S.N.A.K.E.: A Dynamically Reconfigurable Artificial Sensate Skin

by

Gerardo Barroeta Pérez

B.Sc. Electrical Engineering
Instituto Tecnológico y de Estudios Superiores de Monterrey (2002)

Submitted to the Program in Media Arts and Sciences,
School of Architecture and Planning,
in partial fulfillment of the requirements for the degree of

Master of Science in Media Arts and Sciences

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2006

© Massachusetts Institute of Technology 2006. All rights reserved.

Author____
Program in Media Arts and Sciences
August 11, 2006

Certified by______________________________________
Joseph A. Paradiso
Associate Professor of Media Arts and Sciences
Program in Media Arts and Sciences
Thesis Supervisor

Accepted by______________________________________
Andrew B. Lippman
Chair, Departmental Committee on Graduate Students
Program in Media Arts and Sciences

S.N.A.K.E.: A Dynamically Reconfigurable Artificial Sensate Skin

by

Gerardo Barroeta Pérez

Submitted to the Program in Media Arts and Sciences,
School of Architecture and Planning,
on August 11, 2006, in partial fulfillment of the
requirements for the degree of
Master of Science in Media Arts and Sciences

Abstract

The idea of an *Artificial Sensate Skin* device that mimics the characteristics and functions of its analogous living tissue whether human or animal is not new. Yet, most of the current related work has been focused in the development of either materials, flexible electronics or ultra-dense sensing matrices and Wide Area Sensor Networks. The current work describes the design and implementation of a new type of Artificial Sensate Skin. This Artificial Sensate Skin is implemented as a low-power, highly scalable and mechanically flexible Wired Sensor Network. This Skin is composed of one or many *Skin Patches* which in turn are composed of one or many *Skin Nodes*. Each node is able to measure Strain, Pressure, Ambient Light, Pressure, Sound and Mechanoreception. Each Skin Patch can either work as a stand-alone device or as a data extraction device if this is attached to a Personal Computer through a different type of device referred to as *Brains*. Each Skin Node and therefore each Skin Patch is *Dynamically Adaptable* meaning that they can adapt to external stimuli by either modifying their behavior or by completely changing their code. Construction of a sensate skin in such a modular fashion promises intrinsic scalability, where peer-to-peer connections between neighbors can reduce local data, which can then be sent to the brain by the high-speed common backbone. The current project also involves the design and implementation of the software components needed; these include a PC Graphical User Interface, application software and the firmware required by the embedded microcontrollers. Results show that needed resources like bandwidth are greatly minimized because of the addition of embedded processing power. Results also indicate that the platform can be used as a scalable smart material to cover interactive surfaces, or simply to extract a rich set of dense multi-modal sensor data.

Thesis Supervisor: Joseph A. Paradiso
Title: Associate Professor of Media Arts and Sciences, Program in Media Arts and Sciences

S.N.A.K.E.: A Dynamically Reconfigurable Artificial Sensate Skin

by

Gerardo Barroeta Pérez

The following people served as readers for this thesis:

Thesis Reader____

V. Michael Bove
Principal Research Scientist
Program in Media Arts and Sciences

Thesis Reader____

Cynthia Breazeal
Associate Professor of Media Arts and Sciences
MIT Media Lab

Acknowledgments

To **Joe Paradiso** for giving me the opportunity of living two of the best years of my life. For his unquestioning support, flawless technical advice and invaluable friendship.

To **V. Michael Bove**, J. and **Cynthia Breazeal** for providing the necessary extra pairs of critical eyes without which this work would not have been what it is.

To my colleagues: **Ryan Aylward, Matt Laibowitz, Mark Feldmeier, Ari Benbasat, Matt Malinowski and Josh Lifton**, for their huge help throughout these years.

To all of the Media Lab, especially to **Linda Peterson** for helping me through difficult times and to **Lisa Lieberson** for making things happen.

To my mother for making me what I am. To my father for showing me the path. To my sister for always being there. To my grandparents for constantly reminding me to enjoy the really important things in life.

To my friends a very special thanks, you were the my family away from home. It was only because of you that these two years were the best. Thank you **Rogelio, Ricardo, David, Javier, Pepe, Federico, Fernando, Christos, Rocio, Sendi, Vik, Ernesto**, and countless others with which I shared one of many, many great moments.

To everybody at the **Graduate Student Council** for giving me so much for so little. Thank you Emmily, Sylvain, Emmi, Shan and everybody else at the Orientation Committee.

To my Media Lab colleagues, **Ernesto Arroyo, Ernesto Martinez, Ayah, Vincent**, and many others for their friendship, sense of humor and brilliant discussions.

To **Pioneer Circuits** for their fabrication expertise

And finally,

To **Telmex**, for making it possible.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

"Imagination is the beginning of creation. You imagine what you desire, you will what you imagine and at last you create what you will. "

– *George Bernard Shaw*

The integumentary system or in other words, our skin is perhaps one of the most versatile organs in the human body. Our skin is not only a protective layer that shields us from the harshness of the environment, it has many other functions that enable us to survive and communicate. These functions include: Interaction with surrounding environment; Regulation of body temperature; Storage of energy as fat; Excretion of body waste through sweat; Absorption of substances like medicines and oxygen and finally, Alert through multimodal sensations and several forms of physical pain[72].

The human skin is also the largest organ in the human body with the average person having over 20 square feet of skin. From a mechanical point of view it is an amazing material: it is able to regenerate itself from wear and tear (cuts), it is flexible, contractible and expandable and it is able to withstand erosion, mechanical stress and swift temperature changes like few other known materials [20],[36].

From a sensing perspective, the human skin is also quite remarkable. By using different kinds of receptors, it is able to sense touch, vibration, pressure, shear, stretching, temperature (hotness or coolness), mechanoreception by using hair, and several different forms of physical pain[56]. In summary, our skin is much more than a protective surface as it allows us to interact with other people as well as with our physical environment.

If we take into account the aforementioned characteristics and functions of the human skin, it comes as small wonder why people have been trying to emulate it as close as possible. The development of an Artificial Sensate Skin makes economical sense if we consider the huge amount of its potential applications. Some may be evident at first, like using it to increase the autonomy of robots or monitoring a physical quantity in a scientific experiment; others might require more thought, yet the possibilities are endless. Everything from space exploration and aircraft control, to motion capture, patient care, surveillance and even recreation will benefit from a device that closely resembles our skin[43].

1.1 Problem Definition and Motivation

The idea of an Artificial Sensate Skin is anything but new; the literature containing work related to the fields touched on this Thesis, namely Sensor Networks, their associated protocols and Artificial Skins, is vast and broad. It covers the entire span of the technological abstraction layers from microfabrication to software simulations. However, two general trends can be observed as described by Paradiso in [50]: Either they are implemented as dense sensing matrices where individual sensors have to be individually routed to a concentrator or *Central Processing Unit*, or as completely de-centralized Sensor Networks.

This has led to the development of either sensor technologies that are ultra dense in nature such as capacitive fabrics or densely packed VLSI sensor matrices (like CCDs), or extremely distributed like Smart dust that cover areas as large as a city or even more. In the former case, bandwidth can become a problem because of the immense amount of data generated. This kind of sensor must have a dedicated processing unit to be able to operate the sensor matrix. In the second case, because of the distance between nodes and the large areas involved, they are very difficult to use in applications where the stimuli of interest is restricted to smaller areas, probably not larger than a room.

The current project, named S.N.A.K.E Skin or *Sensor Network Array Kapton Embedded* Skin, attempts to attack these problems by having a mixture of both worlds. To do this, two things are done: first, each node is given processing power so that data can be processed locally; second, nodes are brought much closer together, just about enough so that they can be connected to each other to create a skin-like surface that can react to the same kinds of stimuli that our skin encounters.

Furthermore, the addition of processing power enables each node to calculate higher order features, so that if a central unit like a PC needs to extract data from the network, these higher order parameters, like shadow forms and pressure gradients, can be transferred instead of raw data, which saves network resources.

Therefore, the statement of the project goal can be now stated as follows:

To explore and develop a deeper understanding of the different design approaches and technical constraints involved in the fabrication and implementation of a Sensor Network that emulates some of the sensory characteristics and functionality of the Human Skin.

1.2 Synopsis

This work is divided into six chapters. The current chapter gives a general introduction by providing the motivation behind the project, the problem that it is trying to attack, and a general overview of the system developed for such a purpose.

Chapter 2 provides some background work. This chapter will briefly talk about various technologies, projects and ideas in this area that ground the current project.

Chapter 3 describes in detail the Hardware system designed for the project. This chapter is divided into four sections. The first one outlines the design directives followed throughout the entire design of the project. The second one talks about the topology of the S.N.A.K.E. Sensor Network. The third section describes the design of the substrate used for the nodes. The last section describes the design of the Brains.

Chapter 4 talks about the different Software components that had to be developed. This chapter is subdivided into three sections. The first one details the specifics of the Firmware components developed for the embedded processors. The second section describes the PC software that works as a front end to manage the Skin Patches. Finally the last section talks about the details of the communications software.

Chapter 5 details the Results generated by the entire project. This chapter is also subdivided into three main sections. The first one gives the mechanical results of the hardware platform and describes how well they performed. The second presents all the results of the different sensing modalities and modes of operation of the Skin Patches, along with sensor performance and data plots. The last section gives a set of performance metrics under which the entire project performance was judged.

The last part, **Chapter** 6, gives the conclusions derived from the Results Chapter, while also giving some suggestions for possible applications and future work.

1.3 System Overview

Now, a brief system overview of the entire project is provided to give a broad idea of the different components that will make up the S.N.A.K.E. Skin system. For a more detailed treatment of the different components, the reader is advised to read Chapters 3 and 4.

Based on the problem definition and motivations set up before, the current work will therefore focus on the design and implementation of a novel type of sensor network that inherits a subset of the characteristics of the human skin. As such, this Artificial Sensate Skin, will be implemented in a *flexible substrate* as a dense, low-power, and dynamically adaptable Wired Sensor Network. Its design also contemplates that it will be highly scalable and portable, so that it can be formed and "grown" to cover surfaces of different shapes and sizes.

Hardware

Each *Skin Patch* is composed of one or many *Skin Nodes*, each of which is capable of sensing six different physical quantities. Each node is also given processing power, a direct connection to its immediate neighbors, and a connection to a fast backbone bus. Finally, nodes are capable of actuating through the use of an on-board, multi-color LED. This means that each node is an independent and autonomous entity within the Sensor Network.

Data extraction and network management is done through a different entity: the *Brains*. These devices ensure the reliability of the network and deal with administrative tasks like network re-programming, data collection, and overall synchronization. They also arbitrate the use of the backbone bus.

Software

Several different Software components were also created to control the network. First, since each node is capable of changing its behavior by changing its code, application software and a simple Operating System were created. Second, application firmware was developed to indicate the nodes when and how to obtain data from its sensing channels. Finally, a graphical front end was be developed so that a personal computer can be used to gather data and control the network.

Chapter 2

Artificial Sensate Skins

"We stand on the shoulders of giants."

– *Bernard of Chartres*

Following the Introduction, the current chapter talks about several previous projects that in some way or another, inspired or contributed to the idea on which this project is based. Given the Sensor Network nature of the project, the design choices draw heavily upon several different types of technologies that are briefly outlined in this chapter. The first section highlights some of the similarities and differences of several types of Sensor Networks. These were subdivided into two different types according to the level of integration of their sensing elements into *Sensor Networks as Sensate Skins* and *Ultra-dense Sensor Arrays*. The second section talks about other technologies that are also related to this project but had much less influence. These are mentioned so that the current project can be then established within the broader context of its technological field.

2.1 Artificial Sensate Skins

The concept of "Artificial Sensate Skins" refers to a very specific type of device that in some way or another, try to copy either the functionality or appearance (or both) of its analogous living tissue. Particularly in the projects from the Media Lab's Responsive Environments Group, these devices are generally implemented as a Sensor Networks in which several smart sensing elements or nodes are capable of extracting information from the environment.

Although there have been several projects that have attempted to to this, their approaches have been extremely different from each other, the main difference being the amount of integration between sensing elements, and their density. Let us now describe each one of them in detail, so that their relationship to the current project can be better understood.

2.1.1 Sensor Networks as Sensate Skins

The main characteristic of these devices is that the level of integration of their sensors and their density is not as high as with Dense Sensor Arrays but it is much greater than for typical Wireless Sensor Networks. Much of the work related to these Artificial Sensate Skins was either pioneered or heavily developed by faculty and students at the MIT Media Lab. In fact there are three projects that are very closely related to the one presented in this document, and therefore, not only served as major sources of inspiration, but were valuable sources of experience from which this project drew from.

This section will then attempt to summarize the contributions of these three projects along with a selected few from outside of the Media Lab by emphasizing their relative strengths and weaknesses compared to the proposed ongoing work.

Artificial Sensate Skins at the MIT Media Lab

Paintable Computer: Perhaps the starting point of the big interest in Artificial Skins and dense Sensor Networks at the Media Lab, was the work of William Butera[6]. In his Ph.D thesis, Butera presents a new programming paradigm for ultra-dense and ultra-miniaturized computing particles, called *Paintable Computing.* He defines Paintable Computing as: "an agglomerate of numerous, finely dispersed, ultra-miniaturized computing particles; each positioned randomly, running asynchronously and communicating locally". In other words, he

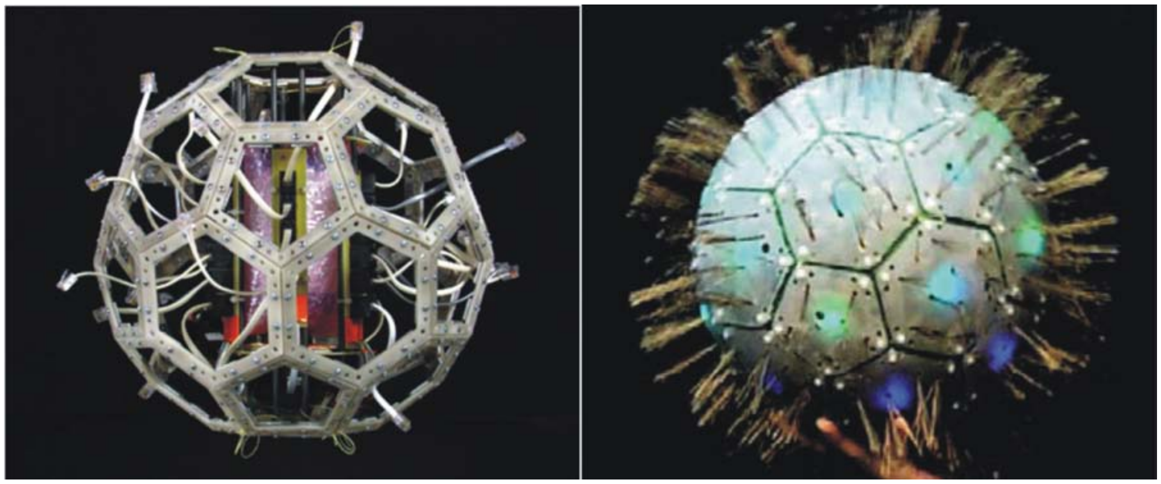


Figure 2-1: Created by MIT Media Lab's Responsive Environments group, Tribble is a Distributed Sensor Network that works like an Artificial Sensate Skin

presented a plausible processing and communications method for programming an otherwise seemingly chaotic computing platform, and therefore enabled much of the future work that would be developed by other groups at the lab.

Tribble: Based on Butera's work, and perhaps the most relevant to the work described here was developed at the Responsive Environments Group by Lifton[41] with the name Tribble or *Tactile Reactive Interface Built By Linked Elements.* Tribble is a dense, multi-modal and peer-to-peer Sensor Network assembled as a plastic sphere that is made up of tiles or Skin Patches resembling a soccer ball. Tribble is shown on Figure 2-1. This was

Each of its nodes is capable of routing data to its neighbors, and it was created as a completely decentralized Sensor Network. Moreover, each one of the nodes is capable of processing its own generated data without the need of a centralized processing unit. It has however, several design characteristics that make it somewhat inflexible and difficult to adapt. First of all, its shape is fixed to that of a sphere-shell-like object, which obviously prevents its from being used to wrap arbitrarily shaped objects.

The second design problem is that even its smaller node is somewhat large, which means that it is difficult to achieve high sensing density. It is also power hungry, heavy and stiff; yet Tribble not only set the standard by which this project is measured, but it was also a

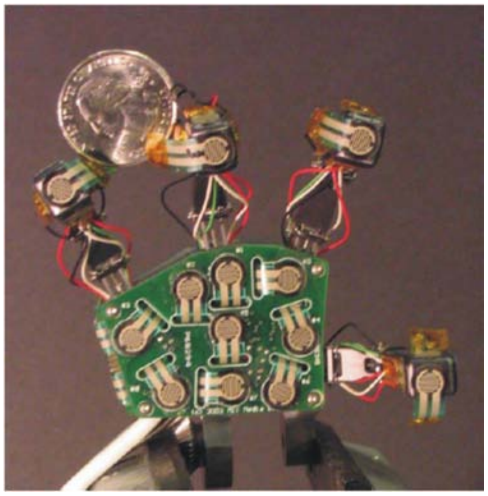


Figure 2-2: Leonardo's Hand

great hardware test-bed in which some of the already existing sensor network algorithms were tested. In short, Tribble is the foundation of the current project and in fact many of the design characteristics that will be outlined in Chapter 3 section will heavily draw upon the experience gained from this project.

Sensate Skin: Another Media Lab project that relates to this research was the work done by Daniel Stiehl regarding Sensate Skins for robots[65]. In his Masters Thesis, Stiehl describes a method to give a robot the ability to sense pressure through a silicon skin. Although the goal of his work is similar to that of the current project, Sensate Skin was more concerned with the mechanical and morphological aspects of the skin, so that this could be adapted to the body parts that make up their test robot Leonardo. It can also only sense pressure, and every sensor must be routed to a central processing unit. This can be seen in Figure 2-2.

Pushpin Computing: Also by Lifton [40],[39], at the Responsive Environments group, a Sensor Network platform with the name of *Pushpin Computing* was created. Its intention was to approach the sensor network world from a ground-up perspective by making a testbed with over a hundred peer-to-peer wireless sensor nodes freely distributed across a table-top as opposed to the more typical top-down approach proceeded by software simulations. The

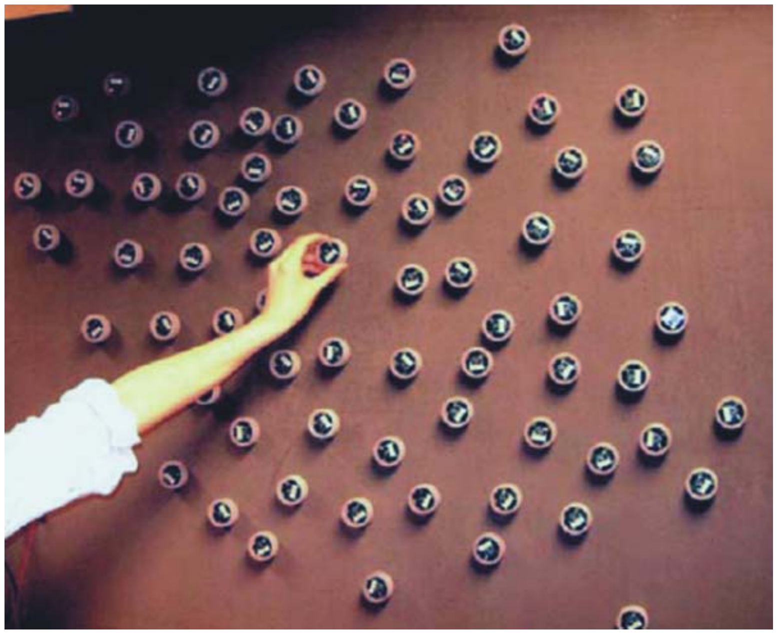


Figure 2-3: Pushpin Computing Platform

Pushpin computing platform is presented in Figure 2-3

This project however, is quite different from that proposed in the current work as it is centered on Sensor Networks and the possible algorithms that can run on the developed platform, rather than on Artificial Sensate Skins. It is also constrained to a fixed rigid substrate and nodes are not directly connected to each other. Still, it is relevant in the sense that it has been a successful attempt to create a dense distributed computing platform in which real applications like shadow tracking and accounts time of flight have been implemented.

Although the S.N.A.K.E. Skin project is not intended to work entirely as a Distributed Sensor Network like the Pushpin platform did, it does attempt to combine the advantages of distributed and centralized processing, so the Pushpin platform proved to be an invaluable source of experience from which the new platform greatly benefited.

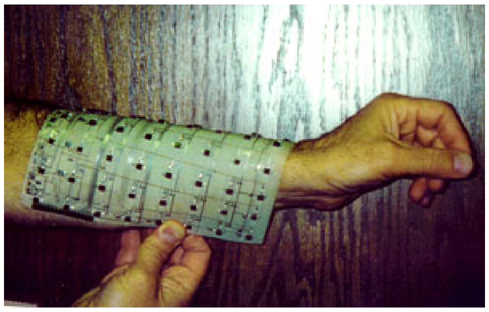
Artificial Sensate Skins elsewhere

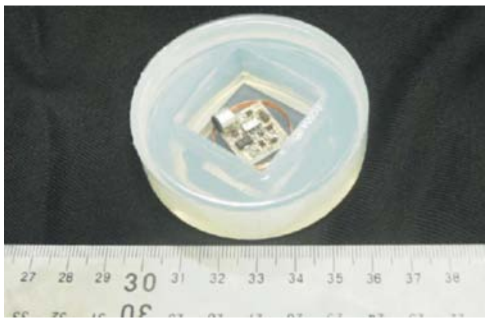
The MIT Media Lab is not the only institute interested in such ideas though, and several other similar projects have been created by people outside of the Lab. But perhaps the main supporter of this concept is Dr. Vladimir J. Lumelsky from the University of Wisconsin-Madison. His 2002 article "Sensitive Skin" [43] is essentially a call-to-arms to generate interest in the scientific community, so that devices with characteristics similar to those of the human skin are created. To demonstrate their idea, they created a prototype of a Skin Patch with infrared lights and receivers that were used as proximity sensors. The prototype presented in the article is shown on Figure 2-4(a).

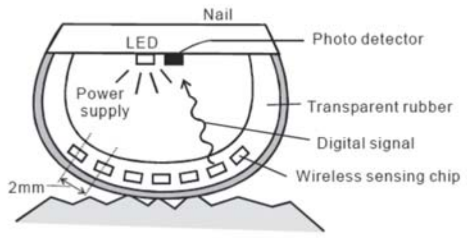
This is not the only recent work outside of the Media Lab though, as many others have attempted different ways to recreating our sense of touch[37][45]. Some of the most relevant works are mentioned in the following list and they can be seen in Figure 2-4.

- Hakozaki[17] proposes a method to create a flexible robot skin to cover wide robot surfaces (fig. 2-4(b)).
- Yamada[75] shows a way of detecting deformations in a rubber membrane by optical means (fig. 2-4(c)).
- Hritsu[22] proposes a similar way of tracking deformations in a finger-like structure (fig. 2-4(d)).
- D. Um presents in [69] a very similar Sensitive Skin to Tribble, however it lacks local processing power (fig. 2-4(e)).
- Rekimoto introduces a capacitive "Smart Skin" sensor in [58] for its use in interactive surfaces (fig. 2-4(f)).

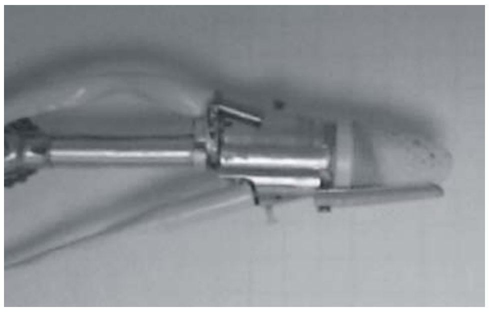
Unlike the work presented earlier, all of these examples rely on a centralized processing unit to process the data extracted from the sensors.

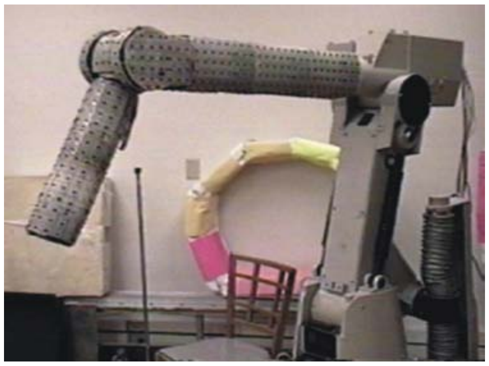

(a) Sensitive Skin by Lumelsky


(b) Flexible robot skin by Hakozaki



(c) Tactile sensor by Yamada


(d) Tactile sensor by Hritsu


(e) Modularized Sensitive Skin by D. Um

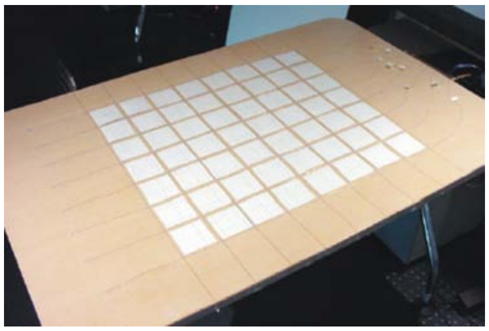

(f) Smart Skin sensor by Rekimoto

Figure 2-4: Dense sensor arrays as artificial skins

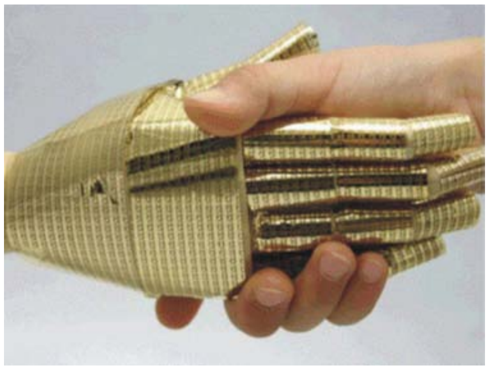
2.1.2 Ultra-dense Sensor Arrays

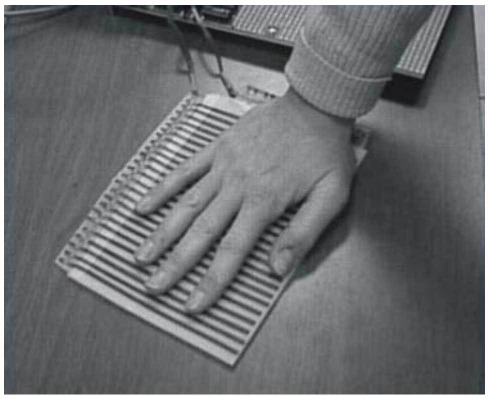
This second category of Artificial Sensate Skins differs from the previous one because of their ultra-high sensor density. The other main difference is that these are also never implemented as sensor networks but rather as sensor matrices. This means that each sensor must be individually routed to a central processing unit. There are numerous works that have developed high-density sensing arrays for their use as Artificial Skins some of which are presented in the following list:

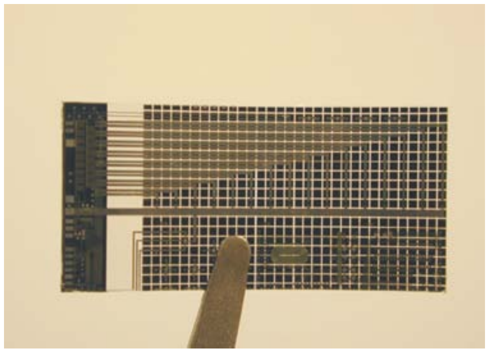
- Takao Someya in [62] presents a large-area flexible sensor matrix with organic field effect transistors (fig. 2-5(a)).
- M. Sergio describes a textile-based capacitive sensor array that can be used as sensitive skin[44] (fig. 2-5(b)).
- Yong Xu, reports an IC-integrated flexible sensor skin that can sense shear stresses for a possible aerospace application[74] (fig. 2-5(c)).
- Papakostas talks about a large-area printable force sensor for smart skin applications based on a piezoresistive effect[49] (fig. 2-5(d)).
- Engel and others[11] talk about a multi-modal flexible skin fabricated with polymer micro-machining methods (fig. 2-5(e)).
- Darren Leigh presents DiamondTouch, a technology to create table-top input devices based on capacitive sensing (fig. 2-5(f)).

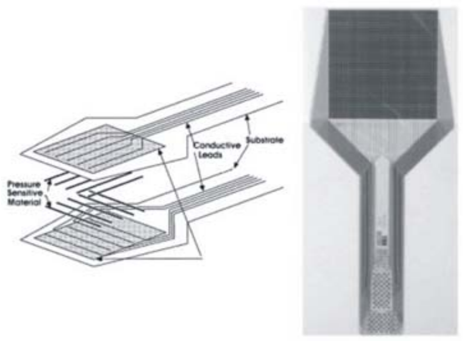
The problem with these approaches though is that as sensor density increases, so does the bandwidth required to retrieve information from the sensors. Moreover, a central processing unit is forcefully required because the sensors can't do anything by themselves. In spite of this, these devices are currently the ones that have the greatest physical and functional resemblance to our skin.

It is also necessary to say that there are several other projects with similar characteristics, however, the previous list only presents a few as a way to exemplify them.

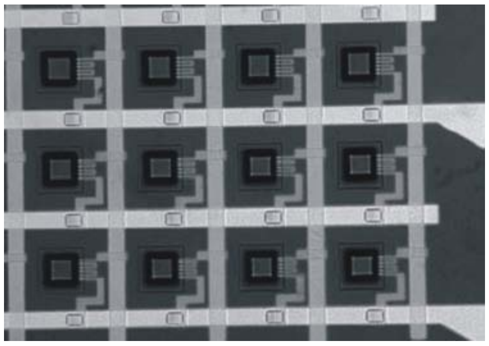

(a) Sensor Matrix by Someya

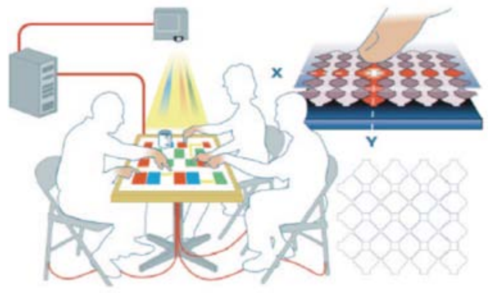

(b) Capacitive-based textile sensor by Sergio M.


(c) IC-integrated flexible skin sensor by Xu



(d) Large-area force sensor by Papakostas


(e) Engel's Multi-modal flexible skin fabricated with polymer micro-machining techniques



(f) Table-top capacitive input device by Leigh

Figure 2-5: Dense sensor arrays as artificial skins

2.2 Flexible Circuits

Although, not exactly Artificial Skins, the subject of flexible circuits is relevant because of two main reasons: first, most of the reference works presented before use some kind of flex circuit technology, because of its ability to bend and in some cases stretch, in in analogy to animal skin. Second because this project will be based on a Sensor Network implemented in flexible nodes fabricated with this technology.

The concept of flexible circuits is not new, actually the concept dates back to 1904[13]. Yet, in spite of their favorable physical characteristics, it wasn't until recently that the use of Flex circuits became more widespread, mainly because the level of integration in today's electronic devices would be impossible without its use. This section is subdivided into two parts. The first one gives a broad overview of the elements of Flex circuit technology. The second one describes other similar technologies that have been recently proposed.

2.2.1 Flex Technology

First of all, what exactly is Flex Technology? From the Handbook of Flexible Circuits by Ken Gilleo[13]: "Flexible circuitry is a patterned array of conductors supported by a flexible dielectric film."

This of course is a very broad definition but it generally refers to any circuit patterned on top of a flexible substrate. It can be as simple as a cable or as complex as the motherboard of a portable computer with several layers made up of different materials. They all however, share some common characteristics, like the thinness of the dielectric materials used for their construction (about $12\mu m$) and their remarkable ability to flex and bend.

The construction of flexible circuits is very different from typical rigid circuits because many different materials can be used for their construction. The following list shows the components that go into the construction of flex circuit boards:

Dielectric Base Film: this is the most important component of a flexible circuit because it will determine its final mechanical, electrical and chemical properties. These are usually very thin layers of a heat-resistant polymer which serves as the circuit substrate

Conductors: since the mechanical properties of flex circuits are determined by the materials they are built with, several different metal alloys can be used if different mechanical properties of the typical copper films are desired.

Adhesives: if the metal is not electro-deposited onto the substrate, adhesive layers must be used to bond the conductor films to the backing base films.

Platings and Finishes: sometimes conductors are treated with special finishes to keep the conductors from oxidizing and maintain good solderability. Gold platings are also common to increase the overall performance of the conductors.

Protective Surfaces: the majority of circuits are usually covered with protective layers to insulate the conductors from the environment. Other covering finishes like soldermasks are used to keep solder only in the specified areas. In the case of flexible circuits though, special covering layers have to be used so that these don't crack when the circuit is bent.

Material selection is not the only different thing between Flex and rigid circuits. Since Flexible circuits can be conformed to the shape of its container or even constantly flex[1], they are generally constructed very differently than their rigid counterparts. Some of the construction types are shown in Figure 2-6[2]

These are only a few of the vast amount of fabrication options available. In fact, every manufacturer offers specialized capabilities that can be used to further enhance the functionality of the Flex circuits. Chapter 3 will describe in more detail some of the other fabrication options and the details of the design choices used for the current project.

[1]This is generally referred to as Dynamic Flex

[2]Extracted from Minco's design guide

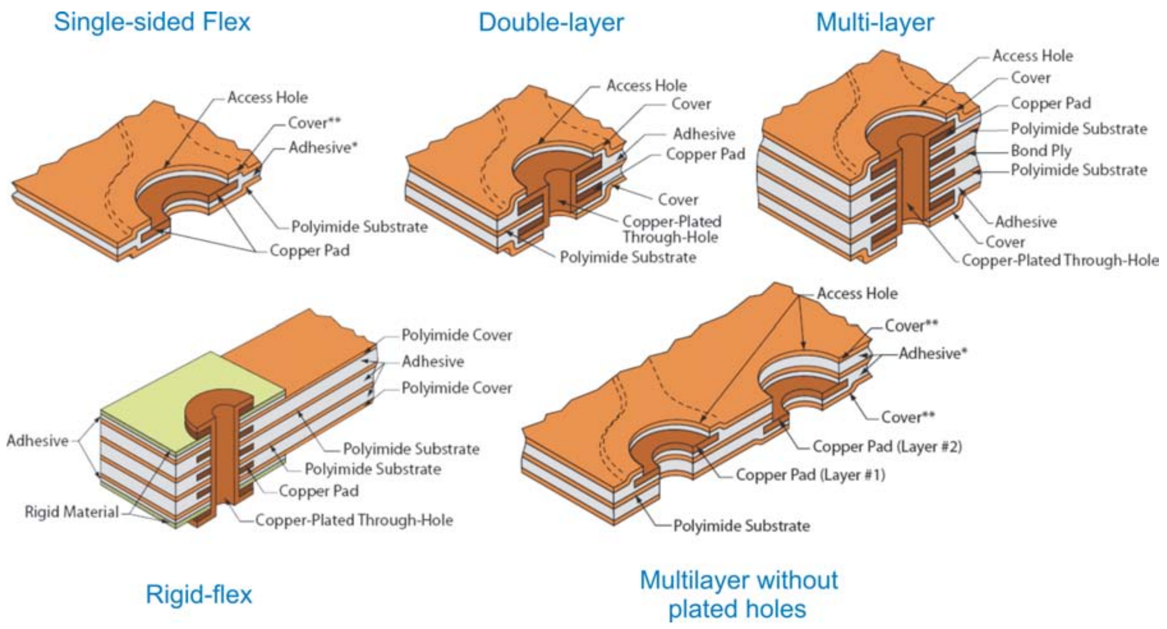


Figure 2-6: Types of Flex circuits[46]

2.2.2 Other Technologies

Flex technology is not the only form of flexible circuitry, however. There other types of flexible circuits that have been developed for other specific applications, and could be applied to develop an Artificial Sensate Skin.

The most prominent example is perhaps the work done by Stéphanie Lacour[35]. By depositing an ultra-thin gold layer on top of a PDMS, substrate she has created a new conducting material that can be stretched much like our own skin.

Other types of flexible circuits are much less related to the area of Artificial Skins like Jacobson's E-paper[33] and printable electronics, but the technologies used to create them might one day also be applied to fabricate an Artificial Sensate Skin. These projects though, are still under research and thus would be better suited for future applications.

Finally, two special issues of IEEE Proceedings (July and August 2005) were devoted to Flexible Electronics.

Chapter 3

Hardware

"Inanimate objects can be classified scientifically into three major categories; those that don't work, those that break down and those that get lost.".

– *Russell Baker*

The current chapter describes the hardware system which is the foundation of the project. The System is based on a set of design directives which are outlined first so that the reader can have a better understanding of the design choices made throughout this and upcoming chapters. Once these directives are mentioned, a thorough description of the hardware components along with their selection criteria will be provided. The chapter ends with a brief description of the three different communications channels used by the three hardware devices that make up the system: the *Skin Network*, Network Hubs or *Brains* and an associated Windows™based Personal Computer.

3.1 Design Philosophy

As it was mentioned in the previous chapter, the main goal of this project is to create a Sensor Network that emulates some of the functions and mechanical characteristics of the human skin. The following design principles thus provided the criteria for the selection of materials and electrical components with which the project elements were created:

Scalability: crucial for covering large surfaces, especially if the node size is small.

High Sensor Density: necessary to achieve high sensor resolution.

Sensing Variety: to allow a broader range of applications and to enable the study of the impact that different sensor types would have on available resources.

Robustness: given that the Artificial Sensate Skin was implemented as a Wired Sensor Network, reliability of network components and communications links are essential for good operation of the system.

Low Maintenance: this is so that the Artificial Skin can operate as a "black box" device and to keep the administrative tasks to a minimum.

Mechanical Flexibility: a key point in the design of the Artificial Skin was to make it flexible so that it can be bent and wrapped around arbitrarily shaped objects.

Ultra-low Power Consumption: ultra-low power is also highly desirable, given that as the network grows, the number of network nodes grows quadratically and so potentially does power consumption.

Small Node Footprint: goes hand by hand with high sensor density; the smaller the footprint with the same number of sensors per node, the higher the density and therefore resolution for a given covered area.

It is important to note that even though the *Skin Network* contains a high speed serial communications backbone, "high speed" is not mentioned as a main design point because of two reasons: first, one of the goals of the project *is* to prove that it is possible to limit the network traffic[1] by including processing power into every node in the Skin. And second, the project will also try to provide clues that will help to understand how the inclusion of

[1]And therefore reduce the overall resources needed to manage a large Skin Network

different sensing modalities will affect the necessary bandwidth that such a system would require.

3.2 Network Design

The high-level design for the Skin follows that of a Sensor Network based on a star network topology. Each spoke of the star is composed of a concentrator node or *Brain* that manages a *Skin Patch* made up of n nodes. The Skin Patches or sub-networks are based on a mesh topology with the addition of a backbone bus. Figure 3-1 shows the Skin Network Topology.

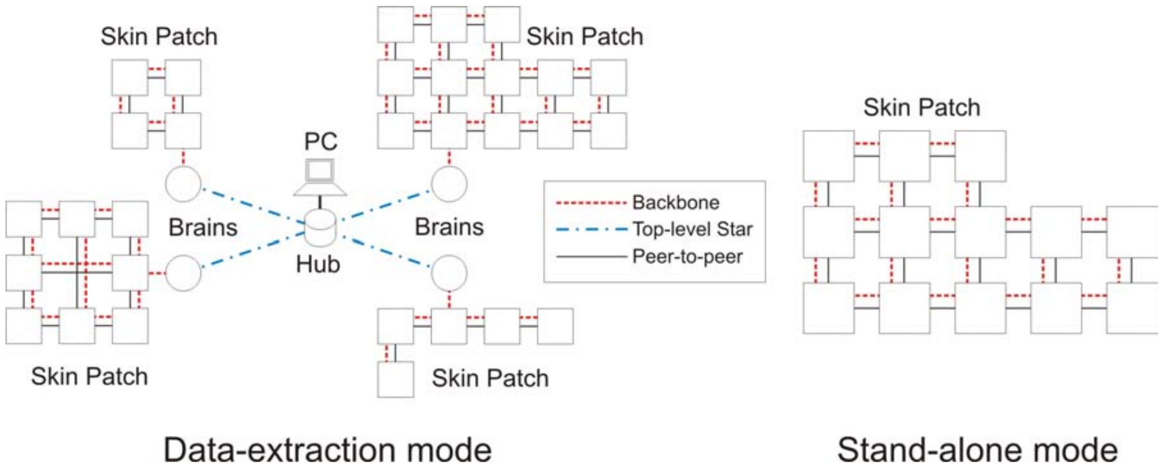


Figure 3-1: S.N.A.K.E. Skin Network Topology

This approach guarantees the scalability of the system at both network levels. Each patch of skin can be made of any arbitrary size[2] without affecting other patches, and at the same time multiple patches of skin can be connected together through their brains to a PC.

This in no way means that a Skin Patch can't be used without a PC connected or even a Brain. Having processing power in each node gives them the ability to process data locally without the need to relay it to a centralized processing unit as in the works presented in [69], [11] and [43]. This also means that skin patches can be used either connected to a

[2]Up to the limit of the backbone bus

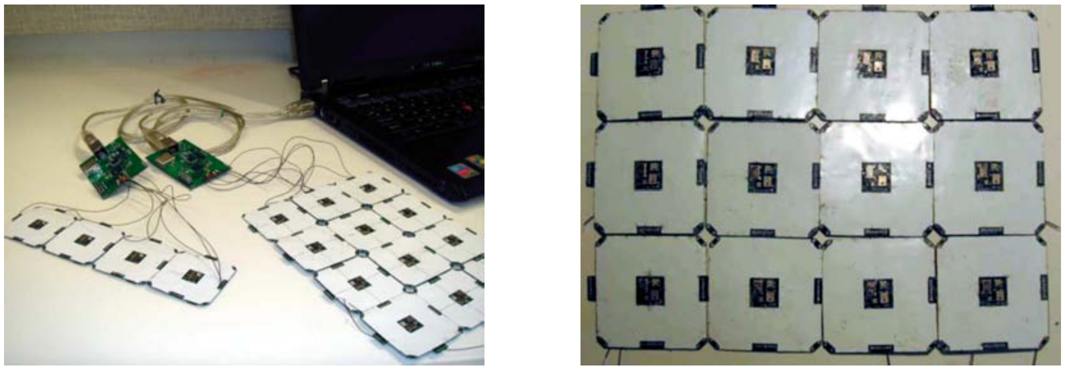

(a) Skin Patches as data extraction devices (b) Skin Patch in stand-alone mode

Figure 3-2: Skin Patch modes of operation

PC with a brain in a data extraction scenario, or by themselves as a stand-alone sensing-processing-actuation distributed platform. The former may be useful to monitor pressure gradients on an airplane's wing, when this is tested inside of a wind-tunnel for example. The later would be better suited if the material is to be used as an interactive architectural material[3]. Figure 3-2 shows the two ways the Skin Patch Network can work.

We now describe the different communication links that the project contains. Although these links are designed according to the specific needs of the involved elements, many of the points described above are observed during the design of these communications channels. Reliability, for example, is a key issue because the integrity of the network depends on the stability of the communications links. For this reason, several redundant links are provided for the backbone network. Scalability is also observed and specific measures were taken to ensure this. We will come back to these points though when the links are individually described.

3.2.1 I2C Backbone

According to the previous section, the Skin Patches are equipped with a network backbone. There are of course many ways of implementing it, but which is the adequate for the project?

[3]More on this point is provided in the Actuation and Sound sensing sections

It is clear that a serial protocol must be used to avoid the high number of lines that a parallel port would require, but then again, which one is the right option? There are a multitude of serial protocols for embedded systems to choose from: SPI, I^{2}C , Microwire, UART, RS-232 and others. To make the right choice, we have to start from what we know:

- The Skin Patch network can be made up of tens up to a few hundred nodes.
- One of the goals of the project is to minimize bandwidth through local processing of data.
- If the skin is working as a stand-alone device, it would be desirable to use the backbone as a multi-master, multi-slave bus.
- If the skin is connected to a computer, it would be desirable to have only one master (the Brain) and multiple slaves (the nodes), depending on whether multiple patches are needed
- Number of communications lines should be minimized to decrease wiring complexity.

By taking this into consideration, we can easily discard SPI because it can't be used as a multi-master bus and needs four lines plus a common ground; we can also discard UART, RS-232, and Microwire, because they can't handle more than 2 devices without multiplexing. I^{2}C on the other hand is capable of doing everything outlined above and it can be run at up to 400Kbps in fast mode and up to 3.4Mbps in high-speed mode. Moreover, the chosen microcontroller has an I^{2}C hardware module so it does not even need to be implemented in software and it also requires only 2 lines: Serial Data (SDA) and Serial Clock (SCL).

Running I^{2}C at the higher speeds requires special considerations, however. The I^{2}C bus can allow multiple devices to be connected connected to the data lines, because it uses what is referred to as a *wired and*, which means that the lines can be pulled down, but not up to avoid short circuits. This is accomplished by using either open collector devices or by alternating the states of the connected devices between high-impedance and a permanent pull-down.

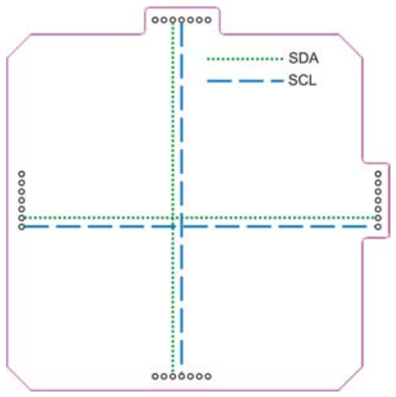


Figure 3-3: I^2C Routing pattern

The bus is then limited only by the rise and fall times specified by the I^2C standard [54], which also means that the bus is limited by its allowed maximum capacitance.

Since there are no devices pulling up the lines of the bus, this has to be done by pull-up devices, generally resistors that connect the bus lines to the power supply. Whenever the bus lines are unused, the bus idles by staying equal to the supply voltage. The problem with this situation though, is that when a device transitions either line from a logical 0 to a logical 1, it must release the line and let the pull-up device drive the line to the supply voltage; if this is done with a resistor, then the line will behave like an RC circuit with a typical exponential response. In other words, it will be slow.

To ensure the reliability goals and scalability that we mentioned before, we have to ensure that the bus can support as many devices as possible and that there are redundant connections so that communications are not lost. The first objective is ensured by routing both I^2C lines to all adjacent neighbors in a + shape shown in Figure 3-3[4]. The second objective, scalability is more difficult to guarantee because the I^2C standard dictates that there should not be more than 400pF in either bus line to ensure the correct operation of the bus. This puts a severe limit on the amount of devices and bus length that can be used. There are, however, two ways around this limitation: the first one requires the use of I^2C expanders like the Philips P82B715 and P82B96 chips, which increase this limit up

[4]Actual routing is shown on Appendix C

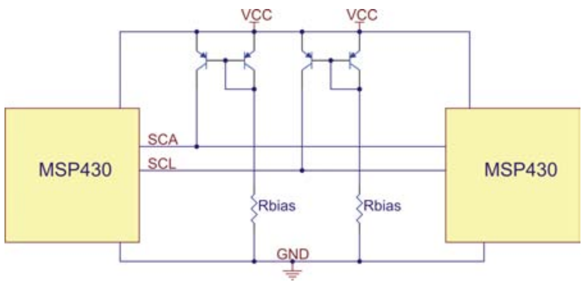


Figure 3-4: I^{2}C Active pull-up devices

to 4000pF; the second one involves using an active pull-up device. The first option involves adding more components to the nodes, which is something we want to avoid, but the second option involves only the modification of the pull-up devices, of which there is only one per patch

Active pull-up devices charge up the bus lines faster than a resistor. The most typical kind is a current-mirror. Current mirrors work as current sources injecting current into the bus lines and thus charging up the capacitance with a linear instead of inverse exponential response. The slope of this line is controlled by the source biasing and it can be made as fast as the current capacity of the transistors used will allow. The schematic of in Figure 3-4 shows this configuration.

This approach is used by the project, and it will be described in more detailed when the hardware of the Brains is mentioned; the actual gain in speed, however, will be shown in the Results Chapter. One last thing to mention is that whenever the skin is working in stand-alone mode, no active pull-ups are used because these are placed in the Brain. This could mean that if there is no Brain connected to the network, the backbone will not work. This is not the case, however, because each node provides a footprint to add a pull-up device if needed. Another option would be to designate a central node in the Skin Patch and add active pull-up devices to it. The backbone is not however meant for node to node communications, this is done through another communications channel which will be described next.

3.2.2 Peer to Peer

As we already mentioned, node to node communications has a different design. What is the reason for this and what kind of communications link is used? The inclusion of a communications link that allows each node to directly communicate to its adjacent immediate neighbors is inspired from the fact that cells in our skin do this to adjust their sensitivity to foreign stimuli. When cells are stimulated they generate an electrical pulse that informs our brain of the event, however when doing so, they release neurochemical transmitters that are received by neighboring cells. This is also used by our eyes to adjust to different amounts of light, and our noses when they become accustomed to certain smells[68],[32].

The addition of a communications link that allows nodes to have peer-to-peer communications is not then responding just to a technical need, but is in fact an emulation of a biological effect observed in living tissues. With this feature, it is expected that the Skin Nodes, will be capable of combining data sensed by their neighbors with that of their own, thus extracting more meaningful information about stimuli events. It will also decrease or eliminate traffic on the backbone bus for inter-node communications.

Peer-to-peer communications is realized by using a custom-created serial protocol similar to Microwire, which uses three lines plus a common ground to transfer information back and forth between nodes. These lines are Serial Transmit (STX), Serial Receive (SRX) and Flow Control (FC). Each node is then connected with three different lines to their four closest neighbors, needing a total of 12 interrupt-capable processor pins. More on the serial protocol will be mentioned in the Software chapter that follows. Also, the full routing of the nodes is presented in Appendix C, which show how these lines are connected.

3.2.3 USB PC link

The last communications link included in the project is a USB PC link. As we have seen before, scalability looms large in the design of the project. USB allows the top level of the network topology to keep with this design philosophy because of its inherent design. USB

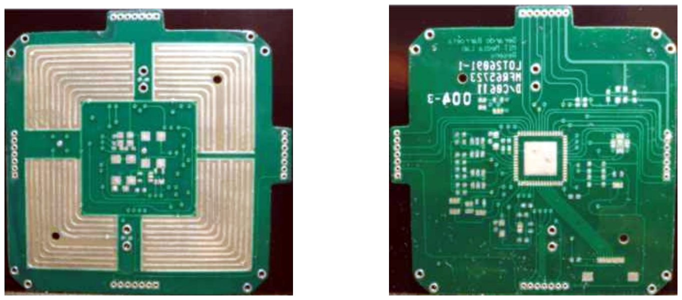

(a) Flex Node Front (b) Flex Node Back

Figure 3-5: Flex Nodes

has a physical bus topology, but a logical star topology which allows multiple Brains to be connected together. USB also provides enough bandwidth to handle the data requirements that connecting multiple brains would involve.

USB is also an industry accepted protocol that has proved to be successful for practically any electronic device created in the last few years from cellphones to portable cameras. It is easy to implement and there are a wide selection of readily available devices that easily add USB functionality to practically anything. A full description of the devices used here will be provided in the Brain section further ahead on this chapter.

3.3 Nodes

Every *Skin Patch* is a Sensor Network composed by one or many *Sensor Nodes*. Each node is composed of a multi-layer, *flexible circuit substrate* or PCB that is capable of sensing six different physical quantities:

1. **Strain/Bending: by using two orthogonal custom-made strain gages**
2. **Proximity/Activity: by using a piezoelectric cantilever**
3. **Absolute Pressure: by using a Quantum-tunneling effect material**
4. **Ambient Light: by adding an integrated sensor**

5. **Audio: by adding a MEMS microphone**

6. **Temperature: by using an integrated temperature sensor**

In addition to the sensors, an ultra-low power microcontroller, programming header, external crystal and RGB LED are also included. This means that each node is capable of sensing, processing and actuating independently from each other. This approach, not only makes the Network more reliable and robust, in accordance with one of the design principles, but it also provides each node with the processing power required to eliminate the need to route signals from every sensor to a Central Processing Unit, typical of dense sensor arrays like the ones presented in Haris[18] and Papakostas[49].

3.3.1 Substrate Design

The Nodes were built using Flex Circuit Technology following the recommendations found in Gilleo[13] and Harper[19]. Flex technology was chosen on top of other technologies described in the previous chapter, because it is a well established technology that has proved to be successful in many commercial products[47]. It also has many of the mechanical properties that conform to the project specifications: flex can be conformed to any arbitrarily shaped container, flexed continuously and virtually indefinitely, is stress compliant, extremely thin and it withstands high temperatures. It is also widely available through several manufacturers, and there is a vast amount of reference information about it[64][13][19]. Choosing a newer technology like the one proposed by Lacour[35], for example, would have created numerous constraints on the project and, because of lack of technological maturity, place the emphasis of the research in the fabrication techniques rather than in the mentioned goals.

