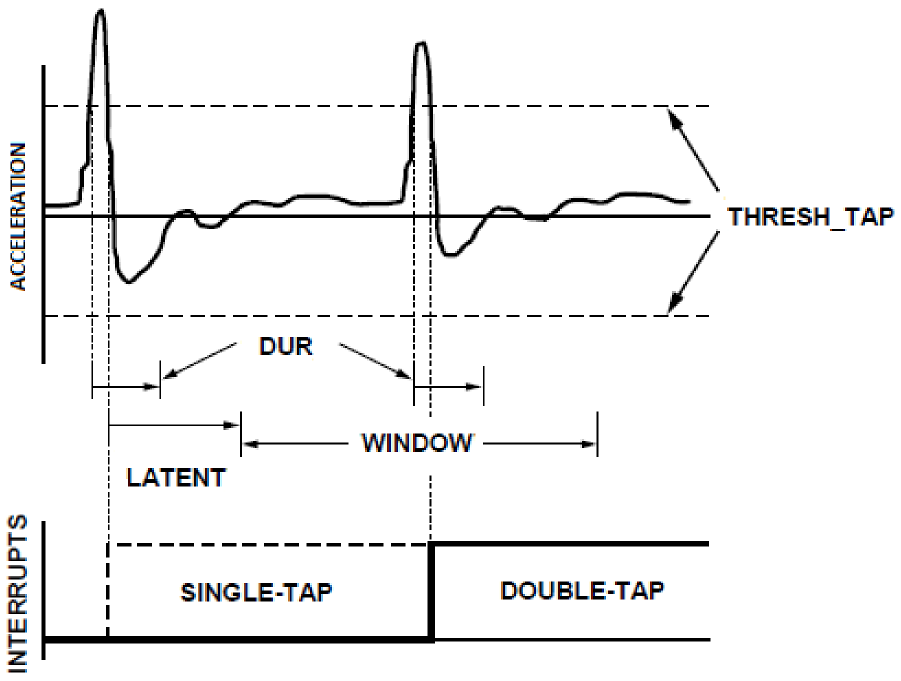


FIGURE 28 Tap detection (ANALOG DEVICES, 2010)

Communication to the accelerometer is a bit more complicated than the previous sensors. The accelerometer requires the address of a specific register to be defined. This is due to the fact that there are over 30 registers on the device. Requesting the value of a specific register follows the logic displayed in FIGURE 29.

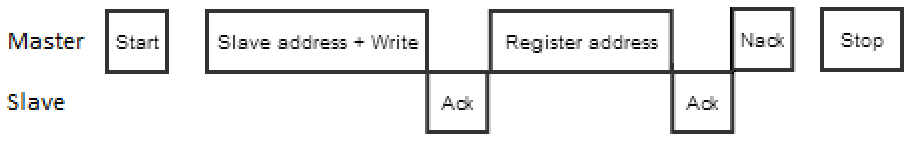
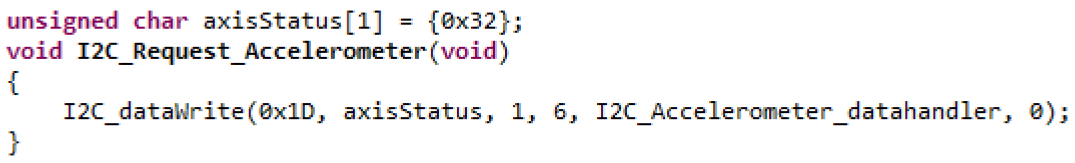


FIGURE 29 Accelerometer data request

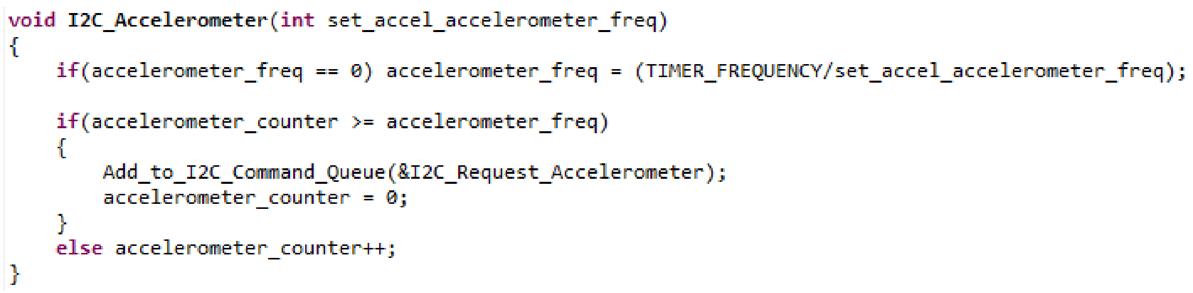
Fetching the data is done exactly the same way as the previous sensors. The amount of bytes read depends on what register is being read. For instance, it is recommended to read both the axis status bits in orientation measurements. It's highly useful to read all six axes status registers so the orientation can be accurately used in calculations. An example of a single register read is the reading of the INT_SOURCE register. This is done to clear tap interrupts. Theoretically it is possible to read all the registers at once. This could done by setting the amount of bytes to match the amount of registers on the device. FIGURE 30 displays how six bytes of data starting from register 0x32 are requested. The accelerometer is located in address 0x1D, the register we're interested in is designated at axisStatus[1]. The amount of bytes we want to read is six and the callback function is designated to be specific for the accelerometer.



```
unsigned char axisStatus[1] = {0x32};
void I2C_Request_Accelerometer(void)
{
    I2C_dataWrite(0x1D, axisStatus, 1, 6, I2C_Accelerometer_datahandler, 0);
}
```

FIGURE 30 I2C accelerometer register 0x32 request (Arponen 2014.04.04)

Timing of accelerometer measurements is done in accordance with Figure 31. If the counter exceeds the value calculated from the desired frequency, an accelerometer measurement request is added to the function queue.



```
void I2C_Accelerometer(int set_accel_accelerometer_freq)
{
    if(accelerometer_freq == 0) accelerometer_freq = (TIMER_FREQUENCY/set_accel_accelerometer_freq);

    if(accelerometer_counter >= accelerometer_freq)
    {
        Add_to_I2C_Command_Queue(&I2C_Request_Accelerometer);
        accelerometer_counter = 0;
    }
    else accelerometer_counter++;
}
```

FIGURE 31 Accelerometer timer (Arponen 2014.04.04)

The accelerometers output format varies greatly depending on the register. In this thesis, the most important outputs were the orientation of the device and the source of the interrupts. The outputs for these registers are explained in Table 13 and Table 14.

TABLE 13 Interrupt source output

INT_SOURCE							
D7	D6	D5	D4	D3	D2	D1	D0
DATA_READY	SINGLE_TAP	DOUBLE_TAP	ACTIVITY	INACTIVITY	FREE_FALL	WATERMARK	OVERRUN/ORIENTATION

TABLE 14 Axis status output

Byte 1							
DATAX0							
D15	D14	D13	D12	D11	D10	D9	D8

Byte 2							
DATAX1							
D7	D6	D5	D4	D3	D2	D1	D0

Byte 3							
DATAY0							
D15	D14	D13	D12	D11	D10	D9	D8

Byte 4							
DATAY1							
D7	D6	D5	D4	D3	D2	D1	D0

Byte 5							
DATAZ0							
D15	D14	D13	D12	D11	D10	D9	D8

Byte 6							
DATAZ1							
D7	D6	D5	D4	D3	D2	D1	D0

As the tap interrupts are directly connected to the processor, INT_SOURCE register only needs to be read to clear interrupts after they occur. The detection and handling of the taps is done by the processor. GS_INT1 and GS_INT2 are connected to P4IN port on the processor. GS_INT1 is connected to P4.5 and GS_INT2 is connected to P4.4. The interrupts are handled on processor side according to Equation 9. If TAP INTERRUPT equals 2, then a single tap has occurred. A value of 3 means a double tap has occurred. This value can then be used to trigger the reading of INT_SOURCE to clear the interrupt.