Flex circuits are different from regular rigid PCB's because they can be manufactured with a very wide selection of materials, and unlike their rigid counterparts, a material stack-up must be provided to the manufacturer as part of the design process. Flex circuits are composed of four major parts: one or many *dielectric base films* whose properties depend on the

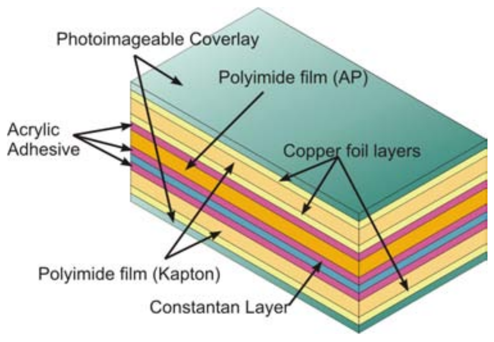


Figure 3-6: Node substrate material stack

material[5], one or more *conducting layers*, their respective *adhesive layers* and any necessary *finishing layers*. However, even though the material selection and the combination between possible elements is very broad, there are typical standardized combinations of adhesives, base materials and conductive layers. Perhaps the most popular combination is Copper film over a Polyimide base because of its low cost and favorable electrical properties. The Skin Nodes are based on this basic material combination with a few necessary modifications.

Lets now describe the material stack-up and design options selected for the project. Figure 3-6 presents the entire material stack layer by layer. First, there were two main options to choose from for the base material: Mylar® (Polyester film) and Kapton® (Polyimide film). According to Gilleo, Mylar is about 20 times cheaper than Kapton, has excellent flexibility, moisture resistance and electrical characteristics but it has very low tear resistance and it can't withstand the high temperatures required for soldering. Kapton films are much more expensive and moisture absorbent than Polyester films but they are easier to prototype with because they can support the temperature used for soldering electrical components. A 2mil ($50.8\mu m$) Kapton layer was therefore chosen as the Node substrate.

The Skin Nodes also contain 4 conductive layers, three of them made of 1oz ($35.5\mu m$) copper and one of a special copper and nickel alloy called **constantan**(Cu55Ni45). Constantan is not as good a conductor as copper, but it has some properties that make it ideal for other purposes [23]. This will be explained in one of the following sections when the different node sensors are explained. Again, there are several options when choosing the right kind

[5]Usually Polyimide or Polyester films but can also be epoxy-glass, glass-resin or others

Thickness (mils)	Layer	IPC Standard
0.001	Photoimageable Coverlay	IPC-SM-840C
0.0014	Top Layer (1 oz Rolled Annealed Cu)	IPC-4204/11 E1E2 M0 W71S/1S
0.002	Polyimide Film (Kapton)	
0.0014	Inner Layer (1 oz Rolled Annealed Cu)	
0.001	Acrylic Adhesive (Pyralux)	IPC-4203/1 E1E2 M2/2
0.002	Polyimide Film (AP)	
0.001	Acrylic Adhesive (Pyralux)	
0.001	Inner Layer 2 (Constantan Foil)	ASTM B267-90, ALLOY 5B
0.001	Acrylic Adhesive (Pyralux)	IPC-4203/18 M2
0.002	Polyimide Film (Kapton)	IPC-4204/11 E1E2 M0 W71S/0
0.0014	Bottom Layer (1 oz Rolled Annealed Cu)	
0.001	Photoimageable Coverlay	IPC-SM-840C
0.0162	**Total thickness**	

Table 3.1: Material thicknesses and standards

of conductor. Gold and silver are better options than copper, but they are evidently more expensive. Polymer Thick Film conductors are also widely used in flexible circuits because they can be applied as inks, but they tend to be more resistive than copper. As with rigid circuits, copper offers the best tradeoff between cost and performance but in the case of flexible circuits, the metal can be either electro-deposited onto the substrate or glued as think *rolled annealed copper laminates*. Rolled annealed copper has the advantage of supporting virtually infinite flexing cycles, while electro-deposited copper tends to crack under these circumstances. Since the idea of the project is to emulate the way a skin would behave, RA copper was chosen in spite of a marginal increase in cost.

Finally, both sides of the nodes had to be covered by a protective layer to isolate the conductive traces from the environment, and to serves as a solder mask. There are four main options: coverfilm, screened solder mask, photoimageable solder mask or dry film solder mask. Coverfilm is the cheapest, but it has to be machined like any other layer which limits the pad size and hole sizes. Photoimageable solder masks or coverlays are flexible versions of their FR-4 cousins; they are a bit more expensive but they allow for pads and vias of practically any size, plus they offer the advantage of being re-workable [73] which is essential for prototyping.

An acrylic adhesive (Pyralux) was used to join the different layers together and make up the stack. All of these materials, along with their exact thicknesses and specifications are included in the IPC-4203[29] and IPC-4204[30] industry standards that can be obtained from

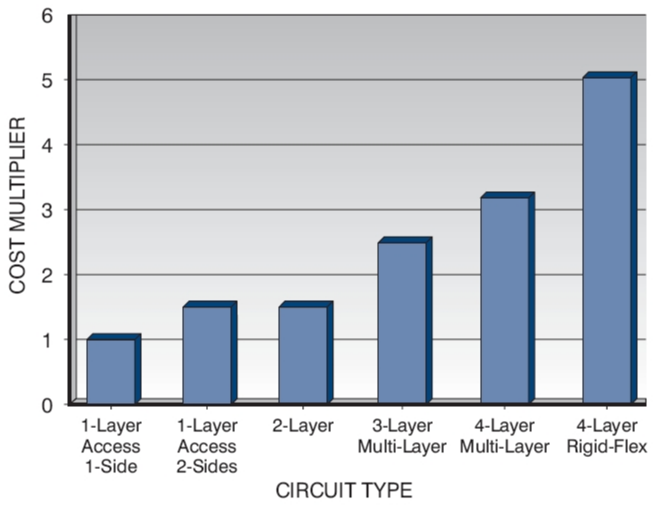


Figure 3-7: Flex circuit cost as a function of fabrication complexity[46]

www.ipc.org. Table 3.1 lists the thicknesses and fabrication standards used to fabricate the substrate[6].

Cost Considerations

Material selection however, is only the first step when designing a flexible PCB. There are many mechanical considerations to make that have a big impact on cost: circuit size, shape, overall thickness, minimum bend radius, mechanical fittings and circuit terminations must all be included into the design phase as well. But before going into these in detail, a brief cost analysis is necessary to illustrate why some design decisions were made.

Flexible circuits are broadly classified depending on the types and number of layers used for their construction[7], and the type chosen will affect the cost, mechanical flexibility, and mechanical stability. Generally speaking, the greater the number of layers per circuit, the greater the fabrication complexity and therefore cost. Layers should be added sparingly then, to keep design/fabrication complexity and costs to a minimum. Keeping the layer count low also increases the flexibility[8] and compliance of the material, which in the case

[6]Courtesy of Dan Hansler of Pioneer Circuits, Inc
[7]Classified as Single-sided, Back-bared, Double Sided, Multilayer, and Rigid-Flex
[8]Characterized by the maximum bend radius

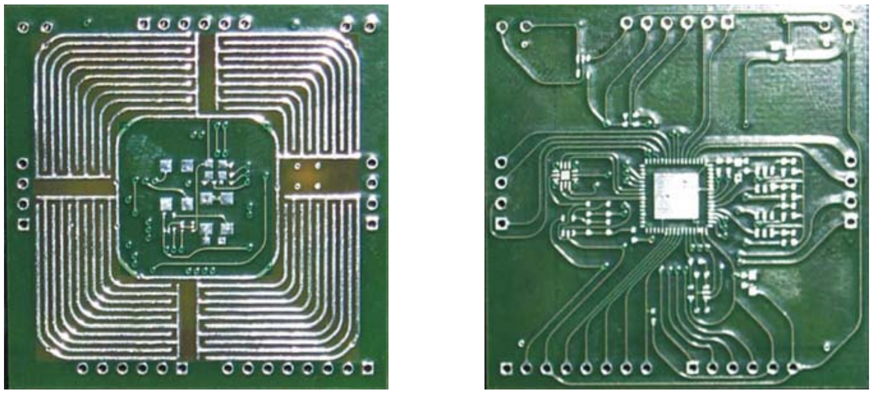

(a) Node Front (b) Node Back

Figure 3-8: Rigid node prototype

of this project are crucial. Figure 3-7[9] shows the relative cost increase for different types of flex. There are several other factors that also increase cost, but analyzing all of them in detail would be beyond the scope of this document; the reader is welcome to check the provided references if further information is needed.

It is clear that multi-layer boards cost more than twice than the average double-layer flex board and more than thrice that of single-layer boards. Yet, it is necessary to resort to the more expensive multi-layer flex boards in order to have enough layers to route all the electronic components required by the goals of the project. Manufacturing the final boards is not the only costly part, though, as they must be first prototyped. A typical four-layer rigid board of this size can be prototyped for less than $50, the same prototype on flex usually costs between $800 and $1,200 because of the labor involved. For this reason, prototyping of the Skin Nodes was done on regular rigid boards, and only the "final" version was fabricated on flex. Not all of the functionality could be tested in the rigid prototypes, but they were essential for the design, debug and first tests. These boards are shown on Figure 3-8.

[9]Courtesy of Minco™: http://www.minco.com/uploadedFiles/Products/Flex_Circuits/aa24-flex_design.pdf

Mechanical Design

Once cost issues have been addressed, we can proceed to describe some of the mechanical aspects behind the design of the Skin substrate. Again, things like mechanical terminations and fittings, shape, size, number and types of layers, overall thickness and minimum bend radius all have to be taken into account when designing the circuit.

The **shape of the Skin Node** was chosen to be roughly square with two overhanging flaps to create mechanical connections to adjacent nodes. This can be seen in any of the layouts of Appendix C or Figure 3-5. Making the nodes square has several advantages: first, unlike complex tiling of polygonal-shaped nodes, square nodes can be made equal to each other avoiding the need to make distinctions between them in the network and thus simplifying the coding of applications; second, they can be easily tiled to form any arbitrary shape, like pixels in a monitor; third, one of the characteristics of the network communications scheme is that every node can communicate directly to its adjacent immediate neighbors, which in the case of a quadrilateral shape is a maximum of four[10] (except for edge and corner nodes). This not only helps to keep the routing and wiring simple but it helps to keep the necessary available processor port-pin-count low, which is important to maximize functionality.

Mechanical terminations are another big design choice. There are many possible ways to connect flexible boards to each other many of which are outlined in Minco's design guide[46]. For the first Skin Network revision, *Physical Reconfigurability* was not included as a design point because of its increased complexity both in hardware and in software. Basically, network reconfigurability in the context of the project means that nodes can be easily rearranged, which would have many advantages over a fixed network configuration, but it requires a type of connector/mechanical terminator that can allow nodes to be easily plugged and unplugged while maintaining a high enough degree of mechanical stability so that communication links (backbone and peer-to-peer) are not frequently lost. None of the evaluated mechanical terminators could achieve this with ease, so a semi-permanent termination method was chosen instead. The chosen Node connection scheme is similar to that presented by Um in [69], and it was implemented with header holes that can be joined

[10]North, South, East, West

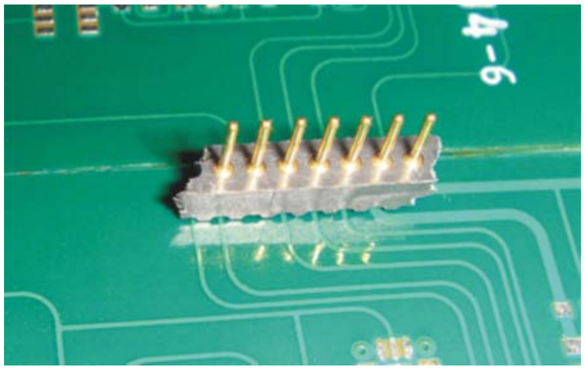


Figure 3-9: Skin Patch mechanical terminations

together by soldering single in-line, break-away headers, which can be seen in Figure 3-9. Header holes are located in the overhanging flaps so that nodes can be tiled without much overlap.

Size was another major consideration; all circuit dimensions can be seen in Figure C-3. How big should the Nodes be such that there is enough space to accommodate all the necessary components, while keeping an acceptable level of sensor density per covered area? This is essentially a cost problem since more expensive fabrication techniques like *blind and buried vias* are needed to increase the component and routing density of the Skin Nodes since there is virtually no space left for compacting the layout. This can be observed in Figure C-4 included in Appendix C. However, given that costs are already high and that these are pricey options, the costs would have become prohibitively expensive. Accordingly, these options were then left for a possible second revision. Finally, Some extra space was needed for to allow bonding of a special material used for the pressure sensors; this is why the Nodes are not routed close to the edges.

Yet the most important mechanical characteristic, and the one that separates this design from rigid PCBs is its ability to bend and flex. Flexibility of the board is influenced by many things, but the three main ones are **material selection, final substrate thickness and routing techniques**. Material selection was already described, but in terms of flexibility, rolled annealed thin films are the best for applications that require constant bending. One could arbitrarily choose extremely thin (up to about 9μm thick) films, but they are not good

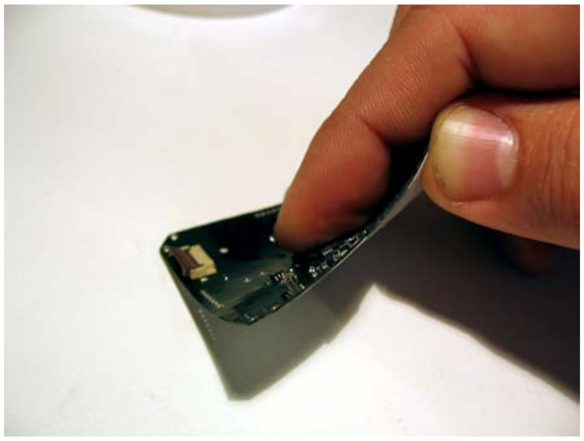


Figure 3-10: Node with electrical components being flexed

at withstanding large stresses which lead to *hairline* cracks on the conductors that cause faulty electrical connections. For this case, standard 1oz copper was chosen. Final substrate thickness simply depends on the number of layers on the material stack; the more layers, the smaller the minimum bend radius that the substrate can take without permanently deforming or cracking. The thicknesses of the entire stack and each layer are shown in Table 3.1. A good rule of thumb is that the minimum bend radius should be at least 24 times the total thickness of the substrate, however this is best defined in IPC's document IPC-2223 [31]. Figure 3-10 shows an example of a fully-assembled node being flexed

Routing and Component Placement Issues

The routing of the board requires special mention. There were several issues that either limited or complicated the electrical routing of the board. First of all, there are only two layers available for electrical components, as the inner layers are only for routing and sensing. The top or *outer skin layer* layer was rendered almost useless for component placement because the real estate was used up by the electrodes required for the pressure sensors; the only available space was occupied by the ambient light sensor, LED, and microphone, which by necessity *had* to be on the outer layer. Second, as it was seen before, layer count must be kept low because of cost and flexibility issues, so *power and ground planes* could not be used, which only further complicate the routing. Third, including a microcontroller on a flex board is difficult because the bending can cause it to pop out. One solution for this is to wire bond the processor chip as thinned bare-die silicon but this is well beyond the scope of the current project.

A more conventional workaround for this is using either *rigidizers* or rigid-flex, but the former can't be used if components have to be placed directly underneath the area to be rigidized, and the later would have been prohibitively expensive. Several routing and component placement tricks, which are shown on the following list, had to be used to reduce overall stress points; these are shown on Figure 3-11.

1. The microcontroller was placed in the center of the bottom layer, and the components that had to be on the top layer plus the crystal were placed directly opposite of the microcontroller on the top layer so that their large bodies worked as rigidizers.

2. Because the node footprint is small, and the center of the board was now somewhat rigid, bending was mainly constrained to the outer regions of the board, so every component that had to be placed in the outer region of the bottom layer was aligned to the bending axis so that stress points at its pins are minimized.

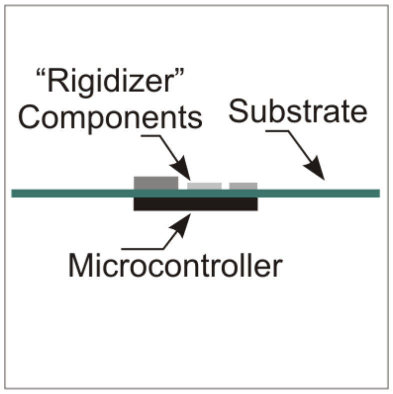
3. Whenever possible, traces were routed so that they would meet at pads/vias through the center, except for those cases where this would create unnecessary extra curves and corners.

4. The *Ibeam* effect shown on Figure 3-11(d), limits bend radius, and is therefore minimized throughout the layout of the circuit. This also helps to keep crosstalk between routed signals to a minimum, which is good for speed as well.

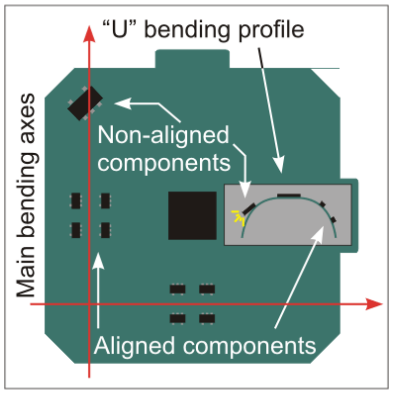
5. Routing Corners are natural stress points, so they were routed underneath components whenever possible.

6. Rounded corners were also used to increase the bending life of the nodes.

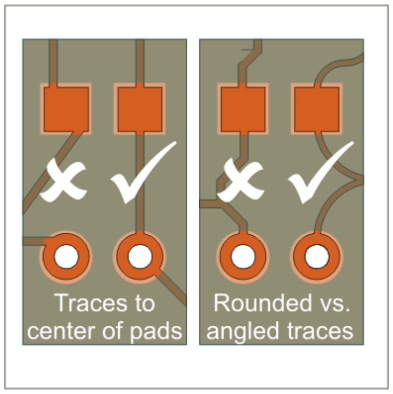
7. Sharp corners create stress points so the curvature radius of the corners was kept large when possible.



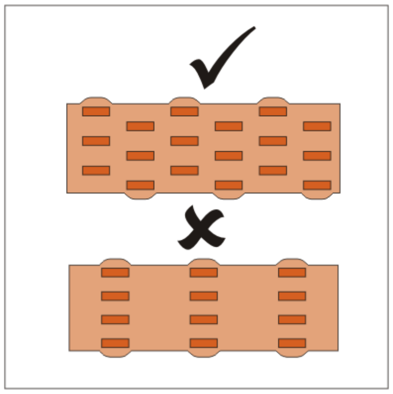
(a) Rigidized center



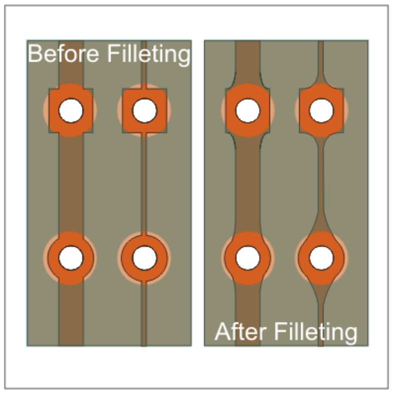
(b) Aligned components



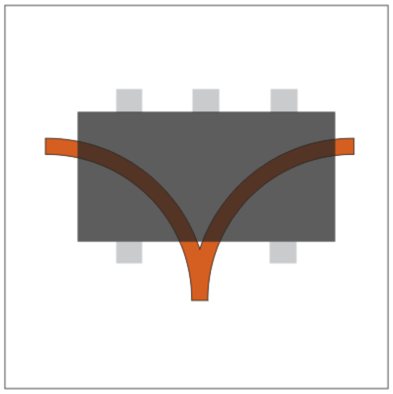
(c) Rounded routing and Centered Routing



(d) IBeam Effect



(e) Teardrops and Fillets



(f) Corners routed under components

Figure 3-11: Routing techniques

8. *Teardrops* and *filleting* were added to all pads and vias.

9. Components were also chosen to be as small as possible, so that bend radius is not limited by this. If components are large, then the substrate will not be able to bend without components breaking off.

There is one last thing to mention. Because of the necessary number of layers and the presence of surface mount components on both layers of the board, the nodes will certainly not be as flexible as a natural skin, plus are not, of course, elastic and 3-axis conformable. These and other results will be mentioned in Chapter 5.

3.3.2 Power Circuitry

As we mentioned in the previous subsection, component selection was made in such a way so that component size and orientation would not much hinder the bending of the circuit. In general, this means selecting components as small and compact (not elongated) as possible, otherwise there would be a risk of components breaking off the substrate as shown in Figure 3-11(b).

Referring back to the design principles of section 3.1, one of the key points is to have a reliable network. Reliability in the case of network power distribution is gained by providing each node with their own power supply or individually wiring each node to a centralized one. Connecting each node to a central power supply would be a step back in the design, because it would mean that each node has to be wired, which is exactly what we are trying to avoid by having processing power in each node. The other option would be to have a coin-cell battery in every node, but these are generally not rechargeable, bulky in relationship to node size and would hinder flexing motion.

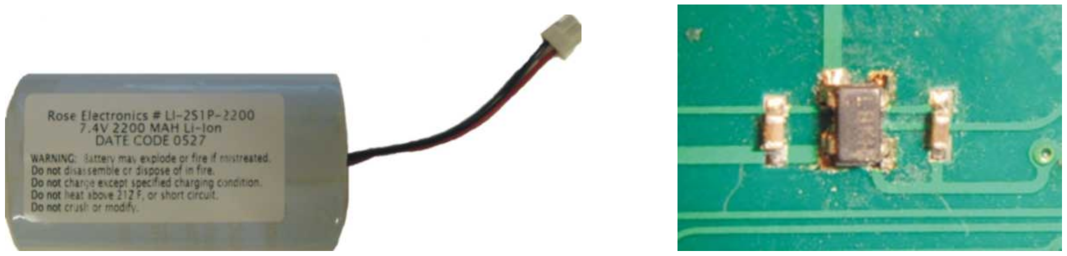
The next best option is to provide as many redundant power and ground connections throughout the network as possible. This is the option implemented in the SNAKE project, and therefore every node has a power and ground connection to each one of their immediate neighbors (North, South, East, West). Since the Skin Network topology is dictated by the shape in which the nodes are connected, the nodes at the edges of whatever network shape is chosen would obviously have less than the four maximum redundant power links. The worst case scenario here, would be when the network topology is a straight link of nodes connected in a line, which provides no redundancy. Power to the network is therefore distributed from a centralized unit, but no individual wiring is necessary, because each node provides power its adjacent neighbors.

Each node is provided with a NCP553SQ33T1 linear voltage regulator that provides a constant 3.3V supply to each node. This regulator is ideal because not only does it come in a ultra-small and compact SC-70-4 package, but it also does not need any external components except for an optional output capacitor. It is an ideal regulator, because it has a 12V maximum input voltage, which allows for a very wide range of power supply voltages; it has an output current capacity of up to 180mA with a 650mV dropout, more than enough for the node circuitry, and it only consumes 2.8μA of quiescent power, which which suits our goal of low power consumption.

Power requirements of the network vary widely, mainly because the size of the Skin will change, depending on the number of nodes connected. Either a large battery or a bench-top power supply can be used, but for the specific case of the skin patch built to test the project, a Rose Electronics LI-2S1P-2200 7.4V Lithium-ion battery pack was chosen, mainly because they were available from previous projects but also because they have a high-discharge rate of 2.20A/h. This means that if each node consumes an average of 15mA, one battery can provide enough power to operate roughly 140 Nodes for about an hour. Actual power consumption is much less than 15mA, but this and other results will be mentioned in Chapter 5. Figure 3-12 shows the power components.

3.3.3 Processor

Microcontroller selection was also a critical part in the design of the Node Hardware. Following with the established design principles, the chosen microcontroller should be capable



(a) 7.4V, 2.2mAh Li-ion Battery (b) 3.3V Regulator in each node

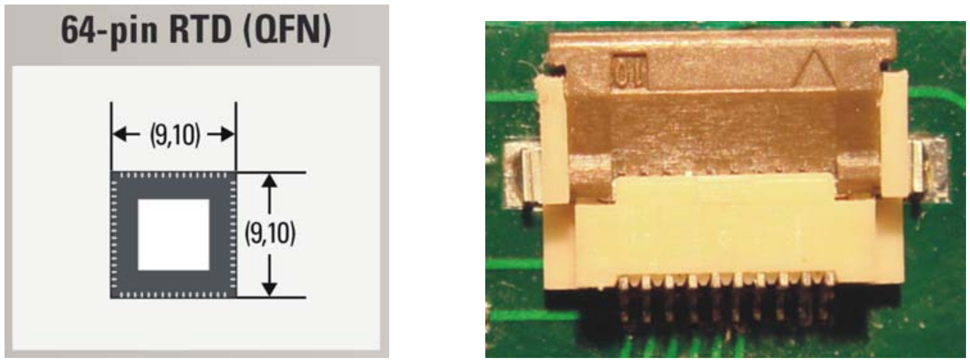
Figure 3-12: Power components

of processing all data generated by the node's 11 sensing channels[11] and still have enough processing power remaining to handle the two communications channels provided(I2C and P2P). It should also be capable of ultra-low power operation and have enough ports and peripherals (timers, ports additional hardware) to handle data and tasks efficiently. Finally, it needs to include a relatively large amount of RAM ($> 10KB$) on die, to handle sampled data and data packets, and come in the smallest possible package.

There are hundreds of microcontrollers available, however only a few manage to have the required characteristics. The PIC24H family of microcontrollers for example, would be an excellent choice, if it wasn't for their large size. Others like the PIC18, Atmel's AVRs, and 8051-based families (Silicon Labs, Intel, Atmel) come in a broad variety of memory sizes, hardware peripherals and special features, however they are all 8-bit microcontrollers, which complicates the handling of large quantities of data, and they are also generally power-hungry, requiring an average of $300\mu A$ per MHz.

The MCU with the most adequate balance of characteristics for the project was a Texas Instruments MSP430F1611. The MSP430 family of microcontrollers has the best-in-class power management modes and ultra-low power consumption [26]. With five different low power modes, 16-bit native operation, less than $6\mu S$ standby wake-up time, and a 300nA deep sleep mode, the MSP430 has the most flexibility when low power is needed[27]. MSP430's can run up to 8.000Mhz with an optional external crystal and up to 5.000Mhz

[11] 2 Strain/Bending, 2 Motion sensors, 4 Pressure sensors, 1 Ambient Light sensor, 1 Microphone, 1 Temperature sensor



(a) MSP430F1610 QFN Package (b) FFC programming header

Figure 3-13: Microcontroller with programming header

with the internal DCO. For this case, an external ABMM2-8.000MHZ-E2-T 8.000Mhz SMD Crystal which was included to mainly drive the serial backbone at a higher speed.

The MSP430F1611 comes in a very compact 9mm-by-9mm leadless 64-QFN package with 48 general purpose I/O pins and two hardware serial modules that can be configured as either USART, SPI or I^2C ports. It also includes 10KB of RAM and 48KB of Flash, a hardware multiplier, a DMA controller, two timers, a supply voltage supervisor, and a 10-input channel, 12-bit ADC module. Although features like the hardware I^2C controller, DMA and hardware multiplier are not absolutely necessary for the system, they are highly desirable because the network backbone uses an I^2C protocol; DMA can be used to transfer sampled sensor data to RAM leaving the processor free for other tasks; and the hardware multiplier greatly speeds many of the calculations.

Probably the only drawbacks to selecting this microcontroller were its relatively big size compared to other MCUs of the same and other families[12] and its price, which at the time this project was created was of roughly \$12.0. These microcontrollers went also unexpectedly out of stock of all major providers when the flexible boards were already created, and the restocking date was beyond the project time frame, so MSP430F1610 had to be used instead with the minor disadvantage of having only half of the RAM. Fortunately, this had no major impact on the operation of the network, but then again this will be mentioned

[12]64-QFN is almost 4 times larger than a 32-QFN used by the MSP430F123

in more detail in the Results Chapter. Finally, since this is a first revision prototype, an FH12-10S-0.5SH FPC programming header was included for debugging purposes. Both, processor and header are shown in Figure 3-13

3.3.4 Sensing Modalities and Actuation

The human skin is capable of sensing many different physical quantities. Every square centimeter of skin is filled with hundreds of nerve fibers sensitive to temperature, light, vibrations, pressure, mechanoreception, and many different forms of pain[13]. This subsection, describes some of the available sensing technologies with which it is possible to emulate some of the sensing capabilities of the human skin while comparing their characteristics to illustrate why one was chosen over the others.

The sensor subsystem of the Skin Nodes is composed of the actual sensors themselves, their associated analog signal processing, and the ADC module of the chosen MSP430 microcontroller. Although they are all independent of each other and use very different sensing technologies, they all share the same selection criteria, and follow the previously mentioned design points. Sensor technologies were selected in such a way so that:

a. Analog-processing component count is minimized.

b. Sensors can be integrated into the substrate if possible.

c. They observe ultra-low power operation.

d. They are analogous to a sensor present in human skin.

Unless otherwise noted, all of the signal conditioning circuits use the same amplifier from Maxim IC: MAX4400AXK which comes in a tiny SC-70-5 package, and consumes only 320μA of quiescent current. Although there are other devices that consume considerably less power, like the Texas Instruments OPA347 or TLV2252, the former was chosen because

[13]Depending on the part of the body, fingertips for example, have up to 2,500 receptors

its better balance between Gain-Bandwidth product and power consumption. These could not be used for the Strain/bending sensors, because a differential amplifier is needed. Other sensors already come pre-amplified and no external amplification was needed. We'll come back to this point when we individually describe each sensor.

Each sensor type is sampled at a frequency high enough to capture the necessary information, but low enough to keep power consumption down and also to avoid saturating the RAM before the processor can extract higher-order features from raw data. In the case that data needs to be extracted from the nodes, these higher-order features are what is generally transferred, thus bandwidth requirements are minimized. This will be described in more detail when we mention some of the applications that run on the Skin Network and again in the Results Chapter.

Before heading on to the sensor suite description, it is important to note that even though receptors in the human skin are capable to sense different types of sensations at once[14] [56], in the world of electronic sensing it is generally easier to use a different sensor for a specific physical quantity. Otherwise effects like temperature drift, electrical coupling and other effects can disturb many measurements and cause ambiguousness in the readings. For this reason, strain was separated from pressure and vibration and a separate sensor with a different technology was chosen for each modality.

Actuation

Due to the nature of the project and the limited amount of node surface real estate, actuation was limited to light response. Skin Nodes are provided with a LTST-C17FB1WT RGB LED. This small 2.4mm-by-2.0mm RGB LED can be easily driven by the 3.3V outputs of the MSP430 MCU without needing a special driver. The LED has a common cathode so the anodes of each LED were connected to three different MCU I/O pins through a current limiting resistor. Since the blue and green LEDs have a DC forward current of 20mA with a forward voltage of 3.5V, a 100Ω resistor was used to limit the current. The red LED

[14]Some examples include: Meissner's corpuscles are sensitive to vibrations and light touch; Pacini's corpuscles are sensitive to pressure and vibration; Merkel's disks, Pincus domes to touch and pressure

has a 30mA forward current with a forward voltage of 2.0V so to keep brightness levels roughly equal a 150Ω was used instead. Moreover, the micrcontroller pins to which the LEDs are connected are timer outputs which allows them to be pulse-width modulated and thus generate practically any color.

Although the LED was added mainly for debugging purposes, it turned out to be a very useful tool for demonstrating the capabilities and functionality of the assembled test network. An interesting thing to note here is that the Skin Network generated some interest for its possible use as an architectural material, so having LEDs makes sense to give the skin a certain type of "chameleon-like" expression.

Strain and Bending

The idea of providing the Skin Nodes with the ability to sense strain and/or bending amount, and its direction was derived from the fact that our human skin is perfectly capable of doing this. The idea is clearly not new, and several ways of achieving this result have been proposed: the works presented by Banks[2], Yamada[75] and Hritsu[22] for example, describe similar optical ways to track the strain and deformation of a skin-like material. In Yamada[76] et al, a vibrotactile strain-sensitive device is used to detect textures. Still, perhaps the example that best demonstrates this emulation are the so-called eFabrics or Smart Fabrics that have been presented in numerous articles describing similar methods to sense the deformation of conductive fabrics and plastic films [61], [71], [44].

Transducer Selection There are indeed, many ways of measuring the strain and/or bending of a material, but as we already saw, the ideal choice for this project will be one that can be integrated into the node substrate while requiring minimal signal conditioning so as to keep power consumption and component count low. Yet, even with these constraints there are still multiple viable options: *FSRs*, *Strain Gages*, and *Piezo-resistive films* are all good candidates.

Taking a closer look though, we find that FSR bend sensors can't be easily integrated into the substrate at the time of fabrication or are difficult to fabricate with any accuracy[34]. because they are made of special, proprietary materials which can't be easily obtained[15]; instead then, they must be added at the time the skin is assembled, which complicates the Skin Node layout and assembly. FSRs also delaminate pretty easily and are not good at resisting high humidity environments. Piezo-resistive (PVDF) films are very good for transducing changes in strain, but they can't be used for cases where static strain is needed; there is also no easy way to integrate them into the fabrication process. Strain gages on the other hand, can be easily customized and streamlined into the fabrication process, they add only one extra layer into the material stack and incurs only a marginal increase in cost. Two orthogonal axes of strain sensing were therefore included into each node.

Strain gages are long traces of a resistive metal for which a change in strain results in a change in electrical resistance[10]. This change can be then amplified and read by a microcontroller. The change in resistance is represented by the equation:

$$\frac{dR}{R} = S \cdot \varepsilon \tag{3.1}$$

Where R is the resistance, ε is the strain, and S is the strain sensitivity "gauge" factor of the material. This sensitivity factor is what determines the effectiveness of a strain gage design and it is highly dependant on the materials and fabrication methods chosen. Another interesting point is that strain can be compressive or tensile, which means that not only the amount of the strain can be extracted but also its direction. As a result, strain gages can be used as bend sensors for the skin.

Strain gages are manufactured by adhesively bonding a thin foil of metal to a flexible plastic substrate like Kapton. The gage is then glued to a surface of the object for which a strain measurement is needed. Ideally, the gages should only change their resistance due to mechanical deformations, however in real applications, factors like the adhesive used to install the gage, temperature, and the gage material, all have an effect on the measurements

[15]Like Tekscan's Flexiforce sensors

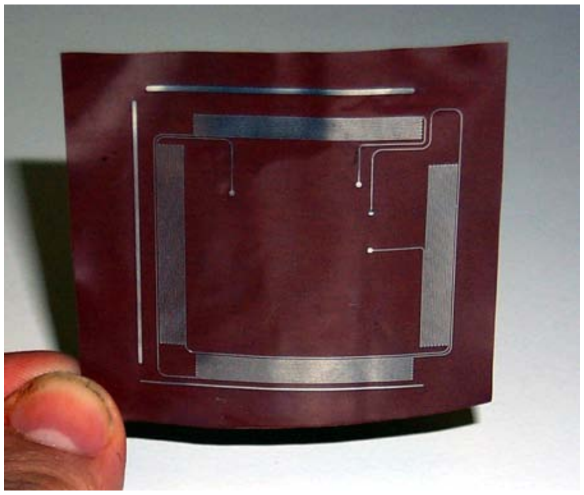


Figure 3-14: Strain gage layer prototype

[48]. Because of this, strain gages are built from special alloys for which the gage factor S is much less sensitive to temperature variations than for regular metals.

Strain Gage Design Material selection directly affects the mechanical properties of the gage, like fatigue life, flexibility, sensitivity and stability. However, even though the materials selection available for fabrication is very broad, constantan has constantly been the most widely used alloy for strain gage applications, and was therefore selected. Constantan has a relatively high sensitivity, high resistivity and high drift temperature which make it ideal for this project.

There are five other variables that have to be taken into consideration when custom-designing a strain gage: gage length, gage width, overall resistance, grid pattern and backing material. In the specific case of our project, backing material was not an issue because the skin is actually made from Kapton, which is the most typical backing material for constantan foil. Here, the length of the sensor grid determines the "active area" of the sensor, and therefore must be adequately fit to cover enough node area so that dead zones[16] are minimized. The width of the gage traces, on the other hand, is the main determinant of the overall resistance of the gage, which translates into its ability for dissipating heat and

[16]Skin areas insensitive to bending

therefore defines its stability. Overall resistance is an issue if the amount of current flowing through the gage needs to be controlled. Finally, The grid pattern is important because it determines the axis of the strain to be measured; grid patterns can be either one of the typical ones (uniaxial, biaxial rosettes, multiple-axis rosettes, radial rosettes) or a customized design. This is provided, however, just as an overview of the available design choices that must be made; for a detailed explanation of these the reader can try Vishay's tech note TN-505-4[24].

For the more specific case of the current project, the strain gage design can be seen in Figures 3-14 or C-7. An observant reader may notice that there are actually four sets of traces, however, these are considered as two gages because the top and bottom sets of traces are connected so they actually make one sensor; this is also the case for the left and right sets. Their design characteristics are outlined in the following list:

Length: designed to cover as much area as possible to diminish dead zones.

Trace width: made as thin as possible; the manufacturer imposes a 5mil trace width limit without imposing a price increase, so this width was chosen.

Overall resistance: determined by trace length and material properties. Although trace length was controlled to make both gages in each node have the same resistance, fabrication variations caused them to have a variation from node to node of up to 20Ωs. The total gage resistances are then in the rage of $168 - 188\Omega$.

Grid pattern: 90-degree T-rosette style with redundant traces (2 sets of traces per strain axis) which gives a resolution of 2 sensors per node.

Strain gages, however, produce very reduced output voltage swings[17] because the changes in resistance due to strain are small. For this reason, they need to be conditioned by a very high-gain amplifier in order to be adequately sampled by the microcontroller. The way this is usually accomplished, and the way it was actually done in the project, is by using

[17]Tens of millivolts

an unbalanced *Wheatstone bridge*, from which a differential output is taken and fed to an instrumentation amplifier. An unbalanced Wheatstone bridge can be seen as nothing more than two independent voltage dividers, from which one of the resistive elements has been replaced by the strain gage. As the gage undergoes some strain, its resistance changes, and the voltage seen at the divider node correspondingly changes[18]. Since one of the voltage divider branches is composed of regular resistors, the voltage seen at its divider node will always be set at $V_{CC}/2$, so when these two bridge outputs are fed to an instrumentation amplifier, this will amplify the difference seen at the divider nodes, causing the output of the amplifier to swing positively or negatively from the bias point, depending on the type of strain experienced by the gage (either tensile or compresive). This arrangement is shown on Figure 3-15.

The resistors that make up each one of the bridge divider branches are generally picked to be of the same value so that the excitation voltage is divided equally among the two resistors, giving the widest possible output swing. However, this configuration has a problem that, although not evident at first, could potentially cause the amplifier to saturate. Since the resistance change of the gages is so small even at their full bending range (around 3Ω maximum change), and given that large gains are needed for the signal to have a noticeable swing at the amplifier output, any minuscule mismatch between the resistance value of the strain gage and its adjacent voltage-divider resistor will cause the amplifier output to saturate. This is never a problem with commercial-grade strain gages, because they are fabricated in such a way that their resistances are within 1% of a commercially available resistor. This is posible because they are mass-produced, which keeps the costs of stringent quality control measures low. Doing this with a multi-layer flex PCB in a custom application is not possible because it would be extremely expensive.

The solution to this problem was to add a matching resistor in series with the strain gage, so that the resistances could be matched to that of the closest commercially-available resistor value. Thus if, for example, the measured effective strain gage resistance was of 166.9, placing a matching 2Ω resistor in series would add enough to the measured value to almost

[18]Since strain can be tensile or compresive, the difference in resistance can be either positive or negative

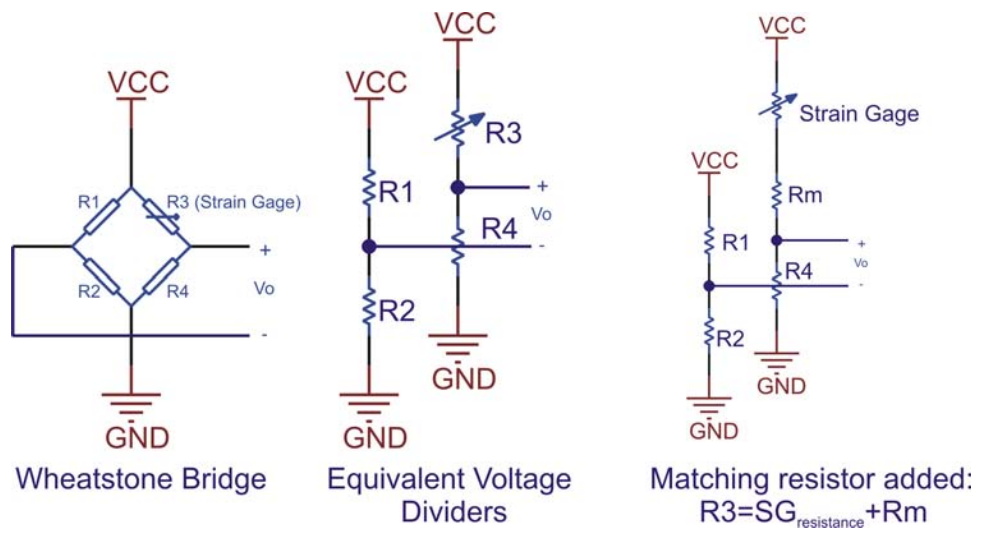


Figure 3-15: Wheatstone bridge used for Strain gage conditioning

perfectly match a 169Ω resistor, which can be obtained commercially. This matching then ensured that the amplifier would not saturate for resistance mismatches, at the expense of manually matching each gauge on each node.

Only the hardware selected for signal conditioning is left to be mentioned. As it was previously said, the Wheatstone bridge output nodes are fed to an instrumentation amplifier. The selection of the amplifier followed the same principles we have been mentioning throughout the document. It was rather difficult to locate an instrumentation amplifier small enough and with a sufficient gain for the required application. Fortunately enough, the MAX4462HEUT from Maxim IC Inc. comes in a small SOT-23 package, and is precisely designed for bridge applications. It even includes a reference input signal for correctly biasing the output voltage of the amplifier[19]. Probably its only drawback is that it comes with fixed gains of 1, 10 and 100, but even with a gain of 100, the output swing for moderate Skin Node bending is of at most 1 volt, which was enough for the requirements of the project. Having the full 0–3.3V swing accepted by the microcontroller would have required a second amplifier stage, which would have been impossible to include, given the already lacking component real estate. Possible ways to improve this and other parts of the design

[19]All amplifiers are actually biased at 1.5V using the external voltage reference provided by the MSP430. This is done so that they have roughly equal voltage swings

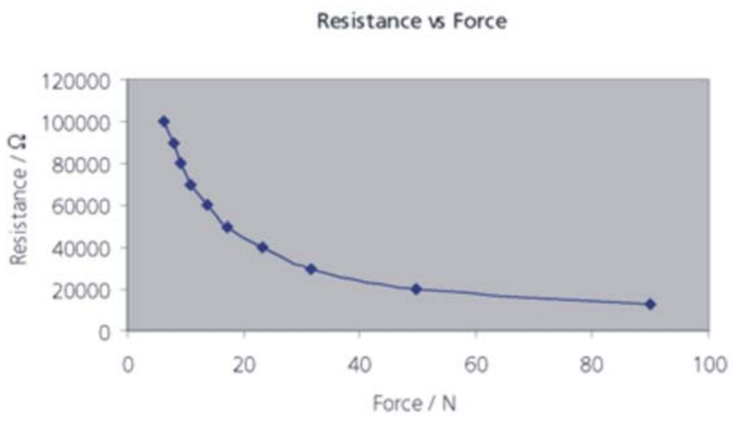


Figure 3-16: QTC response curve for a QTC pressure sensor[52]

are provided in the final chapter.

One factor that has not been mentioned so far about the hardware is its power consumption. This is observed through the entire design of the sensor systems by adopting three measures: use of instrumentation amplifiers that have a low quiescent current, the resistance of the traces is made as high as possible, and gages are gated. Sensor gating is done by pulling down the Wheatstone bridge using a FDG6301N Dual N-channel MOSFET, with the gate connected to a microcontroller output pin. These transistors come in a neat and compact SOT-23-6 package, and allow the microcontroller to turn on the gages only when a measurement needs to be taken greatly reducing the current consumed.

Pressure

Pressure sensors were also custom-made for this application. The reason for this is that even though most of the technologies evaluated before for strain sensing can be also used for pressure, they had to be discarded for same reasons described above and instead a QTC sensor from Peratech was used.

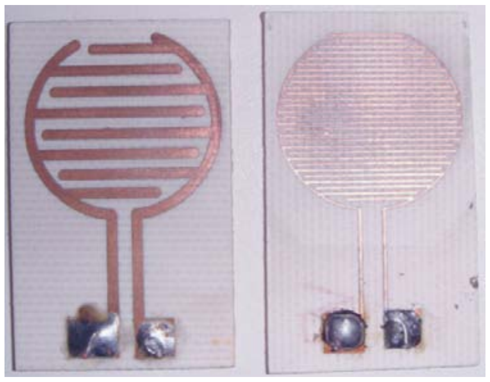
QTC sensors can be obtained as thin films bonded to a paper-like polymer, and can then be therefore easily bonded to the Skin Node substrate. Using these sensors also allows the entire surface of the node to work as a pressure sensor, unlike solid-state pressure transducers, which are localized. QTC sensors are made of metal particles bonded to a thin polymer. These particles are isolated, but as pressure is applied to the material they come closer to each other. As this happens, electrons start to “tunnel” between the isolated material gaps, and increase the conductance of the material. This behavior is not linear though, as it can be seen in Figure 3-16.

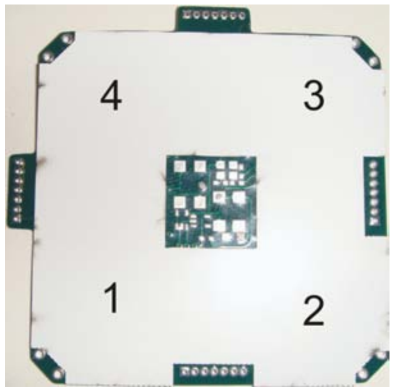
This curve is almost an inverse exponential, meaning that a pressure sensor fabricated with this material will be extremely sensitive to relatively light touches, and have a lesser sensitivity to higher pressures. This can be adjusted with a good signal-conditioning circuit design.

According to Peratech’s integration guidelines [53], the sensor requires the use of electrodes over which the material must be placed, free-floating with the coated side facing them. Accordingly, electrodes were placed on the top or external facing side of the Skin Nodes, and cover almost their entire surface except for the center and edges. The center could not be used, because it was needed for mounting all the electrical components that also need to be on this layer. The edges were left blank because they were used to glue the material to the substrate.

These electrode layouts have two parameters that must be controlled: electrode spacing, and trace width. The right balance between these two will dictate the resolution of the sensor. To ascertain these parameters, several prototype layouts were tested. Some of these are shown on Figure 3-17(a). From these two, conclusions were obtained. First, the width of the traces determines the base resistance of the sensor, which is an important quantity for the circuit design. And second, although electrode width has nonoticeable effects on the sensitivity, slimmer traces give more flexibility and resolution to the sensor. For the Skin Nodes, an electrode width of 20mils was chosen.

The entire surface of the node substrate was subdivided into four corners, so that four separate pressure sensor areas could be created as shown in Figure 3-17(b). These four electrodes were then multiplexed using a Maxim MAX4734EGC 4:1, multiplexer and the output of the multiplexer was amplified using the circuit shown on figure C-1. This is


(a) Prototype boards used to calibrate sensors

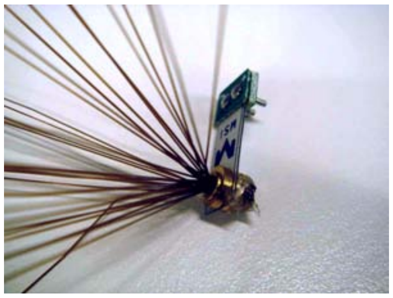


(b) A node covered with QTC material. Each node has four channels of pressure sensing as shown

Figure 3-17: QTC pressure sensors

nothing more than a non-inverting amplifier biased at mid-range by the voltage divider created by the sensor selected by the multiplexer and a resistor. Whenever the pressure sensor is not pressed, it has a resistance in the order of mega ohms. As pressure is applied to the sensor however, the resistance decreases, and the voltage divider voltage changes, which causes the output voltage to swing. One advantage of this circuit is that the sensitivity range of the pressure sensor can be adjusted by changing just one resistor, in this case the pull-down resistor used in the voltage divider.

Finally, there are some mechanical design aspects to consider. Gluing the QTC films onto the substrate only by the edges causes the material to warp and bubble when the substrate is bent. There are work-arounds to avoid this: the first one would be to selectively apply an adhesive around the metal electrodes so that only the parts with no metal stick to the coated side of the sensitive material. A second option would be to encase the entire skin into a silicone rubber; this has the added advantage of evenly distributing the pressure among the four corners of the node, so gradients are better observed.


(a) Mini-sense cantilever with whiskers glued to end mass

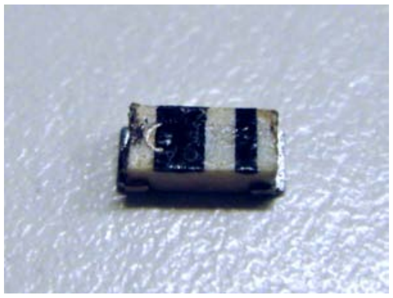

(b) Kyocera's PSAC piezo shock sensor

Figure 3-18: QTC pressure sensors

Proximity/Motion Sensing

Proximity sensing or mechanoreception (motion sensing) was added to the nodes to copy the ability that our skin has to perceive close activity through hair. As with any other sensing modality, there are several different ways to convert the whisker vibrations into electrical signals that can be then sampled by the microcontroller. Two different options based on piezoelectric cantilevers were tested.

The first option was based on the design presented by Lifton in [41]. Taking advantage of piezoelectric film's ability to turn bending strain into high voltages, nylon fibers were glued to the end of a Measurement Specialties MSP6915-ND Mini-Sense piezo cantilever to make up a vibration/mechanoreception sensor. This sensor has a mass attached to the end of the cantilever which also gives significant sensitivity to inertial forces. This design is shown on Figure 3-18.

The second option was based on the same principle of the first one, however instead of using a large piezo cantilever, two different Kyocera PZT vibration sensors, the PSAC380A and PSLC382S, were tested. The only difference between these two sensors is the axis of maximum sensitivity. These sensors have the advantage of being surface mountable, and are very small and sensitive. The only minor drawback is that the sensitive part is located inside of the small package and the whisker can't be directly glued onto it. Fortunately though, these sensors are sensitive enough that they are able to pick up vibrations if the whisker is epoxied to the package of the sensor. For this to be possible the whiskers have to strain the package enough for the sensor to respond, which is easily obtained if the whiskers are thick and stiff enough, and enough gain is given to the amplifier.

The circuit necessary for its signal conditioning depends directly on the type of information that is desired. Vibration, strain, shock or simply activity can all be derived from this sort of sensor depending on the signal conditioning and sampling chosen. A very detailed review of possible circuit options are shown in the Measurement Specialties Piezo Film Technical Manual [63]. Since the idea of including whiskers was mainly to detect the presence of close activity, a very simple circuit was used to generate vibration pulses, which are then connected to an input pin of the microcontroller, which can then either extract the vibration frequency, or simply report if there was or no activity for a certain period of time. This is also shown in Appendix C.

Ambient Light

Although one might not notice the fact that our skin is indeed capable of sensing ambient light, this becomes clear if we take into consideration that it is light and not heat which causes the familiar melanin concentration change in the skin when we spend long times under the sun; in other words we get tanned because of the amount of light received by our skin. Some animals (e.g. Cattlefish have optical sensors distributed in their skin).

Including an ambient light sensor into the nodes would allow several applications to be developed with the network. Some possibilities include the retina-like light edge detection system presented by Lifton in his push-pin distributed-computing-platform [39], and light gradient and source direction detection, and longrange proximity detection by cast shadows.

This capability was implemented by adding a Toshiba TPS851/52 ambient light sensor, shown in Figure 3-19. This tiny device was created as an illuminance sensor for brightness control of mobile device displays, and it is based on a current-amplified photodiode. It is an ideal option for the Skin Nodes because it doesn't need any external amplification, which reduces component count. It also has a low supply-voltage and power consumption that

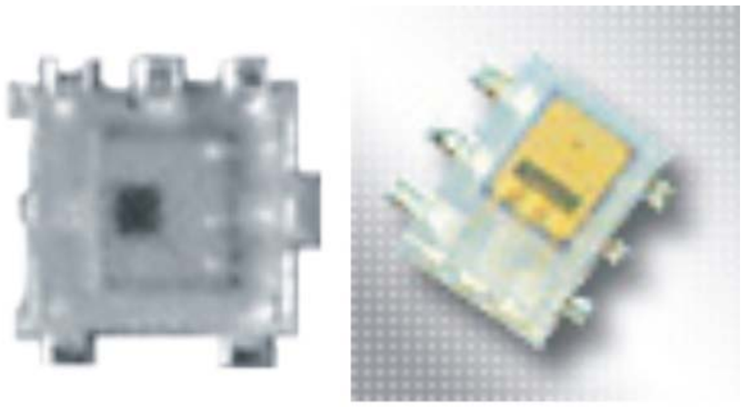


Figure 3-19: Toshiba TPS851/52 Ambient Light Sensors

makes it compatible with the rest of the circuit, it is very sensitive, and it includes a built-in Luminous Efficiency Correction, which adapts to different light sources (incandescent or fluorescent). On top of this, the ambient light sensor has infrared sensitivity suppressed, and has a spectral response close to that of the human eye with a relative sensitivity peak located at 600nm [67].

This sensor has the added advantage of coming in a tiny package that can be surface mounted. There was only one design problem that had to be overcome when mounting the sensor into the nodes. Since both the ambient light sensor and RGB LED have to be placed in the *outer* Skin Node layer, they have to be carefully placed to avoid interference with the sensor measurements. This was accomplished by placing them as far from each other as possible. In actual results, it was seen that some coupling still exists but, is minimal.

Finally, since the sensor outputs current, a resistor was connected at the output to turn it into a voltage that can be then sampled by the ADC. This is the only component needed to condition the signal, which again helps to keep component count low, and it allows the sensitivity of the sensor to be changed by tuning the resistor value (currently $8K\Omega$s).

Sound

The only sensing modality added to the nodes that is not present in the human skin is sound[20]. The reason for doing this is that sound sensing requires a much larger amount of bandwidth than the other sensors, and was therefore useful for pushing the limits of the hardware used, and testing how much data could be processed locally. Having a microphone on each node also enables several exciting applications, such as using differential time of flight to determine sound localization and sound directionality; beamforming is also possible since the skin becomes a conformal microphone array. Moreover, it allows for greater interaction if indeed this material is used as an architectural material.

Sound sensing is realized through a Mini-SiSonic™Knowles SPM0102NE3 MEMS Microphone. This microphone is usually used for mobile phones and PDAs, because unlike electret-based microphones, which can't be subjected to high temperatures and therefore surface mounting, MEMS microphones can be easily soldered and don't require special assembly processes. This was one of the reasons for picking this model. Other reasons include its good frequency response, omni-directionality, and very low current consumption. It can also work with the low voltages used by the microcontroller and other sensors.

The microphone signal conditioning circuit is shown also in Appendix C, and it consists of a capacitor that couples the microphone output to a second-order, low-pass anti-aliasing filter. The amplifier used is the previously-mentioned MAX4400 that has a Gain-Bandwidth product of 800Khz, high slew-rate and low noise. Biasing for the amplifier was accomplished by connecting a 1.5V voltage reference generated by the microcontroller which causes the output voltage of the amplifier to swing across the desired 0–3.3V range. Analog component selection was done to have a cut-off frequency of 4Khz, enough to sample human voices and low-quality audio. The low cutoff frequency chosen responds to the fact that high sampling frequencies need more processing power, and the microcontroller used only has a maximum of 12 RISC MIPS, that has to be divided among sensing, data transfer, and processing

[20]The inclusion of sound sensing into the sensor suite of the Skin Network, is one of the reasons for which the project acronym was chosen as S.N.A.K.E or *Sensor Network Array Kapton Embedded.* Snakes as we know, have the capability of perceiving vibrations and sounds through their skin. The final appearance of the Skin Network when finally assembled was also scale-like like a snake skin

tasks, one of the inherent design tradeoffs that will be mentioned in the software section.

Temperature

For temperature sensing, there was the option of integrating an RTD film into the substrate, however this would have required another Wheatstone bridge for signal conditioning, and a current source to excite it, which translates into a high component count. Including it into the substrate also complicates the fabrication and further increases cost. This option then was discarded and instead a National Instruments LM20CIM7 temperature sensor was chosen. The LM20CIM7 is a precision, low-voltage and low-power, micro SMD temperature sensor. It operates over a -55C–+130 temperature range and has a predominantly linear response with a slight predictable parabolic curvature. Its best feature though is that it outputs an analog voltage that can be directly sampled by the ADC without need for signal conditioning.

Using an integrated temperature instead of a film that can be distributed across the whole substrate has the disadvantage that only the temperature directly affecting the component can be measured instead of that of the bulk substrate, however this can be minimized by encasing the skin in a heat-conductive silicon. This not only helps to average the temperature along the entire surface of the nodes, but also pressure, which as we previously mentioned, had a similar problem.

MSP430 microcontrollers include an internal temperature sensor, but this was not used because it is used to measure the internal core temperature rather than ambient temperature.

3.4 Brain

As outlined in the Network Design section, if a Skin Patch is to be used as a data extraction device, then it must be connected to a PC through a Brain. Brain design did not require much of the previously mentioned design points, because the Brains were implemented with

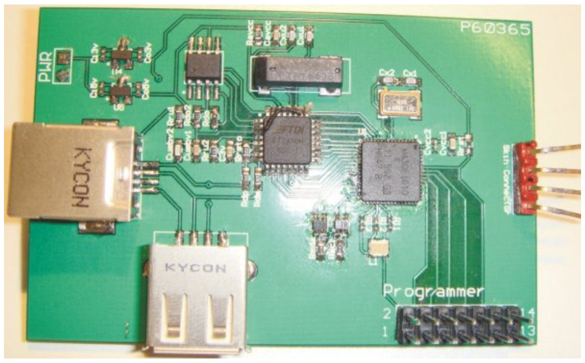


Figure 3-20: Brain

regular PCB fabrication techniques, and their mechanical properties were of no relevant significance. Brains, as opposed by Nodes, need to have the ability of quickly and efficiently relaying information back and forth between the Skin Patches and the PC, while allowing other brains to do so as well.

Given that Brains are not required to do any major data processing, the processing power is not a direct requirement, however larger and more powerful processors are generally needed to handle large amounts of data. In the specific case of this project though, data is locally processed at the nodes and higher-order features rather than pixelated data are transferred to the Brain by the nodes, which allows the Brains to concentrate more on administrative tasks of the network, rather than relaying data back and forth, and it also minimizing the bandwidth required.

Brains are composed of a microcontroller unit, power unit and communications unit; one is shown in Figure 3-20 The schematics and layouts of the brain, which will be briefly outlined in the following subsections, are shown in Appendix C.

3.4.1 Processor unit

The processor unit of the brain is composed by the microcontroller, crystal, programming header and output devices. Having a faster MCU than the one used in the nodes could in theory, transfer data faster. This would be true if the MCU could extract information from the nodes faster than what the bus can provide. However, since the Brain is connected to the Skin Patches using a shared common serial bus, the factor that dictates the transfer speed is the bus and not the microcontroller.

An MSP430F169/10/11 was therefore chosen, which had several advantages. First, it has the same hardware I^{2}C peripheral that nodes have, which makes it easy to interface; second, this is also an ultra-low power microcontroller which is consistent with the overall design goals of the project; and third it is widely available, inexpensive and compact. Keeping in the MSP family also eased potential development overhead

Also part of the processing unit are a JTAG programming header needed for debugging, a crystal, and RGB LED, the same as those used in the nodes. It was considered at some point to add a speaker and other output devices besides the LED to the Brains, however this was discarded because it was decided to use them only as administrative devices to control the flow of data between the nodes and a PC.

3.4.2 Power unit

The power unit is made up of a power supply and power regulators. The batteries used were already mentioned when the node's power unit was described. In fact, the Brains use the same batteries, because power is actually distributed to the Skin Patch from the Brains. This was done because Brains can be encased into a box along with the batteries, as opposed to the Skin patches, which have to be either wrapped around objects or allowed to freely flex and bend.

Two power regulators are needed in the Brains because the processor circuitry requires 3.3V DC, while the chip used for USB communications with the PC needs a 5V power supply. Two SOT-23 Zetex regulators, were chosen: the ZMR500FTA for 5V and the ZMR330FTA for 3.3V. The power provided to the network is unregulated because the nodes already have their own regulators, and is carried by thick traces on the PCB that can withstand higher current loads.

3.4.3 Communications unit

The Communications unit is made up by the I^2C bus that connects it to a Skin Patch, and the USB link that connects it to a PC. Furthermore, the I^2C module is composed of the active pull-up devices described before, a header to connect the brain to a Skin Patch, and the I^2C module of the MSP430 microcontroller. The USB module is formed by the USB controller circuitry, the USB headers, and an external crystal.

Active pull-up devices were created with two XP02401 dual PNP transistors connected as a current mirror, and a biasing resistor. The biasing resistor was chosen so that the current injected by the current mirror to each one of the two I^2C lines was not over the allowed maximum input current of the MSP430 microcontroller (2mA). This can be easily calculated as follows:

$$I_{max} > \frac{V_{cc} - 0.6V}{R_{min}} \tag{3.2}$$

$$R_{min} > \frac{3.3V - 0.6V}{2ma} \tag{3.3}$$

$$R_{min} > 1350\Omega \tag{3.4}$$

A $1.5K\Omega$ resistor was then used for biasing the current mirror.

USB communication was achieved thanks to FTDI's FT245BM USB chip. This device is a USB to FIFO interface chip, that allows any microcontroller to easily have USB connectivity. The device requires its own clocking source, so a 6.000Mhz crystal is also included. Also, if the device is to be named and given a serial number[21] then an EEPROM must be used, and so a 93LC46B EEPROM chip from Microchip was also included. FTDI provides a very detailed design guide[12] for their chips, the Brain's USB circuitry is based on their "self powered 3.3V example".

Finally, it would be prudent to mention that if multiple Brains will be connected to a PC,

[21] This is usually done so that multiple devices of the same type can be later identified

a USB hub is needed. Although in this case the USB controller from the PC was used as the USB hub it is also possible to make each brain a USB hub, itself and daisy chain them one to one another.

Chapter 4

Software

"A Scientist discovers that which exists. An Engineer creates that which never was ".

–Theodore Von Karman

After having described the hardware platform of the S.N.A.K.E. Skin, we now pass on to describe the Software that glues all pieces together and makes it run. The current chapter is divided into three main sections. The first section will describe the embedded firmware that goes into each one of the *Skin Nodes*, and that of *Brains* themselves. The second section outlines the characteristics of the Software necessary to use the Skin Patches as data extraction devices. We end the chapter with a mention of the communications protocols and their associated algorithms, metrics and necessary adaptations.

4.1 Firmware

Given the relatively large amount of embedded processing present in the entire project, a big part of the total Software developed is in fact Firmware. There were four main different firmware components that had to be created for the platform: Node Bootloader, Node Applications, Node Sampling and Brain Firmware.

A key point in the design of the Overall Software and Hardware Systems, has been *Dynamic Reconfigurability*. This term was already mentioned in the previous chapter when we mentioned the mechanical connectors. In the case of the Software System though, this refers to the ability of the Sensor Network to update, change or modify its behavior by modifying its software. This is represented here in two different ways: first, the nodes in the Skin Patches can be reprogrammed at will and second, the application code can dynamically change the sampling rate of the sensors to adjust itself to different stimuli events. We will come back to the later feature when we describe the sampling code in more detail.

Sensor Networks are composed of usually tens or hundreds of nodes; and the current project is no exception[1]. This fact makes ou current method of manually loading code to each one of the nodes using the included programming header very impractical, and we therefore need to look for another option.

Several ways of distributing code in Wireless Sensor Networks have been proposed: Reijers[57], for example, proposes a way to wirelessly reprogram the nodes by only transmitting changes and therefore conserving energy; Boulis[5], in the other hand says that distributing executable images of code to each and every node is inefficient and that having an approach that programs the network as a whole is better suited to large networks; other approaches like the one presented by Lifton[40] use a method called algorithmic self-assembly, in which different algorithms are built with smaller fragments of code.

On top of this, there are even multiple Operating Systems that can be implemented into the MSP430 microcontrollers. Some examples include TinyOS[38], Contiki[9] and FreeRTOS[3]. Which then would be the appropriate combination of code distribution and management technique for our network?

Before answering that question, we have to take into account a fundamental difference between the characteristics of the present project and those of the works presented above, and even further, those of the majority of the literature that is available on the subject: Skin Patches are *Wired Sensor Networks*, and those references specify to Wireless Sensor

[1]A more detailed calculation for the theoretical limit of nodes per Skin Patch will be mentioned in the Communications Software subsection later on this chapter

Networks. Why is this relevant? Mainly because the energy requirements for our project are not as stringent as those for Wireless Sensor Networks. With this in mind, a customized approach was taken: each node was provided with a bootloader capable of dynamically reprogramming application code using the available backbone bus.

4.1.1 Bootloader

Generally speaking, a bootloader is a process that loads programs into memory. These can be one of either an Operating System or an application code. In the more specific case of embedded processors or microcontrollers, however, a bootloader is simply a process that receives a program through a communications channel, writes it to program memory and then executes it. This allows the system to be updateable or upgradeable without the need of special hardware to reprogram the microcontroller.

It is important to note that even though the MPS430 microcontrollers already include a *Bootstrap Loader*[14], this could not be used because it needs to have five lines to be individually routed to every node in the network. So, given the already tight routing of the node layers, making this possible would have entailed adding an extra layer to the material stack. Doing so was highly undesirable because of its impact on cost and mechanical flexibility. On top of this, the MSP430 bootstrap loader would have needed special hardware to be added to the network Brains. Therefore, taking advantage of the MSP430 in-system flash programmer, a customized bootloader was implemented.

Design

There are many things to consider when designing a Bootloader:

- What will be the format of the uploaded code?
- How will this program will be transferred to the microcontroller?

- How much memory is there available for the bootloader and where should this reside in memory?
- Will the bootloader need to relocate or in other way modify the uploaded code?
- Will the bootloader interact with a user or other processors?

But for the particular case of this project, it was decided that:

- Code will be presented in Intel HEX format
- Given that no extra lines can be routed to the nodes, the I^2C Backbone has to be used to upload the code to the nodes
- MSP430F1611/10 have at least 48KB of Flash memory, so the Bootloader should take less than 10% of the memory
- It would be highly desirable if the use of the bootloader is transparent to the user so no changes need to be made to programs for them to work properly
- Since the user will have control over the operation of the Hardware platform, the bootloader will directly interact with a user

The most important aspect to consider though is: Where in memory will the bootloader be located? This is important because the bootloader needs control of the microcontroller after a reset, so it will need to use the reset vector. Interrupt vectors are located at the start of memory, so if the bootloader is placed here, it would then need to redirect the entire interrupt vector to the downloaded program. This would add complexity and latency to the interrupt vectors, so the bootloader was placed at the end of memory instead. Figure 4-1 shows the memory arrangement on the MSP430 once the bootloader has been programmed.

Implementation and Operation

The bootloader was coded using IAR's Embedded Workbench IDE completely in Assembly language. The main reason for choosing ASM over C was purely philosophical, although it

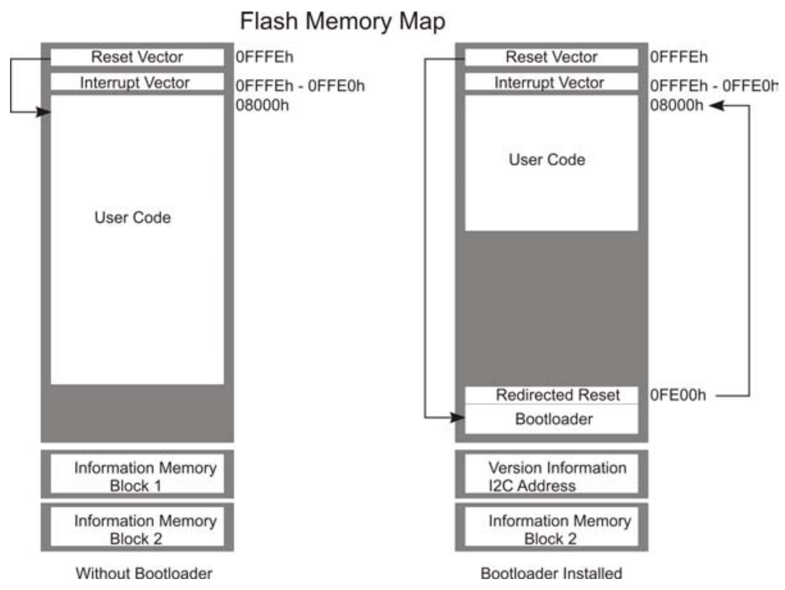


Figure 4-1: Bootloader memory map

Code	Instruction	Description
`0xA0`	`REFLASH_INSTRUCTION`	Reflashing instruction, awaits for code
`0xA4`	`RESET_NODE_INSTRUCTION`	Issues a Watchdog timer fault to reset the node
`0xA6`	`EXECUTE_CODE_INSTRUCTION`	Executes programmed code if any is present

Table 4.1: Bootloader Instructions

has the added advantage of better timing control and less overhead. The total size of the Bootloader HEX file is 794 bytes, which is assembled into the last available flash page.

The Bootloader is started every time a Power-on reset is generated. Once the Bootloader is exectued, it instructs the microcontroller to use the external 8.000Mhz crystal, turn on the Supply-Voltage Supervisor and deactivate the Watchdog timer. It also configures the I^2C port as a slave device. The Bootloader then waits indefinitely for an instruction, which can be any of those listed in Table 4.1:

The program flow can be observed in Figure 4-2, and the full listing of the code is included in Appendix D. Once a `Reflash_Instruction`, is received, the bootloader will expect a file that contains the length of the transmission, the version of the code, and the code itself in a stripped-down version of the Intel HEX format (the start code and type fields are eliminated). If the file is correctly received, and it does not exceed the maximum file size

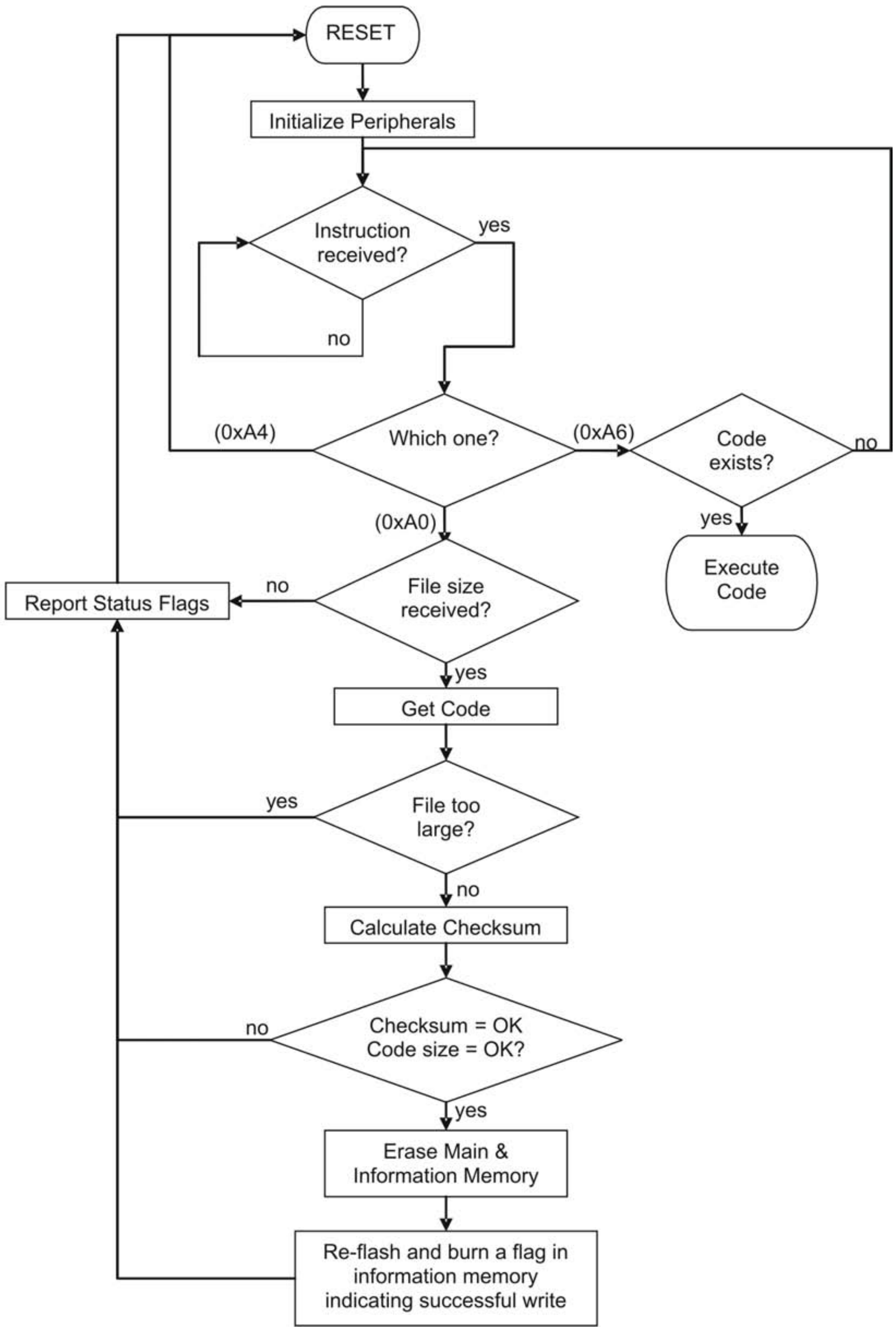


Figure 4-2: Bootloader Flow Chart

Bit	Flag Name	Description
0x01	FILE_RECEIVED_OK	Code file successfully transmitted
0x02	TRANSMISSION_ERROR	I^2C Transmission error
0x04	TRANSMISSION_OVERFLOW	File larger than available RAM
0x08	CHECKSUM_ERROR	Code invalid because of checksum fail
0x10	FILESIZE_OVERFLOW	Code file too large
0x20	INVALID_ADDRESS	Invalid address caused by an address conflict
0x40	FILESIZE_UNDERFLOW	Receiver expects more bytes than actually received
0x80	REFLASH_OK	Device reflashed successfully

Table 4.2: Bootloader Status Byte Flags

allowed, the bootloader will calculate a checksum for every Intel HEX line, and compare it to the one received. If the checksum succeeds, then it proceeds to erase the main and information memories, and burn each line of code into its specified address. The process will only succeed if all the lines were successfully copied into flash, and no invalid addresses were found[2]. The only case in which the code is modified before burning it into flash, is when the reset vector address is detected within a line, in which case the bootloader will redirect it to the to the first address specified in the received code. A flag byte is always reported back to the Brain indicating the status of the operation; if the code was received as a broadcast message, the status byte is not reported until the Brain polls each node, otherwise it is sent back after programming is complete or an error found. Table 4.2 lists the different flags encoded into the response byte

Flag 0x01 means that the node correctly received the file, but it does not necessarily mean that it was burned into flash. Flag 0x02 might be reported if there were problems in the I^2C bus. There are two flags that indicate an overflow; this is because to make efficient use of the I^2C bus, the code file is not transferred (Intel HEX) line by line, but 4KB blocks are sent at once. Flag 0x04 indicates that a block was larger than 4KB and it can't be accomodated into RAM, while flag 0x10 means that the code file can't fit into flash[3]. Flag 0x08 is reported if any line fails the checksum. Flag 0x20 will be set whenever a line has an address occupied by the bootloader which prevents it from being overwritten. Flag 0x40 is set if the transmission file indicates a transmission larger than was actually received, and

[2]An invalid address will be any address that conflicts with the bootloader code space

[3]This might be due to either the code size being larger than the actual Flash, or that it is larger than the Flash minus the size of the bootloader itself

flag 0x80 indicates a successful write to flash.

As previously noted, the Bootloader will wait indefinitely for an instruction, but it could be easily changed to start execution of a program after a determined amount of time. Once execution starts though, the bootloader can't be invoked again unless a reset is generated (hardware or software initiated) or the application code explicitly includes a mean to invoke it. This can be easily accomplished by generating either a Watchdog timer timeout or an invalid write to its counter register.

4.1.2 Sampling Code

The second large firmware component is the application code that deals with sensor sampling. This is probably, the principal software component, given that it is mainly where the*Dynamically Reconfigurable* feature of the Skin Patches lies[4]. As the reader may recall, one of the main goals of the project is to demonstrate that the amount of required bandwidth and other resources can be minimized if each node is given processing power. By doing this, the need to route pixelated data to a centralized processing unit is practically eliminated.

Although this code could be used if the Skin Patches are used as data extraction devices, it is mainly intended for the case when the Skin Patches are being used as stand-alone devices. Its main purpose is to set the rate at which the sensors are sampled, process the generated data, calculate higher order features and possibly transfer these to a PC through a Brain if further processing is needed. Concurrent sampling and processing is possible because the sampling and timing of the application is carried out by the microcontroller peripherals, so the processor core is left free to process the generated data. This is shown in Figure 4-3.

Lets now describe what the microcontroller does after starting up. After a reset, the application configures the clocking unit to run at 8.000 Mhz, initializes the I^{2}C module as a slave device and designates an ADC memory channel for each one of the sensor modes[5].

[4]The other one being its ability to be reprogrammed

[5]The MSP430 ADC module is equipped with 10 sampling channels each one of them with a 16-bit conversion result register

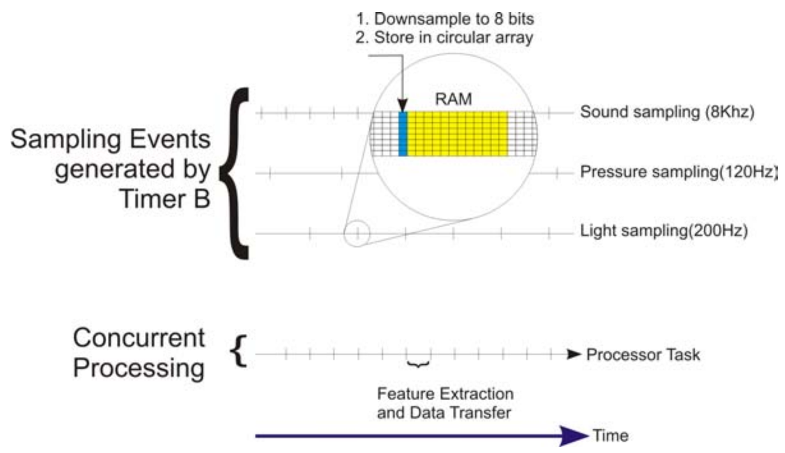


Figure 4-3: Sampling timing example

Then, the Timer B peripheral is configured to generate 6 different timing intervals which trigger the sensor sampling events. With this approach, each sensor can be sampled at a different rate according to its needs. Once a sensing event is generated, its associated sensor is sampled and the result is stored in its assigned memory register. Depending on the sensor type, some additional tasks are sometimes performed before their information is stored, and higher-order features can be then calculated. Sensors that need additional tasks are mentioned in the following list:

Pressure sensing: since there are four pressure sensors controlled by a multiplexer, every time a sampling event is generated (120Hz), the multiplexer address is changed, and a different sensor is sampled. This effectively reduces the sampling rate of each one of the four pressure sensors to 30Hz. After sensing, pressure samples are downsampled to 8-bits, and stored in a 4-byte array, one for every sensor.

Sound sensing: sound is the only sensor which is not downsampled for signal fidelity reasons. It is also the sensor with the fastest sampling rate (8Khz). The result of the samples is stored in a 1Kb circular array that uses 2 bytes per sample.

Strain/Bending: strain gages are gated through a MOSFET transistor so that the Wheatstone bridge used for both strain gages doesn't waste energy when not in use. When a sampling event is generated for either one of the strain gages, they are first turned on,

Sensor Type	Sampling Rate	Bit Depth
Strain/Bending	60Hz	8-bit
Proximity/Activity	120Hz	1-bit
Pressure	120Hz	8-bit
Ambient Light	200Hz	8-bit
Sound	8Khz	12-bit
Temperature	80Hz	8-bit

Table 4.3: Sensor sampling rates and resolution

and once sampling is done, they are turned off to conserve power. Only one sample is stored per sensor.

Whiskers: whiskers are the only sensors that are not sensed through the ADC. The reason for this is that it is mainly the presence or absence of activity that interests us. So instead they are connected to an interrupt-capable input pin, so that they generate one or many interrupts whenever they are stimulated.

Now, as we may recall from the previous chapter, there are a total of six different sensing modalities, but a total of 11 sensing channels (4 pressure, 2 whiskers, 2 strain gages, 1 microphone, 1 ambient light, 1 temperature). The interesting thing is that each one of them can be individually configured to detect different kinds of stimuli. And, even though sensor sampling rates and resolutions are set at start-up, these can adapt and go to nearly zero or increase depending on the history of samples and the current stimuli. Table 4.3 summarizes the default sampling rates and bit depths for the different sensors.

These sampling rates and resolutions may seem arbitrary, however they were chosen so that enough information could be extracted from each sensor to divine some higher order features, but at the same time sampling as low as possible so that the processor can keep up with the generated data. The reader may notice that some sampling rates, like the one for temperature, could be made much lower indeed. This was deemed unnecessary because only processing speed and not sampling speed has an impact on performance. For these cases data was just simply overwritten. Furthermore, having a higher temperature sampling rate may allow to detect human touch.

We also mentioned before that there is an inherent tradeoff in the system as processing power must be divided between sampling, data processing and data transmission. As previously mentioned, these rates, can be changed at any time by either modifying the application code and reprogramming the network or by modifying the code so that it can be dynamically changed without the intervention of a Brain. This is where the beauty of the system lies: it can either statically (through instructions received from a PC or Brain) or dynamically (using data history and neighbor data) adapt to external stimuli based on the needs of the task at hand.

Finally, we have mentioned that sensor data is used to calculate higher-order features, but none have been described so far. There are, however, some features calculated for sound, activity, pressure and bending (strain gages). In the case of sound, a history of 1024 samples is stored (about a quarter of a second) in a circular array, which is then used to calculate the mean and variance of the signal. In the case of the whiskers, a counter stores the amount of positive interrupt edges generated over a second. Strain gages are used as bend sensors, so the reading is converted to an approximate bending angle[6].

Although these are just a few limited examples meant as a proof-of-concept, there are many others that can be calculated and were left for future revisions of the system. These will be mentioned in the last chapter, along with some other future work suggestions.

4.1.3 Node Application Code

The third large firmware component is made up of several other smaller applications that were meant to demonstrate some of the hardware capabilities of the system. They are basically different ways of mapping sensed data to the LEDs and making use of the two available communications channels: I^{2}C and P2P. These applications, served to test the functionality of the Skin Patches as data extraction devices.

The first application uses the light sensors to create a negative feedback loop that turns on the green LED based on the amount of light in the environment. The more light in

[6]An approximation was used based on the data extracted from experimental measurements, which will be shown in the Results chapter later on

the environment, the less active the LED is. This was done by using the timers of the microcontroller for Pulse-Width Modulating the LED light intensity. This application in fact resembles the Retina PFRAG presented by Lifton in his pushpin computing platform[40], which illustrates how this is indeed a Sensor Network, similar to many of those already available. If the Skin Patch is connected to a Brain and an instruction is received to start sending data, data is constantly sent to a Brain every time the node is polled. The interesting thing about the application is that by simply changing the sensor type, any sensor can be constantly monitored on screen in real time for an entire Skin Patch. More on this, however, will be mentioned once the visualization software is described.

The second application was intended to be used as a test for the pressure sensors. This was done by blinking LED lights based on the amount of pressure detected from the pressure sensors. Different corners of the nodes were mapped to different colors of lights.

The third application made use of the P2P communications to transfer a token that could be interchanged between nodes if the pressure sensors were stimulated. This application, however, is only mentioned, and will not be shown but because of its early development phase. Full code listings for applications one and two are shown in Appendix D. Again, the code for the third application was not included because it was not completely debugged and tested.

Just before going on to describe the Brain's firmware, it is important to mention that all application code must be executed explicitly by the user by issuing an Execute Code instruction. This is because, as we previously mentioned, after a reset, the Bootloader always has control over the processor, and it will wait indefinitely for an instruction.

4.1.4 Brain

The fourth and last firmware component is that of the Brains. Brains are the bridge between Skin Patches and a personal computer for the case when the skin is to be used as a data extraction device. Brains were already mentioned in the hardware section, where they were described as Skin Network administration and data relay devices. Brains not only arbitrate

Code	Instruction	Description
`0xA0`	`REFLASH_ALL`	Receives code and broadcasts it to the Skin Patch Network
`0xA2`	`RESET_BRAIN`	Issues a Watchdog timer fault to reset the brain
`0xA4`	`RESET_SKIN`	Broadcasts a reset instruction to the Skin Patch Network
`0xA6`	`EXECUTE_CODE`	Issues an execute instruction for the addressed node
`0xA8`	`REFLASH_DEV`	Receives code and forwards it to the addressed node
`0xAA`	`RESET_DEV`	Issues a reset instruction to the addressed node
`0xAC`	`START_SAMPLING`	Indicates the network to start sampling and sending data
`0xAE`	`RESET_NODE`	Stops sampling and data transfer
`0xB0`	`DEEP_SLEEP`	Hibernation mode (ultra-low power consumption)

Table 4.4: Brain Instructions

the flow of information between the PC and a Skin Patch, they also configure, synchronize and maintain a stable network.

PC software directly interfaces with the firmware of the Brains, so a set of instructions had to be implemented so that the user can communicate to the Brain what needs to be done. Many of these are explicitly requested by the user, while others are internal instructions that usually need to be executed by others. Table 4.4 lists the available instructions of the brain:

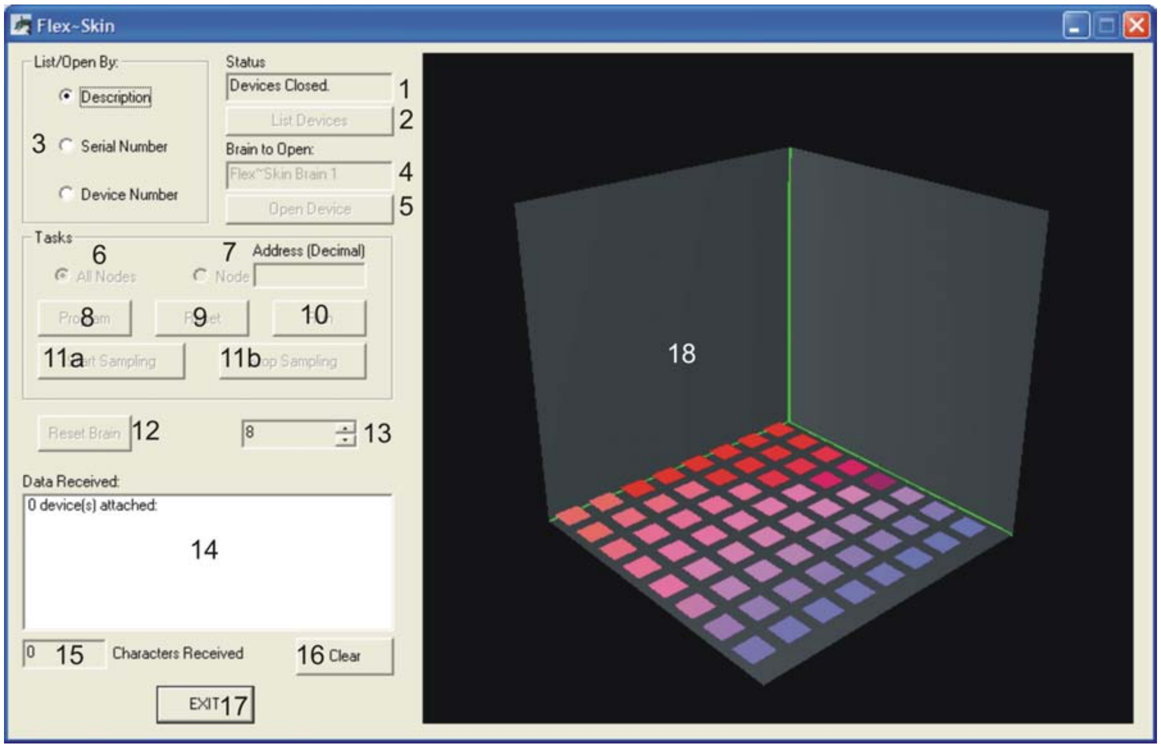
As it can be seen, the brains can selectively reprogram[7] or reset a single node in a Skin Patch Network, or do so for the entire network depending on the command received from the computer. They can also indicate nodes to start/stop execution of application code or sampling code, and/or send a patch of skin to deep-sleep mode to conserve power. An extra instruction was also added to reset the Brain in case this gets stuck.

These are not, however, all tasks carried on by Brains. In fact, most of their tasks are executed automatically and thus are transparent to the user. This was done so as to facilitate their use and minimize the need for administrative tasks. Some of these include: management of both communications channels (USB and I^2C), data flow control and buffering, and synchronization.

[7]If all nodes are programmed at once, programming code is sent to the I^2C broadcast address. This activates the `General Call` flag in the MSP430 microcontroller, which allows it to differentiate if code was sent only to it or to all of the nodes in the network. This is useful because a status byte is transferred back to the brain. If the code was only sent to one node, this status byte is transferred right away, otherwise each node will wait until polled by the Brain.

4.2 PC Graphical User Interface

The second software component developed for the project, is the GUI that serves as a front end to the Skin Patch networks. As the reader may recall, this application is mainly intended for the case in which the Skin Patches are used as data extraction devices, however it could also be used to reprogram a Patch of Skin that is being used as a stand-alone device. The user interface has three main purposes: Network Discovery, Network Programming and Data display and logging. Figure 4-4 shows the application window indicating its components, and Table 4.5 gives a quick description of these.


List/Open By: [✓ Description] Serial Number Device Number

Status: Devices Closed

Brain to Open: Flex~Skin Brain 1

Tasks: [✓ All Nodes] Node

Address (Decimal):

8

Data Received: 0 device(s) attached

0 Characters Received

Figure 4-4: GUI Components

These functions along with others will be described in more detail in the following sections.

Component	Description
1	Indicates the current network status
2	Lists the currently connected Brains
3	Brains can be either listed by any of these three options
4	This box is used to specify which Brain to work with
5	Opens a new device to work with
6	Specifies that all nodes will be affected
7	Specifies that only the node with the typed address will be affected
8	Programs the specified node in the network (affected by 6&7)
9	Resets the network (affected by 6&7)
10	Starts execution of code (affected by 6&7)
11	11a starts network sampling; 11b stops network sampling (affected by 6&7)
12	Resets the currently opened brain
13	Changes the amount of *Display Nodes* in 18
14	Displays messages and echoes received data
15	Displays the number of bytes received
16	Clears 14&15
17	Exits the application
18	Displays received data from the network (size determined by 13)

Table 4.5: GUI Components

4.2.1 Network detection

The reader may recall that one of the design directives of the system was to make it scalable. Software plays an important role in this regard because it has to be able to manage networks of any arbitrary size and shape[8]. With this in mind, the software was designed in such a way that it can work with several different Skin Patches by allowing numerous Brains to be connected concurrently to the system. The ability of FTDI's FT245BM USB chip to allow several identical devices to be operated in the same bus, and its very simple API were in fact the two reasons for which this device was chosen. Before explaining the way that the GUI manages the Brains though, it is necessary to provide a quick overview of the low-level interface between the OS and the device drivers.

Windows requires all hardware devices to have a device driver installed so that the Operating System can understand how to handle it. FTDI provides two different drivers: A **VCP (Virtual COMM Port) driver** and the **D2XX Direct Driver**[28]. The first one turns

[8]Up to the limits set by hardware

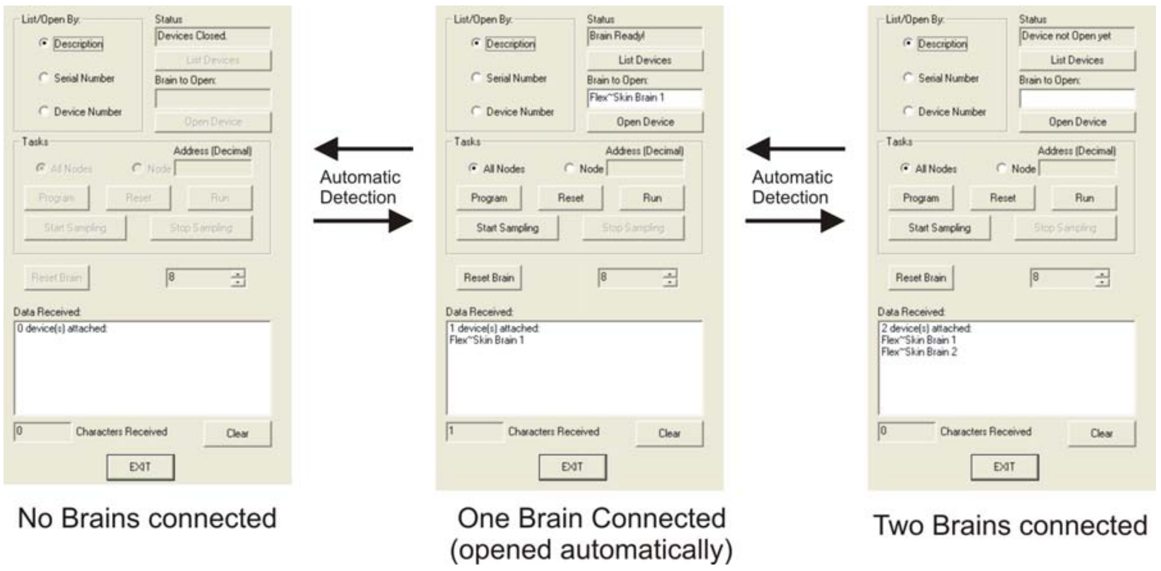
the USB device into a regular serial COMM port that can be accessed using Windows COMM API. The problem with this approach though, is that it needs to install a different driver for every device; so if 10 Brains are connected at once, there will be 10 different Virtual COMM Ports, which is highly undesirable. The second driver, however, allows the device to be accessed through a dynamic library or DLL, and therefore allows multiple devices to be operated concurrently. This was the driver used by the application.

Once Windows detects that a Brain has been connected, it automatically loads its driver. If multiple Brains are connected, however, Windows can't address them individually because the driver has no way of recognizing one Brain from the other because their chips are identical. To get around this, the USB chip uses an EEPROM chip that can be loaded with a serial number so that Windows can differentiate them. This was done using a special application provided in FTDI's website (MProg 2.8a).

Going back to our application, for it to be able to work with a Brain, they have to be successfully *enumerated* and *opened*. Enumeration has to be done in order for the driver to recognize the specifics of the devices connected to the bus. Once devices have been enumerated, the application must open a handler to the device through which communications can then be established. Enumeration is always carried out automatically by the application, but the user must manually open the device[9] with which he wishes to work with; this is done by typing-in either the name, serial number or device number (depending on the selected option) into the edit box (number 4 on Figure 4-4) and then clicking the Open Device button. The only case when a device is automatically opened is when the application detects that only one Brain is connected. This behavior is shown on Figure 4-5.

Automatic Enumeration is done by registering the application so that it can receive notifications whenever hardware changes. This is done by registering the application with a device filter so that it is only notified by Windows whenever a USB device insertion/removal event is generated. Since Brains are therefore detected and enumerated automatically, the system is to a certain extent "plug and play"; it is not fully plug and play because nodes in a skin Patch are not currently automatically detected. This point will be covered in the

[9]In this context, *device* refers to a Brain with an attached Skin Patch



List/Open By: [✓ Description] Status: Devices Closed. Brain to Open: Tasks: [✓ All Nodes] 8
Data Received:
```
0 device(s) attached:
```
0 Characters Received

List/Open By: [✓ Description] Status: Brain Ready! Brain to Open: Flex~Skin Brain 1 Tasks: [✓ All Nodes] 8
Data Received:
```
1 device(s) attached:
Flex~Skin Brain 1
```
1 Characters Received

List/Open By: [✓ Description] Status: Device not Open yet Brain to Open: Tasks: [✓ All Nodes] 8
Data Received:
```
2 device(s) attached:
Flex~Skin Brain 1
Flex~Skin Brain 2
```
0 Characters Received

Figure 4-5: Automatic configuration of Brains

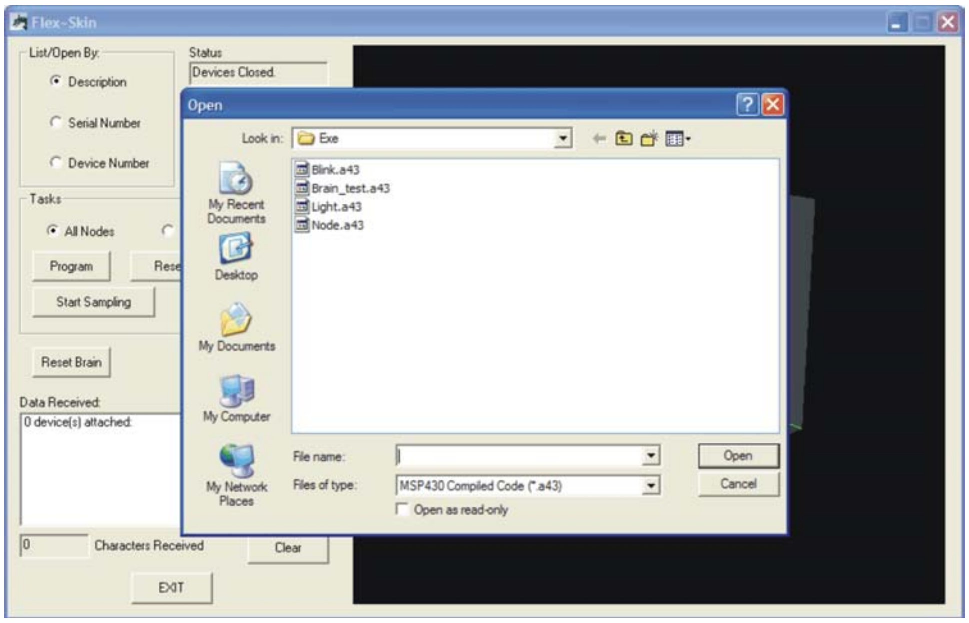
last chapter.

Two other things are left to mention: first, all functions will be disabled until a successful enumeration/opening sequence is executed. And second, enumeration can be always be carried out by selecting an enumeration option in the radio buttons (component number 3). This is useful if several Brains are connected, and their ID's or serial numbers need to be extracted.

4.2.2 Programming

The second major task, and perhaps the most useful one of the PC software is its programming functionality. This has been mentioned before, but what is the role of the GUI? It is actually a really simple one, since the complexities of programming the nodes are mainly carried out by the brain firmware.

There are two ways in which the GUI can program a Skin Patch: programming a single node or the entire Patch. These options are specified by components 6 and 7 (the two radio buttons) shown in Figure 4-4. When the program button is pressed, a modal File Open



Status: Devices Closed

Data Received: 0 device(s) attached

0 Characters Received

Look in: Exe

File name:

Files of type: MSP430 Compiled Code (*.a43)

Figure 4-6: GUI Programming Environment

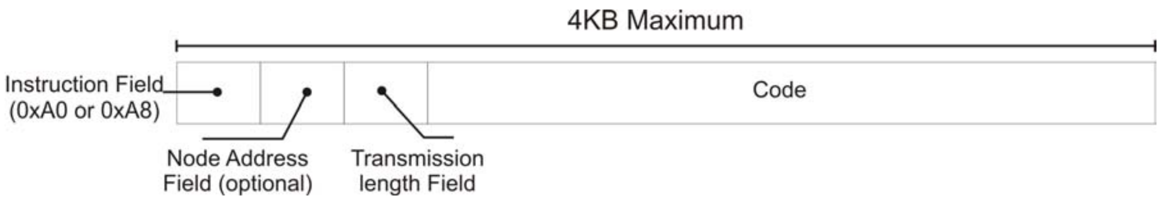

4KB Maximum
Instruction Field (0xA0 or 0xA8)
Code
Node Address Field (optional)
Transmission length Field

Figure 4-7: Programming packet

dialog box is opened so that the user can select the code file that will be used to reprogram the network. This is shown in Figure 4-6.

If no file is selected, the operation is cancelled, otherwise the application will assemble a code package of up to 4KB in length by parsing the file to remove its unused fields (leading colons and field type) and appending extra needed fields. These fields are the appropriate instruction, (either `0xA0` or `0xA8`), depending if it is a single node or the entire Patch, the node address in case this is a single node operation, and the total transmission length. This is then transferred through USB to the currently opened Brain, and then forwarded by this to the appropriate recipients. This can be seen in Figure 4-7

An important point to stress is that the code has to be in Intel HEX format. This can be generated in several different ways. In this case, IAR's Linker was used to generate an output file in this format. The application can be either coded in MSP430 Assembler or C, but it will ultimately depend on the capabilities of the compiler used.

4.2.3 Sampling and Display

The next component of the GUI software is the Sampling and Display sub-unit. This is conformed of components 11a, 11b, 13, 14, 15, 16 and 18 and it allows the user to visualize extracted data in real time, as well as store the information for further processing. These functions can be used in either single node or full patch modes. Visualization and Data logging are always started and stopped at the same time, however they are two separate functions and each will be described separately.

Data visualization is done by embedding an OpenGL picture control into the application window. OpenGL is probably the most widely used graphics programming language because it easily allows 2D and 3D images to be created. The OpenGL control was then used to create an array of 3D-bars to plot data values read from the sensors. Whenever a data value is received, it is mapped to the height value of its corresponding 3D bar, so the sensor value can be visualized in real time by observing how the 3D plot changes.

Several additional effects and controls were added to allow the user to easily change the perspective of the image, and to enhance the visual aspect of the displayed data. The chart can thus be rotated around its X and Z axes, panned (up, down, left and right) and zoomed (in and out), all in real time using the mouse pointer; this was done so that data that may be obstructed by other bars, can be easily visualized. Picture quality was enhanced because in 3D rendered scenes, this depends heavily on the lightning effects used, so a full shading and lightning model was also included in the rendering pipeline. Finally, since the necessary amount of display units (3D bars) can vary according to either Skin Patch size, the type of sensor being monitored[10], or the sampling mode selected (single node or full patch) a spin

[10]Skin Patch size affects the number of necessary display units because it affects the number of nodes that

control was added to change the number of bars that make up the 3D chart. Figure 4-8 shows how these controls affect the chart area[11].

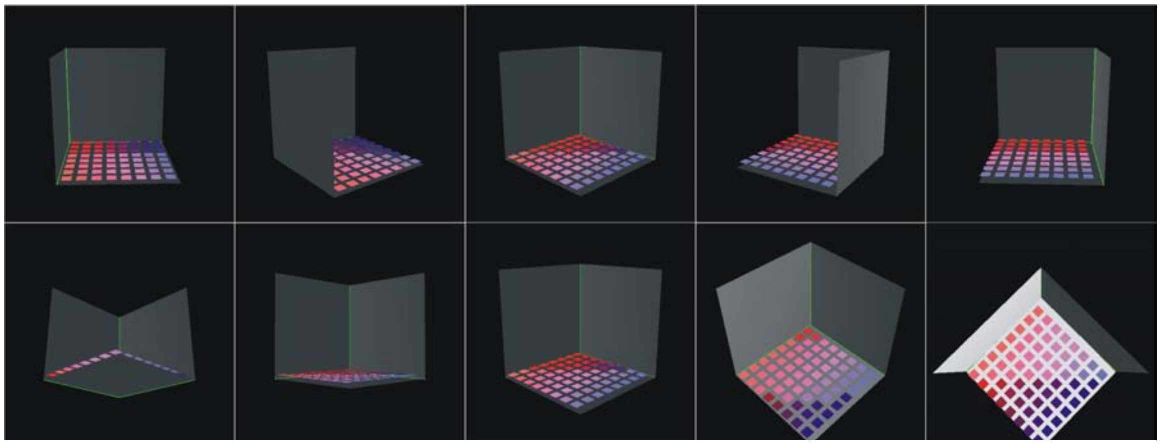
The reader may notice that there is a color gradient in the different columns that make up the plot. This is merely a cosmetic change that was done to better visualize the different bars, because visualization was a bit difficult when bars were all the same color. Yet, this may be used as an extra degree of freedom in case a physical quantity has to be mapped to a display value.

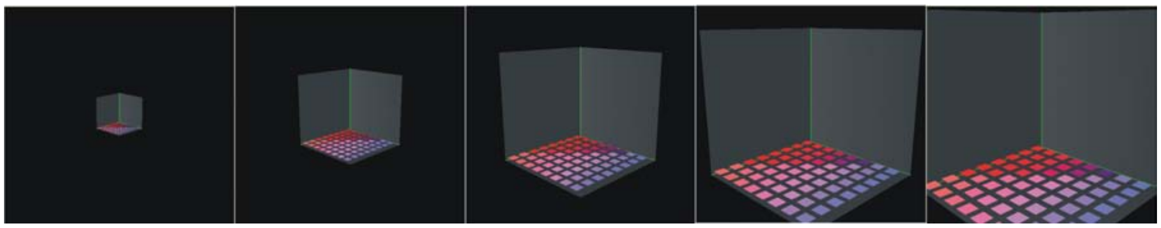
The second function of this sub-unit is data logging. Every time the start sampling button is pressed, a modal File Save dialog box is opened to ask the user for a file name. If this is not provided, then sampling does not start, otherwise, several things will happen:

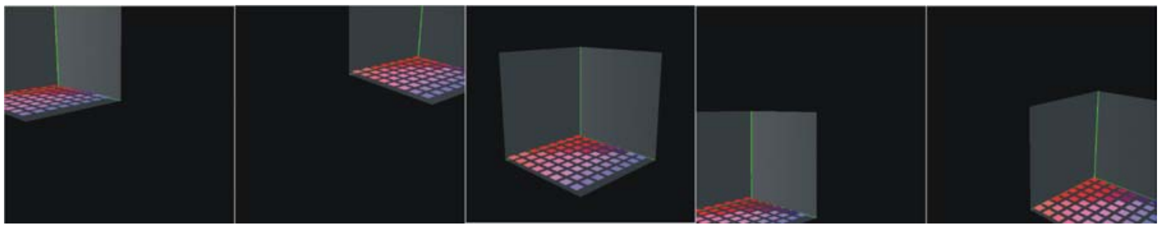
1. The file specified by the user will be opened or created if it does not exist
2. All buttons except for Stop Sampling become disabled
3. An Execute Code instruction is issued to ensure that the nodes are executing the code
4. A thread is started to handle the sampling process
5. If All Nodes is selected, the entire Skin Patch will be scanned for new sensor data by issuing instruction `0xAC` for every node in the Patch. Otherwise, if a node address is specified, only this sensor will be sampled
6. Data from each sensor is then converted to a decimal number, and stored in a different column for every sensor
7. Data is sent to the OpenGL control for visualization
8. The cycle continues until the Stop Sampling button is pressed

have to be monitored. The type of sensor affects it because the amount of sensing channels varies according to the sensing modality being monitored, 4 for pressure, 2 for strain and whiskers, 1 for temperature, ambient light and sound

[11]These charts are shown without any data so that the effect of the transformations can be seen. Other plots with data will be shown


(a) Rotation control


(b) Zooming control


(c) Panning control

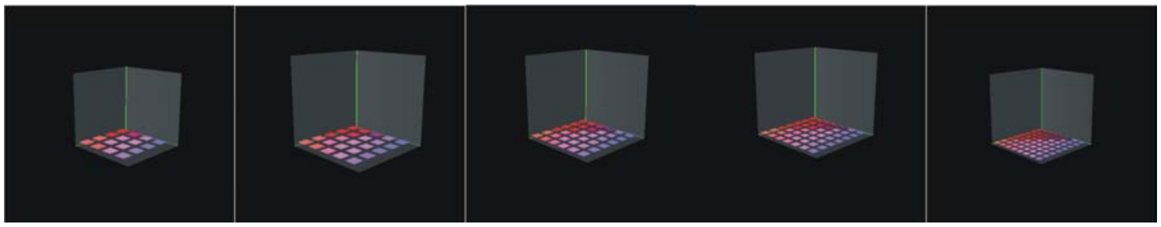

(d) Sizing control

Figure 4-8: Visualization: OpenGL 3D chart transformation modes

Once the process is stopped, the file is closed, the thread is finished and all buttons are now enabled (except for the Stop Sampling button). The file is stored in a Microsoft Excel readable format so that further processing/plotting can be done.