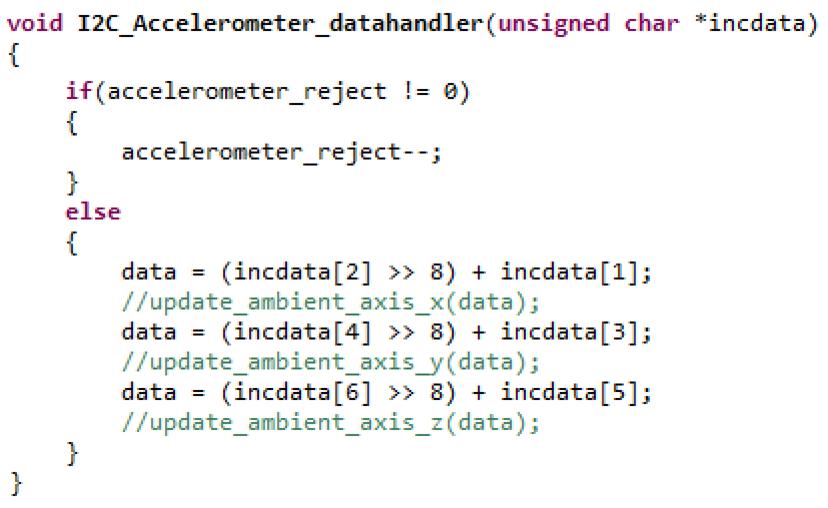
$$\text{TAP INTERRUPT} = \text{P4IN} \ \& \ 0\text{x}30 \qquad (9)$$

It is important to note that axis status for each axis is two bytes long. When sending the axis status information the sensor sends the least significant byte first so the data must be handled in accordance with Equations 10, 11 and 12. The output data is a 16-bit twos complement.

$$\text{X AXIS} = \frac{\text{Byte 2} \gg 8 + \text{Byte 1}}{256} \qquad (10)$$

$$\text{Y AXIS} = \frac{\text{Byte 4} \gg 8 + \text{Byte 3}}{256} \qquad (11)$$

$$\text{Z AXIS} = \frac{\text{Byte 6} \gg 8 + \text{Byte 5}}{256} \qquad (12)$$



```
void I2C_Accelerometer_datahandler(unsigned char *incdata)
{
    if(accelerometer_reject != 0)
    {
        accelerometer_reject--;
    }
    else
    {
        data = (incdata[2] >> 8) + incdata[1];
        //update_ambient_axis_x(data);
        data = (incdata[4] >> 8) + incdata[3];
        //update_ambient_axis_y(data);
        data = (incdata[6] >> 8) + incdata[5];
        //update_ambient_axis_z(data);
    }
}
```

FIGURE 32 Accelerometer data handling

Table 15 contains actual measurement data from the accelerometers axis status registers. The measured value for byte 1 is 0x11, byte 2 is 0x00, byte 3 is 0x0E and, byte 4 is 0x00, byte 5 is 0xFA and byte 6 is 0x00.

TABLE 15 Example calculation

Byte 1 = 0x11							
DATAX0							
D15	D14	D13	D12	D11	D10	D9	D8
0	0	0	1	0	0	0	1

Byte 2 = 0x00							
DATAX1							
D7	D6	D5	D4	D3	D2	D1	D0
0	0	0	0	0	0	0	0

Byte 3 = 0x0E							
DATAY0							
D15	D14	D13	D12	D11	D10	D9	D8
0	0	0	0	1	1	1	0

Byte 4 = 0x00							
DATAY1							
D7	D6	D5	D4	D3	D2	D1	D0
0	0	0	0	0	0	0	0

Byte 5 = 0xFA							
DATAZ0							
D15	D14	D13	D12	D11	D10	D9	D8
1	1	1	1	1	0	1	0

Byte 6 = 0x00							
DATAZ1							
D7	D6	D5	D4	D3	D2	D1	D0
0	0	0	0	0	0	0	0

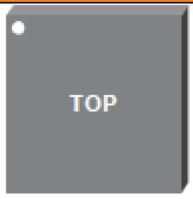
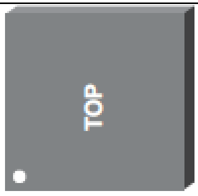
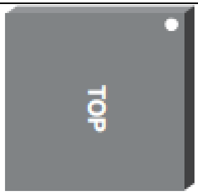
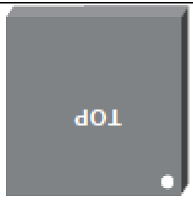
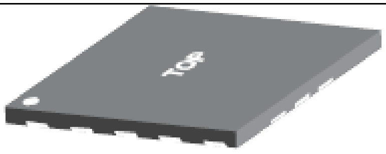
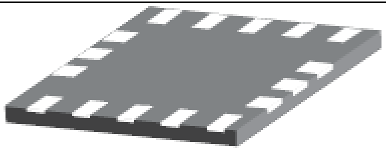
$$\text{X AXIS} = \frac{0\text{x}00 \gg 8 + 0\text{x}11}{256} = \frac{17}{256} \approx 0{,}07$$

$$\text{Y AXIS} = \frac{0\text{x}00 \gg 8 + 0\text{x}0\text{E}}{256} = \frac{14}{256} \approx 0{,}05$$

$$\text{Z AXIS} = \frac{0\text{x}00 \gg 8 + 0\text{xFA}}{256} = \frac{250}{256} \approx 0{,}99$$

These equations describe how the measured bytes from the sensor are handled. They are done in accordance with Equation 10-12. The orientation of the accelerometer can be interpreted from Table 16. The measurement was made when the EVO platform was placed flat on the table. Based on this knowledge the measurement is fairly accurate. Various axis offsets can be calibrated by initializing the OFSX, OFSY and OFSZ registers in accordance with Table 12.

TABLE 16 Orientation interpretation (ANALOG DEVICES, 2010)

X-axis	Y-axis	Z-axis	Orientation
1	0	0	
0	-1	0	
0	1	0	
-1	0	0	
0	0	1	
0	0	-1	

7 DEVELOPMENT TOOLS

7.1 Code Composer Studio

Code Composer Studio, CCS, is an integrated development environment (IDE) and it is made by Texas Instruments. It is used for developing embedded applications on TI-made processor families. CCS is based on an open source software framework called Eclipse. A free version of the CCS is available, but it is code-size limited. CCS supports C/C++ languages. The applications made during this thesis were coded in C.

7.2 Tortoise SVN

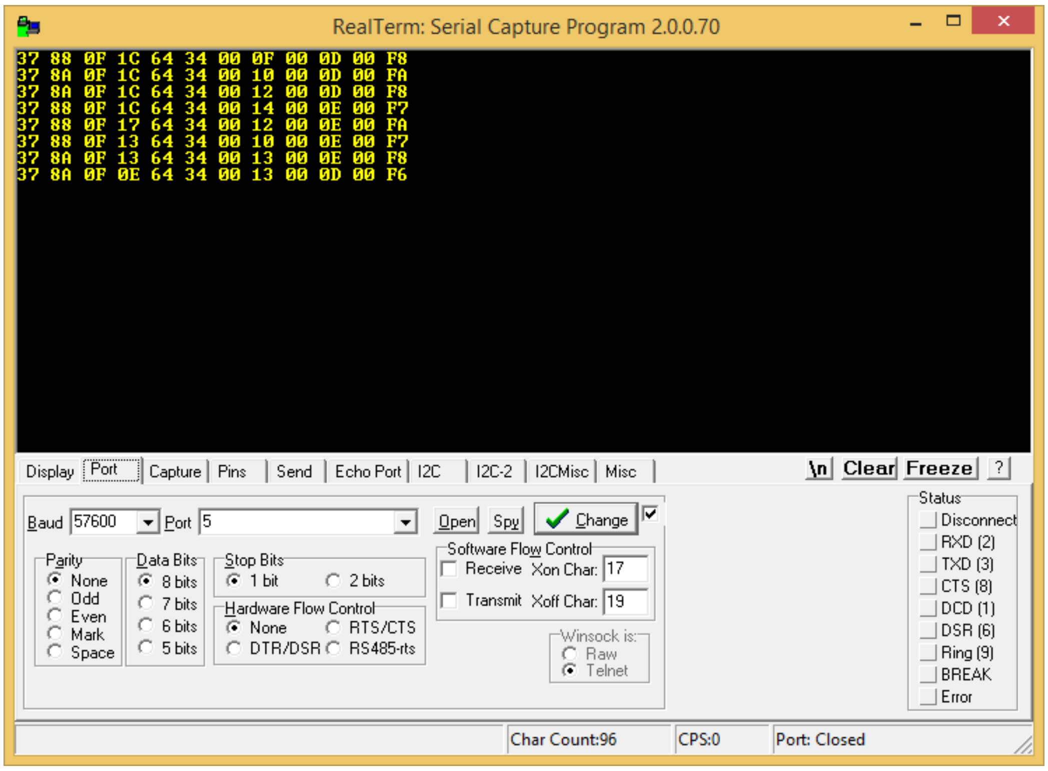
Tortoise SVN is an open sourced version control system. It enables users to copy existing projects into their own workstations. A user can then work on their own copy of the project without altering the original project. Once a user is done with their work, they can submit their work back to the shared folder. All versions are stored and it is possible to track the progression of the entire project. It is also possible to produce various graphs and charts to monitor the branching of the project versions. During the making of this thesis, the source code was periodically submitted into the Tortoise SVN version history folder of Medikro Oy. This enabled several useful features. It enabled the storing of several versions with varying implementations. The best version was naturally always continued. Tortoise SVN also enabled Medikro employees to review the code to make sure it was up to their standards.