It is important to mention that although the application is capable of showing the data from any one of the sensor modalities in the Skin Patches, the functionality to extract data from all sensing modalities at once has not been implemented so far[12] because of the complexity involved in displaying data from many different sensing modalities effectively. Sampled data is also controlled by the installed application on the nodes, therefore if a different sensor needs to be sampled, the application on the nodes has to be changed. Data extracted from all types of sensors will be shown in the results chapter.

4.2.4 Other Functions

Finally there were some other minor functions that were not mentioned before, but that are also useful to have. These are summarized in the following list:

- Resetting Brains (Button 12): useful if either the Brain or the I^2C bus gets stuck
- Executing node applications (Button 10): to indicate the Bootloader to start execution
- Resetting nodes (Button 9): resets either a single node, or the entire patch.
- Amount of bytes received (component 15): useful to debug the network.
- Address validation (component 7): The node address must be in the valid range (0-1023) or the task buttons will not be enabled

Many other functions of the software were left unmentioned intentionally, however the reader is welcome to check the full listing of the different project files in Appendix E.

[12]This means for example that temperature readings and light readings can't be sampled at once, but *not* that several nodes can't be sampled at the same time which as it was previously noted is indeed possible

4.3 Communications Software

The last part of the software section corresponds to the specifics of the communications channels used in the entire project. This will briefly outline some of the necessary design choices that were taken along the whole process.

4.3.1 I2C Backbone

We have talked about the hardware considerations of the I^2C backbone, but nothing has been mentioned about the software that runs the bus. The I^2C bus requires two lines to be routed among all devices which are to be connected to it plus a common ground. This means that whenever a device needs to talk to another one in the bus, it needs a way to address it. In fact the I^2C bus has two different addressing schemes: 7-bit and 10-bit.

The addressing scheme is actually the theoretical limit to the number of devices that can coexist in the same bus. The first one allows a maximum of $2^7 - 1 = 128$ devices while the second one allows $2^{10} - 1 = 1023$ devices. Naturally, the second one was chosen to allow a larger number of devices to be connected to the bus. These numbers are not the actual allowed limit, because the input capacitance of all pins connected to the bus add to the total, which as we already mentioned cannot exceed 400 pico-farads. We already mentioned some ways to get around this limitation, however for the case of the current project, the real limit to this will be presented in the Results Chapter.

There are however several questions that must be answered though: How are these addresses assigned? Is it the Bootloader which assigns the address or is it the application code or both? What happens if there are more than one device in a Skin Patch with the same address? How is the bus arbitrated?

First of all, the scope of an I^2C address is limited to the Patch of Skin to which the node is connected. So although there can't be two nodes with the same address in the same Patch of Skin, addresses can be reused if a different Brain is used. In other words, as long as there are not two nodes with the same address on the same Skin Patch, addresses can be arbitrarily assigned. In the specific case of the Skin Patch assembled to test the project, 12 nodes were connected and addresses `0x100`–`0x10A` were assigned, to consequent nodes.

Now, all nodes are first loaded with the Bootloader program described before, however what was not mentioned before is that when the Bootloader is programmed using the JTAG debugger, the I^2C address assigned to the node is assembled into location `0x1080` in flash memory. This is done so that I^2C addresses are only assigned once in the programming lifetime of the nodes, and no conflicts are created in the bus. If two nodes with the same address are placed on the same bus, the system will eventually reach a deadlock state which must be manually reset, which is why it is so critical to avoid this situation.

The solution to the last question, "How is the bus arbitrated?" is actually really simple: the MSP430 does it. The I^2C hardware module of the MSP430 microcontroller, can detect collisions in the bus and has a method to resolve contentions. Whenever a collision is detected, the device that first takes the data line to ground loses arbitration and an interrupt flag is generated. Processing this interrupt flag allows communications to remain reliable. These cases are rare though, as only one master exists in every Skin Patch: the Brain.

The last consideration to make about the I^2C backbone bus is the speed at which it is run. This is the second parameter that limits the amount of nodes that can be connected to the same patch. Although bandwidth requirements are supposed to be brought down with the addition of processing power into each node, there is still the need to transfer either higher-order features into a PC, or simple raw data if the Skin Patch is to be used as a data extraction device. The amount of bandwidth needed will depend on a variety of factors, like the number of nodes, the sampling rates, and the resolution at which each sensor is sampled. Making the bus run at fast-speed (3.4Mbps) would have placed huge constraints on the length of the bus lines, and running it at the standard speed of 100Kbps would not have been enough for more than a couple of nodes. Therefore high-speed mode was chosen, and the bus is then run at 400Kbps[13].

[13] Although these are the standard speeds of the I^2C standard, the I^2C module of the MSP430 can run the bus at any arbitrary speed, up to a limit of close to 500Kbps

4.3.2 Peer to Peer

Peer to peer communication was added to every node so that data could be exchanged between them without the need to use the backbone bus, which is only to be used for data transfer between a Brain and the Patch Network. With this, it is possible to combine data from each node's own sensors with that of adjacent ones so that more information can be extracted across the footprint of a stimulus event, and either sampling rates can be adjusted, or features like center of mass and sound directionality can be calculated.

No application currently developed for the system utilizes this capability, however this is mentioned before for two reasons: first, because it is part of the design of the system, and second because it sets the basis for future applications and algorithms.

One of these possible algorithms is precisely a Network discovery algorithm, which is critical for a Skin Patch that can be assembled an disassembled. Our current project has a fixed number of nodes in a known topology, so all the software developed for the PC and most of the firmware was created knowing beforehand the configuration of the Skin Patch network. Yet, for the more general case in which the number of nodes, and the way they are connected is not known a priori, a method must be provided so that the Brain firmware and PC software can identify how a particular Skin Patch is connected. The algorithm may look something like this:

1. Every Brain should ping all the addresses to find out which nodes are present
2. For every node in present, the brain will obtain a neighbor list
3. Once a list with all nodes is created, a routing table is generated
4. With the routing table generated, all P2P links can be established and the topology of the Skin Patch Network is found

An alternate way of doing this would be to have each node report itself to the Brain once it is connected, however this would need the I^{2}C bus to work as a multiple master bus. This is actually possible but, once again it is left as a future enhancement to the project.

4.3.3 USB

The last part of the communications software is that of the USB link. As previously mentioned in the Network detection subsection, all USB communications are handled by the D2XX driver. Not much else can be mentioned about it, as most of this was extracted from FTDI's programming guides. For a full reference of the D2XX API the reader is welcome to check FTDI's Programming guide[28], or the full listing of the code in Appendix E.

Chapter 5

Analysis of Results and Evaluation

"No amount of experimentation can ever prove me right; a single experiment can prove me wrong".

– *Albert Einstein*

Once hardware and software components have been described, we proceed to show in the current chapter the obtained results. First, the mechanical performance of the system is presented. Then, the second section will analyze each sensing modality and show data plots for each one. The third section analyzes both communication links, and evaluates the actual operation speeds, data throughput and overall system scalability. The fourth section of the chapter describes how well the system performed as an ultra-low power device under several conditions, so that it can be then evaluated against the design goal. Finally, the last section presents a brief analysis of the involved costs.

5.1 Mechanical Stability

The first of the Results subsections will talk about experience with the mechanical characteristics of the system. These are mentioned first, because several mechanical characteristics of the design greatly affected several of the results presented in this chapter. There were three main characteristics of the mechanical design that affected the overall system performance. The first was related to the substrate, the second to mechanical connections and the third to pressure sensors.

5.1.1 Substrate

Given that the prototyping phase is so expensive it was nearly impossible to exactly know how flexible the material would be until the final design was fabricated. When the nodes were fabricated, the final substrate thickness was actually less than the one specified by the project, however its flexibility and compliance to conform to the shape of the soldered components was far below what was expected. This means that instead of having a substrate with mechanical characteristics similar to that of a thick sheet of paper, it ended up being almost like a credit card. The problem with this is that the substrate does not conform to the body of large components soldered to it like the microcontroller, so whenever the substrate is bent, these components easily break off. This ultimately had an enormous impact on the Skin Patches because they now had to be handled with extreme care to avoid breaking off the microcontrollers. For this reason, the data plots and snapshots extracted for sensing modalities that require physical warping of the Skin Patch like strain and pressure are only shown for a single node to avoid any excess strains.

The fact that components may eventually break off because of skin patch bending was actually expected, however this problem was much worse than was expected in the case of the microcontroller. Not all was a total failure in this regard though, because amazingly enough, none of the other components presented this problem. In fact, all the other components were remarkably resistant to even sharp bending angles; even the larger ones like the MEMS microphone and crystal. What was the difference? Footprint size and type. The microcontroller comes in a 64-pin $9mm \times 9mm$ QFN leadless package which means that when the node was bent all the stress was absorbed by the microcontroller pins. Since there is nothing but a really small amount of solder holding the microcontroller pin to the substrate pad, these break off very easily. This does not happen with other components

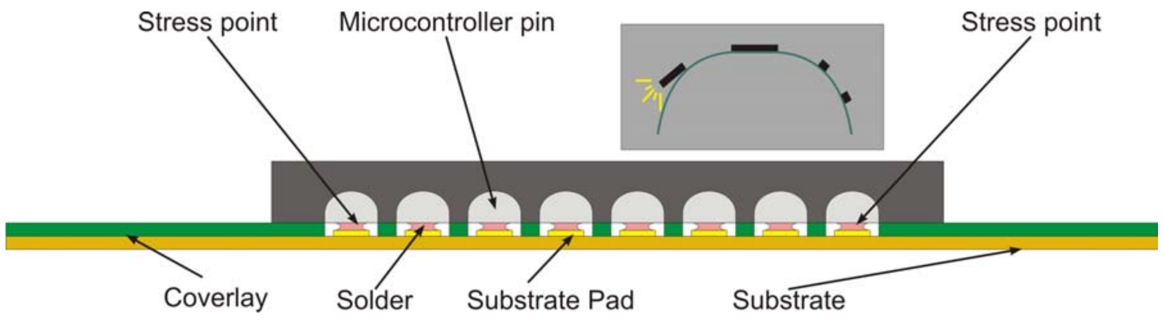


Figure 5-1: Stress concentration points in node microcontrollers

because the pin leads can absorb some of the stress and because they are so small that even sharp bending angles don't stress the soldered connections enough to break them. This effect is shown on Figure 5-1. There are several ways around this problem, some of which will be mentioned in the Conclusions chapter.

5.1.2 Connections

The second mechanical aspect to evaluate was the mechanical terminations used. We mentioned before that mechanical terminations were critical to maintain good network stability, yet there were two problems with the ones chosen in the design: header hole size and connection flap size.

Something that was not foreseen at the design stage of the project is that header holes were so small that they actually caused connections to easily crack when the node was bent. This generated numerous connection flaws which sometimes prevented the correct operation of the I^2C network in the Skin Patches.

The other thing that was not well though when the design was made is that since headers were placed on a flap that protrudes from the body of the Skin Nodes, whenever a Skin Patch experienced some bending, all the bending happened mainly at the flaps and not at the node body. This was actually a good thing to relieve some of the stress that could concentrate on the microcontroller pins, however it was detrimental to the performance of the strain gages. This is shown on Figure 5-2. Because of this, even if the patch is bent,

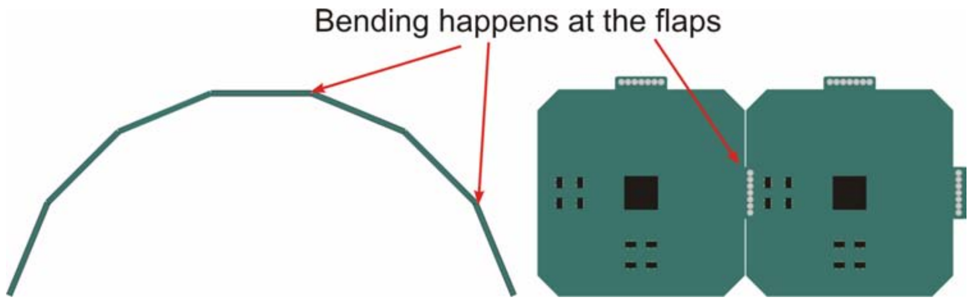


Figure 5-2: Skin Patch bending profile

the strain gages are only minimally stimulated. For this reason, data plots shown for strain gages are not shown for the entire Skin Patch, but rather for individual nodes so that the reader can have an idea of their actual performance.

5.1.3 Pressure Sensors

The third mechanical characteristic that had a deep effect on the sensing behavior of the nodes is related to the pressure sensors. Pressure sensors use a special material that has to be bonded to electrodes placed on the surface of the nodes. However since the electrodes must be in direct touch with the sensing material, no glue can be placed in the active sensor area. Moreover, Skin Nodes had to be very small to maintain a good sensing density, which left very little space that could be used to glue the material to the substrate.

To bond the QTC material to the substrate, a 3M 77 Super Spray Adhesive was used. This glue was applied to the nodes with a mask, so that the sensor electrodes were not coated. However, this glue is semi-liquid when first applied, and because the area available for bonding was so reduced, even if the sensor electrodes were masked, some of the glue was spilled on them making some sensors more sensitive than others and some not sensitive at all.

A way better way of bonding the QTC material to the nodes would have been to use adhesive tape instead of glue, which is easier to correct, cleaner and easier to handle. This lesson was learned later in the assembly process though, and therefore some of the sensors of the assembled skin patch don't work as well as they should. So when these are analyzed, only data plots of the sensors that were not affected will be shown.

Finally, it is important to remark that even though pressure sensors and strain gages were affected by these mechanical flaws, they were actually more sensitive than actually expected, when they were first tested before the entire patch was assembled. Other sensing modalities that did not depend on the mechanical characteristics of the system design were obviously not affected by these problems, so the entire Skin Patch could be easily scanned by the PC sampling application as it was originally intended.

5.2 Sensor Data and Sensing Ranges

Several experiments were carried out to test the functionality of the designed platform. Two patches of skin (one with 12 nodes and another one of 4) and two Brains were fabricated and assembled to test both modes of operation of the Skin Patches: stand-alone mode, and data-extraction mode. This section is divided into two subsections: the first one will show how each one of the sensing modalities performed along with data plots extracted for each one of them using a skin patch as a data extraction device; the second one will demonstrate the functionality of the same patch used in stand-alone mode.

5.2.1 Skin Patches as Data-extraction Devices

To analyze how well each one of the sensing modalities performed, they were first calibrated if necessary and then data was extracted from them. Since each one of the sensing modalities was analyzed separate from each other, results will be presented according to the sensing modality they were obtained from. Although only one data set was extracted for each sensing modality, the extracted information will be shown in two different ways: the first will show some snapshots taken from the real-time 3D OpenGL control from the GUI application when the sampling process started to illustrate its functionality; the second set corresponds to plots created with the data saved by the application at the moment of sampling, along with some images of the stimuli events that generated them. Additionally, there will be a brief evaluation of the performance of the sensors.

Strain and Bending Results

Much was mentioned about strain gages in the hardware section, as they were one of the two sensing modalities that had to be custom-designed. How well did they perform though? To answer this question, we must first calibrate them; this was done by the method of *Shunt Calibration*[15]. Shunt calibration is an indirect way of calibrating strain gages that consists of simulating a strain by changing the resistances of the Wheatstone bridge. This is a popular method because it can be used whenever the required instruments to conduct a direct calibration are not available. This is a tricky exercise though, as the gage factor and total resistances of the gages must be known. Gage factor being an intrinsic property of the used material it is easily obtained from the manufacturer or material property tables; in the other hand, resistances had to be manually measured.

Calibration

Referring back to Chapter 3 when strain gages were mentioned, it was said that it is critical to match the resistances of the strain gages so that the amplifier is not saturated by a resistance mismatch. Table 5.1 shows the actual measured resistances of each one of the two strain gages contained in every one of the 32 fabricated nodes along with the proposed matching and bridge resistors that were used.

The first column is just an index while the second one refers to the serial code imprinted on the node and it was used as a reference. The third and fourth columns indicate the actual measured resistances in Ohms for each one of the two strain gages included in every node. The fifth and sixth columns list the commercially available resistors with the closest resistance values to those listed in the previous two columns. Columns seven and eight show the difference between these two. The next two show the necessary resistor to use in series

	Node	SG1	SG2	R(X)	R(Y)	Δ(X)	Δ(Y)	Rb(X)	Rb(Y)	Δ(Xc)	Δ(Yc)
1	001-1	168.9	173.3	169	174	0.1	0.7	0	0.5	0.1	0.2
2	001-2	161.4	165	162	165	0.6	0	0.5	0	0.1	0
3	001-3	169	173	169	174	0	1	0	1	0	0
4	001-4	178.2	182.2	178	182	-0.2	-0.2	0	0	-0.2	-0.2
5	001-5	175.9	176.5	178	178	2.1	1.5	2	1.5	0.1	0
6	001-6	166.9	167.5	169	169	2.1	1.5	2	1.5	0.1	0
7	001-7	167.2	167	169	169	1.8	2	2	2	-0.2	0
8	001-8	173.5	176.7	174	178	0.5	1.3	0.5	1	0	0.3
9	001-9	171.1	173.9	174	174	2.9	0.1	3	0	-0.1	0.1
10	001-10	165.5	166.1	169	169	3.5	2.9	3.6	3	-0.1	-0.1
11	002-3	184.5	184.7	187	187	2.5	2.3	2.4	2.4	0.1	-0.1
12	002-5	180.8	183.4	182	187	1.2	3.6	1	3.6	0.2	0
13	002-6	183.1	184.7	187	187	3.9	2.3	3.9	2.4	0	-0.1
14	002-10	181.5	182.3	182	182	0.5	-0.3	0.5	0	0	-0.3
15	003-7	173.4	174.3	174	174	0.6	-0.3	0.5	0	0.1	-0.3
16	004-1	172.9	174.7	174	178	1.1	3.3	1	3	0.1	0.3
17	004-2	175.7	179.8	178	180	2.3	0.2	2.4	0	-0.1	0.2
18	004-3	167.8	174.2	169	174	1.2	-0.2	1	0	0.2	-0.2
19	004-6	171	174.9	174	178	3	3.1	3	3	0	0.1
20	004-8	170.3	171.4	174	174	3.7	2.6	3.6	2.4	0.1	0.2
21	004-9	166	171	169	174	3	3	3	3	0	0
22	004-10	173.5	178	174	178	0.5	0	0.5	0	0	0
23	005-1	171.2	177.2	174	178	2.8	0.8	3	1	-0.2	-0.2
24	005-2	170.4	177	174	178	3.6	1	3.6	1	0	0
25	005-3	172.5	177.8	174	178	1.5	0.2	1.5	0	0	0.2
26	005-4	174.6	178.8	178	180	3.4	1.2	3.6	1	-0.2	0.2
27	005-5	178.5	179.8	180	180	1.5	0.2	1.5	0	0	0.2
28	005-6	177	180.3	178	180	1	-0.3	1	0	0	-0.3
29	005-7	175.9	180.9	178	182	2.1	1.1	2	1	0.1	0.1
30	005-8	176.4	180.7	178	182	1.6	1.3	1.5	1	0.1	0.3
31	005-9	176.1	181.4	178	182	1.9	0.6	2	0.5	-0.1	0.1
32	005-10	177.3	183.1	178	187	0.7	3.9	0.5	3.9	0.2	0

Table 5.1: Strain gage measured resistances

with the strain gages to match their resistance values to those listed in columns five and six. The last two columns show the actual measured difference between the resistance of the matched strain gage, and the resistor used in the bridge. It can be seen that the largest difference is of 0.3Ω which is negligible. It is also important to point out that trace lengths for gages one and two of each node were designed to be equal to keep resistance variations to a minimum. Yet as it can be seen, there is still a considerable variation within nodes and from node to node.

With these resistance values it is now possible to calibrate the strain gages. If we refer back to Figure 3-15, the following relationships can be extracted:

$$V_{R2} = V_{CC}\left(1 - \frac{R_1}{R_1 + R2}\right)$$
$$V_{R3} = V_{CC}\left(1 - \frac{R_3}{R_3 + R4}\right)$$
$$V_O = V_{R2} - V_{R3} = V_{CC}\left(\frac{R_3}{R_3 + R4} - \frac{R_1}{R_1 + R2}\right)$$

Or in another form:

$$\frac{V_O}{V_{CC}} = \frac{R_3/R_4}{R_3/R_4 + 1} - \frac{R_1/R_2}{R_1/R_2 + 1} \tag{5.1}$$

Equation 5.1 shows that the output voltage of the bridge is a function of the ratios of the resistances. Moreover, if $R_3/R4 = R_1/R2$ the output is zero which proves why resistances must be matched for a balanced output. Now if we place a resistor R_C in parallel with R_3 which is the added resistance of the strain gage and the matching resistor, the change in resistance of the arm becomes:

$$\Delta R = R_3 - \frac{R_3 R_C}{R_3 + R_C}$$

Or,

$$\frac{\Delta R}{R_3} = \frac{R_3}{R_3 + R_C} \tag{5.2}$$

However, unit resistance change is a function of strain, so if we re-express Equation 5.2 to reflect this, we can then extract a relationship between the simulated strain and the shunting resistance necessary to produce it. Unit resistance change is related to strain by:

$$\frac{\Delta R}{R_G} = F_G \varepsilon \tag{5.3}$$

Where R_G is the nominal resistance of the strain gage (obtained from Table 5.1), F_G is the gage factor, and ε is the material strain. Now, if we combine Equations 5.2 and 5.3, we can obtain:

$$\varepsilon_S = \frac{-R_3}{F_G(R_3 + R_C)} \tag{5.4}$$

Where ε_S is the compressive strain simulated by shunting R_3 with R_C[1]. Now, we already mentioned that the minimum bend radius of a multi-layer flex circuit should be of approximately 24 times the total thickness of the substrate, or in the case of the Skin Nodes $24 \times 0.0162in = 0.3888in = 0.987552cm$. If we take this to be the bending limit of the Skin Nodes, then the strains experienced by the gages will be extremely small. If we consider small strains and we solve for R_C from Equation 5.4 we can now find a resistance value that will produce a known strain which we can then use to calibrate the sensors:

$$R_C = \frac{R_3}{F_G \varepsilon_S} - R_3$$

Since we expect strains to be small, we can arbitrarily pick a relatively small strain value and calculate the resistance needed to produce it. For this case a value of 100^{-6} strain

[1]Although these equations are specific to compressive strain, a similar method can be applied to find tensile strains by shunting R_4 instead of R_3

R_3	R_C
162Ω	809838Ω
165Ω	824835Ω
169Ω	844831Ω
174Ω	869826Ω
178Ω	889822Ω
180Ω	899820Ω
182Ω	909818Ω
187Ω	934813Ω

Table 5.2: Resistances needed to simulate 100 microstrain

units or 100 microstrain was picked. We also know that $F_G = 2.0$ which corresponds to the constantan gage factor[2][16] and that $R_3 = R_{gage} + R_m$. We have all that we need to calculate the resistance values, however since the nominal resistances of the strain gages are not equal, and they were matched to 8 different values (which can be observed in Table 5.1), there are a total of 8 different results which are shown in Table 5.2.

Using these resistance values to shunt the strain gages would emulate a known strain, which can be directly mapped to a bridge output voltage. These values however, are not actually used because it would be difficult to find such commercially-available precision resistors; yet, this exercise serves to have an idea of how large resistor values should be to emulate a microstrain close to what we want. To actually calibrate the sensors, we do the reverse instead: we calculate the strain that corresponds to available resistors, which closely resemble these values, and then use the found strains as the calibration values. Table 5.3 shows the calibration values found with Equation 5.4 for $R_C = 812K\Omega$:

The negative values indicate a compressive strain, but since the gages are essentially located in the bending axis, compressive and tensile stresses are symmetric. This means that these values can be used to adjust tensile strains as well, with only a slight error margin (the constantan film is not exactly at the center). Several assumptions are being made though. First of all, since the bending range will be constrained to the calculated minimum bending radius, there will be no mechanical deformations, and we can assume there will be no

[2]Gage factor changes with temperature, but this will not be taken into account because the temperature range under which the system is expected to work ($0°\sim 80°$C) has only a minimal effect on it

R_3	$Microstrain$
162Ω	$-99.73379695\mu S$
165Ω	$-101.5803439\mu S$
169Ω	$-104.0423853\mu S$
174Ω	$-107.1199029\mu S$
178Ω	$-109.5818897\mu S$
180Ω	$-110.812874\mu S$
182Ω	$-112.0438522\mu S$
187Ω	$-115.1212713\mu S$

Table 5.3: Emulated strain caused by shunting gages with an $812K\Omega$ resistor

histeresis. Second, we are also assuming that constantan is an isotropic material and that the film is so thin that the effects of *Poisson*, and *Shear* strains can be ignored.

The story would end here if all we were interested is the relationship between resistance change and strain however, we also need to find how the bridge output voltage changes with applied strain. To do this, we first record the output voltage when the gages are shunted with a reference resistor value, and then we shunt it again with a different resistor to record the change in output voltage. By doing this we effectively map a difference in simulated strain to a difference in output voltage. However, since we don't know if the relationship between these two is linear, we have to use more than two values to extract the relationship between them. In the actual calibration tests, 7 resistance values were used to calibrate each one of the two gages of a single node; the results are shown in Table 5.4.

The first column shows the shunting resistor used for the calibration, these were the actual values measured with a multimeter and not the one indicated by the resistor color code. The next four columns show the data for strain gages 1 and 2. For each gage, two columns are shown: the first one lists the strain values obtained with Equation 5.3 for each one of the shunt resistors used; the second one lists the voltage seen at the output of the bridge amplifier. Now we can extract the relationship between strain and output voltage by plotting the former as a function of the later using the previous data. Figure 5-3 shows the results obtained, notice that strain sign was ignored because it only indicates the type of strain. The relationship is actually pretty linear, however their curves ended up being displaced from one another, because resistance matching of the bridge branches is not perfect, and

	Node $\#004-3$			
	SG1 (matched to 169Ω)		SG2 (matched to 174Ω)	
R_C	Strain	V_O	Strain	V_O
97000	-869	1.502	-895	1.671
107000	-788	1.492	-811	1.659
118000	-715	1.468	-736	1.646
127000	-664	1.466	-684	1.636
138000	-611	1.457	-629	1.621
148000	-570	1.454	-587	1.611
156000	-541	1.45	-557	1.605

Table 5.4: Calibration results

thus the zero-bias offset is slightly different for both of them.

There is still two more things to take into consideration. The first one is that these results only apply to the calibrated node, to have exact measurements for all nodes each one must be individually calibrated with this procedure; this was not done because it was not a project goal to extract precise measurements. The second one is that we have yet to extract the full sensor range by recording the output voltages experienced when gages are bent to their maximum allowable (set by the minimum bend radius). We do this because we are digitizing the output voltage with the microcontroller's ADC, and we need to figure out how to map the sampled values to an actual strain value. From the lab tests, Strain Gage 1 had a range of approximately $0.765V--1.95V$ with a zero-bias offset of $1.3544V$; Strain Gage 2 had a range of $0.95V--2-135V$ with a zero-bias offset of $1.54V$. This gives a full range for each gage of approximately $1.185V$, half of which is tensile and half compressive. Now, the ADC quantizes the output voltage into 8-bit samples, which means that if the full range of the sensors is of $1.185V$ then, each sampled bit will correspond to approximately $1.185V/256$ or roughly $4mV$. Now finally we can convert this to strain by using either one of the line equations shown on Figure 5-3, if we choose the first one, then each bit will correspond to: $5888.5 \times (4mV) - 7975.5 = 27.25\mu$. Therefore each bit corresponds roughly to 30 microstrain units.

One final note about gage calibration, the actual signal conditioning for the strain gages uses a sleep transistor to maintain power consumption down. This transistor causes a

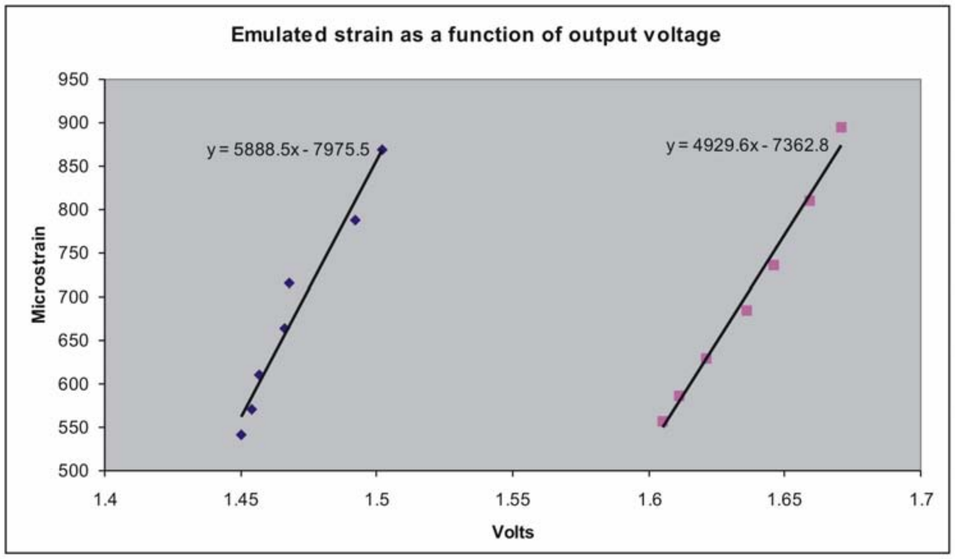


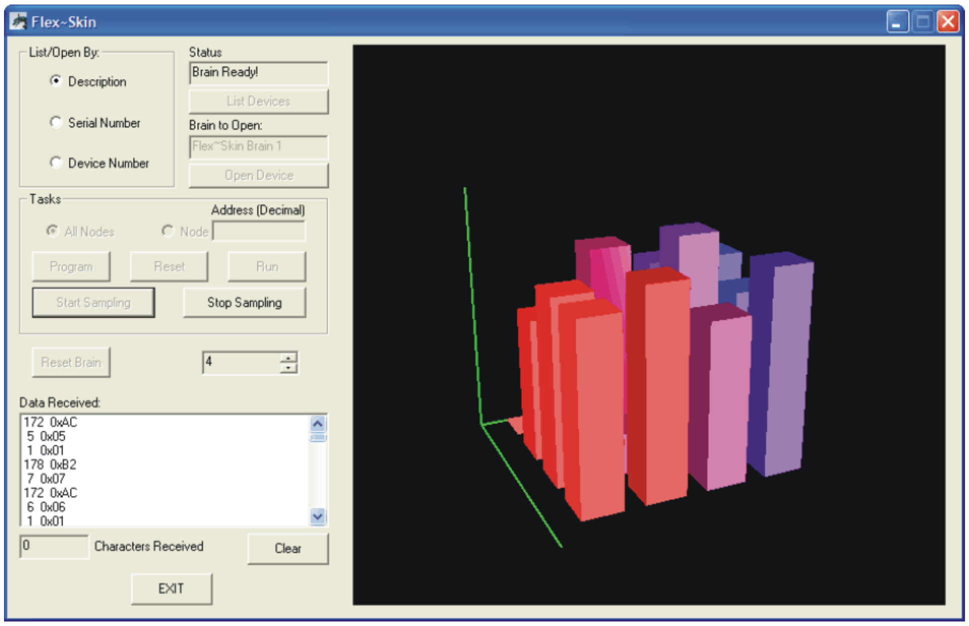
Figure 5-3: Strain Gage electronic calibration for gages 1 and 2 of node #004-3

voltage drop that decreases the total excitation voltage seen at the Wheatstone bridge and this effect was intentionally ignored. Also, if the system is to be used in a wide range of temperatures, then gages must be compensated for its effects; this is easily done though as each node is equipped with a temperature sensor.

Data and Results

Once calibration is done, we can now proceed to show the data extracted from the skin. Figure 5-4 shows some snapshots taken when data from strain gages one and two were being sampled. Although no stimuli could be applied because of the mechanical problems previously mentioned, these are shown merely to demonstrate that raw data can be extracted from both strain gages. Figure 5-4(a) corresponds to data extracted from strain gage 1 of each node and Figure 5-5(c) corresponds to strain gage 2.

Although in the previous figures the strain gages were not submitted to a stimuli, their response can be visualized in Figure 5-5. These plots were extracted from the two strain gages of the node used for the calibration.



Status: Brain Ready!

Brain to Open: Flex~Skin Brain 1

Address (Decimal):

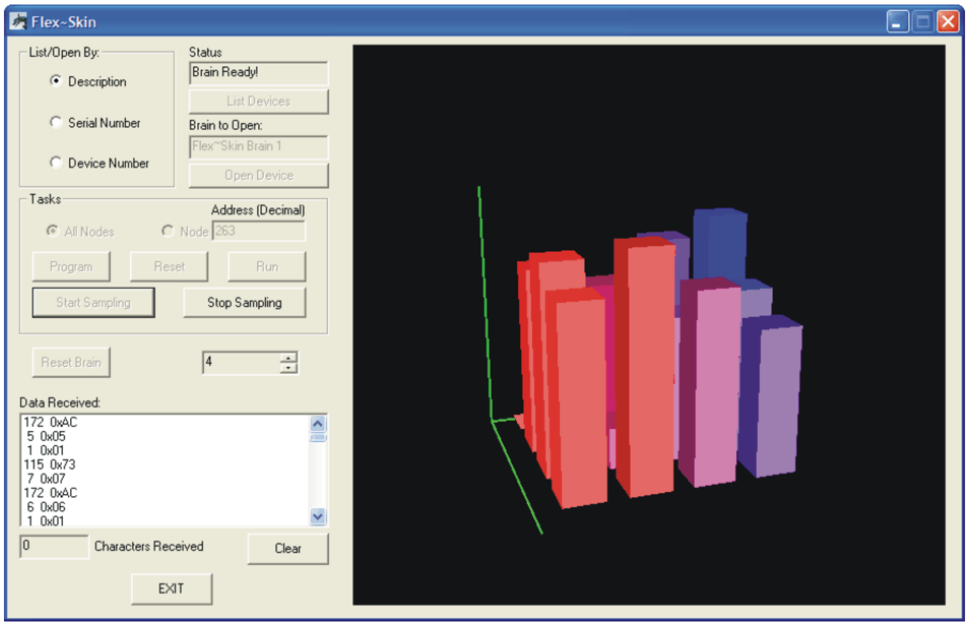
4

Data Received:

```
172  0xAC
 5  0x05
 1  0x01
178  0xB2
 7  0x07
172  0xAC
 6  0x06
 1  0x01
```

0 Characters Received

(a) Data from strain gage 1



Status: Brain Ready!

Brain to Open: Flex~Skin Brain 1

Address (Decimal): 263

4

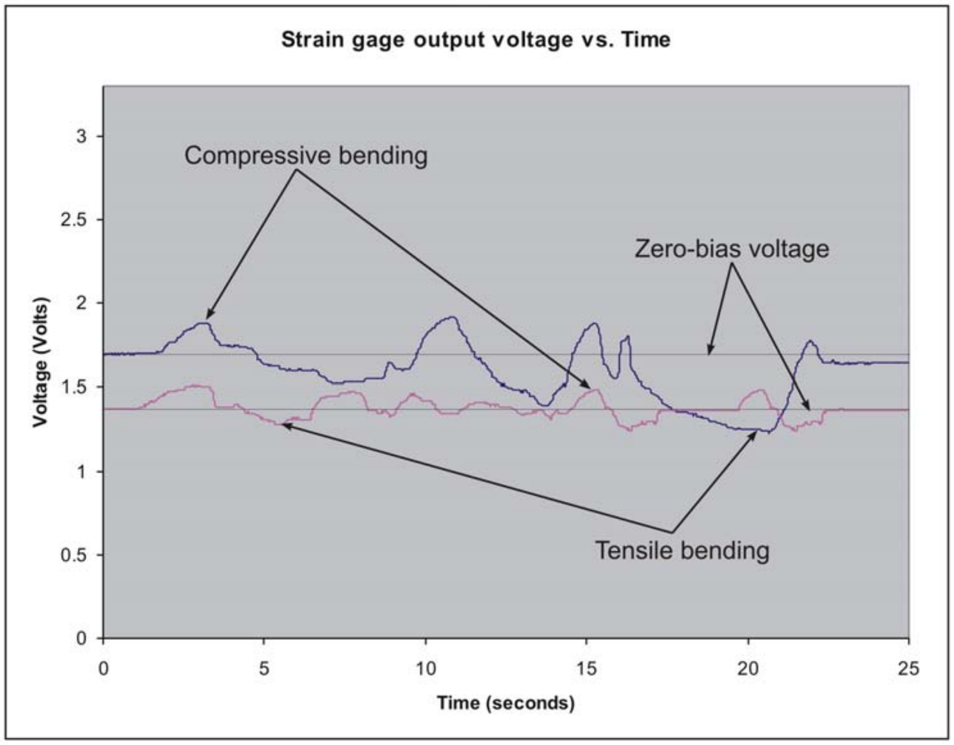
Data Received:

```
172  0xAC
 5  0x05
 1  0x01
115  0x73
 7  0x07
172  0xAC
 6  0x06
 1  0x01
```

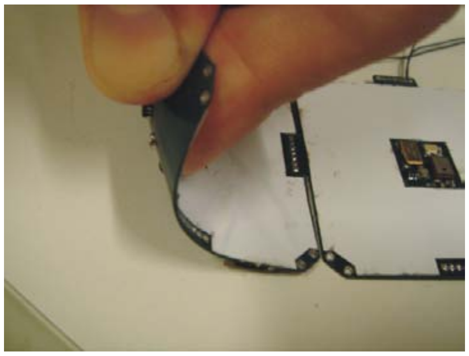
0 Characters Received

(b) Data from strain gage 2

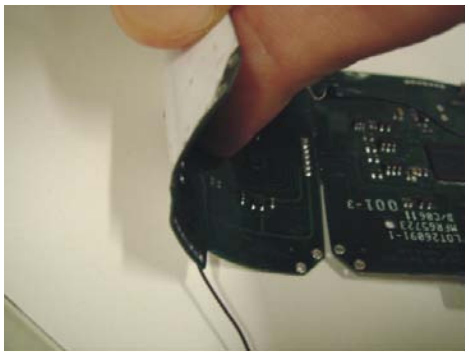
Figure 5-4: Real-time 3D visualization control showing offset from each one of the two strain gages of all nodes (no stimuli applied)



(a) Data from both strain gages



(b) Tensile bending



(c) Compressive bending

Figure 5-5: Strain gage data plots showing strain gage response

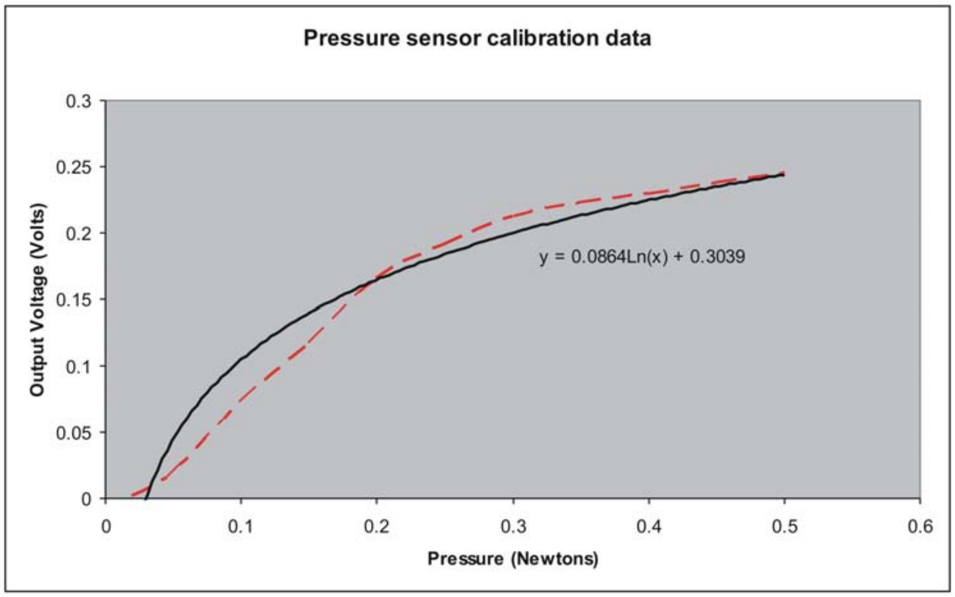


Figure 5-6: Pressure sensor calibration

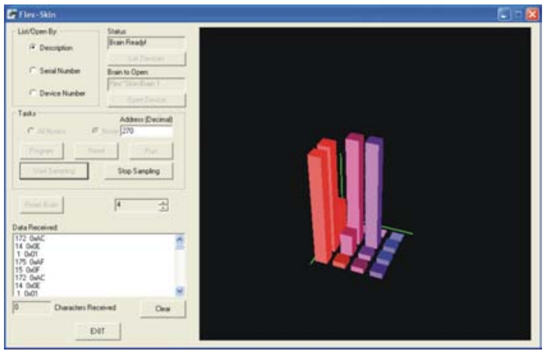
If strain is desired instead of the output voltage, this can be easily calculated using the procedure described before.

Pressure

The second sensing modality that was also custom-designed was pressure. Pressure sensors were designed by using rigid board prototypes, because they could easily be fabricated at the lab. This was done mainly to observe the effect that trace width and length had over the sensitivity of the sensors. However, results showed that these factors had only an effect on the zero-bias resistance of the sensor and not their actual sensitivity.

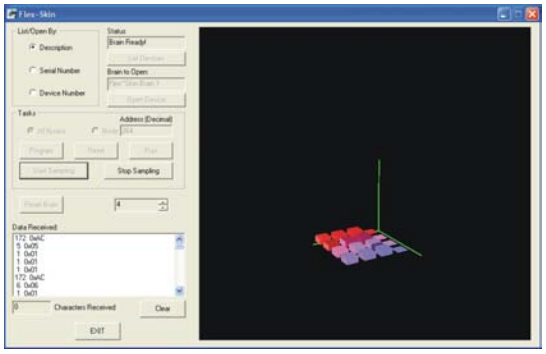
Calibration of pressure sensors was much simpler than that of the strain gages because the instruments needed are easily obtained. In fact all that needed to be done was to apply known loads to the pressure sensors and record the output voltage seen at the output of the amplifier; to do this we only needed a calibrated scale. Figure 5-6 shows the data obtained when one of the pressure sensors was calibrated.

The red (dotted) line corresponds to the data extracted from the calibration measures,



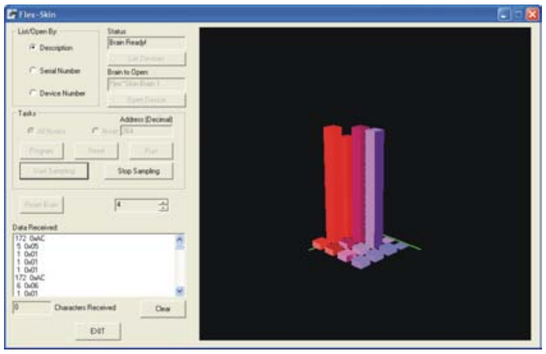
```
172 0xAC
14 0x0E
1 0x01
175 0xAF
15 0x0F
172 0xAC
14 0x0E
1 0x01
```

(a) Pressure data being extracted from pressure channel 1 of all nodes



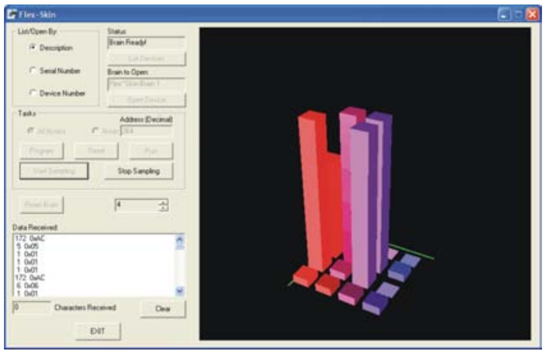
```
172 0xAC
5 0x05
1 0x01
1 0x01
1 0x01
172 0xAC
6 0x06
1 0x01
```

(b) Pressure data being extracted from pressure channel 2 of all nodes



```
172 0xAC
5 0x05
1 0x01
1 0x01
1 0x01
172 0xAC
6 0x06
1 0x01
```

(c) Pressure data being extracted from pressure channel 3 of all nodes



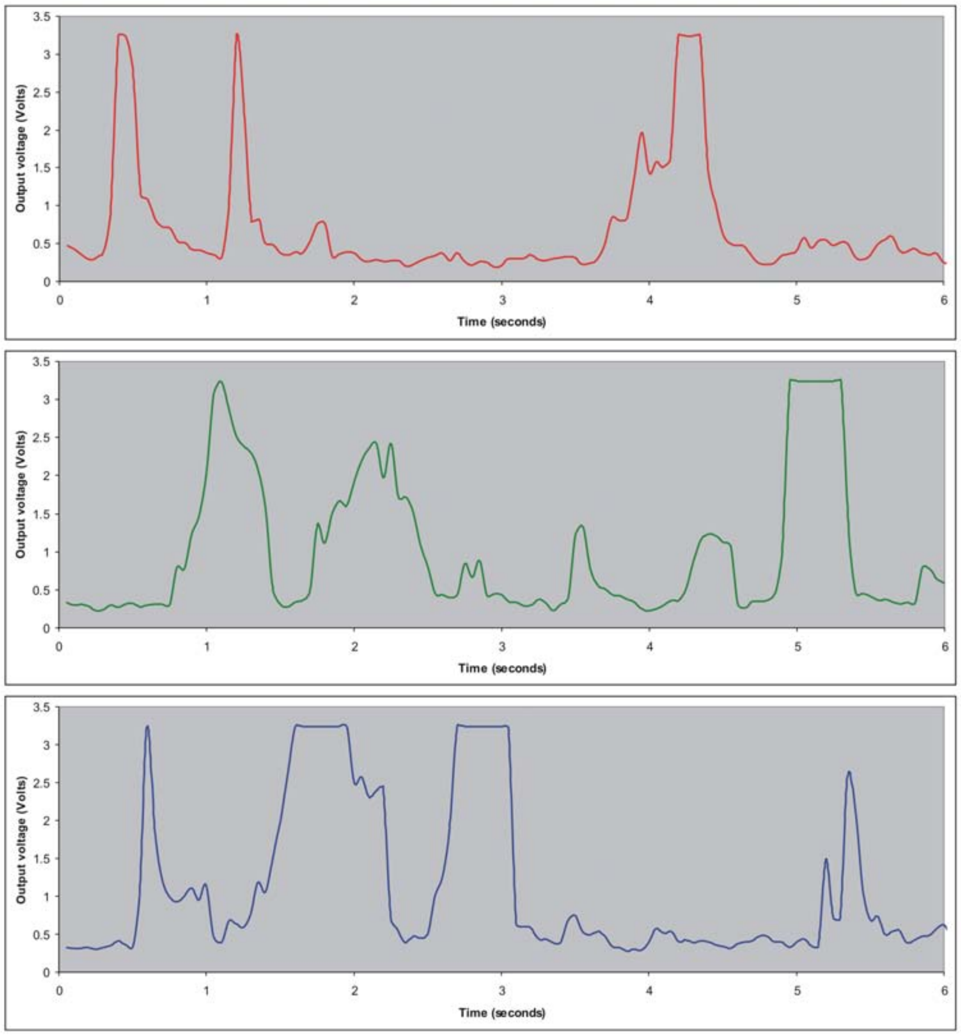
```
172 0xAC
5 0x05
1 0x01
1 0x01
1 0x01
172 0xAC
6 0x06
1 0x01
```

(d) Pressure data being extracted from pressure channel 4 of all nodes

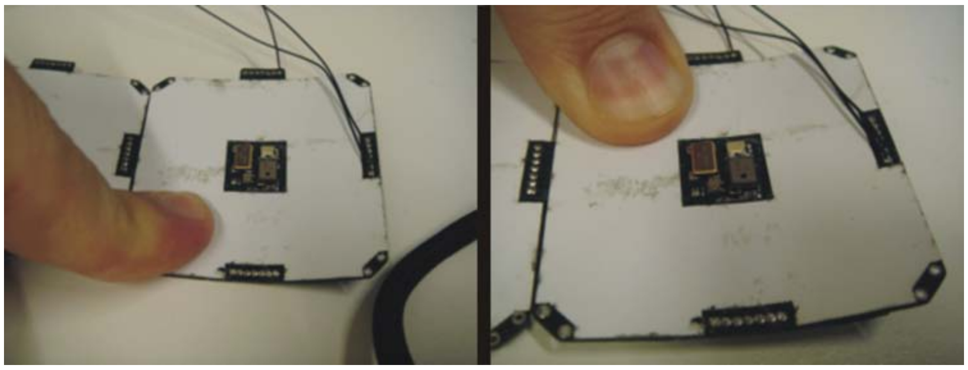
Figure 5-7: Real-time 3D visualization control showing data from each one of the four pressure channels of all nodes

and the black (solid) line is a curve fitted to the data. It is pretty evident that it has a logarithmic response which was actually expected, because as it was previously shown on Figure 3-16 the material resistance has an inverse exponential response to applied pressure. Since the signal conditioning circuit is a non-inverting amplifier, this inverse exponential is inverted and a logarithmic response was obtained. Also from the curve, it can be seen that the sensors are very sensitive, as only a slight touch is needed to generate a distinguishable output. The calibration procedure was done on each one of the four pressure sensors of one node, however this data is not presented because the results were almost identical to those presented in Figure 3-16.

Pressure data was also extracted from the Skin Patches, but as it was previously mentioned,



(a) Data from working pressure sensors and stimuli to generate them



(b) Pressure stimuli

Figure 5-8: Pressure data plots showing pressure sensor response

many of these did not work as expected because of some mechanical problems. Therefore, even though data was extracted from all of the sensors to illustrate how the 3D visualization chart displayed the information, data plots of the pressure sensors are only shown for some of those that actually worked as expected. Again data plots are shown with some pictures that show how the skin was stimulated to generate the data.

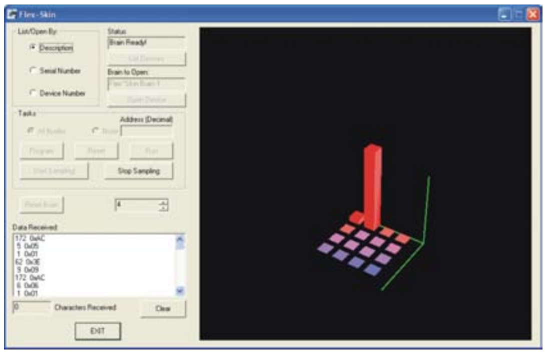
Figure 5-7 shows the real time data, while Figure 5-10(a) shows the data plots of some of the working pressure sensors. Only a few of them are shown though to illustrate their actual response to stimuli. In these plots the output voltage is plotted against time, however if a pressure value is desired it can be calculated using the logarithmic relationship shown above.

One final note on pressure sensing. Although there are multiple “dead” zones insensitive to pressure like the center part of the nodes and the intersection between them, this area only represents less than 10% of the total pressure area. There are many ways to overcome this limitation, some of which will be mentioned in the last chapter.

Ambient Light

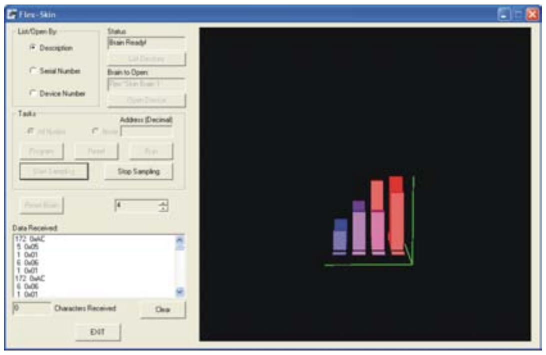
Not much can be said about this sensing modality except that it was probably the best performing one according to the original expectations. The integrated sensor not only is compact and fully integrated, but it has a great sensitivity to all kinds of lightning conditions (incandescent and fluorescent) and it only consumes $620\mu A$. It was also extremely easy to stimulate all of the sensors of the Skin Patch at once.

Several snapshots of the 3D visualization control were taken to show how different lightning conditions affected the skin. These can be observed in Figure 5-9. The first one shows the effect of a laser pointer aimed at the sensor when there was no other light source in the environment. The second one shows how a flashlight shining from the side generates a gradient. The last two show all of the sensors responding at the same time when the amount of ambient light changes.



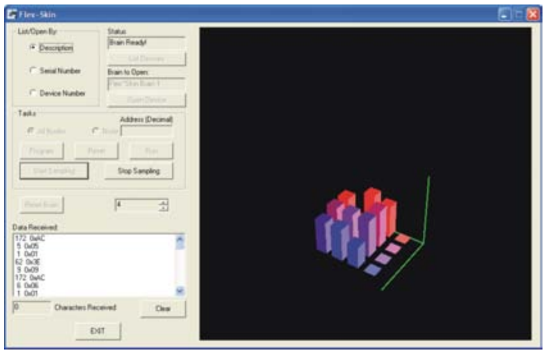
```
172 0xAC
 5 0x05
 1 0x01
62 0x3E
 9 0x09
172 0xAC
 6 0x06
 1 0x01
```

(a) No ambient light present; single node hit with a laser pointer



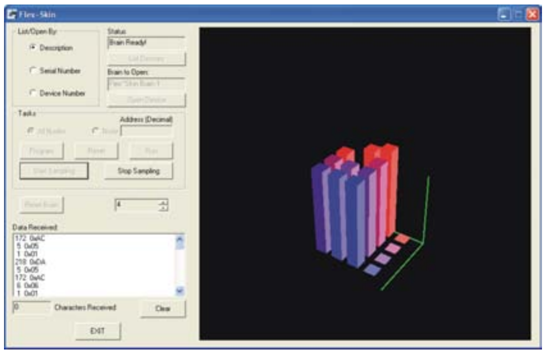
```
172 0xAC
 5 0x05
 1 0x01
 6 0x06
 1 0x01
172 0xAC
 6 0x06
 1 0x01
```

(b) No ambient light present; flashlight illuminating patch from the side



```
172 0xAC
 5 0x05
 1 0x01
62 0x3E
 9 0x09
172 0xAC
 6 0x06
 1 0x01
```

(c) Room with fluorescent ambient light



```
172 0xAC
 5 0x05
 1 0x01
218 0xDA
 5 0x05
172 0xAC
 6 0x06
 1 0x01
```

(d) Room with fluorescent light and flashlight shining on patch

Figure 5-9: Real-time 3D visualization control showing data from different lightning conditions.

Data plots were also extracted for all sensors and are shown with stimuli pictures in Figure 5-10

Finally, light sensors were not calibrated, because in this case the absolute light level was not of interest. Calibration is easily done though, as the manufacturer provides a curve that plots the output current vs. ambient light.

Temperature

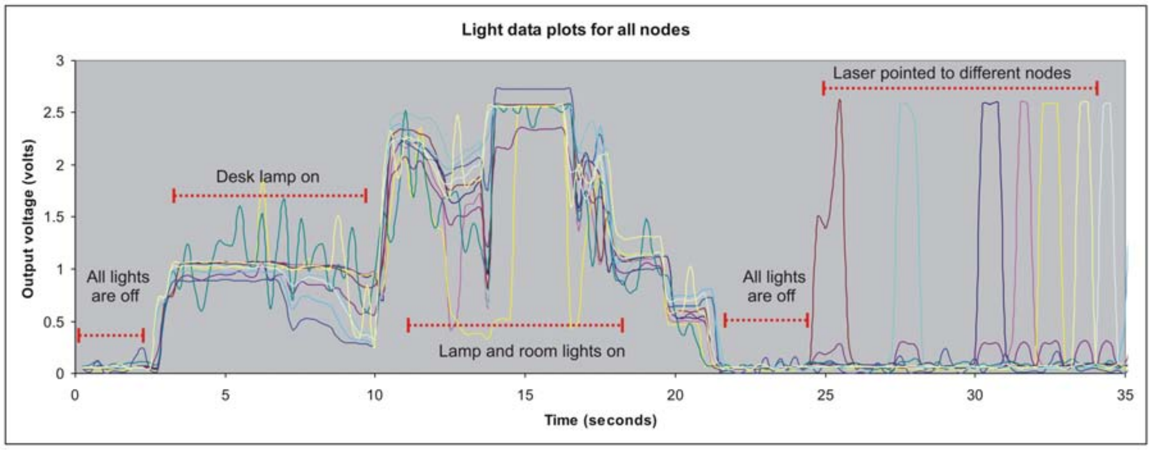
Temperature sensors worked in very much the same way as light sensors in the sense that stimuli could be applied without the need of physical handling of the patch. They also did not need calibration because the sensor datasheet[25] provides enough information to calculate the absolute temperature if this is needed.

The results are shown in a similar way as with the ambient light sensors. Still, since changing the temperature is not as easy as changing the ambient light, especially for just one node, only one general stimulus was applied to the patch to show the response of the sensors. To do this the Skin Patch was warmed with a hot air gun. Figure 5-11 shows the data extracted from all sensors by plotting temperature vs. time. Temperature was calculated by converting the output voltage using the formula provided in the datasheet.

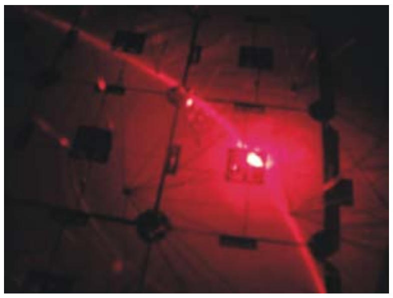
Sound

Sound was the most resource intensive sensing modality. First of all, it must be sampled at least 8Khz, with an 8-bit resolution if human voices or other sounds with a similar frequency range are to be recorded. Considerable processing power is also needed to calculate higher order features like it moving average and variance, and because of this it is also the most power consuming. Yet, also because of this, sound was the best way to push the limits on the system and see how well it performed.

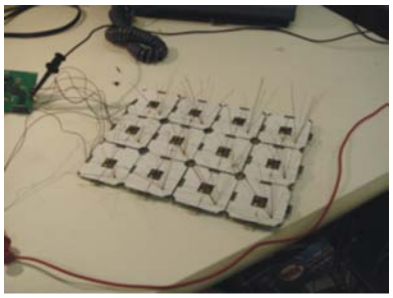
All the sensing modalities up to this point, have been presented in a very similar way: real time data, saved plots and stimuli; this is not done with sound. The reason for this is that



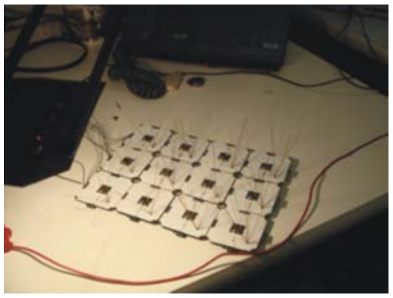
(a) Ambient Light sensor data from all nodes



(b) Laser pointed to a single node

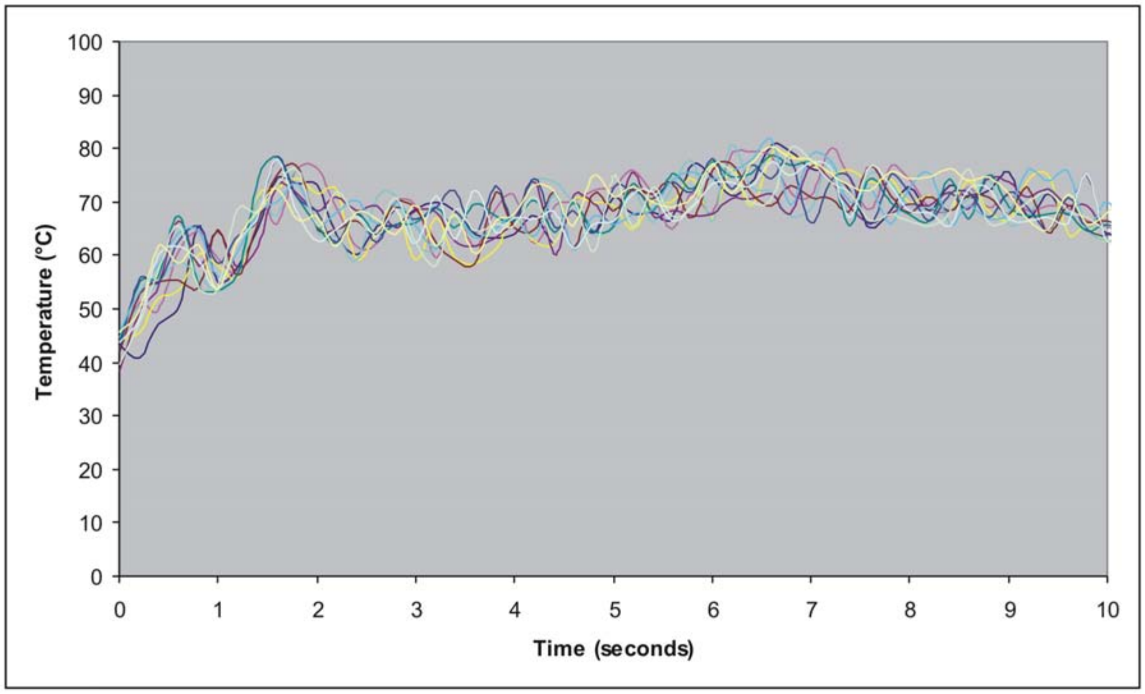


(c) Room lights on

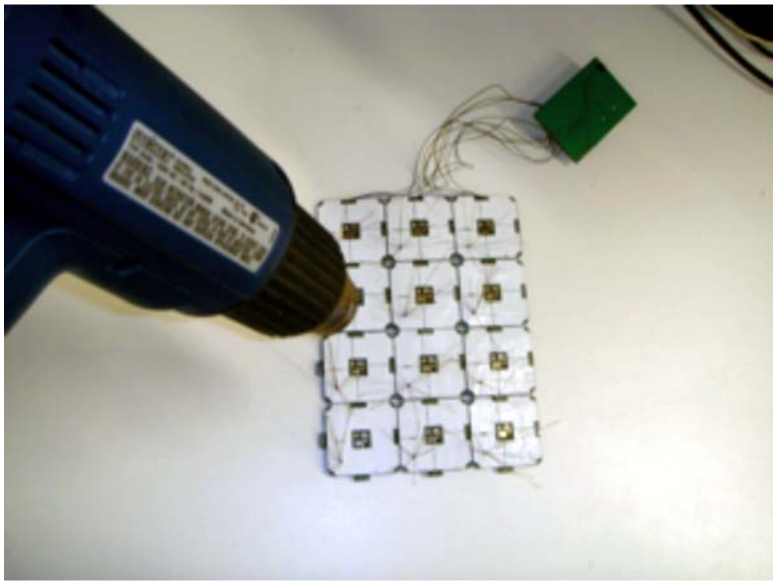


(d) Room lights on and desk lamp on

Figure 5-10: Light data plots showing ambient light sensor responses to different stimuli



(a) Temperature sensor data plots



(b) Hot air gun warming up the patch

Figure 5-11: Temperature sensor response

first of all this is highly undesirable because even a patch of skin with such a low node count (12) already produces a considerable amount of information: if each node is sampled at 8Khz with a 12-bit depth, each node will produce 12Kbits of information per second. That means that if we have 12 nodes, we will have to transfer 144Kbits bits every second, which is already almost half of the available I^2C bandwidth (considered that it is run at 400Kbps).

From a functionality perspective, extracting full sound waves from every node does not make much sense either. Even for applications like sound localization, where sound signals are correlated to extract phase information, three microphones might be enough if they are correctly chosen. So why were microphones included into every node then? Well, first of all if we want to pick three microphones from the whole array, having one in each node gives more flexibility at the time of selecting them. Second, because even if data has to be sampled at high frequencies, it does not mean that it has to be transferred as it can be processed locally. So, as opposed to the raw data obtained from other sensors, when sound data is sampled with the GUI, what each node actually transmits is a feature extracted from the raw data, in this case its moving average.

The moving average of the sound signals is essentially the low-pass filtered version of the original one. This feature was chosen mainly because its simplicity but any other feature like its Timbre, Rhythm, Loudness, Spectrum, and Tonality can be calculated. These features, however, are generally more processing intensive which may cause the processor to be devoted only to this, which is another reason for which calculating them was left as a possible future application. Some sounds with their moving average were extracted from the Skin for illustrative purposes and these are shown in Figure 5-12.

In this case, both raw data and the feature are shown for illustrative purposes, however in a real-world scenario these features can be transferred instead of raw data which reduces the amount of bandwidth needed. Moving averages can also be resource intensive but any other filter can be implemented to further compress the information. A peak-detection algorithm for example could count the peaks in a certain frequency and report this instead.

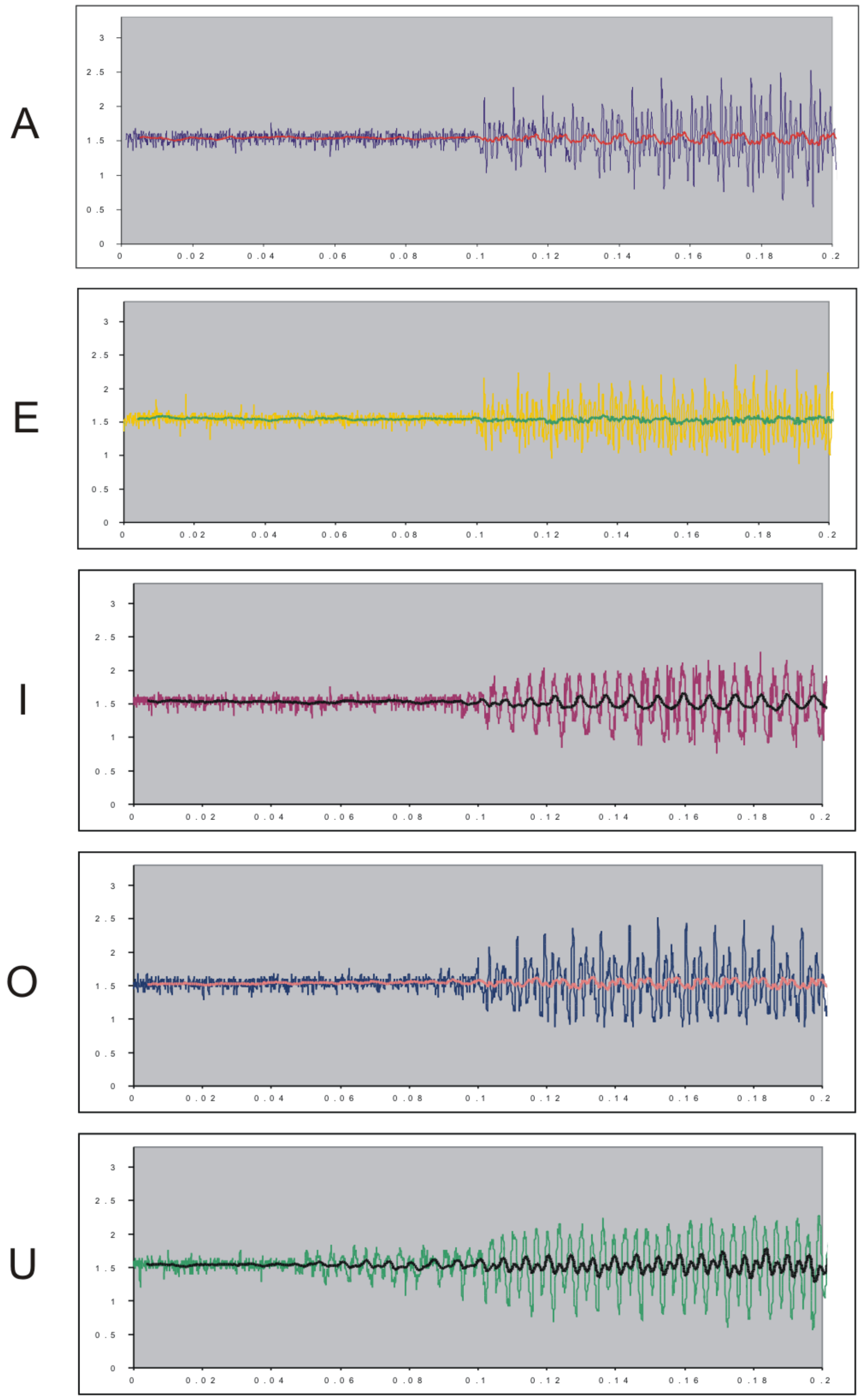


Figure 5-12: Raw microphone data with superimposed moving averages. Window size was 32 samples. Each plot corresponds to a vowel spoken in front of the microphone.

Proximity/Motion Sensing

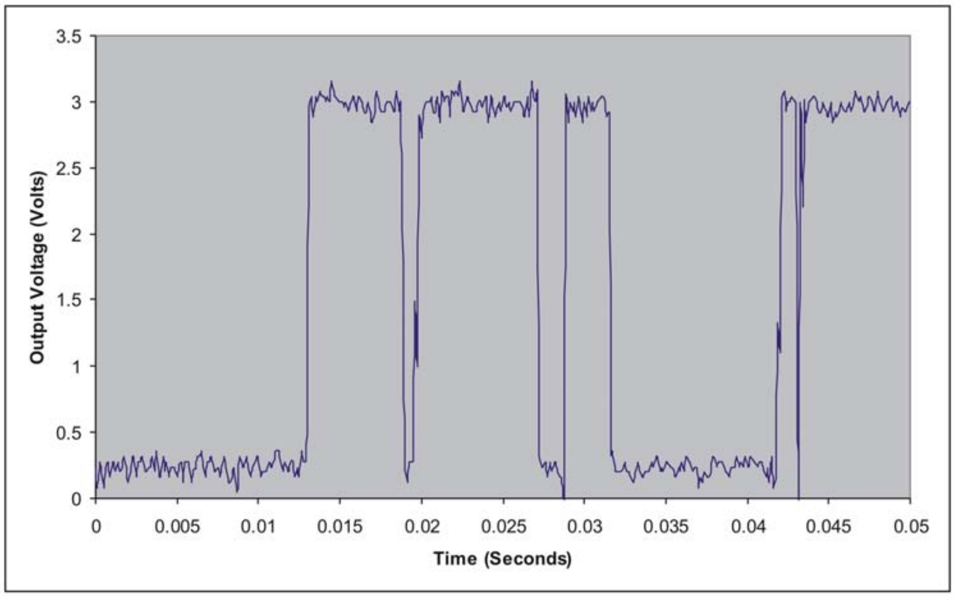
The last sensing modality for which results are shown are the whiskers. Whiskers can be seen as either motion/activity sensors or proximity sensors. Since whiskers are meant to indicate either the presence or absence of activity close to the skin, there is no need to quantize the information received from them as the nature of this information is inherently digital: presence/absence of activity. For this reason the whiskers were connected to interrupt-capable input pins, so that every time they are stimulated, an interrupt is generated, which can be then used to activate more sensors or simply increment a counter.

As it was mentioned in the Hardware chapter, whiskers were implemented with mini-sense piezo cantilevers to which some brush bristles were glued. When these are touched, the cantilever vibrates and it generates a train of impulses that are then detected by the micro-controller. This approach, although effective has two minor drawbacks: first, the cantilevers are mounted on a rigid substrate that hinders the flexibility of the node when this is soldered to it; and second, because the sensor detects the vibrations on the cantilever, there must be available space so that the cantilever can be vibrate freely or it will not work. An alternate method to create the whiskers using a PSAC shock sensor from Kyocera was also considered, but because these sensors were not available at the time the Skin Nodes were designed, they could not be used or tested.

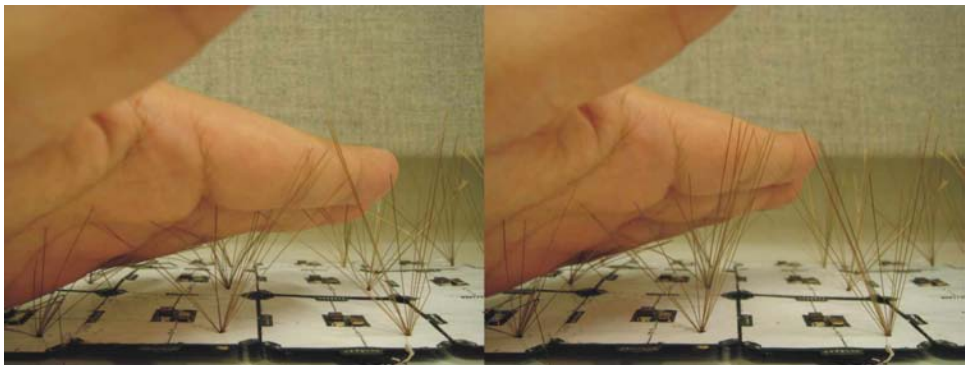
Figure 5-13 shows a plot of data illustrating the behavior of the sensor. Also shown on the figure is a picture that shows how the whiskers are stimulated.

5.2.2 Skin Patches as Stand-alone Devices

It was previously mentioned that Skin Patches can either be used to send data or work as stand-alone devices. The first case is useful if data needs to be further processed or used for another purpose in which case it needs to be extracted from the skin by a personal computer. The second case could be useful for applications where the patch can act like an autonomous device, for example if it is used like an interactive material, where each node is capable of sensing, processing and actuating.



(a) Activity sensor data



(b) Whiskers being stimulated

Figure 5-13: Activity sensor results

To demonstrate the functionality of the Skin Patches as stand-alone devices, a simple application that maps sensor data to the on-board LED was developed. This application samples the ambient light channel and uses the information to modulate the light on the output LED; the less light in the environment, the stronger the LED is lit and viceversa, thus creating an inverse feedback loop that depends on the amount of light in the environment. This application can be used for example, to do shadow tracking which can be seen in Figure 5-14

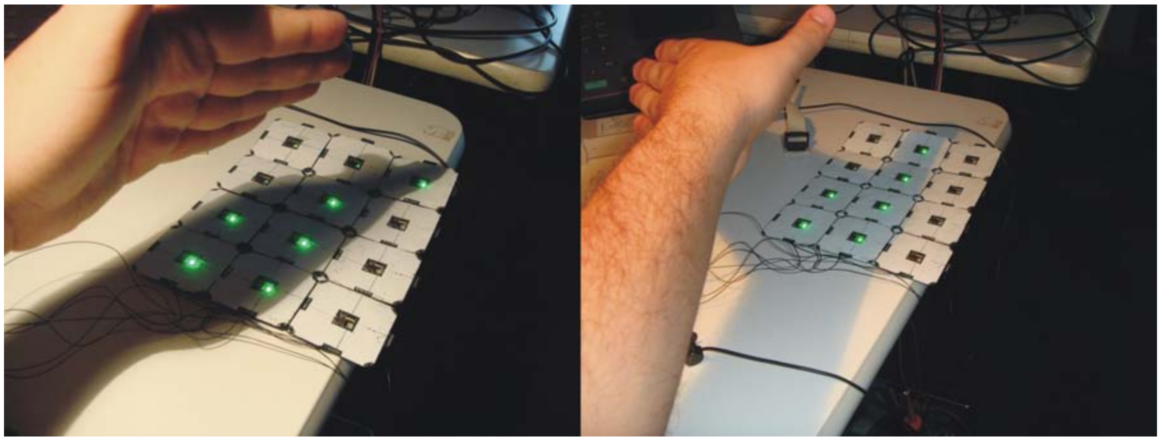


Figure 5-14: Patch used as a stand-alone device. LED's light according to the amount of ambient light

Since the only output device of the nodes is the RGB LED, applications would be probably limited to mapping sensor data to LED lights in different ways. Yet there are many circumstances in which this would be enough, for example if large patches are used to cover up a wall or floor to make it interactive, in fact, Sensor Net Skins covering interactive floors were explored by Richardson in [59]. Then again, the microcontroller has still several unused output pins to which other output devices like a speaker can be connected. This of course would need a redesign of the Skin Nodes so that different output devices can be connected to them, which is easily achieved by adding an extra header.

5.3 Performance Metrics

We have presented results for all the sensing modalities and modes of operation of a Skin Patch. Yet, these results although illustrative, provide little reference as to the overall performance of the project. This subsection will present some metrics that will grant a bit more insight into how well did the system performed against the design goals. In fact, the main goal was to demonstrate that by including processing power in each node, the necessary resources, in special bandwidth could be greatly reduced. The first metric analyzed is then the bandwidth needed to operate a Skin Patch network.

5.3.1 Network Performance

As the reader might recall, scalability is one of the main goals of the project, so ideally, we should be able to connect to a PC as many Skin Patches as possible and every patch of skin should be capable of growing to as many nodes as possible. In reality though these two limits were set by two of the three communications channels used: USB and I^{2}C .

I^{2}C Performance

Peer to peer is never a concern because only two elements are involved, however I^{2}C and USB are shared buses that impose a limit on the amount of devices that can be connected at the same time. I^{2}C sets a limit on the number of nodes per Skin Patch and USB sets it for the number of patches that can be connected to a computer at the same time. The reason for this is mainly bandwidth and network speed.

In the case of I^{2}C , there are two factors that limit the amount of devices that can be connected to it: bus capacitance and network addresses. As we previously mentioned, I^{2}C 10-bit addressing was chosen, which in theory limits the amount of nodes to 1023 ($2^{10} - 1$). However, the I^{2}C bus specification restricts the rise and fall times of both signals to 300ns, and since this time is set by the RC constant of the circuit, the real limiting factors are the pull-up resistance value and the amount of capacitance in the bus.

Because the amount of capacitance in the bus depends on the amount of nodes connected to it, we have to set the resistor value as low as possible. An arbitrarily low value can't be chosen though, because the bus line current would increase and the MSP430 can only handle a maximum of 2mA. With a supply voltage of 3.3 volts, the minimum resistance that can be used is 1350Ωs. However in fast mode, this resistance limits the number of capacitance in the bus to about 300pF[55] instead of the maximum possible of 400pF. This decreases the amount of nodes that can be on the network and is why an active pull-up was used instead. Figure 5-15 shows how an active pull-up device eliminates the RC response of both bus lines (SDA and SCL) giving a speed increase.

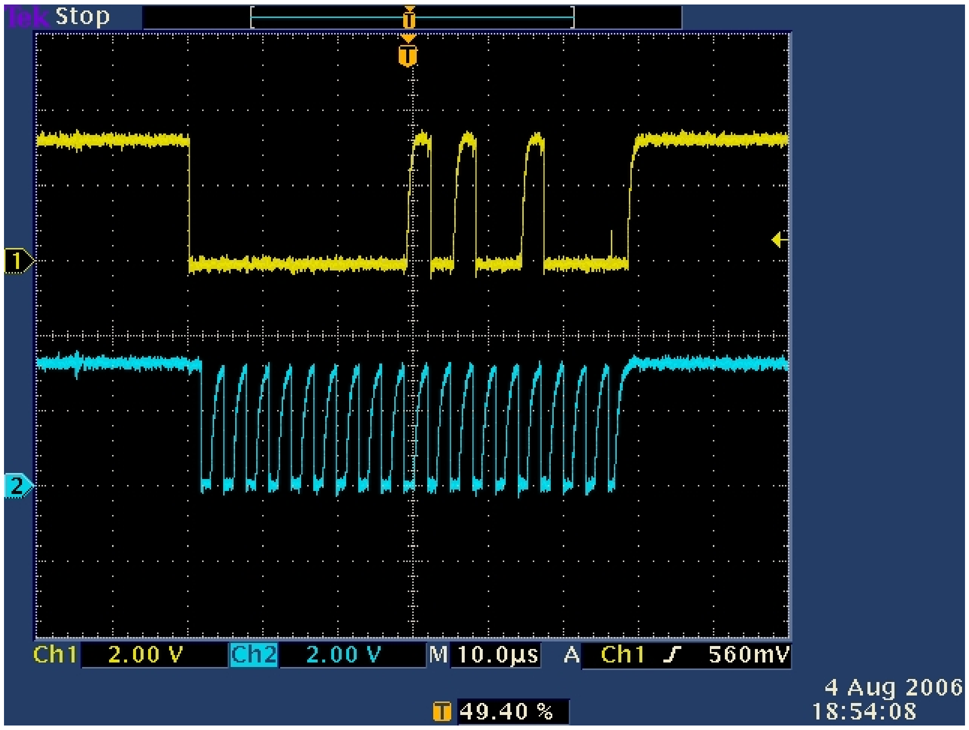
By using the active pull-ups, we can now use the 400pF allowed by the bus specification. The capacitance of the SDA and SCL lines was measured at 9.8pf and 9.6pF respectively with respect to ground, so the maximum number of nodes the bus can have is approximately 40. This is a very small number, yet no special drivers or a ground plane were used. If the driver mentioned in the hardware section is used, the 400pF limit increases to 4000pF. Furthermore if ground planes are used, the capacitance of the traces can be dramatically reduced. With these to improvements the number of nodes per patch can be easily brought to over 500. Once again, the reason why these features were not included were to reduce the amount of components and to keep the number of layers to a minimum. If specialized Skin Patches[3] are designed though, it may make sense to include them.

USB Performance

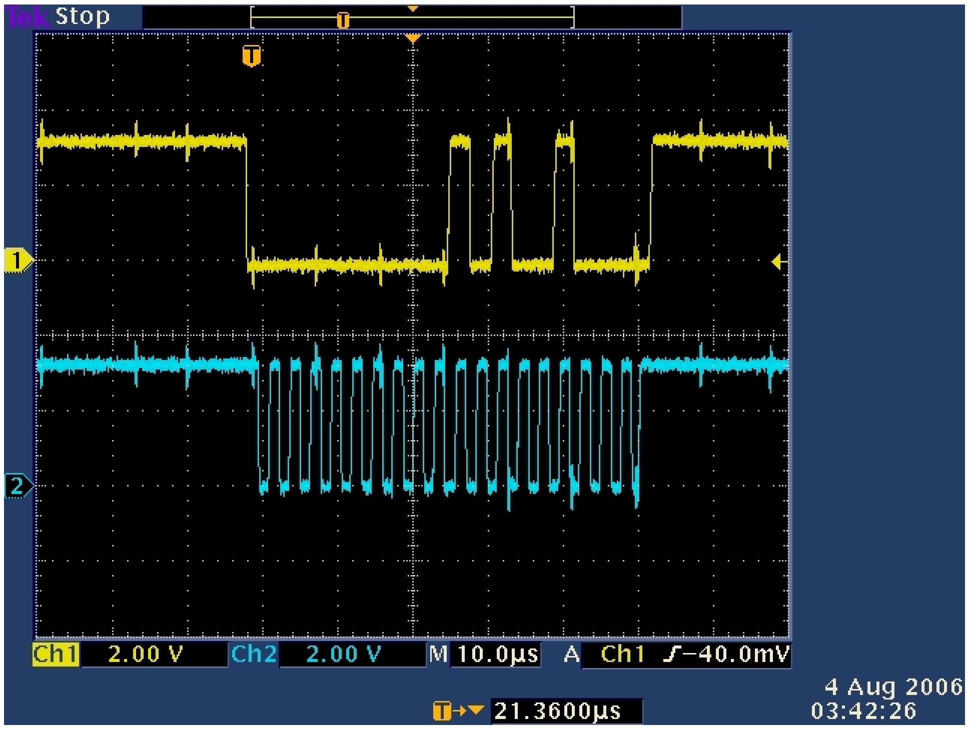
The second limiting factor for network size is the USB bandwidth. USB bandwidth limits the network size because this must be divided among the number of Skin Patches currently connected. This only makes sense if the Skin Patches are connected as data extraction devices, but it does illustrate why it was so important to make every node consume as little bandwidth as possible.

Considering that every node has 11 sensing channels, each one of which will be sampled at the rate and bit depth described in Table 4.3, each node will produce a total of 99800bps

[3]A specialized Skin Patch may sense only a subset of all sensing modalities included in this project



(a) I^2C data when a 1500Ωs resistor was used as pull-up



(b) I^2C when a current mirror was used as pull-up device

Figure 5-15: Difference in speed for both I^2C lines when active and passive pull-up devices are used.

or 12475 bytes every second. The USB chip used has a data rate of 8Mbps or 1MB/s so a total of 84 nodes can be connected in different patches. Once again this number may seem extremely low, but it is only because we are extracting raw data at the full sampling rates. If higher order features are calculated and these are transferred instead of raw data, and if a USB 2.0 capable chip is used instead (480Mbps), this number can grow considerably.

To illustrate this point, lets assume that each node is programmed so that it can change the sampling rates dynamically[4]. Lets also assume that each sensing modality is sampled at 10Hz/8-bits, in a low-bandwidth/ultra-low-power mode. 11 sensing channels will produce a total of 880bps. If USB 2.0 is used and assuming that it is 70% efficient, we would have a total of $480Mbps \times 0.7 = 336Mbps$. With this bandwidth, it would be possible to have more than 380,000 nodes. This is why having processing power in each node is so important: it allows the network to adapt to different conditions, so power consumption and bandwidth requirements are minimized. All of this is in addition to any further gains of calculating higher order features so these are transmitted instead of raw data.

5.3.2 Power Consumption

Power consumption is the second metric used to evaluate the performance of the skin. Although the importance of a low power consumption might not be apparent at first, this becomes evident if Skin Patches with large numbers of nodes are created. Table 5.5 lists the quiescent current consumption of each component of the nodes:

With the data from the table the approximate power consumption can be now calculated. The first component, the microcontroller has a power consumption that depends on the speed at which is run. Since data processing and transmission require full speed, the MCU is set to run at 8.000Mhz, for a total consumption of about 2.1mA. This however can be made as low as 300nA if deep sleep is used. In the case of the RGB LED's, a current-limiting resistor was added to each color to decrease power consumption, so if they are lit at full power, they consume a maximum of 2mA.

[4]Either by adapting to stimuli, or by receiving an instruction from a PC or Brain

Component	# per node	Current consumption
MSP430F1610	1	$300\mu A/MHz$
NCP553SQ33T1	1	$2.8\mu A$
LTST-C17FB1WT	1	2mA/color if lit
MAX4462	2	700μA
MAX4400	4	320μA
MAX4734	1	$4nA$
LM20CIM7	1	10μA
TPS851/52	1	62μA (depending on lighting conditions)
SPM0102NE3	1	$175mA$ (average)
FDG6301N	1	100nA

Table 5.5: Power consumption of Skin Node components

Amplifiers were chosen with a bit more bandwidth than actually needed, so they consume more than could be actually be achieved. If the OPA349 from Texas Instruments was used instead the power consumption per amplifier could be brought down to 70μA per amplifier. Next, The ambient light sensor power consumption depends on the amount of ambient light present, but in regular indoor conditions under artificial lightning it consumes 62μA.

And finally, since the MOSFET is used to gate the Wheatstone bridges used to excite the strain gages, only its leakage current is listed as power consumption. However if the MOSFET is turned on, the bridges will become active and they will consume a current depending on the resistance of their branches. So for example, if a node has a bridge with branches matched to 169Ωs and 174Ωs these will consume:

$$
\begin{aligned}
I_{bridges} &= I_{B1} + I_{B2} \\
I_{bridges} &= \frac{3.3V - V_T}{169\Omega} + \frac{3.3V - V_T}{174\Omega} \\
I_{bridges} &= \frac{3.3V - 0.7V}{169\Omega} + \frac{3.3V - 0.7V}{174\Omega} \\
I_{bridges} &= 0.01538A + 0.014942A = 30.32mA
\end{aligned}
$$

So in average, the strain gages consume a total of 30mA when turned on, which is why it was so important to gate them using the MOSFET. Total power consumption can be then as low or as high as:

$$\begin{aligned} Low \quad & 300nA + 2.8\mu A + 2(700\mu A) + 4(320\mu A) + 4nA + 10\mu \\ & A + 62\mu A + 175\mu A + 100nA = 2.927404mA \\ High \quad & 2.4mA + 2.8\mu A + 2(700\mu A) + 4(320\mu A) + 4nA + 10\mu \\ & A + 620\mu A + 175\mu A + 3mA + 6mA = 14.887904mA \end{aligned}$$

In actual tests, maximum power consumption of one node with the MCU running at full speed, all LEDs on at full power, both strain gages on and a laser aimed at the light sensor, was of 41mA. When everything was turned off however it consumed less than 2mA. If we now combine this with the possible number of nodes in a Skin Patch, we can see that a Skin Patch with 40 nodes at full power would consume 1.64A of current, but this case would be rare. On average each node consumed a total power of less than 9mA in regular operation, so a total of 40 nodes would consume roughly 360mA.

Still, it is important to note that even though 9mA was the average current consumed, this is highly dependant on the application being run by the nodes since most of this will be dissipated by the LEDs. Their total current consumption can be easily brought down though, since they are driven by timer output pins, which means that they can be pulse-width modulated.

Another thing to note is that power dissipated by components does not have any measurable effect in temperature sensor readings. This is mainly because the amplifiers and RGB LED, which are the most power-hungry elements, are placed so far from it that most of their heat dissipates to the surrounding air rather than to the substrate. Moreover, Kapton is not a good heat conductor so it would be extremely rare to see any effects in the temperature readings caused by heat dissipated from nearby components.

Finally, since power is readily available for Brains, and their power consumption is negligible compared to the total power that a large Skin Patch could in theory consume, this was not an issue and was therefore not calculated.

5.3.3 Cost

The final metric used is the overall cost of the project. Cost was a big consideration in the overall design of the project because flex boards are expensive to fabricate and even more expensive to prototype, specially if multilayer boards and other fancy techniques are used. Without a doubt, the major cost component of the project was the fabrication of the node substrates which accounted for roughly two thirds of the total cost. 32 nodes were fabricated at a total cost of \$3,500, so each node cost \$110.

Component prices for nodes and Brains can be observed in the Bill of Materials included in Appendix B. Two prices are listed in this table: the price used for the project and a bulk price for each component. This was done with the intention of providing a comparison between the cost of a prototype project like this one and its possible cost with large-scale manufacturing. It can be seen that the components for each node add-up for a total of \$54 per node or about a third of the total cost. If these components are bought in large quantities though, the cost decreases to about half as much. Even substrates can be manufactured for about a tenth of the cost if they are fabricated in large numbers.

Brains are much cheaper because they use standard fabrication techniques and relatively cheap components. Also the price can be seen in the Bill of Materials.

Chapter 6

Conclusions and Future Work

“So long and thanks for all the fish.”

– Doplhins to humans (Douglas Adams)

We arrive to the last chapter, which presents the conclusions derived from the results presented before. First a summary of the work along with some general conclusions are presented. This is followed by a description of possible applications and future work.

6.1 Conclusions

A Dynamically Adaptable Artificial Sensate Skin was designed. Based on the design goals proposed, 32 nodes were fabricated with Flex circuit technology. Half of these were assembled into two patches of skin of different sizes to test the two modes in which they can operate. Two Brains were also designed and developed so that data could be extracted from one of the patches. Finally several software components were developed for the system. These include a GUI application to serve as a front end to the Skin Patch network and several firmware applications to test the functionality of the skin as a stand-alone device.

Generally speaking, several goals of the project were met. First, the goal of creating a device that emulates some of the functions of the human skin was successful, and it is represented by the two Skin Patches created to test the project. Second, it was demonstrated that having processing power on each node has several advantages:

1. Application code running in the Skin Patches can be changed at any time, either for a single node or an entire patch
2. Each node can adjust the sampling rate of its nodes depending on different criteria
3. Skin Patches can be operated at different power modes

Yet, even though the major goals of the project were met, there are several design choices that greatly reduced the performance of the project as a whole. First, the mechanical problems mentioned before had a great impact on the reliability of I^2C communications. Second, because of the time frame of the project, it was not possible to develop enough application code to thoroughly test all the functionality available; this includes the Peer-to-peer communications channel, which in spite of being present, it is not exploited.

Yet, there are several measures that can be taken to correct most of the problems encountered throughout the design and implementation of this project. For example, the fact that the microcontroller could easily break off when the nodes were bent was caused because of its size and packaging; if a much smaller one with a leaded package[1] is chosen instead, this problem can be easily eliminated. Using a smaller microcontroller would also means sacrificing functionality, which leads to the next point.

The vast majority of the problems found were due to the fact that the number of components in each node was significant. This can be avoided if specialized nodes are created instead. For example, because the MEMS microphone, ambient light sensor and temperature sensors had to be on the outer-facing layer of the skin, the area where these were placed was completely insensitive to pressure. If a pressure-only skin is designed though, the entire top

[1]The leads in the package could absorb some of the stress caused by the node bending, and would therefore help to avoid any ruptures

surface can be used to sense pressure, and since functionality requirements are reduced, a smaller microcontroller can be then chosen.

Another one of the mechanical-caused problems was interconnect conductor cracking. Since the bending of the Skin Patches was mostly restrained to the flaps as previously mentioned, conductors cracked very easily. This can be corrected by making the flaps almost as large as the node itself.

In spite of all these problems though, all components of the project were functional as originally expected, and it was possible to demonstrate both modes of operation, (stand-alone and data extraction) successfully.

6.2 Possible Applications

So far we have extensively talked about the What's and How's but not the Why's. The last part of the chapter, and of this entire document, briefly outlines some of the possible applications of the project.

Given the Sensor Network nature of the Skin Patches, and given that they are equipped with a wide variety of sensing modalities, there are several applications that can be run on the platform. These are presented in the following list:

- Sound Localization: to do this, three or more "sound nodes" could be designated by the Brain. Once this is done, the Brain can then send a broadcast message indicating them to start sampling. After a certain period, samples are recovered by the Brain, and correlations are calculated. From these the direction of the source of the sound can be extracted. Similarly large patches of the skin can be used as an audio beam-forming array, at least for short snippets of skin

- Shape recognition: Basically each node records the amount of light it is receiving and it decides to report a status of either "shadowed" or "lit", so when the Brain scans for the status of all nodes, it can determine the shape of the shadowing object.

- Code Distribution: a very simple means of code distribution was demonstrated in this project, however it can be as complex as needed. Some nodes might be designated as leaders, so that code has to be transferred only to them by the Brain. Once a low activity condition is detected, these would proceed to distribute the code recursively to their neighbors.
- Proximity from shadows: this would be trivial to implement. The average amount of ambient light can be calculated for an entire patch to serve as a reference point. Once this is done, proximity can be calculated from the amount of light blocked from either one or many sensors.
- Pressure Gradients: gathering data from neighbors, nodes can calculate the center of mass of an object placed on top of the skin.
- Motion Sensing: using the whiskers the skin can be used to detect the amount of wind.
- Artistic displays: Skin Patches can be used to wrap walls, floors or ceilings and make them interactive. A passer by could cast a shadow for example, and LEDs would light up, displaying its silhouette.

6.3 Future Work

There are several things that can be done to improve the functionality of the project. Several of these have already been mentioned before, however they are summarized in the following list. A brief description is also provided.

- Larger Network: a larger network would allow more tests and applications to be tested.
- Improve mechanical design: several mechanical characteristics of the design can be improved.

- Implement P2P Communications: currently no applications that use the Peer-to-peer communications channel have been implemented

- Implement an Operating System: a bootloader was implemented for this project, however if more complex tasks are needed, it may be a good idea to add hardware abstraction by implementing an OS.

- Increase Bootloader code size: the current bootloader can only handle a maximum code size of 4KB. This can be changed by modifying the bootloader code so that it reads multiple 4KB code blocks, instead of only one.

- Improve Brain by changing FTDI chip to newer version: there is now a new version of the FTDI chip that does not need any of the external components currently used by the FT245BM. Switching to this version would save on cost and Brain size.

- Create different kinds of skins with different capabilities: as was mentioned before, a better way to implement the Skin Patches would be to create specialized nodes instead of monolithic ones.

- Incorporate QTC Material into substrate or encase in silicon: to avoid the warping and bubbling of the QTC material when the nodes are bent, and to distribute the pressure evenly among the pressure channels, the entire skin could be covered in a silicon rubber. Another option would be to incorporate it into the material stack; this option would be possible if the nodes are specialized, since it would mean fewer components, and therefore less routing needed, which would leave space for an extra layer for the material.

- Increase current capacity of Brain to allow for a larger network: currently the Brain provides the current to the entire Skin Patch connected to it. If a larger Skin Patch is needed, the current capacity of the traces in the Brain responsible for this have to be enlarged.

- Network discovery protocol: this was already mentioned before in the software section. This would be necessary for the Skin to be dynamically reconfigurable so that whenever the Skin Patch topology is changed, this can be detected by the Brain or PC.

- Multiple masters/slaves: currently the I^2C bus is arbitrated by the Brains. No node ever uses the bus unless it is addressed by a Brain. This approach, though, can't be used if a node wants to start communications, so that it can for example report unsolicited data to the Brain. It also means that nodes can't use the line to address any arbitrary neighbor on the network.

It is with this list that we reach the end of the current work, but not before leaving a final note. Even though the goals of the project were broadly met, it was still far from emulating most of the characteristics of the human skin. The technology necessary to create an artificial device that has all the functional characteristics, electrical and mechanical, of our skin is still some years ahead. Much work has to be done yet to research the necessary materials and methods so that we can one day have a device as versatile and functional as our skin.

Appendix A

Abbreviations and Symbols

ADC	Analog to digital converter.
API	Application Programming Interface.
DC	Direct Current.
DCO	Digitally Controlled Oscillator.
DMA	Direct Memory Access.
FIFO	First-In First-Out.
FPC	Flexible Printed Circuits.
FSR	Force Sensitive Resistor.
GUI	Graphic User Interface
I2C	Inter-Integrated Circuit Bus.
I/O	Input/Output.
IDE	Integrated Development Environment.
IPC	Institute of Printed Circuits, now IPC.
JTAG	Joint Test Action Group.
KB	Kilobyte.
LED	Light-Emitting Diode.
MCU	Microcontroller Unit.
MEMS	Micro-electromechanical systems.

MIPS	Million Instructions Per Second.
PCB	Printed circuit board.
PDMS	Polydimethylsiloxane
PNP	A type of semiconductor (P-type, N-type, P-type).
PVDF	Polyvinylidene fluoride.
QFN	Quad flat package no leads.
QTC	Quantum Tunneling Composites.
RAM	Random Access Memory.
RISC	Rapid Instruction Set Computer.
RF	Radio frequency.
RGB	Red, green and blue.
RTD	Resistive Temperature Detector.
SMD	Surface Mount Device.
SNR	Signal to noise ratio.
SPI	Serial Peripheral Interface.
USART	Universal Synchronous-Asynchronous Receive Transmit.
USB	Universal Serial Bus.