7.3 Measurement devices

Various voltages and currents were measured while working on this thesis. Multimeters were used for these measurements. A current voltage setup can be seen in Appendix 3. Voltage measurements were made to check if the inputs and outputs matched the values they were supposed to be. Currents were measured to verify that charge currents were limited properly.

7.4 Realterm

Realterm is a simple serial terminal. Realterm can be used to show communication on the serial line. It was used to debug the software as all the measurements from the sensors can be sent to the serial. The connection to the EVO platform is opened by first checking what port the platform is connected to. This can be done from Devices and Printers menu. It can also be done in Realterm by double-clicking the Port drop-down menu. Once the port is selected, it can be opened by pressing the Open button. Figure 33 displays the Port-page of Realterm. The values seen on the screen are measurement values from the sensors. First two bytes are from the barometer, following four from the humidity sensor and the last six from the accelerometer. Realterm was also used to set the EVO platform into programming state. EVO platform communicates via the USB connection and the device driver on the PC generates a Virtual Serial Port for communications.



```
37 88 0F 1C 64 34 00 0F 00 0D 00 F8
37 8A 0F 1C 64 34 00 10 00 0D 00 FA
37 8A 0F 1C 64 34 00 12 00 0D 00 F8
37 88 0F 1C 64 34 00 14 00 0E 00 F7
37 88 0F 17 64 34 00 12 00 0E 00 FA
37 88 0F 13 64 34 00 10 00 0E 00 F7
37 8A 0F 13 64 34 00 13 00 0E 00 F8
37 8A 0F 0E 64 34 00 13 00 0D 00 F6
```

Baud 57600 Port 5

Parity: [✓ None] Odd Even Mark Space

Data Bits: [✓ 8 bits] 7 bits 6 bits 5 bits

Stop Bits: [✓ 1 bit] 2 bits

Hardware Flow Control: [✓ None] RTS/CTS DTR/DSR RS485-rts

Software Flow Control: Receive Xon Char: 17 Transmit Xoff Char: 19

Winsock is: Raw [✓ Telnet]

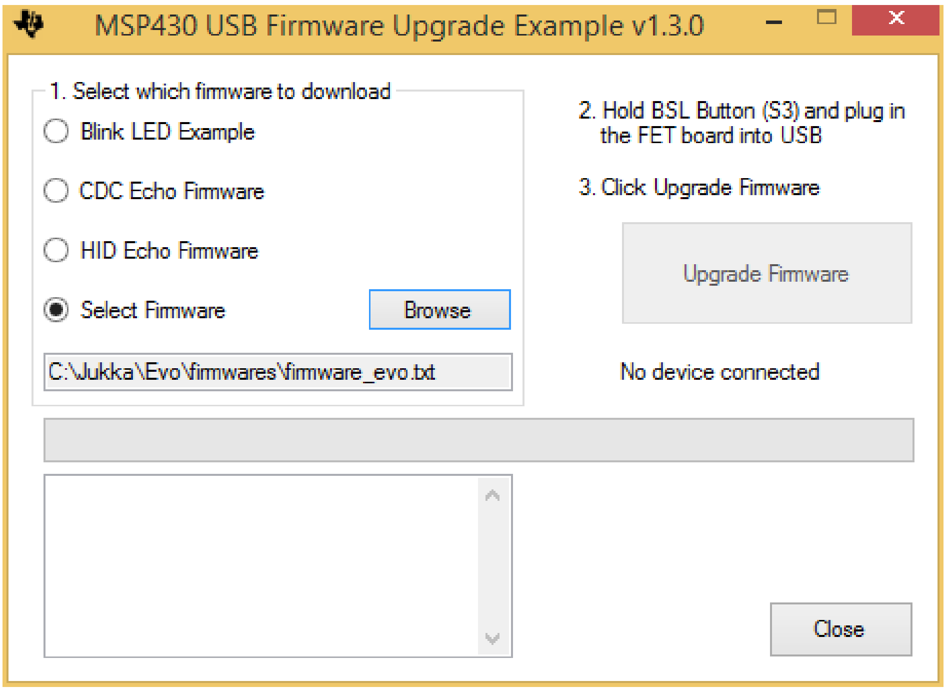
Status: Disconnect RXD (2) TXD (3) CTS (8) DCD (1) DSR (6) Ring (9) BREAK Error

Char Count:96 CPS:0 Port: Closed

FIGURE 33 Realterm Port select (Arponen, 2014.03.13)

7.5 MSP430 USB Firmware Upgrade

MSP430 USB Firmware Upgrade is used to upload the code into the EVO platform. The text file containing the code we want to upload is selected by pressing Browse. If the device is not connected or set to programming state, the program reports that no device is connected.


MSP430 USB Firmware Upgrade Example v1.3.0
1. Select which firmware to download
Blink LED Example
CDC Echo Firmware
HID Echo Firmware
Browse
2. Hold BSL Button (S3) and plug in the FET board into USB
3. Click Upgrade Firmware
Upgrade Firmware
Close

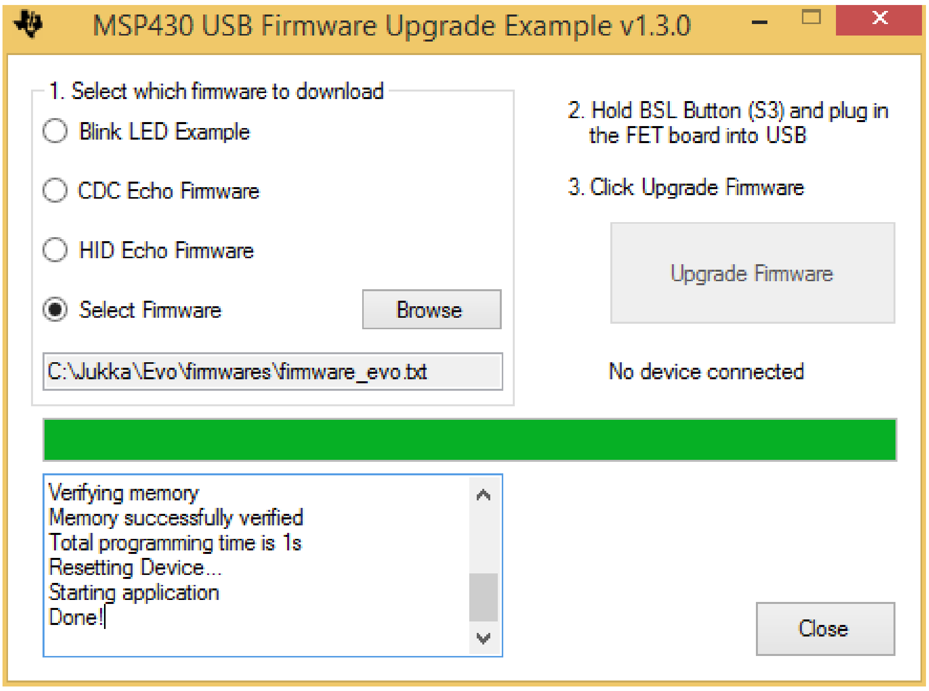
● Select Firmware

C:\Jukka\Evo\firmwares\firmware_evo.txt

No device connected

FIGURE 34 MSP430 USB Firmware (Arponen, 2014.03.13)

Once the device is connected and set to programming state, code can be uploaded by pressing the Upgrade Firmware button. A successfully uploaded code is displayed in Figure 35. After uploading the code the device will function with it until a new code is uploaded.


MSP430 USB Firmware Upgrade Example v1.3.0
1. Select which firmware to download
Blink LED Example
CDC Echo Firmware
HID Echo Firmware
Browse
2. Hold BSL Button (S3) and plug in the FET board into USB
3. Click Upgrade Firmware
Upgrade Firmware
Close

● Select Firmware

C:\Jukka\Evo\firmwares\firmware_evo.txt

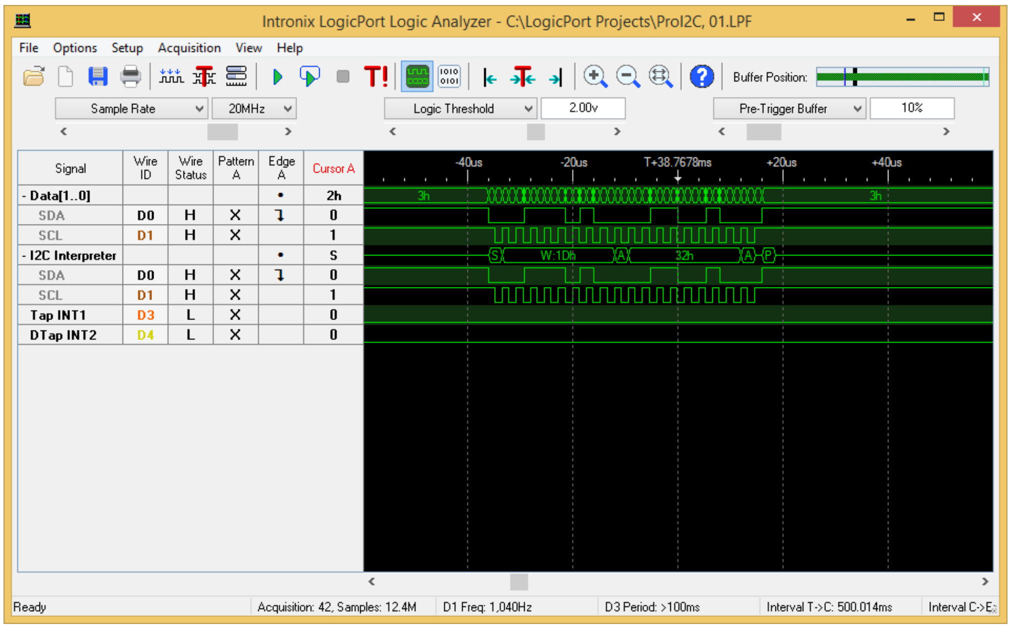
No device connected

```
Verifying memory
Memory successfully verified
Total programming time is 1s
Resetting Device...
Starting application
Done!
```