Appendix B

Bill of Materials

Node components

Type	Name	Part Number	Description	Manufacturer	Package	# per node	Unit Price	Total	Bulk Price	Total Bulk
Mechanical	FPC	FH12-10S-0.5SH	CONN FPC/FFC 10POS .5MM HORZ SMD	Hirose Electronic	SMD	1	$1.7900	$1.38	$0.73416 / 1000	0.73416
Capacitor	Crystal Cap.	MCH155A160JK	CAP CERAMIC 16PF 5% 50V C0G 0402	Rohm	0402	2	$0.0450	$0.09	$0.007 / 10000	0.014
Capacitor	VCC Decoupling	ECJ-1VB0G106M	CAP CERAMIC 10UF 4V X5R 0603	Panasonic ECG	0603	1	$0.3250	$0.33	$0.124 / 4000	0.124
Capacitor	VCC Decoupling	ECJ-0EF1C104Z	CAP .1UF 16V CERAMIC Y5V 0402	Panasonic ECG	0603	1	$0.0190	$0.02	$0.006 / 10000	0.006
Capacitor	VREF Storage	ECJ-1VB0G106M	CAP CERAMIC 10UF 4V X5R 0603	Panasonic ECG	0603	1	$0.3250	$0.33	$0.124 / 4000	0.124
Capacitor	VREF Storage	ECJ-0EF1C104Z	CAP .1UF 16V CERAMIC Y5V 0402	Panasonic ECG	0603	1	$0.0190	$0.02	$0.006 / 10000	0.006
Capacitor	Vreg Input	ECJ-0EB0J105M	CAP 1UF 6.3V CERAMIC X5R 0402	Panasonic ECG	0402	1	$0.1120	$0.11	$0.04 / 10000	0.04
Capacitor	Vreg Output	ECJ-0EB0J105M	CAP 1UF 6.3V CERAMIC X5R 0402	Panasonic ECG	0402	1	$0.1120	$0.11	$0.04 / 10000	0.124
Capacitor	Audio Filter	ECJ-0EB1H392K	CAP 3900PF 50V CERAMIC X7R 0402	Panasonic ECG	0402	2	$0.0190	$0.04	$0.006 / 10000	0.012
Capacitor	Mic. Coupling	ECJ-1VB0J475K	CAP CERAMIC 4.7UF 6.3V X5R 0603	Panasonic ECG	0603	1	$0.1970	$0.20	$0.054 / 4000	0.054
Resistor	Reset	ERJ-2GEJ473X	RES 47K OHM 1/16W 5% 0402 SMD	Panasonic ECG	0402	1	$0.0840	$0.08	$0.00567 / 10000	0.00567
Resistor	Mic. Coupling	ERJ-2RKF1001X	RES 1.00K OHM 1/16W 1% 0402 SMD	Panasonic ECG	0402	1	$0.0980	$0.10	$0.00851 / 10000	0.00851
Resistor	Audio Filter	ERJ-2RKF1022X	RES 10.2K OHM 1/16W 1% 0402 SMD	Panasonic ECG	0402	2	$0.0980	$0.20	$0.00851 / 10000	0.01702
Resistor	Pressure Rin	ERJ-2RKF1001X	RES 1.00K OHM 1/16W 1% 0402 SMD	Panasonic ECG	0402	1	$0.0980	$0.10	$0.00851 / 10000	0.00851
Resistor	Pressuure Rf	ERJ-2RKF1001X	RES 1.00K OHM 1/16W 1% 0402 SMD	Panasonic ECG	0402	1	$0.0980	$0.10	$0.00851 / 10000	0.00851
Resistor	Pressure Rpr	ERJ-2GEJ334X	RES 330K OHM 1/16W 5% 0402 SMD	Panasonic ECG	0402	1	$0.0840	$0.08	$0.00567 / 10000	0.00567
Resistor	Bridge Resistors	RR0510P-161-D	RES 160 OHM 1/16W .5% 0402 SMD	Susumu Co.	0402	6	$0.1220	$0.73	$0.00882 / 10000	0.05292
Resistor	SG Zero Ohm	ERJ-2GE0R00X	RES ZERO OHM 1/16W 5% 0402 SMD	Panasonic ECG	0402	2	$0.0840	$0.17	$0.00567 / 10000	0.01134
Resistor	I2C Pull-up	RR0510P-472-D	RES 4.7K OHM 1/16W .5% 0402 SMD	Susumu Co.	0402	2	$0.1220	$0.24	$0.00882 / 10000	0.01764
Resistor	LED (GB)	RR0510P-101-D	RES 100 OHM 1/16W .5% 0402 SMD	Susumu Co.	0402	2	$0.1220	$0.24	$0.00882 / 10000	0.01764
Resistor	LED (R)	RR0510P-201-D	RES 200 OHM 1/16W .5% 0402 SMD	Susumu Co.	0402	1	$0.1220	$0.12	$0.00882 / 10000	0.00882
Power	LDO Vreg	NCP553SQ33T1	IC REG LDO 80MA 3.3V SC70-4	ON Semiconductor	SC70-4	1	$0.7000	$0.70	$0.228 / 3000	0.228
Output	LED	LTST-C17FB1WT	0805 RGB Diffused	Lite-On	0805	1	$3.0000	$3.00	$1.48 / 500	1.48
Signal	MCU	MSP430F1610	IC MCU 16BIT 32K FLASH 64-QFN, 5K RAM	Texas Instruments	QFN	1	$13.7400	$13.74	$8.25 / 1000	8.25
Signal	MOSFET	FDG6301N	MOSFET N-CHAN DUAL 25V SC70-6	Fairchild Semiconductor	SC-70-6	1	$0.4300	$0.43	$0.12040/1000	0.1204
Signal	Crystal	ABMM2-8.000MHZ	ABRACON SMD Crystals 8 MHz	Abracon	SMD	1	$1.1900	$1.19	$0.57 / 2500	0.57
Signal	In. Amp	MAX4462HEUT	SOT23, 3V/5V, Single-Supply, Rail-to-Rail In. Amp	Maxim	SOT23-6	2	$2.3300	$4.66	$0.98 / 2500	1.96
Signal	Amplifier	MAX4400AXK	IC OPAMP GP R-TO-R SOT23-5	Maxim	SC70-5	4	$1.1800	$4.72	$0.33 / 2500	1.32
Signal	MUX	MAX4734EGC	0.8 Ohm, Low-Voltage, 4-Channel Analog Multiplexe	Maxim	uMAX	1	$1.4000	$1.40	$1.05 / 2500	1.05
Sensor	Temperature	LM20CIM7	IC TEMP SENSOR MICROSMD SC-70-5	National Semiconductor	SC70-5	1	$1.0900	$1.09	$0.36975 / 3000	0.36975
Sensor	Vibration	MSP6915-ND	MINI SENSE VERTICAL	Measurement Specialties	DIP	2	$3.0000	$6.0000	$1.245 / 100	2.49
Sensor	Light	TPS851/52	IC SENSOR PHOTO ILLUMINANCE SMD	Toshiba	SMD	1	$1.0800	$1.08	$0.46 / 3000	0.46
Sensor	Sound	SPM0102NE3	MIC SISONIC SMD 1.5-5.5V MINI	Knowles	SMD	1	$6.5000	$6.50	$4.25299 / 1200	4.25299
Sensor	Pressure	QTC Film	QTC FORCE SENSORS	Peratech	Special	1/8 Sheet	$40.0000	$5	30	3
TOTAL								$54.30		26.21739

Brain Components

Type	Name	Part Number	Description	Manufacturer	Package	# per node	Unit Price	Total	Bulk Price	Total Bulk
Capacitor	C3V	ECJ-1VB1C333K	CAP 33000PF 16V CERM X7R 0603	Panasonic - ECG	0603	1	$0.0350	0.035	$0.001292/1000	$0.0064
Capacitor	Cavcc	ECJ-1VB1C104K	CAP .1UF 16V CERAMIC X7R 0603	Panasonic - ECG	0603	1	$0.0350	0.035	$0.001292/1000	$0.0064
Capacitor	Vreg	ECJ-1VB1C104K	CAP .1UF 16V CERAMIC X7R 0604	Panasonic - ECG	0603	4	$0.0350	0.140	$0.001292/1001	$0.0255
Capacitor	Decoupling cap1	ECJ-1VB1C104K	CAP .1UF 16V CERAMIC X7R 0605	Panasonic - ECG	0603	2	$0.0350	0.070	$0.001292/1002	$0.0127
Capacitor	Decoupling cap2	ECJ-1VF1A105Z	CAP 1UF 10V CERAMIC Y5V 0603	Panasonic - ECG	0603	2	$0.0360	0.072	$0.01472/1000	$0.2944
Capacitor	Crystal caps	ECJ-0EC1H120J	CAP 12PF 50V CERAMIC 0402 SMD	Panasonic - ECG	0402	4	$0.0190	0.076	$0.00637/5000	$0.0255
Resistor	Ravcc	ERA-V33J471V	RES 470 OHM 1/16W 1500PPM 5%0603	Panasonic - ECG	0603	1	$0.5520	0.552	$0.14608/5000	$0.1461
Resistor	Rdm	ERJ-3GEYJ270V	RES 27 OHM 1/10W 5% 0603 SMD	Panasonic - ECG	0603	2	$0.0800	0.160	$0.01682/1000	$0.0336
Resistor	Rdo	ERJ-3GEYJ222V	RES 2.2K OHM 1/10W 5% 0603 SMD	Panasonic - ECG	0603	1	$0.0800	0.080	$0.01682/1000	$0.0168
Resistor	Rdo2	ERJ-3EKF1101V	RES 1.10K OHM 1/10W 1% 0603 SMD	Panasonic - ECG	0603	1	$0.0900	0.090	$0.01899/1000	$0.0190
Resistor	Rest MSP	ERJ-3GEYJ473V	RES 47K OHM 1/10W 5% 0603 SMD	Panasonic - ECG	0603	1	$0.0800	0.080	$0.01682/1000	$0.0168
Resistor	Rrsto	ERJ-3EKF1501V	RES 1.50K OHM 1/10W 1% 0603 SMD	Panasonic - ECG	0603	1	$0.0900	0.090	$0.01899/1000	$0.0190
Resistor	Rru1	ERJ-3EKF1101V	RES 1.10K OHM 1/10W 1% 0603 SMD	Panasonic - ECG	0603	1	$0.0900	0.090	$0.01899/1000	$0.0190
Resistor	Rru2	ERJ-3EKF1502V	RES 15.0K OHM 1/10W 1% 0603 SMD	Panasonic - ECG	0603	1	$0.0900	0.090	$0.01899/1000	$0.0190
Signal	Crystal	ABMM2-8.000MHZ	ABRACON SMD Crystals 8 MHz	Abracon	SMD	1	$1.1900	$1.19	$0.57 / 2500	0.57
Signal	Crystal	ATP060SM	CRYSTAL 6.0000 MHZ 20PF SMT	CTS	SMD	1	$2.8900	2.890	$0.99/1000	$0.9900
Signal	MCU	MSP430F1610	IC MCU 16BIT 32K FLASH 64-QFN, 5K RAM	Texas Instruments	QFN	1	$13.7400	$13.74	$8.25 / 1000	8.25
Signal	USB Chip	FT245BM	FTDI USB Interface ICs FT245BM FTDI CHIP USB to Parallel FIFC	FTDI	QFP	1	$4.8500	$4.85	$3.49/500	$3.4900
Signal	USB EEPROM	93LC46B	IC SRL EE 1K 64X16 2.5V 8MSOP	Microchip	MSOP	1	$0.5300	$0.53	$0.38/100	$0.3800
Signal	Active pull-ups	XP02401	TRANS ARRAY PNP/PNP SMINI-5P	Panasonic - SSG	S-mini 5P	2	$0.4200	$0.84	$0.11765/1000	$0.2353
Header	USB header	KUSB-SMT-BS1N-W	USB Connectors B TYPE SMT RECPT WHITE	Kycon	SMD	1	$1.6300	$1.63	$0.69/1000	$0.6900
Power	Vreg	LM3480IM3-3.3	IC REGULATOR LDO 100MA SOT-23	National Semiconductor	SOT-23-5	1	$0.3900	$1.13	$0.36/25000	$0.3600
Power	Vreg	LM3480IM3-5.0	IC REGULATOR LDO 100MA SOT-23	National Semiconductor	SOT-23-5	1	$0.3900	$1.13	$0.36/25000	$0.3600
Output	LED	LTST-C17FB1WT	0805 RGB Diffused	Lite-On	0805	1	$3.0000	$3.00	$1.48 / 500	1.48
TOTAL								$32.59		$17.4655

Appendix C

Schematics and PCB Layouts

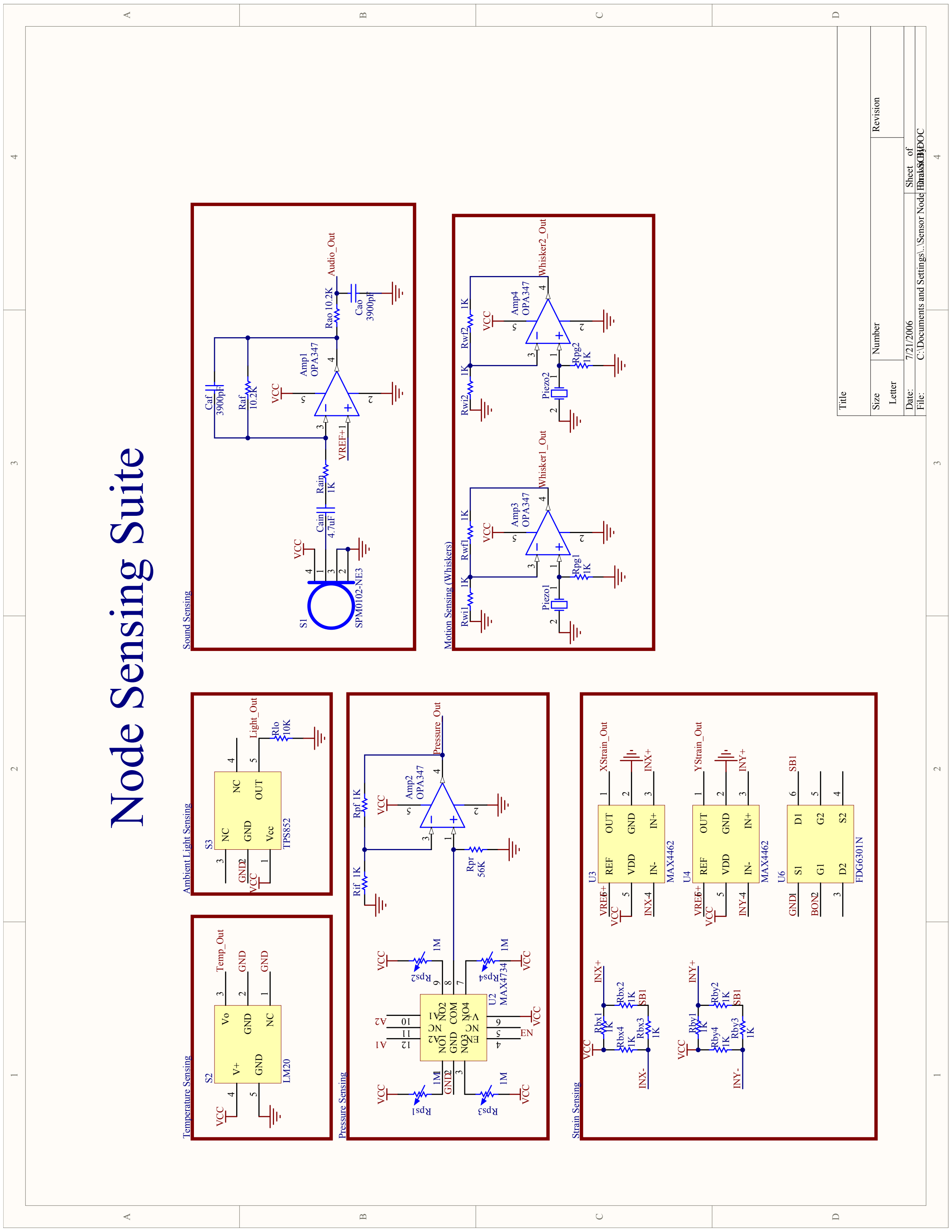


Figure C-1: Node Sensing Suite

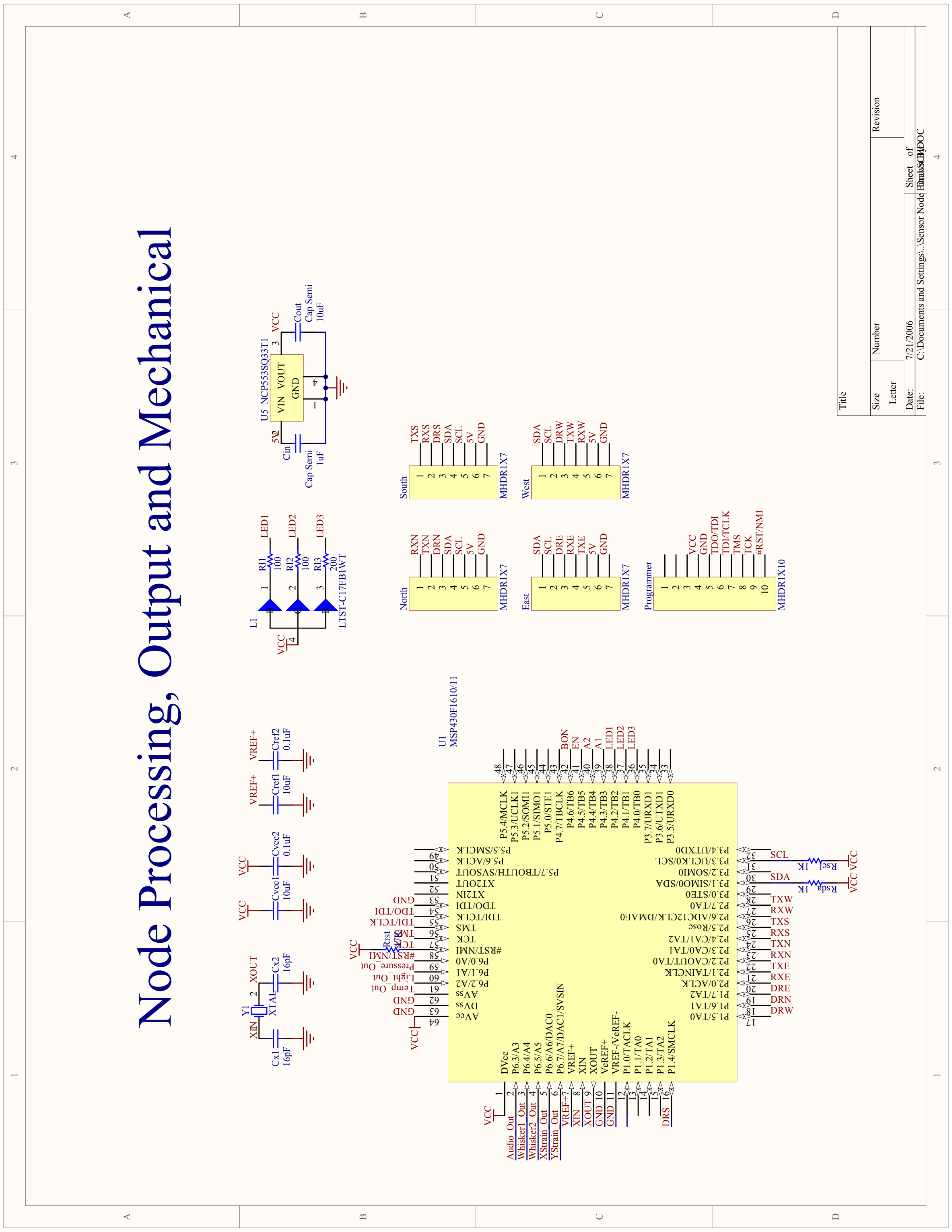


Figure C-2: Node Processing

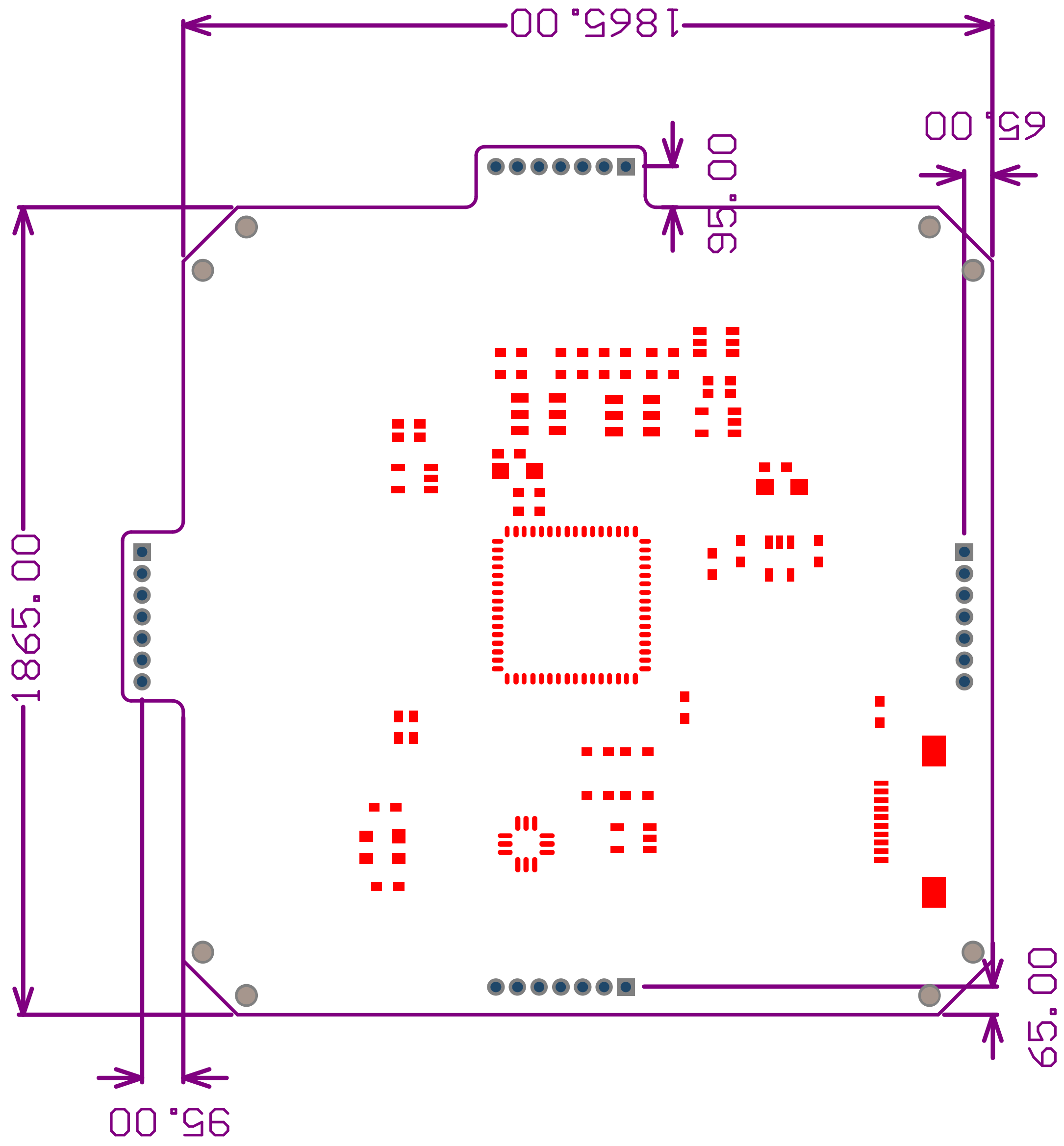


Figure C-3: Node Dimensions (mils)

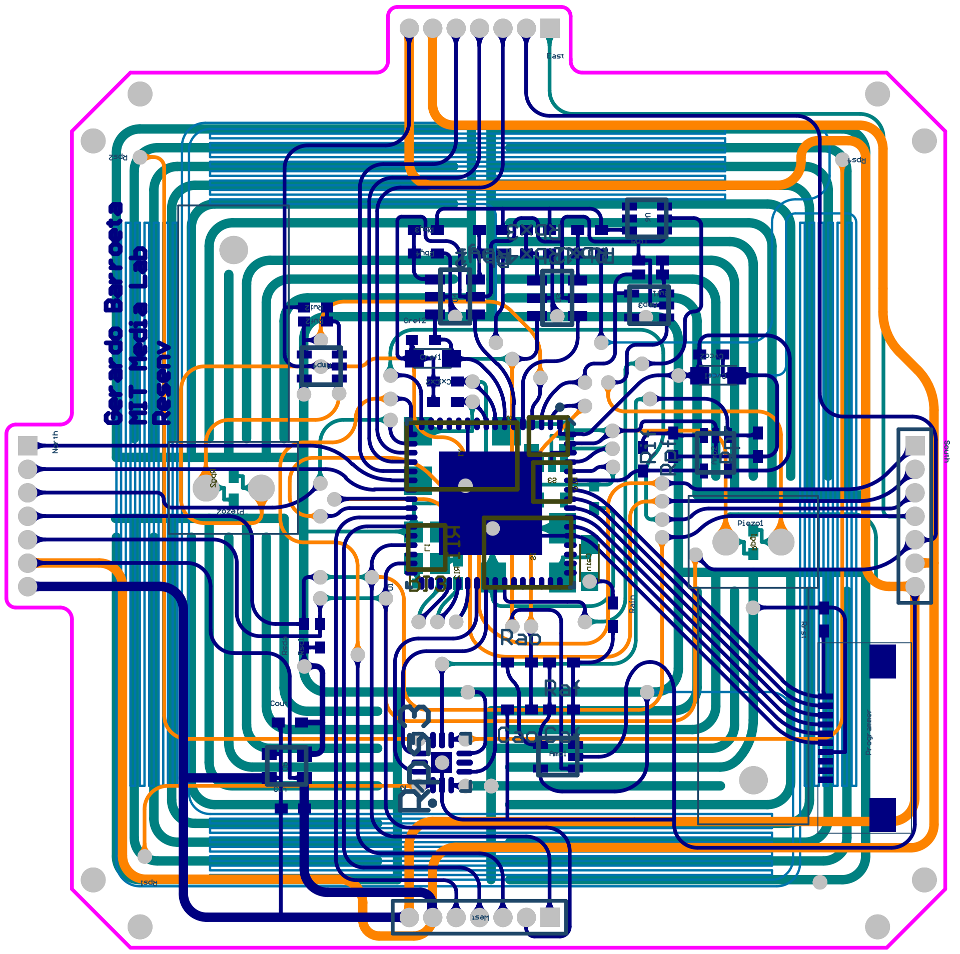


Figure C-4: Node Full Routing Stack

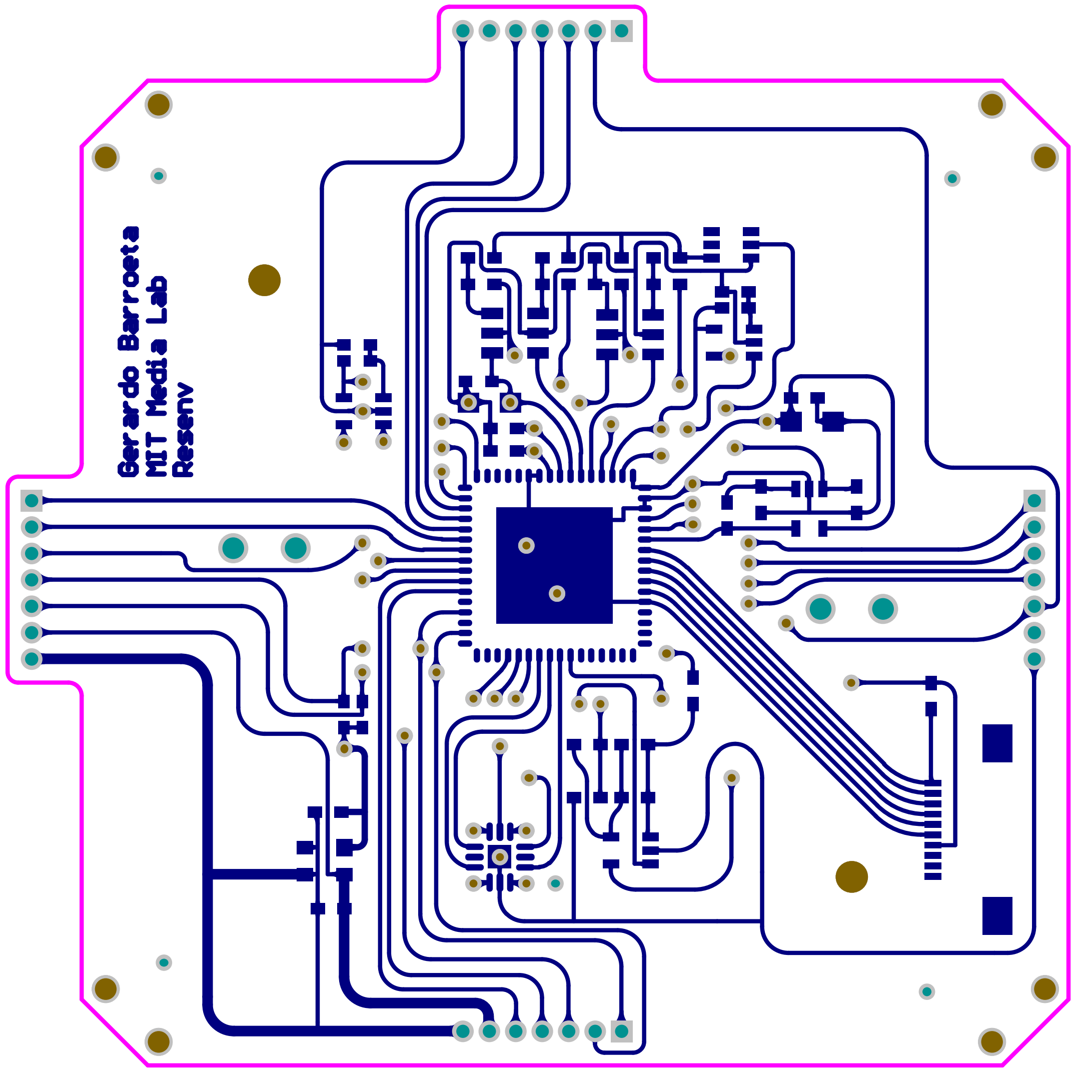


Figure C-5: Node Top Routing Layer

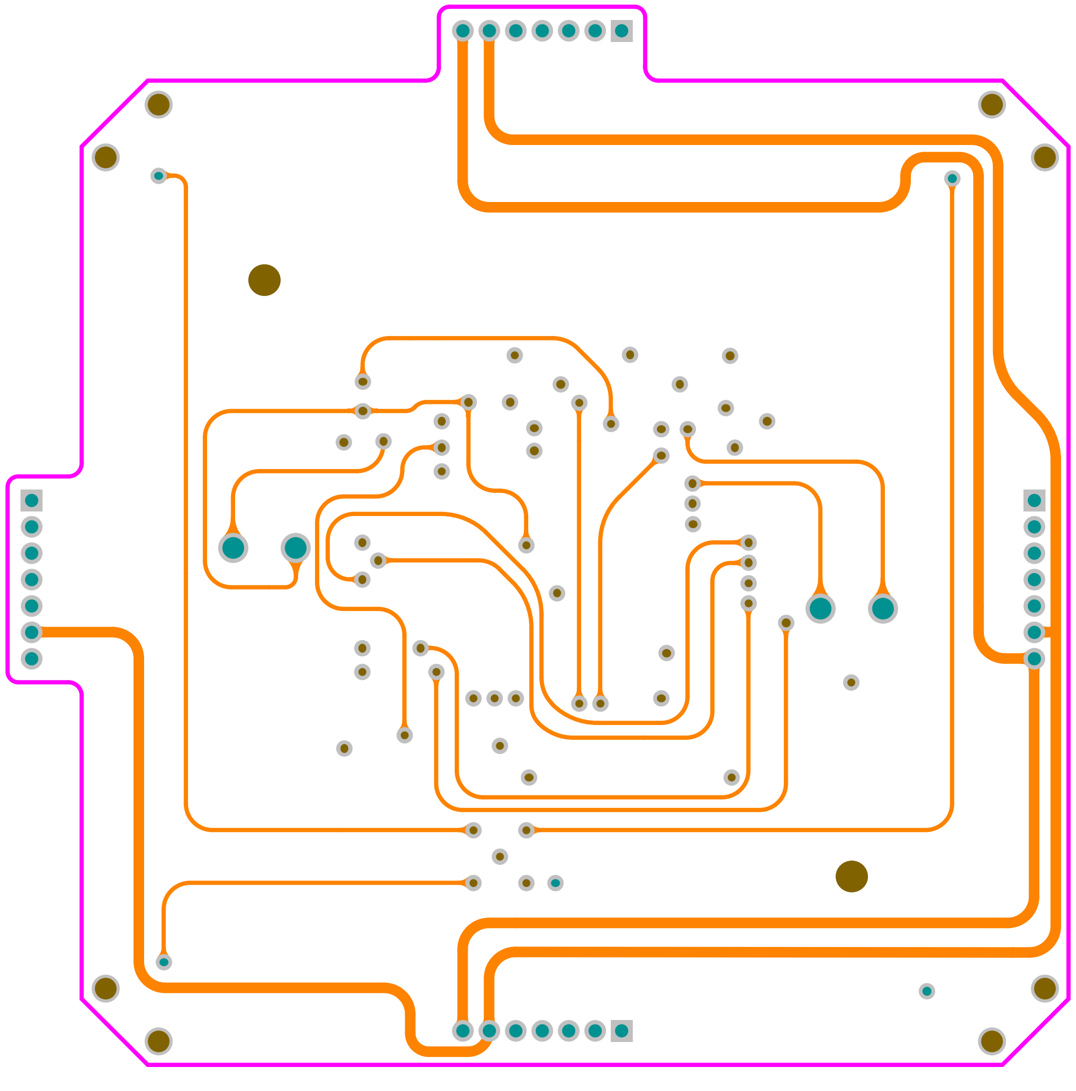


Figure C-6: Node Internal Layer 1

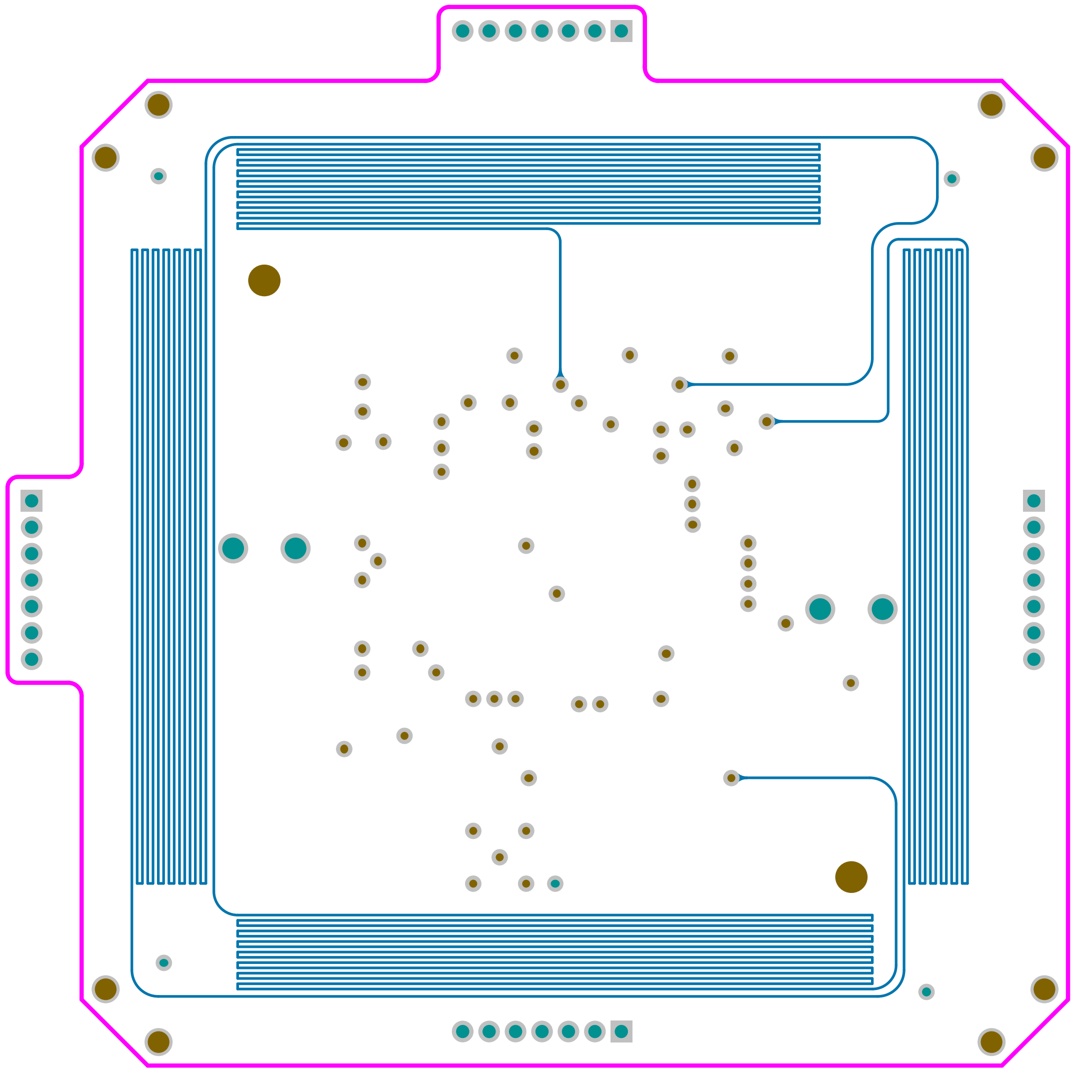


Figure C-7: Strain Gages (Internal Layer 2)

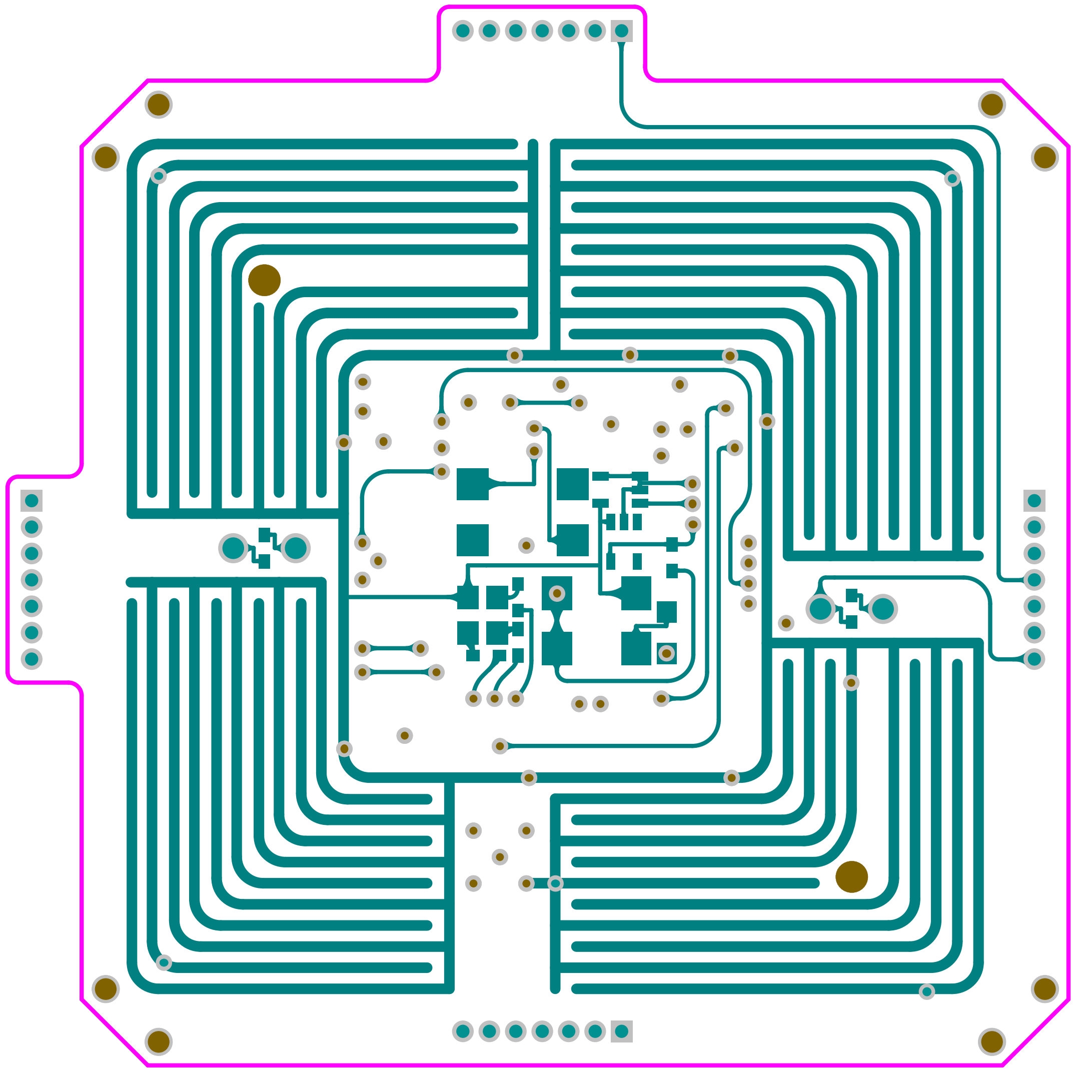


Figure C-8: Node Bottom Routing Layer

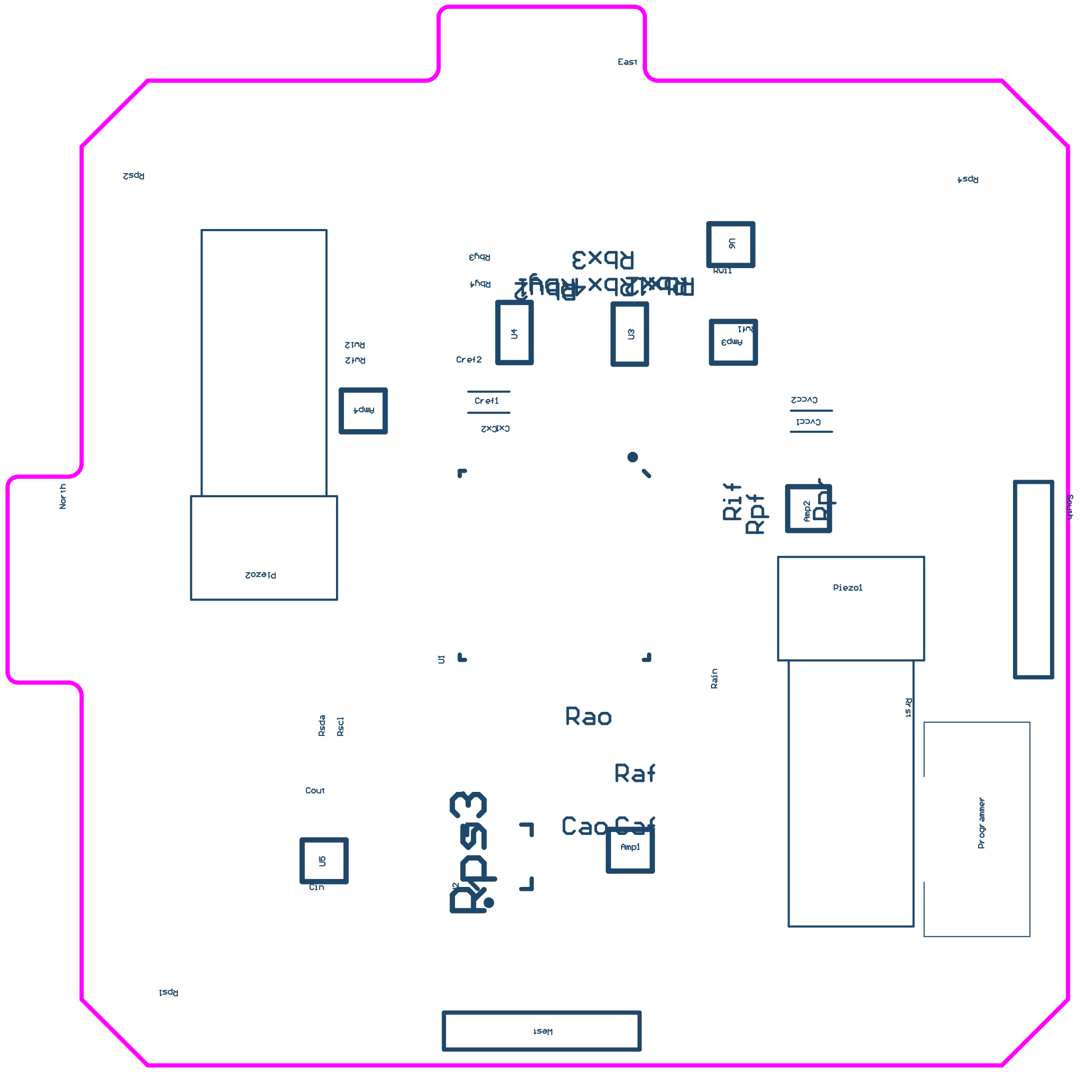


Figure C-9: Node Top Overlay

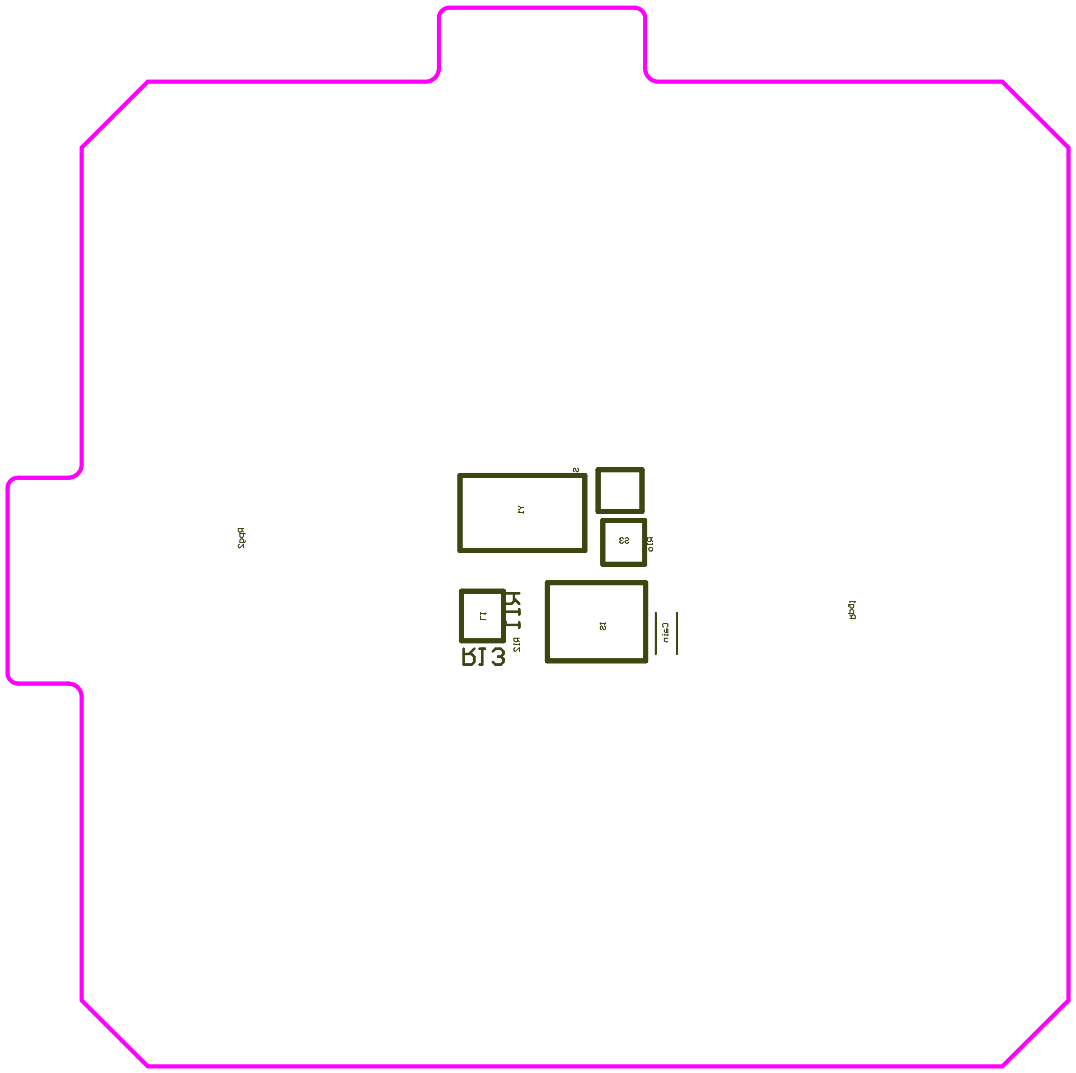


Figure C-10: Node Bottom Overlay

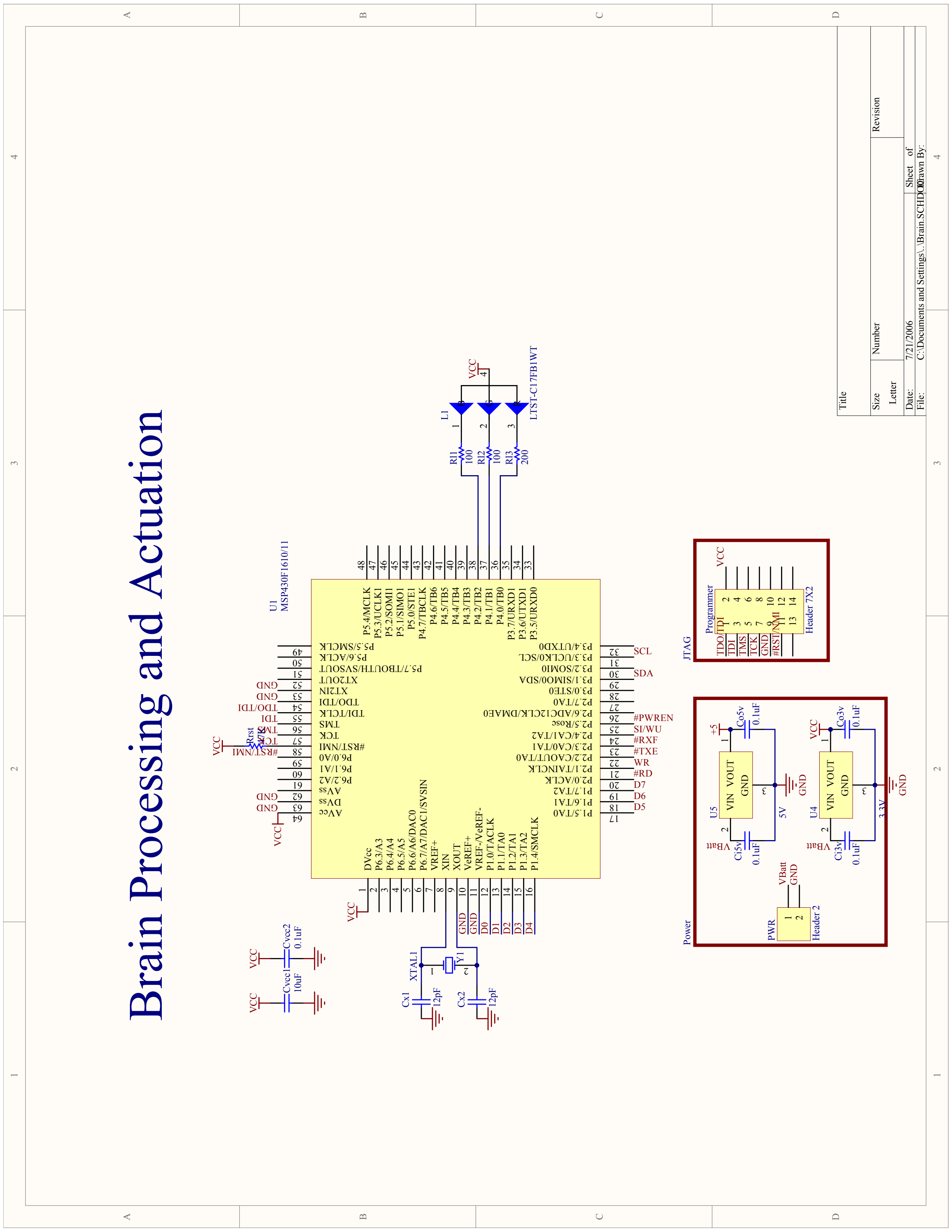


Figure C-11: Brain Processing

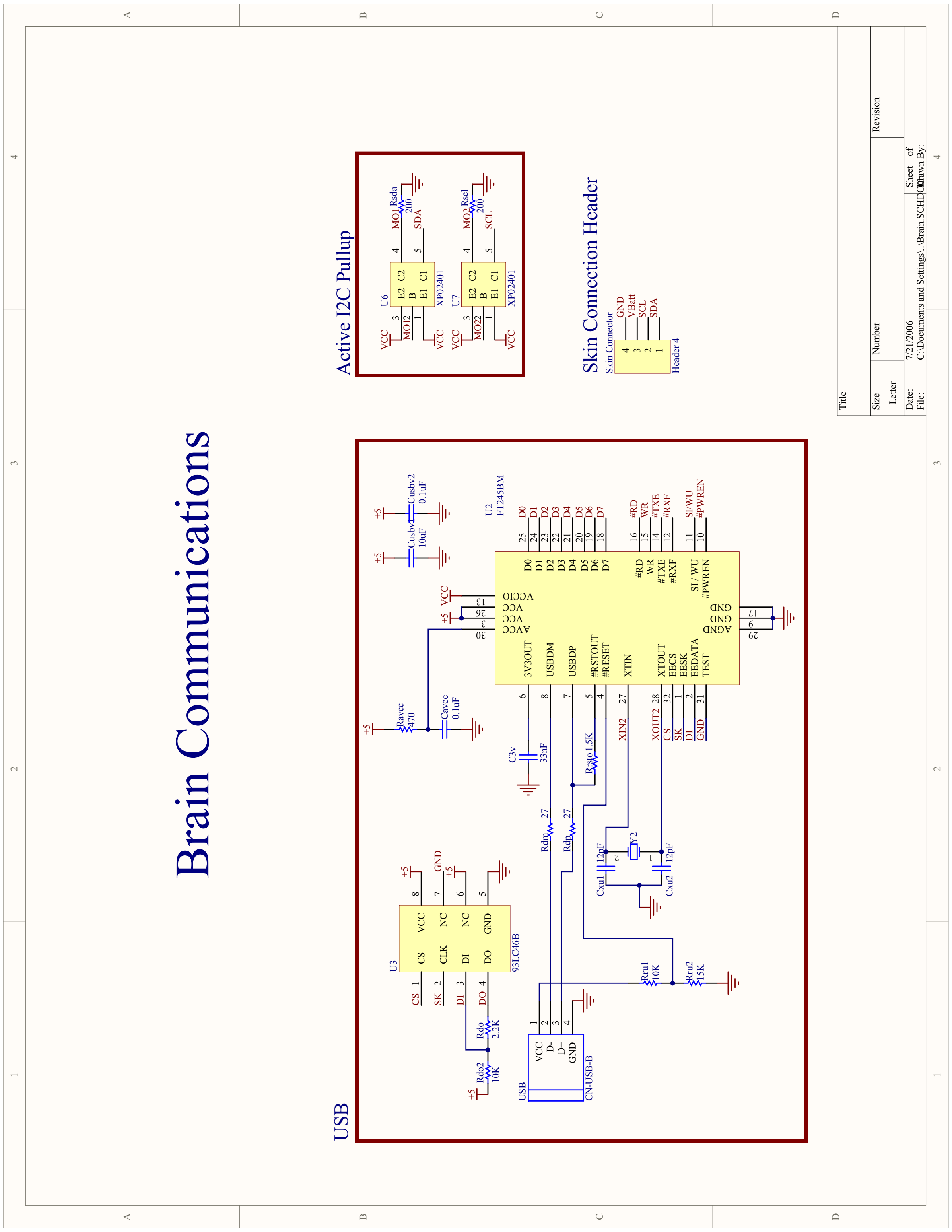


Figure C-12: Brain Communications

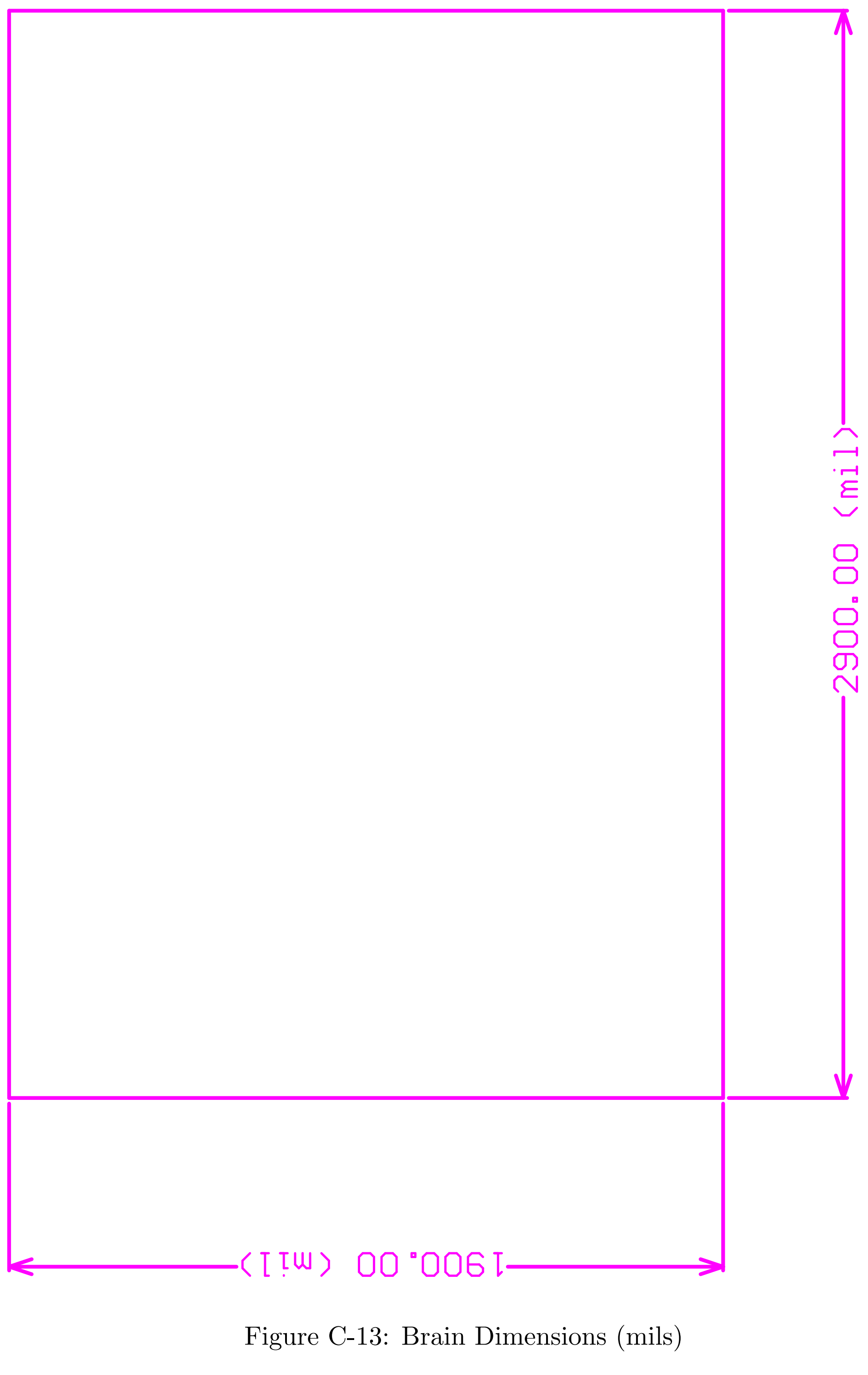


Figure C-13: Brain Dimensions (mils)