FIGURE 35 MSP430 Firmware upload (Arponen, 2013.03.13)

7.6 Logicport

Logicport is an extremely useful logic analyser. Various signals can be measured by connecting a Logicport measurement device into the components. The measurement device is connected to a PC through a USB-connection and the signals can be followed on a program. This enables the reading of messages sent on the I^{2}C. The inputs are fully user definable. This makes following the communication very easy. Figure 36 shows Logicsport's view of communication on the I^{2}C. We can see that the communication follows the logic of a measurement request. The device address is 0x1D, so we know that the device is the accelerometer. The value following the Ack is the register being requested. In this case, it is register 0x32. This register contains values is explained by Table 12.


Intronix LogicPort Logic Analyzer - C:\LogicPort Projects\ProI2C, 01.LPF
File Options Setup Acquisition View Help
Buffer Position:
-40us -20us T+38.7678ms +20us +40us
3h
S W:1Dh A 32h A P

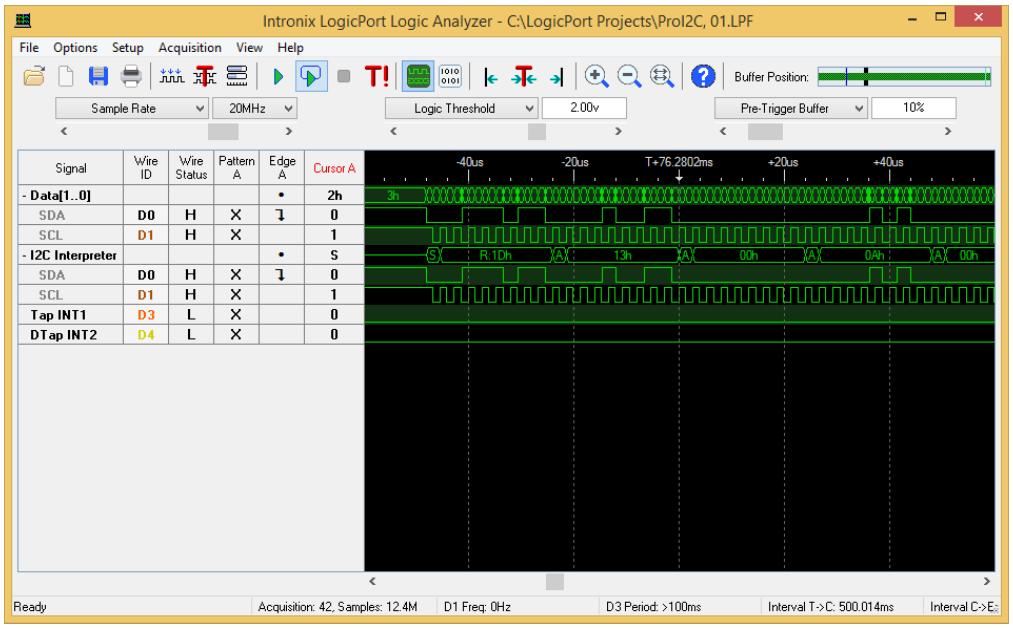
Sample Rate: 20MHz
Logic Threshold: 2.00v
Pre-Trigger Buffer: 10%

Signal	Wire ID	Wire Status	Pattern A	Edge A	Cursor A
- Data[1..0]				•	2h
SDA	D0	H	X	↴	0
SCL	D1	H	X		1
- I2C Interpreter				•	S
SDA	D0	H	X	↴	0
SCL	D1	H	X		1
Tap INT1	D3	L	X		0
DTap INT2	D4	L	X		0

Ready | Acquisition: 42, Samples: 12.4M | D1 Freq: 1,040Hz | D3 Period: >100ms | Interval T->C: 500.014ms | Interval C->E:

FIGURE 36 Logicport data request (Arponen, 2014.03.13)

The data fetch following the request is displayed in Figure 37. We can clearly see that the accelerometer is outputting several bytes of data. Logic errors in the communication are very easy to see when using this software.


Intronix LogicPort Logic Analyzer - C:\LogicPort Projects\ProI2C, 01.LPF
File Options Setup Acquisition View Help
Buffer Position:
-40us -20us T+76.2802ms +20us +40us
3h
S R:1Dh A 13h A 00h A 0Ah A 00h

Sample Rate: 20MHz
Logic Threshold: 2.00v
Pre-Trigger Buffer: 10%

Signal	Wire ID	Wire Status	Pattern A	Edge A	Cursor A
- Data[1..0]				•	2h
SDA	D0	H	X	↴	0
SCL	D1	H	X		1
- I2C Interpreter				•	S
SDA	D0	H	X	↴	0
SCL	D1	H	X		1
Tap INT1	D3	L	X		0
DTap INT2	D4	L	X		0

Ready | Acquisition: 42, Samples: 12.4M | D1 Freq: 0Hz | D3 Period: >100ms | Interval T->C: 500.014ms | Interval C->E:

FIGURE 37 Logicport data fetch (Arponen, 2014.03.13)

8 RESULTS

The structure of the entire system was made modular by separating the layers of operation. This included making new files for each device, as well as separating the entire I^{2}C into its own file. All of the restructuring was taken into account in the code. The structure model makes it easy to add new functionality and devices due to the modularity of the software.

The I^{2}C communication functions were created in a way that enables the user to communicate with any device in the I^{2}C. Setting values to the registers of a device was made possible, as well as reading values from them. The functions were simple and easy to use.

Measurements were made possible by the functions in the I^{2}C layer. Reading and writing to the sensors was made possible. The sensors were initialized by writing values to their registers, if necessary. The values read from the sensors were returned to a device specific data handler. The ambient conditions were then updated and used in calculations.

Power settings from the regulator were studied and tested. Functions were created to read the outputs of the regulator. This enabled the system to know what state the power management is. It was then possible to take further action, e.g. send a note to the user that the battery is overheating. It is also possible to control the power settings programmatically.

The timer logic allows functions to be called at a desired frequency. The frequency is calculated from timer interrupts. This method allows the system to remain interrupt driven. When a measurement time is reached, the function in question is added to the queue. Commands are executed from the queue based on the timer in a controlled and logical manner.

Queueing function calls functions based on the timer. Several functions can be added to the queue in the same clock interrupt. Commands are executed from the queue once every clock interrupt. The queue worked perfectly and allowed flexibility in the timing routine.

9 DEVELOPMENT IDEAS

In the future, the use of LEDs could be implemented to signal statuses of various inputs and outputs. For instance, it could be useful to use the LEDs with the global outputs of the power regulator. This way the power settings could be monitored, even without having to print the outputs to the PC.

The switches located in the B-side of the EVO platform could be used to control outputs. The switches can be seen on Appendix 1. For example, these switches could be used to toggle the power regulator settings. They could also be used to add commands to the command queue. This could be used in a situation where commands need to be added to the queue manually. Tap interrupts are cleared by reading the tap source register from the accelerometer. This could be done by the switch, so that the interrupt is cleared after it has been detected by a user.

The EVO platform contains a Bluetooth communication chip. In the future this chip could be used for wireless communication. Measurement data could be sent to PC or phone wirelessly. However, setting up Bluetooth communication needs a lot of work that has to be done.

The wireless power source could also be implemented in the future. This would bring additional flexibility into the power management.

Tap interrupts could be used for control or for discarding data. Tapping a measurement device while a measurement is being made may alter the results. In this case, it would be wise to discard these measurements. Taps could also be used for adding commands to the command queue. Tilting the accelerometer can also be used as an interrupt. All three axes could be used as separate tilt interrupts.