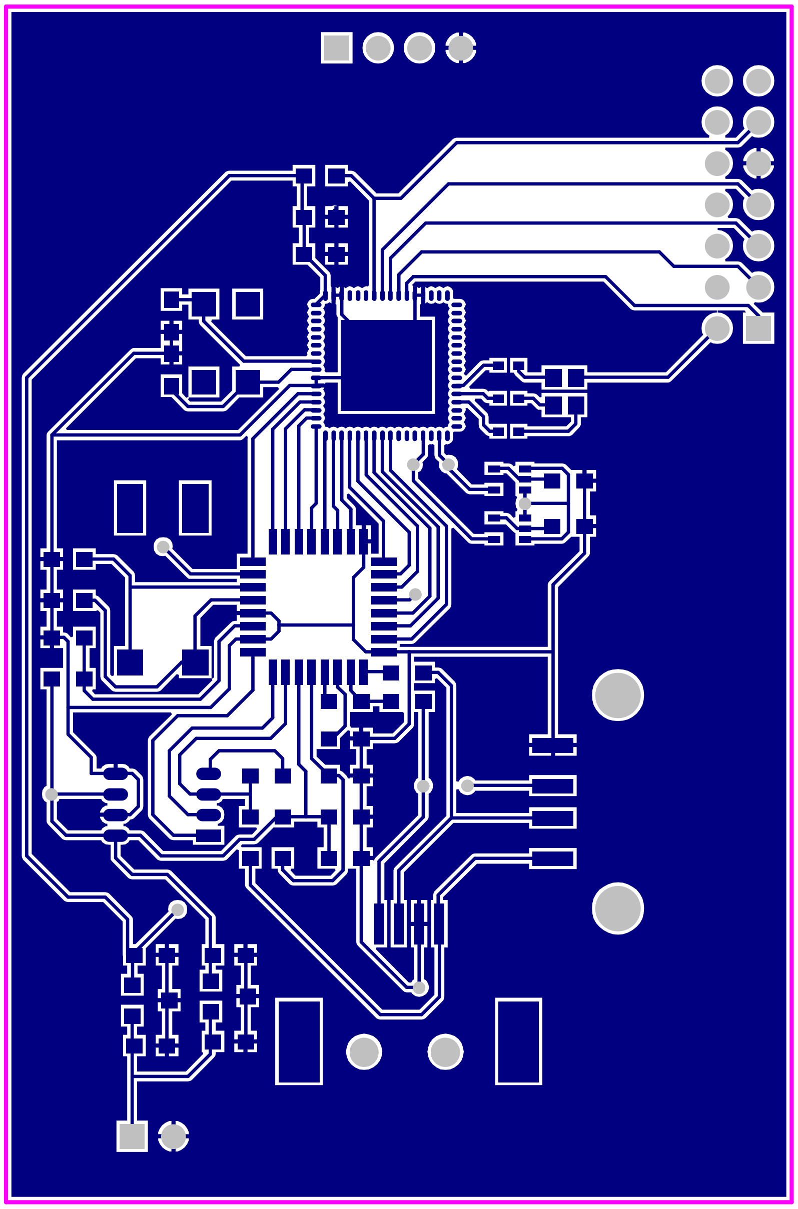


Figure C-14: Brain Top Layer

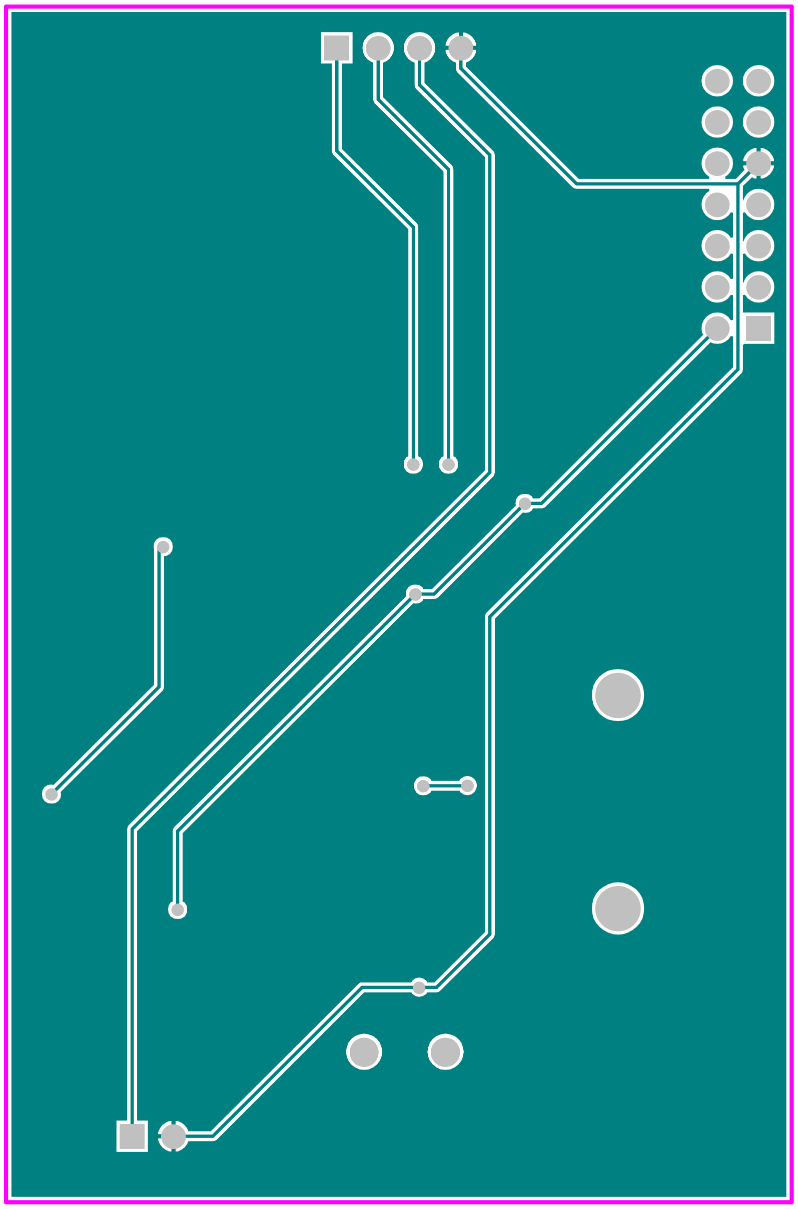


Figure C-15: Brain Bottom Layer

Appendix D

Firmware

```
#include  "msp430x16x.h"
#define MAX_CODE_SIZE           (0x1000)
#define CODE_START_ADD          (0x8000)
#define REFLASH_INST            (0xA0)
#define RESET_BRAIN_INST        (0xA2)
#define RESET_NODE_INST         (0xA4)
#define EXECUTE_CODE_INST       (0xA6)
#define I2C_OWN_ADDRESS         (261)

;-------------------------------------------------------------------------------
                ORG 1140h
;-------------------------------------------------------------------------------

fileBuf         DS MAX_CODE_SIZE
fileBufPtr      DS 2
Address         DS 2
nBytes          DS 2
codeStartAdd    DS 2
fileSize        DS 2
codeSize        DS 2
DataFlags       DS 1
fileFlags       DS 1
I2CFlags        DS 1

;-------------------------------------------------------------------------------
                ORG 0FA00h                 ; Progam Start
;-------------------------------------------------------------------------------
RESET
        MOV.W   #1130h,SP               ; Initialize 'x1x9 stackpointer

        //STOP WATCHDOG
        MOV.W   #WDTPW+WDTHOLD, &WDTCTL ; stop watchdog

        //INITIALIZE SVS
        MOV.B   #060h+PORON,&SVSCTL     ; SVS POR enabled @ 2.6V

        //INITIALIZE CRYSTAL (8.000 MHZ)
        BIC     #OSCOFF, SR
        BIS.B   #XTS, &BCSCTL1          ; LFXTL high-freq mode
L1                                      ; wait for crystal to stabilize
        BIC.B   #OFIFG,&IFG1            ; Clear OFIFG
        MOV     #0FFh,R15               ; Delay
L2
        DEC R15
        JNZ L2
        BIT.B   #OFIFG,&IFG1            ; Re-test OFIFG
        JNZ L1                          ; Repeat test if needed

        //INITIALIZE PORTS
        BIS.B   #07Fh,&P4DIR            ; Set as outputs
        MOV.B   #0B5h,&P4OUT            ; Turn off Strain gauges

        //INITIALIZE I2C PORT
        BIS.B   #0Ah,&P3SEL             ; Select I2C pins
        BIS.B   #I2C+SYNC+XA,&U0CTL     ; Recommended init procedure
        BIC.B   #I2CEN,&U0CTL           ; Recommended init procedure
        BIS.B   #I2CSSEL0+I2CRM,&I2CTCTL; ACLK
        MOV.B   #7, I2CSCLH             ; Set Baud Rate to 400Khz
```

```
        MOV.B   #7, I2CSCLL             ; Set Baud Rate to 400Khz
        MOV     #I2C_OWN_ADDRESS,&I2COA ; Slave Address is 048h
        BIS.B   #I2CEN,&U0CTL           ; Enable I2C

        //INITIALIZE VARIABLES
        MOV.W   #0, fileBufPtr          ; Init pointer
        MOV.B   #0, fileFlags           ; Init flags
        MOV.B   #0, I2CFlags
        CLR     R6

        MOV.W   #130, R10               ; Delay to signal reset
        CALL    #DELAY                  ;
        MOV.B   #0BFh, &P4OUT           ; Turn off Led's
        EINT                            ; Enable interrupts

MainLoop
        MOV     #1, R10                 ; Wait until An instruction
        CALL    #I2C_GET_BYTE           ; is received, and then decode it
        BIT.B   #1, &I2CFlags           ; if flag is set, nothing was received
        JC      MainLoop                ; jump back to main loop
        BIC.B   #OAIFG+GCIFG, &I2CIFG

DecodeInstruction
        CMP.B   #REFLASH_INST, R5       ; If it is a reflash instruction
        JZ      ReflashSelf             ; get file first, then reflash device
        CMP.B   #RESET_NODE_INST, R5    ; If the instruction is to reset
        JZ      ResetSelf               ; Write invalid code to Watchdog for PUC
        CMP.B   #EXECUTE_CODE_INST, R5  ; If there is a code loaded and it
        JZ      ExecuteCode             ; needs to execute, jump to its start address
        JMP     MainLoop                ; Invalid instruction/data

ReflashSelf
        MOV     #200h, R10
        CALL    #I2C_GET_BYTE
        MOV     R5, R6
        CALL    #I2C_GET_BYTE
        SWPB    R5
        XOR     R6, R5
        MOV     R5, fileSize
        BIT.B   #01, I2CFlags
        JNC     GetFile
        BIS.B   #06, fileFlags
        MOV     #0FFh, R10
        MOV.B   fileFlags, R5
        CALL    #I2C_SLAVE_READ
        JMP     MainLoop
        //Then, receive the file
GetFile
        CALL    #GET_FILE
        CALL    #REFLASH_SELF
        JMP     MainLoop

ResetSelf
        MOV     #0, WDTCTL              ; Invalid watchdog write to cause PUC

ExecuteCode
        MOV     #01000h, R5             ; Load pointer
        CMP     #0, 0(R5)               ; This address contains a flag indicating
        JNZ     MainLoop                ; if there is a valid code programmed.

        MOV     2(R5), codeStartAdd
        MOV.B   #0, &I2CIFG
        BIC.B   #I2CEN,&U0CTL           ; Reconfigure I2C
        DINT
        BR      &codeStartAdd           ; Jump to start address of flashed code

;_______________________________________________________________________________
I2C_SLAVE_READ  ; Send a byte through I2C
;_______________________________________________________________________________
        RET

;_______________________________________________________________________________
I2C_GET_BYTE    ; Obtain a byte through I2C
;_______________________________________________________________________________
        BIT.B   #RXRDYIFG, &I2CIFG
        JC      GetI2CByte
        DEC.B   R10
        JNZ     I2C_GET_BYTE
        BIS.B   #01h, I2CFlags
        RET
GetI2CByte
        MOV.B   &I2CDRB, R5
        BIC.B   #RXRDYIFG, &I2CIFG
        BIC.B   #01h, I2CFlags
        RET

;_______________________________________________________________________________
GET_FILE  ; Gets a file through I2C
;_______________________________________________________________________________
        MOV     #0FFh, R10
        CALL    #I2C_GET_BYTE
        BIT.B   #01, I2CFlags
        JNC     SaveByte
        BIS.B   #02h, fileFlags
        RET

SaveByte
        MOV     fileBufPtr, R6
        MOV.B   R5, fileBuf(R6)
        INC     R6
        CMP     #1000h, R6
        JNZ     NotFileSizeLimit
        CLR     R6
        BIS.B   #04h, fileFlags

NotFileSizeLimit
        MOV     R6, fileBufPtr
        MOV     R6, nBytes
        DEC     fileSize
        JZ      fileDone
        //MOV   R8, fileSize
        JMP     GET_FILE
fileDone
        MOV     nBytes, fileSize
        MOV     #0, fileBufPtr
        BIS.B   #1, fileFlags
```

```
        RET

;_______________________________________________________________________________
REFLASH_SELF  ; Burns a received code file into Flash memory
;_______________________________________________________________________________
        //FIRST CALCULATE CHECKSUM FOR EVERY LINE, IF ANYTHING IS WRONG, ABORT.
        CLR     fileBufPtr
        CLR     codeSize
        CLR     R5
        MOV     fileSize, R6

CheckNextLine
        CLR     R7                      ; R7 Holds line Accumulator
        MOV.B   fileBuf(R5), R8         ; First byte contains # of data bytes
        ADD     R8, codeSize            ; Calculate total code size
        ADD     R8, R7                  ; Accumulate
        SUB     R8, R6                  ; Decrease total bytes to process
        INC     R5                      ; Increment pointer
        ADD.B   fileBuf(R5), R7         ; Accumulate address 1st byte
        INC     R5                      ; Increment pointer
        ADD.B   fileBuf(R5), R7         ; Accumulate address 2nd byte
AddNextByte
        INC     R5                      ; Increment pointer
        ADD.B   fileBuf(R5), R7         ; Accumulate nth data byte
        DEC.B   R8                      ; Decrease data byte counter
        JNZ     AddNextByte             ; Again
        INV.B   R7                      ; Calculate checksum by
        INC.B   R7                      ; taking the 2's complement
        INC     R5                      ; of the sum
        CMP.B   fileBuf(R5), R7         ; compare to stored checksum
        JNZ     ErrorCS                 ; Was there an error?
        INC     R5                      ; increment pointer
        SUB     #4, R6                  ; Are we done?
        JNZ     CheckNextLine           ;
        SUB     #3, R5                  ; Now calculate code Start Address
        MOV.B   fileBuf(R5), R6         ;
        INC     R5                      ;
        MOV.B   fileBuf(R5), R7         ;
        SWPB    R7                      ;
        BIS     R7, R6                  ;
        MOV     R6, codeStartAdd        ; Save code Start Address
        ADD     codeSize, R6

        //EVERYTHING WAS CORRECT, REFLASH DEVICE
        CMP     #1000h, codeSize        ; Before doing that, is code bigger
        JGE     FileTooBig              ; than the allowed limit?
        CMP     #0FA00h, R6             ; Is code size + code start add bigger than
        JGE     FileTooBig              ; allowed? yes, abort
        MOV     #8000h, R5              ; Nope, move the start address to pointer
        MOV.W   #WDTPW+WDTHOLD, &WDTCTL ; stop watchdog
        DINT                            ; Disable Interrupts
        MOV.W   #FWKEY+FSSEL_1, &FCTL2  ; Configure Flash controller
        MOV.W   #FWKEY+FN4+FN1+FN0, &FCTL2 ; Configure Flash controller
        MOV.W   #FWKEY,&FCTL3           ; Clear LOCK

        //ERASE FLASH MEMORY EXCEPT FOR BL SEGMENTS
EraseNext
        MOV     #FWKEY+ERASE,&FCTL1     ; Enable segment erase
        CLR     0(R5)                   ; Dummy Write to erase
        ADD     #200h, R5               ; Point to next segment
        CMP     #0FA00h,R5              ; Stop when address equal to BL address
        JNE     EraseNext               ;
        //ERASE INTERRUPT VECTORS AND INFO MEMORY
        MOV     #0FE00h, R5             ; Erase interrupt vectors
        MOV     #FWKEY+ERASE,&FCTL1     ; Enable segment erase
        CLR     0(R5)                   ; Dummy Write to erase
        MOV     #01000h, R5             ; Now erase Info Memory
        MOV     #FWKEY+ERASE,&FCTL1     ; Enable segment erase
        CLR     0(R5)                   ; Dummy Write to erase
        MOV     #01080h, R5             ; Now erase Info Memory
        MOV     #FWKEY+ERASE,&FCTL1     ; Enable segment erase
        CLR     0(R5)                   ; Dummy Write to erase
        MOV     #0FFFEh, R5             ; Restore BL reset vector
        MOV     #FWKEY+WRT,&FCTL1       ; Enable segment write
        MOV     #0FA00h, 0(R5)          ; Dummy Write to restore RESET vector

        //NOW BURN CODE TO FLASH!
        CLR     R5
        MOV     codeSize, R9
FlashNextLine
        MOV.B   fileBuf(R5), R8         ; Get # of data bytes of current line
        INC     R5                      ;
        MOV.B   fileBuf(R5), R6         ; Calculate address of line
        INC     R5                      ;
        MOV.B   fileBuf(R5), R7         ;
        SWPB    R6                      ;
        BIS     R7, R6                  ;
        MOV     R8, R7                  ;
        CMP     #0FA00h, R6             ; Is it a valid address?
        JGE     IVCheck                 ;
        JMP     FlashLine
IVCheck
        CMP     #0FFE0h, R8             ; Is is an interrupt vector?
        JL      InvalidAddress
        CMP     #0FFFEh, R6             ; Is it the Reset vector?
        JNE     FlashLine               ; Nope, flash it as a regular line
        ADD     #3, R5
        DECD    R9
        JNZ     FlashNextLine
        JMP     DoneFlashing
FlashLine
        INC     R5
        MOV     #FWKEY+WRT,&FCTL1       ; Enable segment write
        MOV.B   fileBuf(R5), 0(R6)      ; Write to Flash
        INC     R6
        DEC.B   R8
        JNZ     FlashLine
        INCD    R5
        SUB     R7, R9
        JNZ     FlashNextLine
DoneFlashing
        MOV     #1000h, R5              ; Inidicate a successful write
        MOV     #FWKEY+WRT,&FCTL1       ; Enable segment write
        MOV     #0, 0(R5)
        INCD    R5
        MOV     codeStartAdd, 0(R5)     ;
```

```
        MOV     #1080h, R5
        MOV     #I2C_OWN_ADDRESS, 0(R5) ; Burn the I2C address in flash
        MOV     #FWKEY+LOCK, &FCTL3     ; Done Writing, set lock
        EINT                            ; re-enable interrupts
        BIS.B   #80h, fileFlags         ; Indicate Successful writing!
        RET                             ; DONE!!!

ErrorCS
        BIS.B   #08, fileFlags
        RET
FileTooBig
        BIS.B   #10h, fileFlags
        RET
InvalidAddress
        BIS.B   #20h, fileFlags
        RET


        RET
;______________________________________________________________________________
DELAY ; Delay function (3n^2 + 4n + delay-9)
;______________________________________________________________________________
        PUSH R8
        PUSH R9
        MOV R10, R9
LoopDelay1
        MOV R10, R8
LoopDelay2
        DEC R8
        JNZ LoopDelay2
        DEC R9
        JNZ LoopDelay1
        POP R9
        POP R8
        RET

;------------------------------------------------------------------------------
;           Interrupt Vectors Used MSP430x13x/14x/15x/16x
;------------------------------------------------------------------------------
        ORG     0FFFEh                  ; MSP430 RESET Vector
        DW      RESET                   ;

END
```

```
#include  "msp430x16x.h"
#define REFLASH_ALL_INST        (0xA0)
#define RESET_BRAIN_INST        (0xA2)
#define RESET_SKIN_INST         (0xA4)
#define EXECUTE_CODE_INST       (0xA6)
#define DEEP_SLEEP              (0xB0)

;______________________________________________________________________
; Variables
;______________________________________________________________________
;RAM VARIABLES
;-------------------------------------------------------------------------------
        ORG 1140h
;-------------------------------------------------------------------------------
Packet          EQU     Pressure
Sound           DS      1024
Light           DS      256
Temp            DS      8
Pressure        DS      4
SoundAvg        DS      2
SoundVar        DS      2
XStrain         DS      1
YStrain         DS      1
LightValue      DS      1
TempValue       DS      1
TempVar         DS      0,0,0,0
SoundPtr        DS      2
SoundPtrL       DS      2
LightPtr        DS      2
TempPtr         DS      2
PressurePtr     DS      2
PacketPtr       DS      2
Flags           DS      1
BootLoaderAdd   DS      1
SoundFlag       DS      1
;VALUES
LIGHT_MAX   EQU 0800h

;-------------------------------------------------------------------------------
        ORG     08000h                  ; Progam Start
;-------------------------------------------------------------------------------
RESET
        MOV.W   #1120h,SP               ; Initialize 'x1x9 stackpointer

        //STOP WATCHDOG
        MOV.W   #WDTPW+WDTHOLD, &WDTCTL ; stop watchdog

        //INITIALIZE SVS
        MOV.B   #060h+PORON,&SVSCTL     ; SVS POR enabled @ 2.5V

        //INITIALIZE CRYSTAL (8.000 MHZ)
        BIC     #OSCOFF,SR              ; Turn on osc.
        BIS.B   #XTS, &BCSCTL1          ; LFXTL high-freq mode
L1                                      ; wait for crystal to stabilize
        BIC.B   #OFIFG,&IFG1            ; Clear OFIFG
        MOV     #0FFh,R15               ; Delay
L2
        DEC R15
```

```
        JNZ L2
        BIT.B   #OFIFG,&IFG1            ; Re-test OFIFG
        JNZ L1                          ; Repeat test if needed
        BIS.B   #SELM_3,&BCSCTL2        ; Select XT1 as source for MCLK
        BIS.B   #SELS, &BCSCTL2         ; select XT1CLK as source for SMCLK

        //INITIALIZE PORTS
        BIS.B   #7FH, &P4DIR            ; Set as outputs
        BIS.B   #0CFh, &P6SEL           ; Enable A/D channel inputs
        MOV.B   #0A3h, &P4OUT
        BIS.B   #07h, &P4SEL

        //INITIALIZE I2C
        BIS.B   #0Ah,&P3SEL             ; Select I2C pins
        BIS.B   #I2C+SYNC+XA,&U0CTL     ; Recommended init procedure
        BIC.B   #I2CEN,&U0CTL           ; Recommended init procedure
        BIS.B   #I2CSSEL0,&I2CTCTL      ; ACLK
        MOV.B   #07h, I2CSCLH           ; Set Baud Rate to 400Khz
        MOV.B   #07h, I2CSCLL           ; Set Baud Rate to 400Khz
        BIS.B   #RXRDYIFG+GCIE, &I2CIE  ; Activate I2C receive Interrupt
        MOV     #0107h,&I2COA           ; Own Address is 1AAh
        BIS.B   #I2CEN,&U0CTL           ; Enable I2C

        //INITIALIZE ADC
        MOV     #ADC12ON+SHT0_3+REFON,&ADC12CTL0  ; Turn on ADC12, set MSC
        MOV     #SHP,&ADC12CTL1         ; Use samp. timer, single sequence
        BIS.B   #INCH_0,&ADC12MCTL0     ; AVcc=ref+, channel=A0 (pressure channel)
        BIS.B   #INCH_1,&ADC12MCTL1     ; AVcc=ref+, channel=A0 (light channel)
        BIS.B   #INCH_2,&ADC12MCTL2     ; AVcc=ref+, channel=A0 (temperature channel)
        BIS.B   #INCH_3,&ADC12MCTL3     ; AVcc=ref+, channel=A0
        BIS.B   #INCH_4,&ADC12MCTL4     ; AVcc=ref+, channel=A0
        BIS.B   #INCH_5,&ADC12MCTL5     ; AVcc=ref+, channel=A0
        BIS.B   #INCH_6,&ADC12MCTL6     ; AVcc=ref+, channel=A0
        BIS.B   #INCH_7+EOS,&ADC12MCTL7 ; AVcc=ref+, channel=A0
        MOV     #00BFh, &ADC12IE        ; Activate Interrupts
        BIS     #ENC,&ADC12CTL0         ; Enable conversions

        //INITIALIZE TIMER B
        MOV.W   #OUTMOD_4+CCIE,&TBCCTL0 ; CCR1 reset/set
        MOV.W   #OUTMOD_4+CCIE,&TBCCTL1 ; CCR1 reset/set
        MOV.W   #OUTMOD_4+CCIE,&TBCCTL2 ; CCR1 reset/set
        MOV.W   #OUTMOD_4+CCIE,&TBCCTL3 ; CCR1 reset/set
        MOV.W   #OUTMOD_4+CCIE,&TBCCTL4 ; CCR1 reset/set
        MOV.W   #OUTMOD_4+CCIE,&TBCCTL5 ; CCR1 reset/set
        MOV.W   #OUTMOD_4+CCIE,&TBCCTL6 ; CCR1 reset/set
        MOV.W   #TBIE+TBSSEL_1+MC_2,&TBCTL   ; ACLK, upmode

        //INITIALIZE VARIABLES
        MOV.W   #0, SoundPtr
        MOV.W   #2, SoundPtrL
        MOV.W   #0, LightPtr
        MOV.W   #0, PressurePtr
        MOV.W   #0, TempPtr
        MOV.W   #0, SoundAvg
        MOV.W   #0, PacketPtr
        EINT

MainLoop
```

```
        JMP     MainLoop

;_______________________________________________________________________
DISABLE_TBCCRIRQ ; Disable TBCCRx interrupts to allow sampling of ADC channel
;_______________________________________________________________________
        BIC     #CCIE,&TBCCTL1          ; Disable all TBCCR IRQ's to allow ADC conv.
        BIC     #CCIE,&TBCCTL2          ; Disable all TBCCR IRQ's to allow ADC conv.
        BIC     #CCIE,&TBCCTL3          ; Disable all TBCCR IRQ's to allow ADC conv.
        BIC     #CCIE,&TBCCTL4          ; Disable all TBCCR IRQ's to allow ADC conv.
        BIC     #CCIE,&TBCCTL5          ; Disable all TBCCR IRQ's to allow ADC conv.
        BIC     #CCIE,&TBCCTL6          ; Disable all TBCCR IRQ's to allow ADC conv.
        RET

;_______________________________________________________________________
ENABLE_TBCCRIRQ ; ENABLE TBCCRx interrupts to allow sampling of ADC channel
;_______________________________________________________________________
        BIS     #CCIE,&TBCCTL1          ; Disable all TBCCR IRQ's to allow ADC conv.
        BIS     #CCIE,&TBCCTL2          ; Disable all TBCCR IRQ's to allow ADC conv.
        BIS     #CCIE,&TBCCTL3          ; Disable all TBCCR IRQ's to allow ADC conv.
        BIS     #CCIE,&TBCCTL4          ; Disable all TBCCR IRQ's to allow ADC conv.
        BIS     #CCIE,&TBCCTL5          ; Disable all TBCCR IRQ's to allow ADC conv.
        BIS     #CCIE,&TBCCTL6          ; Disable all TBCCR IRQ's to allow ADC conv.
        RET
;_______________________________________________________________________
DELAY ; Delay function (3n^2 + 4n + delay-9)
;_______________________________________________________________________
        PUSH    R8
        PUSH    R9
        MOV     R10, R9
LoopDelay1
        MOV     R10, R8
LoopDelay2
        DEC     R8
        JNZ     LoopDelay2
        DEC     R9
        JNZ     LoopDelay1
        POP     R9
        POP     R8
        RET

;-----------------------------------------------------------------------------
ADC12ISR        ; Interrupt Service Routine for ADC12
;-----------------------------------------------------------------------------
        ADD     &ADC12IV,PC             ; Add offset to PC 3
        RETI                            ; Vector 0: No interrupt 5
        JMP     ADOV                    ; ADC Overflow
        JMP     ADTOV                   ; ADC Timing Overflow
        JMP     ADM0                    ; ADC Mem0 ISR
        JMP     ADM1                    ; ADC Mem1 ISR
        JMP     ADM2                    ; ADC Mem2 ISR
        JMP     ADM3                    ; ADC Mem3 ISR
        JMP     ADM4                    ; ADC Mem4 ISR
        JMP     ADM5                    ; ADC Mem5 ISR
        JMP     ADM6                    ; ADC Mem6 ISR
        JMP     ADM7                    ; ADC Mem7 ISR
        JMP     ADM8                    ; ADC Mem8 ISR
        JMP     ADM9                    ; ADC Mem9 ISR
        JMP     ADM10                   ; ADC Mem10 ISR
```

```
        JMP     ADM11                   ; ADC Mem11 ISR
        JMP     ADM12                   ; ADC Mem12 ISR
        JMP     ADM13                   ; ADC Mem13 ISR
        JMP     ADM14                   ; ADC Mem14 ISR

ADM15                                   ; Add ADC12MEM12 ISR code Here
        MOV     &ADC12MEM15, R5
        JMP     ADC12ISR
        //THIS INTERRUPT FIRES WHEN PRESSURE CONVERSION IS DONE
ADM0                                    ; Add ADC12MEM0 ISR code Here
        MOV     &ADC12MEM0,R5           ; Move A0 result
        BIC.B   #20h, &P4OUT            ; Turn Off MUX
        CALL    #ENABLE_TBCCRIRQ
        //Downsample to 8 bits
        RRA     R5
        RRA     R5
        RRA     R5
        RRA     R5
        MOV.B   PressurePtr, R6         ;
        MOV.B   R5, Pressure(R6)
        INC     R6
        CMP.B   #4, R6
        JNZ     DonePressure
        MOV     #0, R6
DonePressure
        MOV     R6, PressurePtr
        CMP.B   #0F0h, R5
        JGE     Lower
        BIC.B   #01h, &P4OUT
        JMP     EndPressure
Lower
        BIS.B   #01, &P4OUT
EndPressure
        RETI                            ; Return from interrupt

        //THIS INTERRUPT FIRES WHEN LIGHT CONVERSION IS DONE
ADM1                                    ; Add ADC12MEM1 ISR code Here
        MOV     &ADC12MEM1, R5
        CALL    #ENABLE_TBCCRIRQ
        RRA     R5
        RRA     R5
        RRA     R5
        RRA     R5
        MOV     LightPtr, R6
        MOV.B   R5, Light(R6)
        MOV.B   R5, LightValue
        INC     LightPtr
        CMP.B   #255, LightPtr
        JNZ     DoneLight
        MOV     #0, LightPtr
DoneLight
        RETI                            ; Return from interrupt

        //THIS INTERRUPT FIRES WHEN TEMPERATURE CONVERSION IS DONE
ADM2                                    ; Add ADC12MEM2 ISR code Here
        MOV     &ADC12MEM2, R5
        CALL    #ENABLE_TBCCRIRQ
        RRA     R5
        RRA     R5
        RRA     R5
        RRA     R5
        MOV     TempPtr, R6
        MOV.B   R5, Temp(R6)
        MOV.B   R5, TempValue
        INC     TempPtr
        CMP     #8, TempPtr
        JNZ     DoneTemp
        MOV     #0, TempPtr
DoneTemp
        RETI                            ; Return from interrupt

        //THIS INTERRUPT FIRES WHEN SOUND CONVERSION IS DONE
ADM3                                    ; Add ADC12MEM3 ISR code Here
        MOV     SoundPtr, R6
        MOV     &ADC12MEM3, Sound(R6)   ; Save byte
        CALL    #ENABLE_TBCCRIRQ
        INCD    SoundPtr
        CMP     #1024, SoundPtr
        JNZ     DoneSound
        MOV     #0, SoundPtr
DoneSound
        RETI                            ; Return from interrupt
ADM4                                    ; Add ADC12MEM4 ISR code Here
        MOV     &ADC12MEM4, R5
        RETI                            ; Return from interrupt
ADM5                                    ; Add ADC12MEM5 ISR code Here
        MOV     &ADC12MEM5, R5
        RETI
ADM6                                    ; Add ADC12MEM6 ISR code Here
        MOV     &ADC12MEM6, R5
        RETI
ADM7                                    ; Add ADC12MEM7 ISR code Here
        MOV     &ADC12MEM6, R5
        MOV     R5, XStrain             ; Save Result
        MOV     &ADC12MEM7, R5
        CALL    #ENABLE_TBCCRIRQ
        MOV     R5, YStrain
        BIC.B   #40h, &P4OUT            ; Turn OFF gauges
        BIC     #ENC,&ADC12CTL0         ; Turn off to configure
        BIC     #MSC, &ADC12CTL0        ; Turn off multiple conversions
        BIS     #ENC,&ADC12CTL0         ; Enable ADC12
        RETI                            ; Return from interrupt
ADM8                                    ; Add ADC12MEM8 ISR code Here
        RETI
ADM9                                    ; Add ADC12MEM9 ISR code Here
        RETI
ADM10                                   ; Add ADC12MEM10 ISR code Here
        RETI
ADM11                                   ; Add ADC12MEM11 ISR code Here
        RETI
ADM12                                   ; Add ADC12MEM12 ISR code Here
        RETI
ADM13                                   ; Add ADC12MEM13 ISR code Here
        RETI
ADM14                                   ; Add ADC12MEM14 ISR code Here
        RETI
```

```
ADOV                                    ; Add ADC12 Overflow ISR code Here
        RETI
ADTOV                                   ; Add ADC12 Timer Overflow ISR code Here
        RETI

;-----------------------------------------------------------------------------
TB0_ISR;    Toggle P1.0
;-----------------------------------------------------------------------------
        ADD.W   #200,&TBCCR0            ; Offset until next interrupt
        RETI                           ;

;-----------------------------------------------------------------------------
TBX_ISR;    Common ISR for TBCCR1-6 and overflow
;-----------------------------------------------------------------------------
        ADD.W   &TBIV,PC                ; Add Timer_B offset vector
        RETI                            ; CCR0 - no source
        JMP     TBCCR1_ISR              ; TBCCR1
        JMP     TBCCR2_ISR              ; TBCCR2
        JMP     TBCCR3_ISR              ; TBCCR3
        JMP     TBCCR4_ISR              ; TBCCR4
        JMP     TBCCR5_ISR              ; TBCCR5
        JMP     TBCCR6_ISR              ; TBCCR6
TB_over
        RETI                            ; Return from overflow ISR

TBCCR1_ISR                              ; Generates frequency to sample audio
        BIT     #ADC12BUSY, &ADC12CTL1  ; Wait for ADC to finish conversions
        JC      TBCCR1_ISR
        BIC     #ENC,&ADC12CTL0         ; Turn off to configure
        MOV     #CSTARTADD_3+SHP, &ADC12CTL1
        BIS     #ENC,&ADC12CTL0
        CALL    #DISABLE_TBCCRIRQ
        BIS     #ADC12SC,&ADC12CTL0     ; Start conversions
        ADD.W   #500,&TBCCR1            ; Offset until next interrupt (Sample sound @ 8 KHz)
        //JMP     TBX_ISR                 ;
        RETI

TBCCR2_ISR                              ; Generates Frequency to sample light
        BIT     #ADC12BUSY, &ADC12CTL1
        JC      TBCCR2_ISR
        BIC     #ENC,&ADC12CTL0         ; Turn off to configure
        MOV     #CSTARTADD_1+SHP, &ADC12CTL1; Specify channel to sample (light)
        BIS     #ENC,&ADC12CTL0
        CALL    #DISABLE_TBCCRIRQ
        BIS     #ADC12SC,&ADC12CTL0     ; Start conversions
        ADD.W   #20001,&TBCCR2          ; Offset until next interrupt (Sample light @ 200Hz)
//        JMP     TBX_ISR                 ;
        RETI

TBCCR3_ISR                              ; Generates samp. freq. for pressure channels and whiskers
        BIT     #ADC12BUSY, &ADC12CTL1
        JC      TBCCR3_ISR
        MOV.B   PressurePtr, R6         ; get pointer
        RLA.B   R6                      ; shift to set mux address
        RLA.B   R6                      ; shift to set mux address
        RLA.B   R6                      ; '
        BIS.B   #20h, R6                ; Turn on MUX
        CLR     R7
        XOR.B   &P4OUT, R7
        BIC.B   #0F8h, R7
        XOR.B   R7, R6
        MOV.B   R6, &P4OUT
        BIC     #ENC,&ADC12CTL0         ; Turn off to configure
        MOV     #SHP, &ADC12CTL1        ; Specify channel to sample (pressure)
        BIS     #ENC,&ADC12CTL0
        CALL    #DISABLE_TBCCRIRQ
        BIS     #ADC12SC,&ADC12CTL0     ; Start conversions
        //SAMPLE WHISKERS
        BIT.B   #10h, &P6IN             ; Whiskers generate square pulses
        JC      Whisker1On              ; in their respective input pin
        //BIC.B #01h, &P4OUT            ; when they are touched.
        JMP     Whisker2                ; An associated LED will flicker
Whisker1On                              ; at the same frequency that its
        BIS.B   #01h, &P4OUT            ; associated whisker
Whisker2
        BIT.B   #20h, &P6IN
        JC      Whisker2On
        //BIC.B #04, &P4OUT
        JMP     EndTBCCR3
Whisker2On
        BIS.B   #04, &P4OUT
EndTBCCR3
        ADD.W   #33333,&TBCCR3          ; Offset until next interrupt (Sample
//        JMP     TBX_ISR                 ; pressure and whiskers @ 120Hz)
        RETI

TBCCR4_ISR
        BIT     #ADC12BUSY, &ADC12CTL1
        JC      TBCCR4_ISR
        BIC     #ENC,&ADC12CTL0         ; Turn off to configure
        MOV     #CSTARTADD_2+SHP, &ADC12CTL1; Specify channel to sample (temp)
        BIS     #ENC,&ADC12CTL0
        CALL    #DISABLE_TBCCRIRQ
        BIS     #ADC12SC,&ADC12CTL0     ; Start conversions
        ADD.W   #50000,&TBCCR4          ; Offset until next interrupt (Sample temp @ 80Hz)
//        JMP     TBX_ISR                 ;
        RETI

TBCCR5_ISR
        BIT     #ADC12BUSY, &ADC12CTL1
        JC      TBCCR5_ISR
        BIC     #ENC,&ADC12CTL0         ; Turn off to configure
        MOV     #CSTARTADD_6+SHP+CONSEQ_1, &ADC12CTL1; Specify channel to sample (temp)
        BIS     #MSC, &ADC12CTL0        ; Multiple conversions
        BIS     #ENC,&ADC12CTL0         ; Enable ADC12
        BIS.B   #40h, &P4OUT            ; Turn on gauges
        CALL    #DISABLE_TBCCRIRQ
        BIS     #ADC12SC,&ADC12CTL0     ; Start conversions
        ADD.W   #0FFFFh,&TBCCR5         ; Offset until next interrupt (Sample SG's @60Hz )
//        JMP     TBX_ISR                 ;
        RETI

TBCCR6_ISR
        ADD.W   #10000,&TBCCR6          ; Offset until next interrupt
        //JMP     TBX_ISR                 ;
        RETI
```

```
;-----------------------------------------------------------------------------
I2C_ISR;    Common ISR for I2C Module
;-----------------------------------------------------------------------------
        ADD     &I2CIV,PC               ; Add I2C offset vector
        RETI                            ; No interrupt
        RETI                            ; Arbitration Lost
        RETI                            ; No Acknowledge
        RETI                            ; Own Address
        RETI                            ; Register Access Ready
        JMP     RXRDY_ISR               ; Receive Ready
        JMP     TXRDY_ISR               ; Transmit Ready
        JMP     GC_ISR                  ; General Call
        RETI                            ; Start Condition

RXRDY_ISR
        MOV.B   &I2CDRB, R6             ; RX data in R6
        RETI

TXRDY_ISR
        MOV     PacketPtr, R6
        MOV.B   Packet(R6), &I2CDRB
        INC     PacketPtr
        BIC.B   #0F0h, PacketPtr
        RETI

GC_ISR
        BIC.B   #GCIFG+RXRDYIFG, &I2CIFG
        MOV     #0, &WDTCTL             ; Yup, reset

EndI2CISR
        RETI

;-----------------------------------------------------------------------------
;           Interrupt Vectors Used MSP430x13x/14x/15x/16x
;-----------------------------------------------------------------------------
        ORG     0FFFEh                  ; MSP430 RESET Vector
        DW      RESET                   ;
        ORG     0FFEEh                  ; ADC12 Interrupt Vector
        DW      ADC12ISR                ;
        ORG     0FFFAh                  ; Timer_B0 Vector
        DW      TB0_ISR                 ;
        ORG     0FFF8h                  ; Timer_BX Vector
        DW      TBX_ISR                 ;
        ORG     0FFF0h                  ; I2C interruplt vector
        DW      I2C_ISR                 ;

END
```

```
;******************************************************************************
;   MSP-FET430P140 Demo - Software Toggle P1.0
;
;   Description; Toggle P1.0 by xor'ing P0.1
;   ACLK = n/a, MCLK = SMCLK = default DCO ~ 800k
;
;                MSP430F149
;             -----------------
;         /|\|              XIN|-
;          | |                 |
;          --|RST          XOUT|-
;            |                 |
;            |             P1.0|-->LED
;
;   M.Buccini
;   Texas Instruments, Inc
;   January 2004
;******************************************************************************
#include  "msp430x16x.h"

#define REFLASH_ALL_INST        (0xA0)
#define RESET_BRAIN_INST        (0xA2)
#define RESET_SKIN_INST         (0xA4)
#define EXECUTE_CODE_INST       (0xA6)
#define SAMPLE_INST             (0xA8)
#define LIGHT_MAX               (0x0100)
#define I2C_OWN_ADDRESS         (269)

;_______________________________________________________________________
; Variables
;_______________________________________________________________________
;RAM VARIABLES
;------------------------------------------------------------------------------
        ORG 1140h
;------------------------------------------------------------------------------
Light           DS   16
Temp            DS   20
LightPtr        DS   1
Flags           DS   1
BootLoaderAdd   DS   1

;------------------------------------------------------------------------------
        ORG     08000h                  ; Progam Start
;------------------------------------------------------------------------------
RESET
        MOV.W   #1120h,SP               ; Initialize 'x1x9 stackpointer

        //STOP WATCHDOG
        MOV.W   #WDTPW+WDTHOLD, &WDTCTL ; stop watchdog

        //INITIALIZE SVS
        MOV.B   #060h+PORON,&SVSCTL     ; SVS POR enabled @ 2.5V

        //INITIALIZE CRYSTAL (8.000 MHZ)
        BIC     #OSCOFF,SR              ; Turn on osc.
        BIS.B   #XTS, &BCSCTL1          ; LFXTL high-freq mode
L1                                      ; wait for crystal to stabilize
        BIC.B   #OFIFG,&IFG1            ; Clear OFIFG
```

```
        MOV     #0FFh,R15               ; Delay
L2
        DEC R15
        JNZ L2
        BIT.B   #OFIFG,&IFG1            ; Re-test OFIFG
        JNZ L1                          ; Repeat test if needed
        BIS.B   #SELM_3,&BCSCTL2        ; Select XT1 as source for MCLK
        BIS.B   #SELS, &BCSCTL2         ; select XT1CLK as source for SMCLK

        //INITIALIZE PORTS
        BIS.B   #7FH, &P4DIR            ; Set as outputs
        BIS.B   #02H, &P4SEL            ; Set as timer outputs for PWM
        BIS.B   #07h, &P6SEL            ; Enable A/D channel inputs
        MOV.B   #0A7h, &P4OUT

        //INITIALIZE I2C
        BIS.B   #0Ah,&P3SEL             ; Select I2C pins
        BIS.B   #I2C+SYNC+XA,&U0CTL     ; Recommended init procedure
        BIC.B   #I2CEN,&U0CTL           ; Recommended init procedure
        MOV.B   #I2CSSEL0,&I2CTCTL      ; ACLK
        MOV.B   #07h, I2CSCLH           ; Set Baud Rate to 400Khz
        MOV.B   #07h, I2CSCLL           ; Set Baud Rate to 400Khz
        BIS.B   #GCIE+TXRDYIE, &I2CIE   ; Activate I2C receive Interrupt
        MOV     #1080h, R5              ; Get I2C address from flash (from BL)
        MOV     0(R5), &I2COA           ; Slave Address is 048h
        BIS.B   #I2CEN,&U0CTL           ; Enable I2C

        //INITIALIZE ADC
        MOV     #ADC12ON+SHT0_3+REFON,&ADC12CTL0  ; Turn on ADC12, set MSC
        MOV     #SHP,&ADC12CTL1         ; Use samp. timer, single sequence
        BIS.B   #INCH_1,&ADC12MCTL0     ; AVcc=ref+, channel=A0
        MOV     #BIT0,&ADC12IE          ; Enable ADC12IFG.0 for ADC12MEM0
        BIS     #ENC,&ADC12CTL0         ; Enable conversions

        //INITIALIZE TIMER B
        MOV   #LIGHT_MAX,&TBCCR0        ; PWM Period
        MOV   #OUTMOD_7,&TBCCTL1        ; CCR1 reset/set
        MOV   #TBSSEL_1+MC_1,&TBCTL     ; SMCLK, upmode        EINT
        EINT

        MOV     #0, LightPtr
        CLR     R7
MainLoop
        BIS     #ADC12SC,&ADC12CTL0     ; Start conversions
        MOV.W   #20, R10                ; Delay
        CALL    #DELAY
        MOV     R5, &TBCCR1
        JMP     MainLoop

;_______________________________________________________________________
; Delay function (3n^2 + 4n + delay-9)
;_______________________________________________________________________
DELAY
        PUSH R8
        PUSH R9
        MOV R10, R9
LoopDelay1
        MOV R10, R8
```

```
LoopDelay2
        DEC R8
        JNZ LoopDelay2
        DEC R9
        JNZ LoopDelay1
        POP R9
        POP R8
        RET

;-----------------------------------------------------------------------------
ADC12ISR        ; Interrupt Service Routine for ADC12
;-----------------------------------------------------------------------------
        MOV     &ADC12MEM0,R5           ; Move A0 result
        MOV     LightPtr, R6
        MOV     R5, Light(R6)
        INCD    LightPtr
        BIC.B   #0F0h, LightPtr
        CMP.W   #LIGHT_MAX,R5           ; Set To maximum value to
        JNC     EndISR                  ; avoid timer overflow
        MOV.W   #LIGHT_MAX, R5          ; set to max.
EndISR
        RETI                            ; Return from interrupt

;-----------------------------------------------------------------------------
I2C_ISR;    Common ISR for I2C Module
;-----------------------------------------------------------------------------
        ADD     &I2CIV,PC               ; Add I2C offset vector
        RETI                            ; No interrupt
        RETI                            ; Arbitration Lost
        RETI                            ; No Acknowledge
        RETI                            ; Own Address
        RETI                            ; Register Access Ready
        JMP     RXRDY_ISR               ; Receive Ready
        JMP     TXRDY_ISR               ; Transmit Ready
        JMP     GC_ISR                  ; General Call
        RETI                            ; Start Condition

RXRDY_ISR
        MOV.B   &I2CDRB, R6             ; RX data in R6
        MOV.B   Light(R6), &I2CDRB
        RETI

TXRDY_ISR
        MOV     LightPtr, R6
        MOV.B   Light(R6), &I2CDRB
        BIC.B   #TXRDYIFG, &I2CIFG
        MOV.B   #0FFh, R5
WaitTX
        DEC.B   R5
        JZ      EndI2CISR
        BIT.B   #TXRDYIFG, &I2CIFG
        JNC     WaitTX
        BIC.B   #TXRDYIFG, &I2CIFG
        INC     R6
        MOV.B   Light(R6), &I2CDRB
        //XOR.B #01, &P4OUT
        RETI
```

```
GC_ISR
        MOV.B   &I2CDRB, R6             ; RX data in R6
        CMP.B   #0A4h, R6
        JNZ     EndI2CISR
        MOV     #2500, R5
        MOV     #0, 0(R5)
        BIC.B   #GCIFG+RXRDYIFG, &I2CIFG
        MOV     #0, &WDTCTL             ; Yup, reset

EndI2CISR
        RETI

;-----------------------------------------------------------------------------
;           Interrupt Vectors Used MSP430x13x/14x/15x/16x
;-----------------------------------------------------------------------------
        ORG     0FFFEh                  ; MSP430 RESET Vector
        DW      RESET                   ;
        ORG     0FFEEh                  ; ADC12 Interrupt Vector
        DW      ADC12ISR                ;
        ORG     0FFF0h                  ; I2C interruplt vector
        DW      I2C_ISR                 ;

        END
```

```
;******************************************************************************
;   MSP-FET430P140 Demo - Software Toggle P1.0
;
;   Description; Toggle P1.0 by xor'ing P0.1
;   ACLK = n/a, MCLK = SMCLK = default DCO ~ 800k
;
;                MSP430F149
;             -----------------
;         /|\|              XIN|-
;          | |                 |
;          --|RST          XOUT|-
;            |                 |
;            |             P1.0|-->LED
;
;   M.Buccini
;   Texas Instruments, Inc
;   January 2004
;******************************************************************************
#include  "msp430x16x.h"
#define REFLASH_ALL_INST        (0xA0)
#define RESET_BRAIN_INST        (0xA2)
#define RESET_SKIN_INST         (0xA4)
#define EXECUTE_CODE_INST       (0xA6)

;------------------------------------------------------------------------------
            ORG     01140h                  ; Progam Start
;------------------------------------------------------------------------------
flag    DS 1

;VALUES
LIGHT_MAX   EQU 0800h

;------------------------------------------------------------------------------
            ORG     08000h                  ; Progam Start
;------------------------------------------------------------------------------
RESET
        MOV.W   #1120h,SP               ; Initialize 'x1x9 stackpointer

        //STOP WATCHDOG
        MOV.W   #WDTPW+WDTHOLD, &WDTCTL ; stop watchdog

        //INITIALIZE SVS
        MOV.B   #060h+PORON,&SVSCTL     ; SVS POR enabled @ 2.5V

        //SET MAX DCO FREQUENCY
        BIS.B #RSEL2+RSEL1+RSEL0,&BCSCTL1 ;
        BIS.B #DCO2+DCO1+DCO0,&DCOCTL ; Set max DCO frequency

        //INITIALIZE CRYSTAL (8.000 MHZ)
        BIC     #OSCOFF,SR              ; Turn on osc.
        BIS.B   #XTS, &BCSCTL1          ; LFXTL high-freq mode
L1                                      ; wait for crystal to stabilize
        BIC.B   #OFIFG,&IFG1            ; Clear OFIFG
        MOV     #0FFh,R15               ; Delay
L2
        DEC R15
        JNZ L2
        BIT.B   #OFIFG,&IFG1            ; Re-test OFIFG
```

```
        JNZ L1                          ; Repeat test if needed
        BIS.B   #SELM_3,&BCSCTL2        ; Select XT1 as source for MCLK
        BIS.B   #SELS, &BCSCTL2         ; select XT1CLK as source for SMCLK

        //INITIALIZE PORTS
        BIS.B   #7FH, &P4DIR            ; Set as outputs
        BIS.B   #07h, &P6SEL            ; Enable A/D channel inputs
        MOV.B   #0B3h, &P4OUT

        //INITIALIZE I2C
        BIS.B   #0Ah,&P3SEL             ; Select I2C pins
        BIS.B   #I2C+SYNC+XA,&U0CTL     ; Recommended init procedure
        BIC.B   #I2CEN,&U0CTL           ; Recommended init procedure
        BIS.B   #I2CSSEL0+I2CRM,&I2CTCTL     ; ACLK
        MOV.B   #07h, I2CSCLH           ; Set Baud Rate to 400Khz
        MOV.B   #07h, I2CSCLL           ; Set Baud Rate to 400Khz
        BIS.B   #GCIE, &I2CIE           ; Activate I2C receive Interrupt
        BIS.B   #I2CEN,&U0CTL           ; Enable I2C
        EINT

        MOV.B   #0, flag

MainLoop
        XOR.B   #05h, &P4OUT
        MOV.W   #450h, R10                ; Delay
        CALL    #DELAY
        JMP     MainLoop

;_______________________________________________________________________
; Delay function (3n^2 + 4n + delay-9)
;_______________________________________________________________________
DELAY
        PUSH R8
        PUSH R9
        MOV R10, R9
LoopDelay1
        MOV R10, R8
LoopDelay2
        DEC R8
        JNZ LoopDelay2
        DEC R9
        JNZ LoopDelay1
        POP R9
        POP R8
        RET


;------------------------------------------------------------------------------
I2C_ISR;    Common ISR for I2C Module
;------------------------------------------------------------------------------
        ADD     &I2CIV,PC               ; Add I2C offset vector
        RETI                            ; No interrupt
        RETI                            ; Arbitration Lost
        RETI                            ; No Acknowledge
        RETI                            ; Own Address
        RETI                            ; Register Access Ready
        JMP     RXRDY_ISR               ; Receive Ready
        RETI                            ; Transmit Ready
        JMP     GC_ISR                  ; General Call
        RETI                            ; Start Condition

RXRDY_ISR
        BIC.B   #RXRDYIFG, &I2CIFG
        RETI
GC_ISR
        MOV.B   &I2CDRB,R5              ; RX data in R5
        BIC.B   #GCIFG+RXRDYIFG, &I2CIFG
        CMP.B   #RESET_SKIN_INST, R5    ; Is it a reset?
        JNZ     EndI2CISR               ; No
        MOV     #0, &WDTCTL             ; Yup, reset

EndI2CISR
        RETI

;-----------------------------------------------------------------------------
;           Interrupt Vectors Used MSP430x13x/14x/15x/16x
;-----------------------------------------------------------------------------
        ORG     0FFFEh                  ; MSP430 RESET Vector
        DW      RESET                   ;
        ORG     0FFF0h                  ; I2C interruplt vector
        DW      I2C_ISR                 ;

END
```

```
#include  "msp430x16x.h"
#define RD_                     (0x0001)
#define WR                      (0x0002)
#define TXE_                    (0x0004)
#define RXF_                    (0x0008)
#define SI_WU                   (0x0010)
#define PWREN_                  (0x0020)
#define MAX_CODE_SIZE           (0x1000)
#define REFLASH_INST            (0xA0)
#define RESET_BRAIN_INST        (0xA2)
#define RESET_SKIN_INST         (0xA4)
#define EXECUTE_CODE_INST       (0xA6)
#define REFLASH_DEV_INST        (0xA8)
#define RESET_DEV_INST          (0xAA)
#define START_SAMPLING_INST     (0xAC)
#define STOP_SAMPLING_INST      (0xAE)
#define I2C_BRAIN_ADDRESS       (0x0100)
#define DATA_BYTES              (2)
;________________________________________________________________________
; Variables
;________________________________________________________________________
;RAM VARIABLES
      ORG 1140h
USBBuf          DS MAX_CODE_SIZE
USBBufPtr       DS 2
nBytes          DS 2
Address         DS 2
DataPacket      DS 18
fileFlag        DS 1
fileStatus      DS 1
I2CFlags        DS 1

;-------------------------------------------------------------------------------
            ORG     08000h                  ; Progam Start
;-------------------------------------------------------------------------------
RESET
        MOV.W   #1130h,SP               ; Initialize 'x1x9 stackpointer

        //STOP WATCHDOG
        MOV.W   #WDTPW+WDTHOLD, &WDTCTL ; stop watchdog

        //INITIALIZE SVS
        MOV.B   #060h+PORON,&SVSCTL     ; SVS POR enabled @ 2.6V

        //INITIALIZE CRYSTAL (8.000 MHZ)
        BIC     #OSCOFF,SR              ; Turn on osc.
        BIS.B   #XTS, &BCSCTL1          ; LFXTL high-freq mode
L1                                      ; wait for crystal to stabilize
        BIC.B   #OFIFG,&IFG1            ; Clear OFIFG
        MOV     #0FFh,R15               ; Delay
L2
        DEC R15
        JNZ L2
        BIT.B   #OFIFG,&IFG1            ; Re-test OFIFG
        JNZ L1                          ; Repeat test if needed
        BIS.B #SELM0+SELM1,&BCSCTL2     ; Select XT1 as source for MCLK
        BIS.B #SELS, &BCSCTL2           ; select XT1CLK as source for SMCLK
```

```
        //INITIALIZE PORTS
        BIS.B   #0FFh, &P2OUT           ; Set all pins to 1
        BIS.B   #RD_+WR+SI_WU, &P2DIR   ; Set P2 directions
        BIS.B   #08h, &P2IES            ; Interrupt generated with negative transition1
        BIS.B   #08h, &P2IE             ; Activate RD interrupt
        MOV.B   #0FEh, &P4OUT           ; Turn off LED's
        BIS.B   #07h, &P4DIR            ; Set as outputs

        //INITIALIZE I2C PORT
        BIS.B   #70h, P5SEL
        BIS.B   #0Ah,&P3SEL             ; Select I2C pins
        BIS.B   #I2C+SYNC,&U0CTL        ; Recommended init procedure
        BIC.B   #I2CEN,&U0CTL           ; Recommended init procedure
        BIS.B   #XA,&U0CTL              ; Recommended init procedure
        BIS.B   #I2CSSEL0,&I2CTCTL      ; ACLK
        MOV.B   #7, I2CSCLH             ; Set Baud Rate to 400Khz
        MOV.B   #7, I2CSCLL             ; Set Baud Rate to 400Khz
        BIS.B   #I2CEN,&U0CTL           ; Enable I2C

        //INITIALIZE VARIABLES
        MOV.B   #0, I2CFlags
        MOV.W   #0, USBBufPtr
        MOV.B   #0, fileFlag
        MOV.W   #130, R10
        CALL    #DELAY
        MOV.B   #0FFh, &P4OUT
        //CALL  #I2C_RESET
        EINT

MainLoop                                ; Wait here until a code file is transfered
        BIT.B   #80h, I2CFlags          ; Is sampling on?
        JNC     MainLoop                ; Nope, don't do anything

        BIC.B   #I2CEN,&U0CTL           ; Reconfigure I2C
        BIS.B   #XA,&U0CTL              ; Reconfigure I2C
        BIC.B   #I2CRM,&I2CTCTL         ; deactivate Repeat Mode
        MOV.B   #DATA_BYTES, &I2CNDAT   ; Read n bytes
        BIC.B   #I2CTRX, I2CTCTL        ; Read bytes
        BIS.B   #I2CEN,&U0CTL           ; Recommended init procedure
        BIS.B   #MST,&U0CTL             ; Master mode
        //MOV   #272, Address           ; Last node in network

        //Read sensor info from node X
ReadNode
        MOV     Address, &I2CSA
        CLR     R6
        MOV     #DATA_BYTES, nBytes
        BIS.B   #I2CSTT+I2CSTP,&I2CTCTL ; Initiate transfer
ReadByte
        MOV     #0200h, R10             ; Timeout for reception
        CALL    #I2C_GET_BYTE           ; Get byte
        CALL    #USB_SEND               ; Send it through USB
        //MOV.B R5, DataPacket(R6)      ; Save byte
        INC     R6
        DEC.B   nBytes
        JNZ     ReadByte
        //MOV   #400, R10
        //CALL  #DELAY
        BIC.B   #80h, I2CFlags
        JMP     MainLoop

SampleNextNode
        DEC     Address
        CMP     #257, Address

        JMP     MainLoop                ; not yet
;________________________________________________________________________
REFLASH ; Updates firmware in all nodes
;________________________________________________________________________
        BIT.B   #04, fileFlag           ; Are we reflashing one or all nodes?
        JC      SingleDevice            ; If not single device, broadcast code
        BIC.B   #I2CEN,&U0CTL           ; Reconfigure I2C
        BIC.B   #XA,&U0CTL              ; Reconfigure I2C
        BIS.B   #I2CRM,&I2CTCTL         ; Activate Repeat Mode
        BIS.B   #I2CEN,&U0CTL           ; Recommended init procedure
        MOV     #0,&I2CSA               ; Reflash all nodes
        CLR     R6
        JMP     InitI2CTX

SingleDevice
        BIC.B   #I2CEN,&U0CTL           ; Reconfigure I2C
        BIS.B   #XA,&U0CTL              ; Reconfigure I2C
        BIS.B   #I2CRM,&I2CTCTL         ; Activate Repeat Mode
        BIS.B   #I2CEN,&U0CTL           ; Recommended init procedure
        SUB     #2, nBytes              ; Decrease nBytes
        CLR     R6
        MOV     USBBuf(R6), R5          ; Read Address
        MOV     R5,&I2CSA               ; Broadcast Address
        MOV.B   #2, R6

InitI2CTX
        MOV     nBytes, R7              ; Get number of bytes to transfer

        //Indicate a reflash to BootLoader
        BIS.B   #MST,&U0CTL             ; Master mode
        BIS.B   #I2CSTT+I2CTRX,&I2CTCTL ; Initiate transfer
        MOV.B   #REFLASH_INST, R5       ; Now indicate a reflash instruction
        CALL    #I2C_SEND_BYTE          ; Send instruction
        //Send code size to indicate lenght of transmission
        MOV     nBytes, R5              ; Get code file size from memory
        CALL    #I2C_SEND_BYTE          ; Send low end of nBytes
        SWPB    R5                      ; Send High end of nBytes
        CALL    #I2C_SEND_BYTE          ; Send low end of nBytes

        //Send File!
SendLoop
        MOV.B   USBBuf(R6), R5          ; Get byte from memory
        CALL    #I2C_SEND_BYTE          ; Send byte
        INC     R6                      ; Increment pointer
        MOV     #0FFh, R5
WaitClk
        DEC     R5
        JZ      Stp
        BIT.B   #I2CSCLLOW, &I2CDCTL    ; Is slave holding clock line low?
        JC      WaitClk                 ; Yup, wait
        DEC     R7                      ; Decrease bytes to send
```

```
        JNZ     SendLoop                ; Are we done?
Stp
        MOV     #0FFh, R5
        BIS.B   #I2CSTP,&I2CTCTL        ; Issue STOP
ChkSTP1
        DEC     R5
        JZ      EndReFlash
        BIT.B   #I2CSTP,&I2CTCTL        ; Check STP condition
        JNZ     ChkSTP1

        BIC.B   #02, fileFlag           ; File Processed!
        BIC.B   #02, &P4OUT
        MOV     #500h, R10
        CALL    #DELAY
        BIS.B   #02, &P4OUT
        MOV     #0, I2CFlags
        MOV     #0, fileFlag
        MOV     #0, fileStatus

EndReFlash
        RET

;________________________________________________________________________
I2C_SEND_BYTE ; Sends a byte through I2C (byte in R5)
;________________________________________________________________________
        MOV.B   #0FFh, R10              ; Send timeout
        MOV.B   R5, &I2CDRB             ; Send byte
WaitTX
        DEC     R10                     ; Nope
        JZ      EndI2C_Send             ; Enough tries?
        BIT.B   #TXRDYIFG, &I2CIFG      ; Done with transmission?
        JNC     WaitTX                  ; Nope wait
EndI2C_Send
        BIC.B   #TXRDYIFG, &I2CIFG      ; Clear flag
        RET

;________________________________________________________________________
I2C_GET_BYTE ; Sends a byte through I2C (byte in R5)
;________________________________________________________________________
        BIT.B   #RXRDYIFG, &I2CIFG      ; Receive ready?
        JC      GetByte                 ; Yep, get byte
        DEC     R10                     ; Nope
        JNZ     I2C_GET_BYTE            ; Enough tries?
        BIS.B   #01h, I2CFlags          ; Error receiving byte!
        RET
GetByte
        MOV.B   &I2CDRB, R5             ; Read I2C data
        BIC.B   #RXRDYIFG, &I2CIFG      ; Clear RX Flag
        BIC.B   #01h, I2CFlags          ; Did we get something?
        RET

;________________________________________________________________________
I2C_RESET ; Sends a byte through I2C (byte in R5)
;________________________________________________________________________
        BIC.B   #0Ah, &P3SEL            ; Deselect I2C pins
TestSDA
        BIS.B   #0Ah, &P3DIR            ; Set as outputs
        BIS.B   #02h, &P3OUT            ; Bring SDA back up (Assert a 1)
        BIC.B   #02h, &P3DIR            ; Set as input
        BIT.B   #02h, &P3IN             ; Read result
        JC      GenerateStop
        BIS.B   #08h, &P3OUT            ;
        BIC.B   #08h, &P3OUT            ;
        BIC.B   #08h, &P3OUT            ;
        JMP     TestSDA

GenerateStop
        BIS.B   #08h, &P3DIR            ; Set as outputs
        BIS.B   #08h, &P3OUT            ; Generate stop condition
        BIC.B   #0Ah, &P3SEL            ; Restore I2C pins
        RET

;____________________________________________________________________
USB_GET ; Reads data from FTDI Chip
;____________________________________________________________________
        BIC.B   #RXF_, &P2IFG           ; Clear IFG
        BIC.B   #0FFh, &P1DIR           ; Set USB data port as input
        BIC.B   #RD_, &P2OUT            ; Strobe RD line to read data
        MOV.B   &P1IN, R5               ; Read data
        BIS.B   #RD_, &P2OUT            ; Strobe RD line back
        RET

;____________________________________________________________________
USB_SEND ; Sends data to FTDI Chip
;____________________________________________________________________
        BIT.B   #TXE_, &P2IN            ; Is transfer buffer full?
        JC      USB_GET                 ; Yes, wait
        BIS.B   #0FFh, &P1DIR           ; Set USB data port as output
        BIS.B   #WR, &P2OUT             ; Strobe write line
        MOV.B   R5, &P1OUT              ; Echo data received
        BIC.B   #WR, &P2OUT             ; Strobe line down
        RET
;____________________________________________________________________
DELAY ; Delay function (3n^2 + 4n + delay-9)
;____________________________________________________________________
        PUSH R8
        PUSH R9
        MOV R10, R9
LoopDelay1
        MOV R10, R8
LoopDelay2
        DEC R8
        JNZ LoopDelay2
        DEC R9
        JNZ LoopDelay1
        POP R9
        POP R8
        RET

;-----------------------------------------------------------------------------
USB_TX_ISR     ;    USB Receive ISR
;-----------------------------------------------------------------------------
        //Receive USB byte
        CALL    #USB_GET
        //Echo byte to ack
        CALL    #USB_SEND
```

```
        //THIS PART DECODES INSTRUCTIONS RECEIVED FROM USB PORT
DecodeInstruction
        SUB.B   #0A0h, R5
        JN      End_USB_TX_ISR
        CMP.B   #0Fh, R5
        JGE     End_USB_TX_ISR
        ADD     R5, PC
        JMP     ReflashAll
        JMP     ResetBrain
        JMP     ResetSkin
        JMP     ExecuteSkinCode
        JMP     ReflashDev
        JMP     ResetDev
        JMP     StartSampling
        JMP     StopSampling
        //Other instructions go here

        //REFLASH_DEV / REFLASH_ALL INSTRUCTION
ReflashAll
        MOV.B   #01, fileFlag           ; Indicate that data is on the line
        JMP     StoreByte
ReflashDev
        MOV.B   #05, fileFlag           ; Indicate that data is on the line
StoreByte
        CALL    #USB_GET                ; Get byte
        CALL    #USB_SEND
        MOV     USBBufPtr, R6           ; Get pointer value
        MOV.B   R5, USBBuf(R6)          ; Store data
        INC     R6                      ; Increment pointer
        CMP     #MAX_CODE_SIZE, R6      ; Overflow?
        JNZ     StorePtr                ; No, store pointer value
        CLR     R6                      ; Overflowed pointer, reset
        BIS.B   #04h, fileFlag          ; Indicate file size overflow
StorePtr
        MOV     R6, USBBufPtr           ; Store pointer
        MOV     R6, nBytes              ; Store number of bytes
        BIT.B   #RXF_, &P2IN            ; Is there more data to read?
        JNC     StoreByte               ; yep
        XOR.B   #03,  fileFlag          ; File waiting...
        CLR     USBBufPtr               ; Clear pointer
        CALL    #REFLASH
        RETI

        //RESET BRAIN INSTRUCTION
ResetBrain
        MOV.W   #0, &WDTCTL             ; Write WDT with no password (RESET!)

        //RESET SKIN/NODE INSTRUCTION
ResetDev
ResetSkin
        BIC.B   #I2CEN,&U0CTL           ; Reconfigure I2C
        BIC.B   #XA,&U0CTL              ; Reconfigure I2C
        BIC.B   #I2CRM,&I2CTCTL         ; ACLK
        BIS.B   #I2CEN,&U0CTL           ; Recommended init procedure
        MOV     #0,&I2CSA               ; RESET ALL NODES!
        MOV.B   #1, &I2CNDAT
        BIS.B   #MST,&U0CTL             ; Master mode
        BIS.B   #I2CSTT+I2CSTP+I2CTRX,&I2CTCTL ; Initiate transfer
        MOV.B   #RESET_SKIN_INST, R5    ; Indicate a reflash instruction
        CALL    #I2C_SEND_BYTE          ; Send instruction
        RETI

        //EXECUTE SKIN CODE INSTRUCTION
ExecuteSkinCode
        BIC.B   #I2CEN,&U0CTL           ; Reconfigure I2C
        BIC.B   #XA,&U0CTL              ; Reconfigure I2C
        BIC.B   #I2CRM,&I2CTCTL         ; ACLK
        BIS.B   #I2CEN,&U0CTL           ; Recommended init procedure
        MOV     #0,&I2CSA               ; RESET ALL NODES!
        MOV.B   #1, &I2CNDAT
        BIS.B   #MST,&U0CTL             ; Master mode
        BIS.B   #I2CSTT+I2CSTP+I2CTRX,&I2CTCTL ; Initiate transfer
        MOV.B   #EXECUTE_CODE_INST, R5  ; Indicate a reflash instruction
        CALL    #I2C_SEND_BYTE          ; Send instruction
        RETI

StartSampling
        CLR     R6
        CALL    #USB_GET                ; Get byte
        CALL    #USB_SEND               ; echo
        MOV.B   R5, Address(R6)
        CALL    #USB_GET                ; Get byte
        CALL    #USB_SEND               ; echo
        INC     R6
        MOV.B   R5, Address(R6)
        BIS.B   #80h, I2CFlags
        RETI

StopSampling
        BIC.B   #80h, I2CFlags
        RETI

End_USB_TX_ISR
        RETI

;-----------------------------------------------------------------------------
I2C_ISR;    Common ISR for I2C Module
;-----------------------------------------------------------------------------
        ADD     &I2CIV,PC               ; Add I2C offset vector
        RETI                            ; No interrupt
        RETI                            ; Arbitration Lost
        RETI                            ; No Acknowledge
        RETI                            ; Own Address
        RETI                            ; Register Access Ready
        JMP     RXRDY_ISR               ; Receive Ready
        RETI                            ; Transmit Ready
        RETI                            ; General Call
        RETI                            ; Start Condition

RXRDY_ISR
        MOV.B   &I2CDRB,R5              ; RX data in R5
        bic.B   #LPM0,0(SP)             ; Clear LPM0
        RETI

;-----------------------------------------------------------------------------
```

```
;           Interrupt Vectors Used MSP430x13x/14x/15x/16x
;-----------------------------------------------------------------------------
        ORG     0FFFEh                  ; MSP430 RESET Vector
        DW      RESET                   ;
        ORG     0FFE2h                  ; P2 Interrupt vector
        DW      USB_TX_ISR              ;
        ORG     0FFF0h                  ; I2C interruplt vector
        DW      I2C_ISR                 ;

END
```