REFERENCES

ANALOG DEVICES, 2010. ADXL346 Ultralow Power Digital Accelerometer. [Product datasheet]. Available: http://www.analog.com/static/imported-files/data_sheets/ADXL346.pdf

HONEYWELL, 2010 July. TruStability™ Silicon Pressure Sensors: SSC Series-Standard Accuracy. [Product datasheet]. Available: http://sensing.honeywell.com/honeywell-sensing-ssc-silicon-pressure-sensors-analog-product-sheet-008215-2-en.pdf

HONEYWELL a, 2012 June. I2C Communication with the Honeywell HumidIcon™ Digital Humidity/Temperature Sensors. [Technical note]. Available: http://sensing.honeywell.com/i2c%20comms%20humidicon%20tn_009061-2-en_final_07jun12.pdf

HONEYWELL b, 2012 June. Entering and Using Command Mode on the Honeywell HumidIcon™ Digital Humidity/Temperature Sensors. [Technical Note]. Available: http://sensing.honeywell.com/command%20mode%20humidicon%20tn_009062-3-en_final_07jun12.pdf

MAXIM INTEGRATED, 2010. Dual-Input Linear Charger, Smart Power Selector with Advanced Battery Temperature Monitoring. [Product datasheet]. Available: http://datasheets.maximintegrated.com/en/ds/MAX8934G.pdf

MEDIKRO OY, 2013 May 31st. Medikro EVO platform. [PowerPoint slides].

NXP SEMICONDUCTORS, 2012 October. UM10204 I^{2}C-bus specification and user manual. [User manual]. Available: http://www.nxp.com/documents/user_manual/UM10204.pdf

PHILIPS SEMICONDUCTORS, 2003 March. AN20216-01 I^{2}C Manual. [Application Note]. Available: http://www.nxp.com/documents/application_note/AN10216.pdf

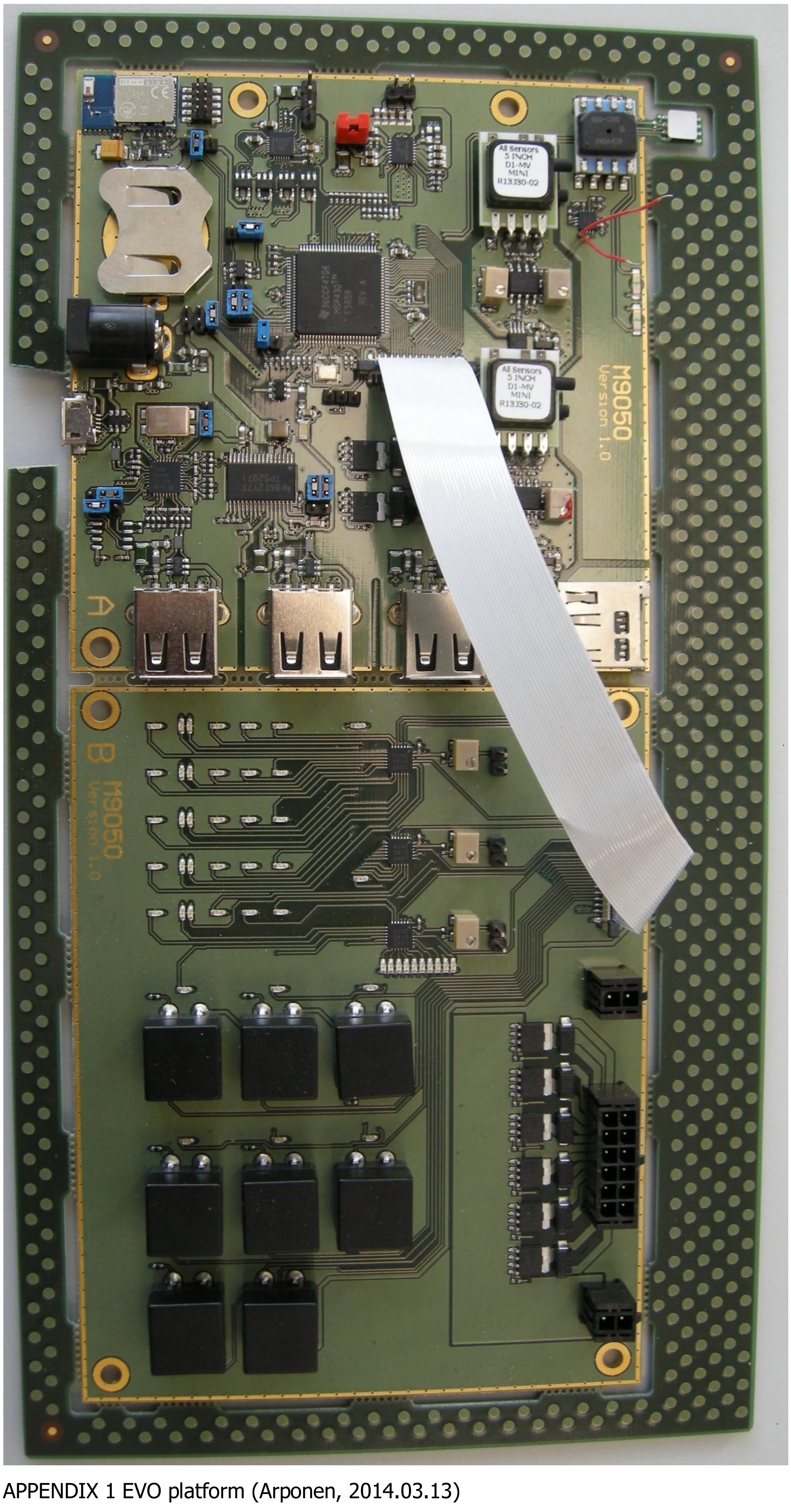
TEXAS INSTRUMENTS, 2014. MSP430™ Ultra-Low-Power Microcontrollers. [Product brochure]. Available: http://www.ti.com/lit/sg/slab034w/slab034w.pdf

TEXAS INSTRUMENTS, 2008 June. MSP430x5xx and MSP430x6xx Family. [User' Guide]. (Revised 2013 February). Available: http://www.ti.com/lit/ug/slau208m/slau208m.pdf

TEXAS INSTRUMENTS, 2012. MSP430F665X, MSP430F645X, MSP430F565X, MSP430F535X MIXED SIGNAL MICROCONTROLLERS. [Product datasheet]. Available: http://www.ti.com/lit/ds/symlink/msp430f5658.pdf

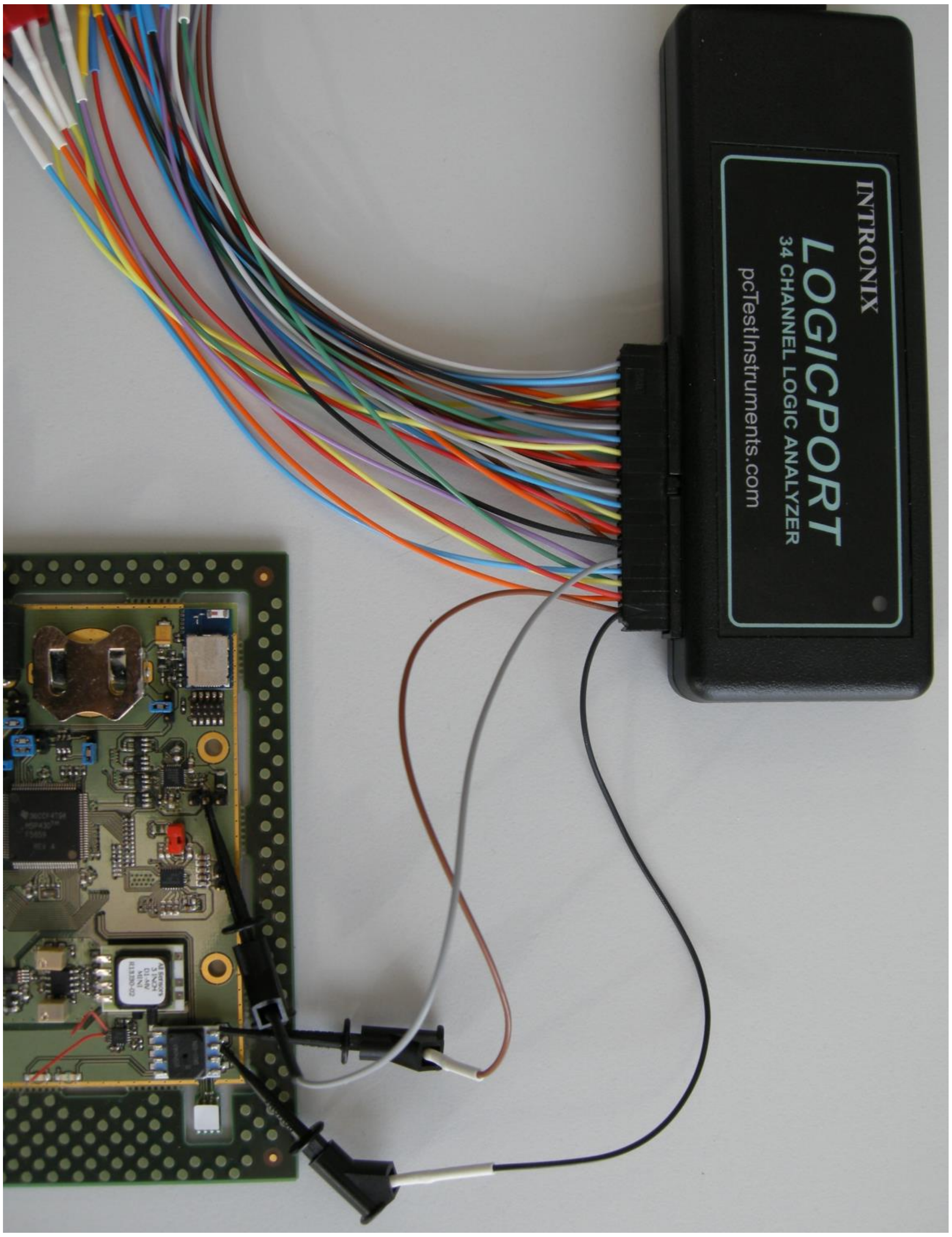
The Function Pointer Tutorials, 2011. [Referenced 27.3.2014]. Available: http://www.newty.de/fpt/index.html

APPENDIX 1: EVO PLATFORM



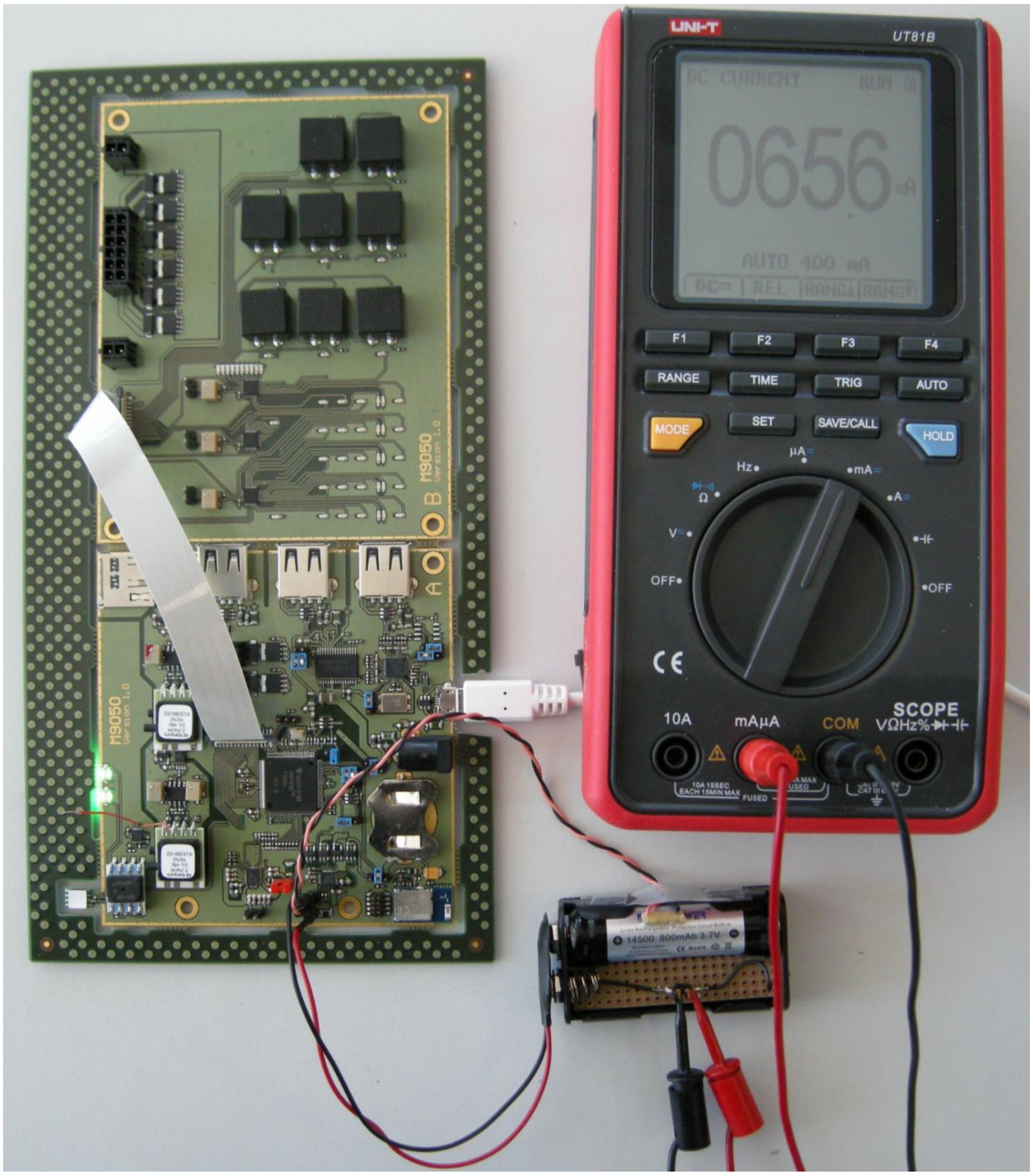
APPENDIX 1 EVO platform (Arponen, 2014.03.13)

APPENDIX 2: LOGICPORT CONNECTION



APPENDIX 2 Logicport connection. (Arponen, 2014.03.13)

APPENDIX 3: BATTERY CHARGE CURRENT



APPENDIX 3 Charge current with 95 mA limit. (Arponen 2014.03.31)

APPENDIX 4: ACCELEROMETER TAP DETECTION VALUES

Register value DEC	Register value HEX	Threshold (g)	Duration (µs)	Latency (ms)	Window (ms)
0	0x00	0,00	0	0,00	0,00
1	0x01	0,06	625	1,25	1,25
2	0x02	0,13	1250	2,50	2,50
3	0x03	0,19	1875	3,75	3,75
4	0x04	0,25	2500	5,00	5,00
5	0x05	0,31	3125	6,25	6,25
6	0x06	0,38	3750	7,50	7,50
7	0x07	0,44	4375	8,75	8,75
8	0x08	0,50	5000	10,00	10,00
9	0x09	0,56	5625	11,25	11,25
10	0x0A	0,63	6250	12,50	12,50
11	0x0B	0,69	6875	13,75	13,75
12	0x0C	0,75	7500	15,00	15,00
13	0x0D	0,82	8125	16,25	16,25
14	0x0E	0,88	8750	17,50	17,50
15	0x0F	0,94	9375	18,75	18,75
16	0x10	1,00	10000	20,00	20,00
17	0x11	1,07	10625	21,25	21,25
18	0x12	1,13	11250	22,50	22,50
19	0x13	1,19	11875	23,75	23,75
20	0x14	1,25	12500	25,00	25,00
21	0x15	1,32	13125	26,25	26,25
22	0x16	1,38	13750	27,50	27,50
23	0x17	1,44	14375	28,75	28,75
24	0x18	1,51	15000	30,00	30,00
25	0x19	1,57	15625	31,25	31,25
26	0x1A	1,63	16250	32,50	32,50
27	0x1B	1,69	16875	33,75	33,75
28	0x1C	1,76	17500	35,00	35,00
29	0x1D	1,82	18125	36,25	36,25
30	0x1E	1,88	18750	37,50	37,50
31	0x1F	1,95	19375	38,75	38,75
32	0x20	2,01	20000	40,00	40,00
33	0x21	2,07	20625	41,25	41,25
34	0x22	2,13	21250	42,50	42,50
35	0x23	2,20	21875	43,75	43,75
36	0x24	2,26	22500	45,00	45,00
37	0x25	2,32	23125	46,25	46,25
38	0x26	2,38	23750	47,50	47,50
39	0x27	2,45	24375	48,75	48,75
40	0x28	2,51	25000	50,00	50,00
41	0x29	2,57	25625	51,25	51,25
42	0x2A	2,64	26250	52,50	52,50
43	0x2B	2,70	26875	53,75	53,75
44	0x2C	2,76	27500	55,00	55,00
45	0x2D	2,82	28125	56,25	56,25
46	0x2E	2,89	28750	57,50	57,50
47	0x2F	2,95	29375	58,75	58,75
48	0x30	3,01	30000	60,00	60,00
49	0x31	3,07	30625	61,25	61,25
50	0x32	3,14	31250	62,50	62,50
51	0x33	3,20	31875	63,75	63,75
52	0x34	3,26	32500	65,00	65,00
53	0x35	3,33	33125	66,25	66,25
54	0x36	3,39	33750	67,50	67,50
55	0x37	3,45	34375	68,75	68,75
56	0x38	3,51	35000	70,00	70,00
57	0x39	3,58	35625	71,25	71,25
58	0x3A	3,64	36250	72,50	72,50
59	0x3B	3,70	36875	73,75	73,75
60	0x3C	3,76	37500	75,00	75,00

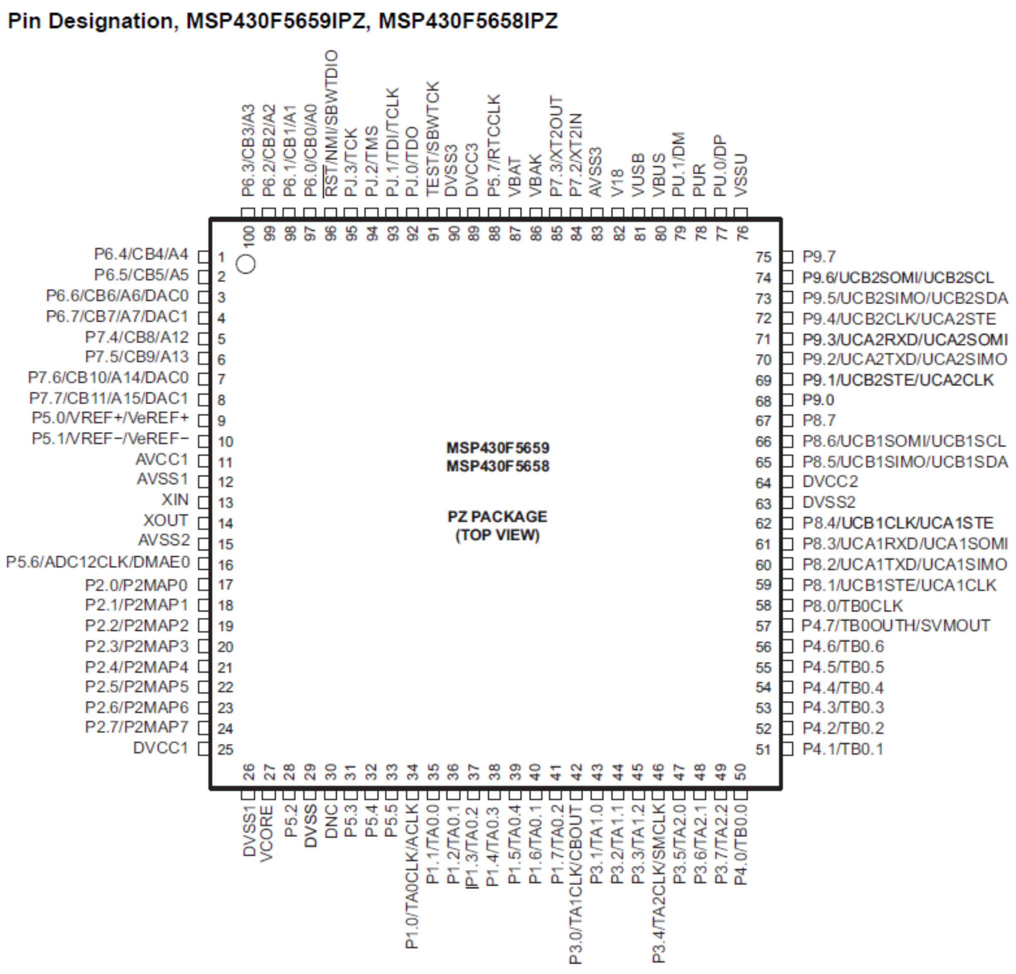
61	0x3D	3,83	38125	76,25	76,25
62	0x3E	3,89	38750	77,50	77,50
63	0x3F	3,95	39375	78,75	78,75
64	0x40	4,02	40000	80,00	80,00
65	0x41	4,08	40625	81,25	81,25
66	0x42	4,14	41250	82,50	82,50
67	0x43	4,20	41875	83,75	83,75
68	0x44	4,27	42500	85,00	85,00
69	0x45	4,33	43125	86,25	86,25
70	0x46	4,39	43750	87,50	87,50
71	0x47	4,45	44375	88,75	88,75
72	0x48	4,52	45000	90,00	90,00
73	0x49	4,58	45625	91,25	91,25
74	0x4A	4,64	46250	92,50	92,50
75	0x4B	4,71	46875	93,75	93,75
76	0x4C	4,77	47500	95,00	95,00
77	0x4D	4,83	48125	96,25	96,25
78	0x4E	4,89	48750	97,50	97,50
79	0x4F	4,96	49375	98,75	98,75
80	0x50	5,02	50000	100,00	100,00
81	0x51	5,08	50625	101,25	101,25
82	0x52	5,15	51250	102,50	102,50
83	0x53	5,21	51875	103,75	103,75
84	0x54	5,27	52500	105,00	105,00
85	0x55	5,33	53125	106,25	106,25
86	0x56	5,40	53750	107,50	107,50
87	0x57	5,46	54375	108,75	108,75
88	0x58	5,52	55000	110,00	110,00
89	0x59	5,58	55625	111,25	111,25
90	0x5A	5,65	56250	112,50	112,50
91	0x5B	5,71	56875	113,75	113,75
92	0x5C	5,77	57500	115,00	115,00
93	0x5D	5,84	58125	116,25	116,25
94	0x5E	5,90	58750	117,50	117,50
95	0x5F	5,96	59375	118,75	118,75
96	0x60	6,02	60000	120,00	120,00
97	0x61	6,09	60625	121,25	121,25
98	0x62	6,15	61250	122,50	122,50
99	0x63	6,21	61875	123,75	123,75
100	0x64	6,27	62500	125,00	125,00
101	0x65	6,34	63125	126,25	126,25
102	0x66	6,40	63750	127,50	127,50
103	0x67	6,46	64375	128,75	128,75
104	0x68	6,53	65000	130,00	130,00
105	0x69	6,59	65625	131,25	131,25
106	0x6A	6,65	66250	132,50	132,50
107	0x6B	6,71	66875	133,75	133,75
108	0x6C	6,78	67500	135,00	135,00
109	0x6D	6,84	68125	136,25	136,25
110	0x6E	6,90	68750	137,50	137,50
111	0x6F	6,96	69375	138,75	138,75
112	0x70	7,03	70000	140,00	140,00
113	0x71	7,09	70625	141,25	141,25
114	0x72	7,15	71250	142,50	142,50
115	0x73	7,22	71875	143,75	143,75
116	0x74	7,28	72500	145,00	145,00
117	0x75	7,34	73125	146,25	146,25
118	0x76	7,40	73750	147,50	147,50
119	0x77	7,47	74375	148,75	148,75
120	0x78	7,53	75000	150,00	150,00
121	0x79	7,59	75625	151,25	151,25
122	0x7A	7,65	76250	152,50	152,50
123	0x7B	7,72	76875	153,75	153,75
124	0x7C	7,78	77500	155,00	155,00
125	0x7D	7,84	78125	156,25	156,25
126	0x7E	7,91	78750	157,50	157,50

127	0x7F	7,97	79375	158,75	158,75
128	0x80	8,03	80000	160,00	160,00
129	0x81	8,09	80625	161,25	161,25
130	0x82	8,16	81250	162,50	162,50
131	0x83	8,22	81875	163,75	163,75
132	0x84	8,28	82500	165,00	165,00
133	0x85	8,35	83125	166,25	166,25
134	0x86	8,41	83750	167,50	167,50
135	0x87	8,47	84375	168,75	168,75
136	0x88	8,53	85000	170,00	170,00
137	0x89	8,60	85625	171,25	171,25
138	0x8A	8,66	86250	172,50	172,50
139	0x8B	8,72	86875	173,75	173,75
140	0x8C	8,78	87500	175,00	175,00
141	0x8D	8,85	88125	176,25	176,25
142	0x8E	8,91	88750	177,50	177,50
143	0x8F	8,97	89375	178,75	178,75
144	0x90	9,04	90000	180,00	180,00
145	0x91	9,10	90625	181,25	181,25
146	0x92	9,16	91250	182,50	182,50
147	0x93	9,22	91875	183,75	183,75
148	0x94	9,29	92500	185,00	185,00
149	0x95	9,35	93125	186,25	186,25
150	0x96	9,41	93750	187,50	187,50
151	0x97	9,47	94375	188,75	188,75
152	0x98	9,54	95000	190,00	190,00
153	0x99	9,60	95625	191,25	191,25
154	0x9A	9,66	96250	192,50	192,50
155	0x9B	9,73	96875	193,75	193,75
156	0x9C	9,79	97500	195,00	195,00
157	0x9D	9,85	98125	196,25	196,25
158	0x9E	9,91	98750	197,50	197,50
159	0x9F	9,98	99375	198,75	198,75
160	0xA0	10,04	100000	200,00	200,00
161	0xA1	10,10	100625	201,25	201,25
162	0xA2	10,16	101250	202,50	202,50
163	0xA3	10,23	101875	203,75	203,75
164	0xA4	10,29	102500	205,00	205,00
165	0xA5	10,35	103125	206,25	206,25
166	0xA6	10,42	103750	207,50	207,50
167	0xA7	10,48	104375	208,75	208,75
168	0xA8	10,54	105000	210,00	210,00
169	0xA9	10,60	105625	211,25	211,25
170	0xAA	10,67	106250	212,50	212,50
171	0xAB	10,73	106875	213,75	213,75
172	0xAC	10,79	107500	215,00	215,00
173	0xAD	10,85	108125	216,25	216,25
174	0xAE	10,92	108750	217,50	217,50
175	0xAF	10,98	109375	218,75	218,75
176	0xB0	11,04	110000	220,00	220,00
177	0xB1	11,11	110625	221,25	221,25
178	0xB2	11,17	111250	222,50	222,50
179	0xB3	11,23	111875	223,75	223,75
180	0xB4	11,29	112500	225,00	225,00
181	0xB5	11,36	113125	226,25	226,25
182	0xB6	11,42	113750	227,50	227,50
183	0xB7	11,48	114375	228,75	228,75
184	0xB8	11,55	115000	230,00	230,00
185	0xB9	11,61	115625	231,25	231,25
186	0xBA	11,67	116250	232,50	232,50
187	0xBB	11,73	116875	233,75	233,75
188	0xBC	11,80	117500	235,00	235,00
189	0xBD	11,86	118125	236,25	236,25
190	0xBE	11,92	118750	237,50	237,50
191	0xBF	11,98	119375	238,75	238,75
192	0xC0	12,05	120000	240,00	240,00

193	0xC1	12,11	120625	241,25	241,25
194	0xC2	12,17	121250	242,50	242,50
195	0xC3	12,24	121875	243,75	243,75
196	0xC4	12,30	122500	245,00	245,00
197	0xC5	12,36	123125	246,25	246,25
198	0xC6	12,42	123750	247,50	247,50
199	0xC7	12,49	124375	248,75	248,75
200	0xC8	12,55	125000	250,00	250,00
201	0xC9	12,61	125625	251,25	251,25
202	0xCA	12,67	126250	252,50	252,50
203	0xCB	12,74	126875	253,75	253,75
204	0xCC	12,80	127500	255,00	255,00
205	0xCD	12,86	128125	256,25	256,25
206	0xCE	12,93	128750	257,50	257,50
207	0xCF	12,99	129375	258,75	258,75
208	0xD0	13,05	130000	260,00	260,00
209	0xD1	13,11	130625	261,25	261,25
210	0xD2	13,18	131250	262,50	262,50
211	0xD3	13,24	131875	263,75	263,75
212	0xD4	13,30	132500	265,00	265,00
213	0xD5	13,36	133125	266,25	266,25
214	0xD6	13,43	133750	267,50	267,50
215	0xD7	13,49	134375	268,75	268,75
216	0xD8	13,55	135000	270,00	270,00
217	0xD9	13,62	135625	271,25	271,25
218	0xDA	13,68	136250	272,50	272,50
219	0xDB	13,74	136875	273,75	273,75
220	0xDC	13,80	137500	275,00	275,00
221	0xDD	13,87	138125	276,25	276,25
222	0xDE	13,93	138750	277,50	277,50
223	0xDF	13,99	139375	278,75	278,75
224	0xE0	14,05	140000	280,00	280,00
225	0xE1	14,12	140625	281,25	281,25
226	0xE2	14,18	141250	282,50	282,50
227	0xE3	14,24	141875	283,75	283,75
228	0xE4	14,31	142500	285,00	285,00
229	0xE5	14,37	143125	286,25	286,25
230	0xE6	14,43	143750	287,50	287,50
231	0xE7	14,49	144375	288,75	288,75
232	0xE8	14,56	145000	290,00	290,00
233	0xE9	14,62	145625	291,25	291,25
234	0xEA	14,68	146250	292,50	292,50
235	0xEB	14,75	146875	293,75	293,75
236	0xEC	14,81	147500	295,00	295,00
237	0xED	14,87	148125	296,25	296,25
238	0xEE	14,93	148750	297,50	297,50
239	0xEF	15,00	149375	298,75	298,75
240	0xF0	15,06	150000	300,00	300,00
241	0xF1	15,12	150625	301,25	301,25
242	0xF2	15,18	151250	302,50	302,50
243	0xF3	15,25	151875	303,75	303,75
244	0xF4	15,31	152500	305,00	305,00
245	0xF5	15,37	153125	306,25	306,25
246	0xF6	15,44	153750	307,50	307,50
247	0xF7	15,50	154375	308,75	308,75
248	0xF8	15,56	155000	310,00	310,00
249	0xF9	15,62	155625	311,25	311,25
250	0xFA	15,69	156250	312,50	312,50
251	0xFB	15,75	156875	313,75	313,75
252	0xFC	15,81	157500	315,00	315,00
253	0xFD	15,87	158125	316,25	316,25
254	0xFE	15,94	158750	317,50	317,50
255	0xFF	16,00	159375	318,75	318,75

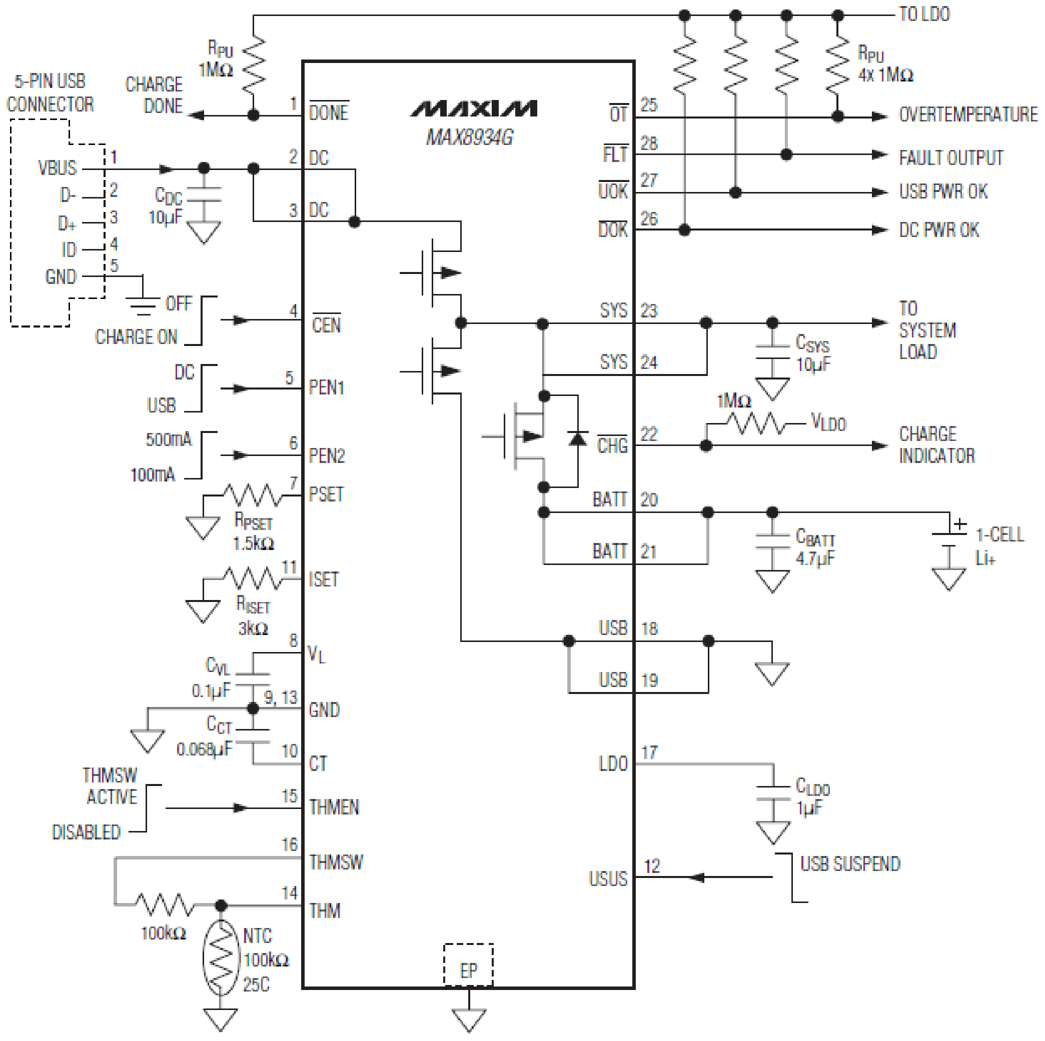
APPENDIX 4 Accelerometer tap detection values (Arponen 2014.03.31)

APPENDIX 5: MSP430 PIN CONFIGURATION



APPENDIX 5 MSP430 Pin configuration (TEXAS INSTRUMENTS, 2012.)

APPENDIX 5: MAX8934 PIN CONFIGURATION



APPENDIX 6 MAX8934 Pin configuration (MAXIM INTEGRATED, 2010).