Appendix E

PC Code

```cpp
//----------------------------------------------------------------------------
//
// Copyright (C) Gerardo Barroeta, MIT Media Lab
//
//----------------------------------------------------------------------------


#pragma once
#include <gl/gl.h>
#include <gl/glu.h>

typedef struct vertex_t
{
    float x, y, z;
} vertex_t;

typedef struct vector_t
{
    float x, y, z;
} vector_t;

typedef struct polygon_s
{
    vertex_t v[3];
} polygon_t;

void normalize(vector_t *v);
void normal(vertex_t v[], vector_t *normal);

class COpenGLControl :
    public CWnd
{
public:
    /*****************/
    /* PUBLIC MEMBERS */
    /*****************/
    // Timer
    UINT_PTR m_unpTimer;
    COpenGLControl(void);
public:
    ~COpenGLControl(void);

private:
    /******************/
    /* PRIVATE MEMBERS */
    /******************/
    // Window information
    CWnd    *hWnd;
    HDC     hdc;
    HGLRC   hrc;
    int     m_nPixelFormat;
    CRect   m_rect;
    CRect   m_oldWindow;
    CRect   m_originalRect;
    float   m_fLastX;
    float   m_fLastY;
    float   m_fLastZ;
    float   m_fRotX;
    float   m_fRotY;
    float   m_fRotZ;
    float   m_fPosX;
    float   m_fPosY;
    float   m_fZoom;
    float   m_XMatrix;
    float   m_YMatrix;
    float   m_Height[100][100];

```

```cpp

public:
    void oglCreate(CRect rect, CWnd *parent);
    void oglInitialize(void);
    void oglDrawScene(void);
    void setHeight(int x, int y, float height);
    void setMatrixSize(int x, int y);

public:
    afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
    afx_msg void OnPaint();
    afx_msg void OnDraw(CDC *pDC);
    afx_msg void OnTimer(UINT_PTR nIDEvent);
    afx_msg void OnSize(UINT nType, int cx, int cy);
    afx_msg void OnMouseMove(UINT nFlags, CPoint point);

public:
    DECLARE_MESSAGE_MAP()
};

```

```
//{{NO_DEPENDENCIES}}
// Microsoft Visual C++ generated include file.
// Used by USBtest.rc
//
#define IDM_ABOUTBOX                    0x0010
#define IDD_ABOUTBOX                    100
#define IDS_ABOUTBOX                    101
#define IDD_USBTEST_DIALOG              102
#define IDR_MAINFRAME                   128
#define IDC_BUTTON_SENDFILE             1004
#define IDC_SLIDER_WRITE_RATE           1006
#define IDC_EDIT_PORT_SELECTED          1014
#define IDC_EDIT_PORT_STATUS            1015
#define IDC_LIST1                       1019
#define IDC_EDIT_NUM_RECD               1021
#define IDC_BUTTON_CLEAR                1023
#define IDC_BUTTON_OPEN                 1024
#define IDC_EDIT_ADDRESS                1025
#define IDC_EDIT_NAME_NUMBER            1026
#define IDC_RADIO_NAME_NUM              1027
#define IDC_RADIO_SERNUM                1028
#define IDC_RADIO_DEVNO                 1029
#define IDC_BUTTON_LIST                 1030
#define IDC_LIST2                       1032
#define IDC_BUTTON_RESET                1037
#define IDC_BUTTON_RUN_NODES            1038
#define IDC_BUTTON_RS                   1039
#define IDC_OPENGL                      1041
#define IDC_RADIO_SINGLE_NODE           1043
#define IDC_RADIO_ALL_NODES             1044
#define IDC_BUTTON_START                1046
#define IDC_BUTTON_STOP                 1047
#define IDC_EDIT1                       1061
#define IDC_EDITSPIN                    1061
#define IDC_SPIN                        1063

// Next default values for new objects
//
#ifdef APSTUDIO_INVOKED
#ifndef APSTUDIO_READONLY_SYMBOLS
#define _APS_NEXT_RESOURCE_VALUE        130
#define _APS_NEXT_COMMAND_VALUE         32771
#define _APS_NEXT_CONTROL_VALUE         1064
#define _APS_NEXT_SYMED_VALUE           101
#endif
#endif

```

```
// stdafx.h : include file for standard system include files,
//  or project specific include files that are used frequently, but
//      are changed infrequently
//

#if !defined(AFX_STDAFX_H__07909308_D75E_11D4_A644_93781B97CC41__INCLUDED_)
#define AFX_STDAFX_H__07909308_D75E_11D4_A644_93781B97CC41__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

#define VC_EXTRALEAN        // Exclude rarely-used stuff from Windows headers

#include <afxwin.h>         // MFC core and standard components
#include <afxext.h>         // MFC extensions
#include <afxdisp.h>        // MFC Automation classes
#include <afxdtctl.h>       // MFC support for Internet Explorer 4 Common Controls
#ifndef _AFX_NO_AFXCMN_SUPPORT
#include <afxcmn.h>         // MFC support for Windows Common Controls
#endif // _AFX_NO_AFXCMN_SUPPORT


//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert additional declarations immediately before the
    previous line.

#endif // !defined(AFX_STDAFX_H__07909308_D75E_11D4_A644_93781B97CC41__INCLUDED_)

```

```cpp
//---------------------------------------------------------------------------
//
// Copyright (C) Gerardo Barroeta, MIT Media Lab
//
//---------------------------------------------------------------------------


// USBtest.h : main header file for the USBTEST application
//

#if !defined(AFX_USBTEST_H__07909304_D75E_11D4_A644_93781B97CC41__INCLUDED_)
#define AFX_USBTEST_H__07909304_D75E_11D4_A644_93781B97CC41__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

#ifndef __AFXWIN_H__
    #error include 'stdafx.h' before including this file for PCH
#endif

#include "resource.h"       // main symbols

/////////////////////////////////////////////////////////////////////////////
// CUSBtestApp:
// See USBtest.cpp for the implementation of this class
//

class CUSBtestApp : public CWinApp
{
public:
    CUSBtestApp();

// Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CUSBtestApp)
    public:
    virtual BOOL InitInstance();
    //}}AFX_VIRTUAL

// Implementation

    //{{AFX_MSG(CUSBtestApp)
        // NOTE - the ClassWizard will add and remove member functions here.
        //    DO NOT EDIT what you see in these blocks of generated code !
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};


/////////////////////////////////////////////////////////////////////////////
//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert additional declarations immediately before the
    previous line.

#endif // !defined(AFX_USBTEST_H__07909304_D75E_11D4_A644_93781B97CC41__INCLUDED_)
```

```
//---------------------------------------------------------------------------
//
// Copyright (C) Gerardo Barroeta, MIT Media Lab
//
//---------------------------------------------------------------------------

// USBtestDlg.h : header file
//

#if !defined(AFX_USBTESTDLG_H__07909306_D75E_11D4_A644_93781B97CC41__INCLUDED_)
#define AFX_USBTESTDLG_H__07909306_D75E_11D4_A644_93781B97CC41__INCLUDED_
#define GUID_CLASS_USB_DEVICE      GUID_DEVINTERFACE_USB_DEVICE

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

#include "ftd2xx.h"
#include "OpenGLControl.h"
#include <windows.h>
#include <dbt.h>
#include <initguid.h>
#include "afxwin.h"
#include "afxcmn.h"

UINT ThreadProc1(LPVOID param);
const int ON = 100;
const int OFF = 101;
static GUID GUID_DEVINTERFACE_USB_DEVICE = { 0xA5DCBF10L, 0x6530, 0x11D2, { 0x90, 0x1F↙
    , 0x00, 0xC0, 0x4F, 0xB9, 0x51, 0xED } };

// CUSBtestDlg dialog
class CUSBtestDlg : public CDialog
{
// Construction
public:
    CUSBtestDlg(CWnd* pParent = NULL);  // standard constructor
    void CUSBtestDlg::Buttons(int onoff);
    FT_STATUS CUSBtestDlg::OpenBy();

// Dialog Data
    //{{AFX_DATA(CUSBtestDlg)
    enum { IDD = IDD_USBTEST_DIALOG };
    CListBox    m_Received;
    long    m_NumRecd;
    int     m_SerDesc;
    int     m_Nodes;
    CString m_128status;
    CString m_NameNmbr;
    CString m_Address;
    CString m_PortStatus;
    COpenGLControl m_oglWindow;
    CString m_EditSpin;
    CSpinButtonCtrl m_Spin;
    //}}AFX_DATA

    // ClassWizard generated virtual function overrides
public:
    afx_msg void OnLvnItemchangedList2(NMHDR *pNMHDR, LRESULT *pResult);
    afx_msg void OnClose();
    afx_msg void OnEnable(BOOL bEnable);
    afx_msg void OnVScroll(UINT nSBCode, UINT nPos, CScrollBar* pScrollBar);
    FT_STATUS Write(LPVOID, DWORD, LPDWORD);
    FT_STATUS Read(LPVOID, DWORD, LPDWORD);
    FT_STATUS GetQueueStatus(LPDWORD);
    FT_STATUS Purge(ULONG);

```

```
// Implementation, typedefs for DLL functions
protected:
    HICON m_hIcon;
    HMODULE m_hmodule;
    FT_HANDLE m_ftHandle;
    void LoadDLL();
    void RegisterNotification(HDEVNOTIFY *hDevNotify);
    void UnregisterNotification(HDEVNOTIFY *hDevNotify);

    typedef FT_STATUS (WINAPI *PtrToOpen)(PVOID, FT_HANDLE *);
    PtrToOpen m_pOpen;
    FT_STATUS Open(PVOID);

    typedef FT_STATUS (WINAPI *PtrToOpenEx)(PVOID, DWORD, FT_HANDLE *);
    PtrToOpenEx m_pOpenEx;
    FT_STATUS OpenEx(PVOID, DWORD);

    typedef FT_STATUS (WINAPI *PtrToListDevices)(PVOID, PVOID, DWORD);
    PtrToListDevices m_pListDevices;
    FT_STATUS ListDevices(PVOID, PVOID, DWORD);

    typedef FT_STATUS (WINAPI *PtrToClose)(FT_HANDLE);
    PtrToClose m_pClose;
    FT_STATUS Close();

    typedef FT_STATUS (WINAPI *PtrToRead)(FT_HANDLE, LPVOID, DWORD, LPDWORD);
    PtrToRead m_pRead;
    //FT_STATUS Read(LPVOID, DWORD, LPDWORD);

    typedef FT_STATUS (WINAPI *PtrToWrite)(FT_HANDLE, LPVOID, DWORD, LPDWORD);
    PtrToWrite m_pWrite;
    //FT_STATUS Write(LPVOID, DWORD, LPDWORD);

    typedef FT_STATUS (WINAPI *PtrToResetDevice)(FT_HANDLE);
    PtrToResetDevice m_pResetDevice;
    FT_STATUS ResetDevice();

    typedef FT_STATUS (WINAPI *PtrToPurge)(FT_HANDLE, ULONG);
    PtrToPurge m_pPurge;
    //FT_STATUS Purge(ULONG);

    typedef FT_STATUS (WINAPI *PtrToSetTimeouts)(FT_HANDLE, ULONG, ULONG);
    PtrToSetTimeouts m_pSetTimeouts;
    FT_STATUS SetTimeouts(ULONG, ULONG);

    typedef FT_STATUS (WINAPI *PtrToGetQueueStatus)(FT_HANDLE, LPDWORD);
    PtrToGetQueueStatus m_pGetQueueStatus;
    //FT_STATUS GetQueueStatus(LPDWORD);

    // Generated message map functions
    //{{AFX_MSG(CUSBtestDlg)
    virtual BOOL OnInitDialog();
    afx_msg void OnSysCommand(UINT nID, LPARAM lParam);
    afx_msg BOOL OnDeviceChange(UINT nEventType, DWORD_PTR dwData);
    afx_msg void OnPaint();
    afx_msg HCURSOR OnQueryDragIcon();
    virtual void OnOK();
    afx_msg void OnButtonSendFile();
    afx_msg void OnTimer(UINT nIDEvent);
    afx_msg void OnButtonClear();
    afx_msg void OnButtonOpen();
    afx_msg void OnRadioNameNum();
    afx_msg void OnRadioSernum();
    afx_msg void OnRadioDevno();
    afx_msg void OnRadioAllNodes();
    afx_msg void OnRadioSingleNode();
    afx_msg void OnChangeEditNameNumber();
    afx_msg void OnChangeEditAddress();
    afx_msg void OnButtonList();
    afx_msg void OnSelchangeList1();
    afx_msg void OnButtonReset();
    afx_msg void OnButtonRs();
    afx_msg void OnButtonRunNodes();
    afx_msg void OnButtonStart();
    afx_msg void OnButtonStop();
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()

    //{{AFX_VIRTUAL(CUSBtestDlg)
protected:
    virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV support
    virtual BOOL OnCommand(WPARAM wParam, LPARAM lParam);
    //}}AFX_VIRTUAL
};

//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert additional declarations immediately before the    ↙
    previous line.

#endif // !defined(AFX_USBTESTDLG_H__07909306_D75E_11D4_A644_93781B97CC41__INCLUDED_)
```

```cpp
//---------------------------------------------------------------------------
//
// Copyright (C) Gerardo Barroeta, MIT Media Lab
//
//---------------------------------------------------------------------------

#include "stdafx.h"
#include "OpenGLControl.h"
#include <math.h>           // sqrt

COpenGLControl::COpenGLControl(void)
{
   m_fPosX = 0.0f;    // X position of model in camera view
   m_fPosY = -10.0f;   // Y position of model in camera view
   m_fZoom = 80.0f;   // Zoom on model in camera view
   m_fRotX = -75.0f;  // Rotation on model in camera view
   m_fRotZ = -45.0f;    // Rotation on model in camera view
   m_XMatrix = 8;
   m_YMatrix = 8;
   for (int j=0; j<100; j++)
   {
        for (int k=0; k<100; k++)
        {
            //m_Height[j][k] = (float)j+(float)k;
            m_Height[j][k] = 0;
        }
   }
}

COpenGLControl::~COpenGLControl(void)
{
}

void COpenGLControl::oglCreate(CRect rect, CWnd *parent)
{
   CString className = AfxRegisterWndClass(CS_HREDRAW |
      CS_VREDRAW | CS_OWNDC, NULL,
      (HBRUSH)GetStockObject(BLACK_BRUSH), NULL);

   CreateEx(0, className, "OpenGL", WS_CHILD | WS_VISIBLE |
            WS_CLIPSIBLINGS | WS_CLIPCHILDREN, rect, parent, 0);

   // Set initial variables' values
   m_oldWindow    = rect;
   m_originalRect = rect;

   hWnd = parent;
}

BEGIN_MESSAGE_MAP(COpenGLControl, CWnd)
    ON_WM_PAINT()
    ON_WM_CREATE()
    ON_WM_TIMER()
    ON_WM_SIZE()
    ON_WM_MOUSEMOVE()
END_MESSAGE_MAP()


int COpenGLControl::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    if (CWnd::OnCreate(lpCreateStruct) == -1)
        return -1;

    // TODO:  Add your specialized creation code here
    oglInitialize();

    return 0;
```

```cpp
}

void COpenGLControl::OnPaint()
{
    //CPaintDC dc(this); // device context for painting
    // TODO: Add your message handler code here
    // Do not call CWnd::OnPaint() for painting messages
    ValidateRect(NULL);
}

void COpenGLControl::OnDraw(CDC *pDC)
{
    // TODO: Camera controls.
    glLoadIdentity();
    glTranslatef(0.0f, 0.0f, -m_fZoom);
    glTranslatef(m_fPosX, m_fPosY, 0.0f);
    glRotatef(m_fRotX, 1.0f, 0.0f, 0.0f);
    glRotatef(m_fRotZ, 0.0f, 0.0f, 1.0f);
}

void COpenGLControl::oglInitialize(void)
{
    // Initial Setup:
    //
    static PIXELFORMATDESCRIPTOR pfd =
    {
      sizeof(PIXELFORMATDESCRIPTOR),
      1,
      PFD_DRAW_TO_WINDOW | PFD_SUPPORT_OPENGL | PFD_DOUBLEBUFFER,
      PFD_TYPE_RGBA,
      32,    // bit depth
      0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
      16,    // z-buffer depth
      0, 0, 0, 0, 0, 0, 0,
    };

    // Get device context only once.
    hdc = GetDC()->m_hDC;

    // Pixel format.
    m_nPixelFormat = ChoosePixelFormat(hdc, &pfd);
    SetPixelFormat(hdc, m_nPixelFormat, &pfd);

    // Create the OpenGL Rendering Context.
    hrc = wglCreateContext(hdc);
    wglMakeCurrent(hdc, hrc);

    // Basic Setup:
    //
    // Set color to use when clearing the background.
    glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
    glClearDepth(1.0f);

    // Turn on backface culling
    glFrontFace(GL_CCW);
    glCullFace(GL_BACK);

    // Turn on depth testing
    glEnable(GL_DEPTH_TEST);
    glDepthFunc(GL_LEQUAL);
    glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);          // Really Nice ↙
    Perspective Calculations


    // Create light components
    GLfloat ambientLight0[] = { 0.0f, 0.0f, 0.0f, 1.0f };
    GLfloat diffuseLight0[] = { 0.2f, 0.2f, 0.2f, 1.0f };
    GLfloat specularLight0[] = { 0.2f, 0.2f, 0.2f, 1.0f };
    // Create light components
    GLfloat ambientLight1[] = { 0.0f, 0.0f, 0.0f, 1.0f };
    GLfloat diffuseLight1[] = { 0.2f, 0.2f, 0.2f, 1.0f };
    GLfloat specularLight1[] = { 0.2f, 0.2f, 0.2f, 1.0f };
    // Create light components
    GLfloat ambientLight2[] = { 0.0f, 0.0f, 0.0f, 1.0f };
    GLfloat diffuseLight2[] = { 0.2f, 0.2f, 0.2f, 1.0f };
    GLfloat specularLight2[] = { 0.2f, 0.2f, 0.2f, 1.0f };

    GLfloat global_ambient[] = { 0.999f, 0.999f, 0.999f, 1.0f };

    glLightModelfv(GL_LIGHT_MODEL_AMBIENT, global_ambient);

    // Assign created components to GL_LIGHT0
    glLightfv(GL_LIGHT0, GL_AMBIENT, ambientLight0);
    glLightfv(GL_LIGHT0, GL_DIFFUSE, diffuseLight0);
    //glLightfv(GL_LIGHT0, GL_SPECULAR, specularLight0);
    // Assign created components to GL_LIGHT0
    glLightfv(GL_LIGHT1, GL_AMBIENT, ambientLight1);
    glLightfv(GL_LIGHT1, GL_DIFFUSE, diffuseLight1);
    //glLightfv(GL_LIGHT1, GL_SPECULAR, specularLight1);
    // Assign created components to GL_LIGHT0
    glLightfv(GL_LIGHT2, GL_AMBIENT, ambientLight2);
    glLightfv(GL_LIGHT2, GL_DIFFUSE, diffuseLight2);
    //glLightfv(GL_LIGHT2, GL_SPECULAR, specularLight2);

    GLfloat position0[] = { 0, 200, 0, 1.0f };
    GLfloat position1[] = { 50, 200, 0, 1.0f };
    GLfloat position2[] = { -50, 200, 0, 1.0f };

    // Assign light positions
    glLightfv(GL_LIGHT0, GL_POSITION, position0);
    glLightfv(GL_LIGHT1, GL_POSITION, position1);
    glLightfv(GL_LIGHT2, GL_POSITION, position2);

    glShadeModel(GL_SMOOTH);
    glEnable(GL_LINE_SMOOTH);

    // Blending Function For Translucency Based On Source Alpha Value ( NEW )
    //glBlendFunc(GL_SRC_ALPHA,GL_ONE);
    //glEnable(GL_BLEND);
    //glDisable(GL_DEPTH_TEST);

    //Set up lighning model
    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    //glEnable(GL_LIGHT1);
    //glEnable(GL_LIGHT2);

    // Send draw request
    OnDraw(NULL);
}

void COpenGLControl::OnTimer(UINT nIDEvent)
{
   switch (nIDEvent)
   {
      case 1:
      {
         // Clear color and depth buffer bits
         glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

         // Draw OpenGL scene
         oglDrawScene();

         // Swap buffers
```

```
         SwapBuffers(hdc);

         break;
      }

      default:
         break;
   }

   CWnd::OnTimer(nIDEvent);
}

void COpenGLControl::OnSize(UINT nType, int cx, int cy)
{
    CWnd::OnSize(nType, cx, cy);

    if (0 >= cx || 0 >= cy || nType == SIZE_MINIMIZED) return;

    // Map the OpenGL coordinates.
    glViewport(0, 0, cx, cy);

    // Projection view
    glMatrixMode(GL_PROJECTION);

    glLoadIdentity();

    // Set our current view perspective
    gluPerspective(35.0f, (float)cx / (float)cy, 0.01f, 2000.0f);

    // Model view
    glMatrixMode(GL_MODELVIEW);
}

void COpenGLControl::oglDrawScene(void)
{
    // Filled Mode
    glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
    // enable color tracking
    glEnable(GL_COLOR_MATERIAL);

    //Set up material properties
    float red=1.0f, green=0.0f, blue=0.0f, alpha=1.0f;
    float mcolor[] = { red, green, blue, alpha };
    glMaterialfv(GL_FRONT, GL_AMBIENT_AND_DIFFUSE, mcolor);
    glFrontFace(GL_CCW);
    glColor3f(red, green, blue);

    //Specular highlights
    float specReflection[] = { 0.4f, 0.4f, 0.4f, 1.0f };
    glMaterialfv(GL_FRONT, GL_SPECULAR, specReflection);

    polygon_t face;
    GLfloat a=-(3*m_XMatrix/2-1), b, offset;
    vector_t vecNormal = { 0.0f, 0.0f, 0.0f };
    for (int i=0; i<m_XMatrix; i++)
    {
        b=-(3*m_YMatrix/2-1);
        for (int j=0; j<m_YMatrix; j++)
        {
            glBegin(GL_QUADS);
                // Top Side
                face.v[0].x=1.0f+a;
                face.v[0].y=1.0f+b;
                face.v[0].z=2.0f;
                face.v[1].x=1.0f+a;
                face.v[1].y=1.0f+a;
                face.v[1].z=0;
                face.v[2].x=-1.0f+a;
                face.v[2].y=1.0f+a;
                face.v[2].z=0;
                //Calculate normal vector
                normal(face.v, &vecNormal);
                //Paint the polygon
                glVertex3f( 1.0f+a, 1.0f+b,  m_Height[i][j]);
                glVertex3f( 1.0f+a, 1.0f+b, 0.0f);
                glVertex3f(-1.0f+a, 1.0f+b, 0.0f);
                glVertex3f(-1.0f+a, 1.0f+b,  m_Height[i][j]);
                glNormal3f(face.v[0].x + vecNormal.x, face.v[0].y + vecNormal.y, face.↙
    v[0].z + vecNormal.z);

                // Bottom Side
                face.v[0].x=-1.0f+a;
                face.v[0].y=-1.0f+b;
                face.v[0].z=0;
                face.v[1].x=1.0f+a;
                face.v[1].y=-1.0f+b;
                face.v[1].z=0;
                face.v[2].x=1.0f+a;
                face.v[2].y=-1.0f+b;
                face.v[2].z=2.0f;
                //Calculate normal vector
                normal(face.v, &vecNormal);
                //Paint the polygon
                glVertex3f(-1.0f+a, -1.0f+b, 0.0f);
                glVertex3f( 1.0f+a, -1.0f+b, 0.0f);
                glVertex3f( 1.0f+a, -1.0f+b,  m_Height[i][j]);
                glVertex3f(-1.0f+a, -1.0f+b,  m_Height[i][j]);
                glNormal3f(face.v[0].x + vecNormal.x, face.v[0].y + vecNormal.y, face.↙
    v[0].z + vecNormal.z);

                // Front Side
                face.v[0].x=1.0f+a;
                face.v[0].y=1.0f+b;
                face.v[0].z=2.0f;
                face.v[1].x=-1.0f+a;
                face.v[1].y=1.0f+b;
                face.v[1].z=2;
                face.v[2].x=-1.0f+a;
                face.v[2].y=-1.0f+b;
                face.v[2].z=2.0f;
                //Calculate normal vector
                normal(face.v, &vecNormal);
                //Paint the polygon
                glVertex3f( 1.0f+a,  1.0f+b, m_Height[i][j]);
                glVertex3f(-1.0f+a,  1.0f+b, m_Height[i][j]);
                glVertex3f(-1.0f+a, -1.0f+b, m_Height[i][j]);
                glVertex3f( 1.0f+a, -1.0f+b, m_Height[i][j]);
                glNormal3f(face.v[0].x + vecNormal.x, face.v[0].y + vecNormal.y, face.↙
    v[0].z + vecNormal.z);

                // Back Side
                face.v[0].x=-1.0f+a;
                face.v[0].y=-1.0f+b;
                face.v[0].z=0;
                face.v[1].x=-1.0f+a;
                face.v[1].y=1.0f+b;
                face.v[1].z=0;
                face.v[2].x=1.0f+a;
                face.v[2].y=1.0f+b;
                face.v[2].z=0;
                //Calculate normal vector
                normal(face.v, &vecNormal);
                //Paint the polygon
                glVertex3f(-1.0f+a, -1.0f+b, 0.0f);
```

```
                glVertex3f(-1.0f+a,  1.0f+b, 0.0f);
                glVertex3f( 1.0f+a,  1.0f+b, 0.0f);
                glVertex3f( 1.0f+a, -1.0f+b, 0.0f);
                glNormal3f(face.v[0].x + vecNormal.x, face.v[0].y + vecNormal.y, face.↙
    v[0].z + vecNormal.z);

                // Left Side
                face.v[0].x=-1.0f+a;
                face.v[0].y=-1.0f+b;
                face.v[0].z=0;
                face.v[1].x=-1.0f+a;
                face.v[1].y=-1.0f+b;
                face.v[1].z=2.0f;
                face.v[2].x=-1.0f+a;
                face.v[2].y=1.0f+b;
                face.v[2].z=2.0f;
                //Calculate normal vector
                normal(face.v, &vecNormal);
                //Paint the polygon
                glVertex3f(-1.0f+a, -1.0f+b, 0.0f);
                glVertex3f(-1.0f+a, -1.0f+b, m_Height[i][j]);
                glVertex3f(-1.0f+a,  1.0f+b, m_Height[i][j]);
                glVertex3f(-1.0f+a,  1.0f+b, 0.0f);
                glNormal3f(face.v[0].x + vecNormal.x, face.v[0].y + vecNormal.y, face.↙
    v[0].z + vecNormal.z);

                // Right Side
                face.v[0].x=1.0f+a;
                face.v[0].y=1.0f+b;
                face.v[0].z=2.0f;
                face.v[1].x=1.0f+a;
                face.v[1].y=-1.0f+b;
                face.v[1].z=2.0f;
                face.v[2].x=1.0f+a;
                face.v[2].y=-1.0f+b;
                face.v[2].z=0;
                //Calculate normal vector
                normal(face.v, &vecNormal);
                //Paint the polygon
                glVertex3f( 1.0f+a,  1.0f+b, m_Height[i][j]);
                glVertex3f( 1.0f+a, -1.0f+b, m_Height[i][j]);
                glVertex3f( 1.0f+a, -1.0f+b, 0.0f);
                glVertex3f( 1.0f+a,  1.0f+b, 0.0f);
                glNormal3f(face.v[0].x + vecNormal.x, face.v[0].y + vecNormal.y, face.↙
    v[0].z + vecNormal.z);
            glEnd();
            b+=3;
            red -= 1/(m_YMatrix*m_XMatrix);
            //Change the color
            glColor3f(red, green, blue);
        }
        a+=3;
        blue += 5/(m_YMatrix*m_XMatrix);
    }
    a=(3*m_XMatrix/2);
    // Draw Walls
    float specReflection2[] = { 0.0f, 0.0f, 0.0f, 1.0f };
    glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, specReflection2);

    glBegin(GL_QUADS);
        glColor4f(0.2f, 0.2f, 0.2f, 0.5f);
        face.v[0].x=-a-1.0f;
        face.v[0].y=a+1.0f;
        face.v[0].z=-1.0f;
        face.v[1].x=a;
        face.v[1].y=a+1.0f;
        face.v[1].z=-1.0f;
        face.v[2].x=a;
        face.v[2].y=a+1.0f;
        face.v[2].z=a+2.0f;
        //Calculate normal vector
        normal(face.v, &vecNormal);
        glVertex3f(-a-1.0f, a+1.0f, -1.0f);
        glVertex3f(a, a+1.0f, -1.0f);
        glVertex3f(a, a+1.0f, 2.0f*a);
        glVertex3f(-a-1.0f, a+1.0f, 2.0f*a);
        glNormal3f(face.v[0].x + vecNormal.x, face.v[0].y + vecNormal.y, face.v[0].z +↙
     vecNormal.z);

        face.v[0].x=-a-1.0f;
        face.v[0].y=a+1.0f;
        face.v[0].z=-1.0f;
        face.v[1].x=-a-1.0f;
        face.v[1].y=-a;
        face.v[1].z=-1.0f;
        face.v[2].x=-a-1.0f;
        face.v[2].y=-a;
        face.v[2].z=2.0f*a;
        //Calculate normal vector
        normal(face.v, &vecNormal);
        glVertex3f(-a-1.0f,a+1.0f,-1.0f);
        glVertex3f(-a-1.0f,-a,-1.0f);
        glVertex3f(-a-1.0f,-a,2.0f*a);
        glVertex3f(-a-1.0f,a+1.0f,2.0f*a);
        glNormal3f(face.v[0].x + vecNormal.x, face.v[0].y + vecNormal.y, face.v[0].z +↙
     vecNormal.z);

        face.v[0].x=-a-1.0f;
        face.v[0].y=a+1.0f;
        face.v[0].z=-1.0f;
        face.v[1].x=a;
        face.v[1].y=a+1.0f;
        face.v[1].z=-1.0f;
        face.v[2].x=a;
        face.v[2].y=-a;
        face.v[2].z=-1.0f;
        //Calculate normal vector
        normal(face.v, &vecNormal);
        glVertex3f(-a-1,a+1,-1);
        glVertex3f(a,a+1,-1);
        glVertex3f(a,-a,-1);
        glVertex3f(-a-1,-a,-1);
        glNormal3f(face.v[0].x + vecNormal.x, face.v[0].y + vecNormal.y, face.v[0].z +↙
     vecNormal.z);
    glEnd();

    glLineWidth(1.0f);
    offset = 0.9;
    glBegin(GL_LINES);
        glColor3f(0.0f, 1.0f, 0.0f);
        glVertex3f(-a-offset,a+offset,-1);
        glVertex3f(-a-offset,-a,-1);
        glVertex3f(-a-offset,a+offset,-1);
        glVertex3f(a, a+offset,-1);
        glVertex3f(-a-offset,a+offset,-1);
        glVertex3f(-a-offset,a+offset,2*a);
    glEnd();
}

void COpenGLControl::OnMouseMove(UINT nFlags, CPoint point)
{
   int diffX = (int)(point.x - m_fLastX);
   int diffY = (int)(point.y - m_fLastY);
   m_fLastX  = (float)point.x;
```

```
   m_fLastY  = (float)point.y;

   // Left mouse button
   if (nFlags & MK_LBUTTON)
   {
      m_fRotX += (float)0.5f * diffY;

      if ((m_fRotX > 360.0f) || (m_fRotX < -360.0f))
      {
         m_fRotX = 0.0f;
      }

      m_fRotZ += (float)0.5f * diffX;

      if ((m_fRotZ > 360.0f) || (m_fRotZ < -360.0f))
      {
         m_fRotZ = 0.0f;
      }
   }

   // Right mouse button
   else if (nFlags & MK_RBUTTON)
   {
      m_fZoom -= (float)0.1f * diffY;
   }

   // Middle mouse button
   else if (nFlags & MK_MBUTTON)
   {
      m_fPosX += (float)0.05f * diffX;
      m_fPosY -= (float)0.05f * diffY;
   }

   OnDraw(NULL);

   CWnd::OnMouseMove(nFlags, point);
}




void COpenGLControl::setHeight(int x, int y, float height)
{
    m_Height[x][y] = height;
}
void COpenGLControl::setMatrixSize(int x, int y)
{
    m_XMatrix=(float)x;
    m_YMatrix=(float)y;
}
void normalize (vector_t *v)
{
    // calculate the length of the vector
    float len = (float)(sqrt((v->x * v->x) + (v->y * v->y) + (v->z * v->z)));

    // avoid division by 0
    if (len == 0.0f)
        len = 1.0f;

    // reduce to unit size
    v->x /= len;
    v->y /= len;
    v->z /= len;
}

void normal (vertex_t v[3], vector_t *normal)
{
    vector_t a, b;

    // calculate the vectors A and B
    // note that v[3] is defined with counterclockwise winding in mind
    // a
    a.x = v[0].x - v[1].x;
    a.y = v[0].y - v[1].y;
    a.z = v[0].z - v[1].z;
    // b
    b.x = v[1].x - v[2].x;
    b.y = v[1].y - v[2].y;
    b.z = v[1].z - v[2].z;

    // calculate the cross product and place the resulting vector
    // into the address specified by vector_t *normal
    normal->x = (a.y * b.z) - (a.z * b.y);
    normal->y = (a.z * b.x) - (a.x * b.z);
    normal->z = (a.x * b.y) - (a.y * b.x);

    // normalize
    normalize(normal);
}
```

```
//---------------------------------------------------------------------------
//
// Copyright (C) Gerardo Barroeta, MIT Media Lab
//
//---------------------------------------------------------------------------
// USBtest.cpp : Defines the class behaviors for the application.

#include "stdafx.h"
#include "USBtest.h"
#include "USBtestDlg.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

//Globals

/////////////////////////////////////////////////////////////////////////////
// CUSBtestApp

BEGIN_MESSAGE_MAP(CUSBtestApp, CWinApp)
    //{{AFX_MSG_MAP(CUSBtestApp)
        // NOTE - the ClassWizard will add and remove mapping macros here.
        //    DO NOT EDIT what you see in these blocks of generated code!
    //}}AFX_MSG
    ON_COMMAND(ID_HELP, CWinApp::OnHelp)
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CUSBtestApp construction

CUSBtestApp::CUSBtestApp()
{
    // TODO: add construction code here,
    // Place all significant initialization in InitInstance
}

/////////////////////////////////////////////////////////////////////////////
// The one and only CUSBtestApp object

CUSBtestApp theApp;

/////////////////////////////////////////////////////////////////////////////
// CUSBtestApp initialization

BOOL CUSBtestApp::InitInstance()
{
    AfxEnableControlContainer();

    // Standard initialization
    // If you are not using these features and wish to reduce the size
    //  of your final executable, you should remove from the following
    //  the specific initialization routines you do not need.

/*#ifdef _AFXDLL
    Enable3dControls();         // Call this when using MFC in a shared DLL
#else
    Enable3dControlsStatic();   // Call this when linking to MFC statically
#endif
*/
    CUSBtestDlg dlg;
    m_pMainWnd = &dlg;
    int nResponse = dlg.DoModal();
    if (nResponse == IDOK)
    {
```

```
        // TODO: Place code here to handle when the dialog is
        //  dismissed with OK
    }
    else if (nResponse == IDCANCEL)
    {
        // TODO: Place code here to handle when the dialog is
        //  dismissed with Cancel
    }

    // Since the dialog has been closed, return FALSE so that we exit the
    //  application, rather than start the application's message pump.
    return FALSE;
}
```

```
// USBtestDlg.cpp : implementation file
#include "stdafx.h"
#include "USBtest.h"
#include "USBtestDlg.h"
#include "ftd2xx.h"
#include <stdio.h>

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

#pragma region Definitions
#define BYTES_TO_READ           1000
#define BYTES_TO_SEND           45
#define SOF_BYTES               2
#define EOF_BYTES               2
#define REFLASH_ALL_INST        0xA0
#define RESET_BRAIN_INST        0xA2
#define RESET_SKIN_INST         0xA4
#define EXECUTE_CODE_INST       0xA6
#define REFLASH_DEV_INST        0xA8
#define RESET_DEV_INST          0xAA
#define START_SAMPLING_INST     0xAC
#define STOP_SAMPLING_INST      0xAE
#define BYTES_PER_LINE          2
#pragma endregion

#pragma region Globals
//globals
bool m_connect=false;
bool m_sampling=false;
DWORD numDevs;
CFile sampleFile;
int x=0, y=0;
int running;
int board_present;
int bus_busy;
short R_Add;
HDEVNOTIFY *hdn=NULL;
int     m_SpinValue;
CWinThread* pWThread;
#pragma endregion

#pragma region AboutDlg
// CAboutDlg dialog used for App About
class CAboutDlg : public CDialog
{
public:
    CAboutDlg();

// Dialog Data
    //{{AFX_DATA(CAboutDlg)
    enum { IDD = IDD_ABOUTBOX };
    //}}AFX_DATA

    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CAboutDlg)
    protected:
    virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV support
    //}}AFX_VIRTUAL

// Implementation
protected:
    //{{AFX_MSG(CAboutDlg)
    //}}AFX_MSG
```

```
    DECLARE_MESSAGE_MAP()
};

CAboutDlg::CAboutDlg() : CDialog(CAboutDlg::IDD)
{
    //{{AFX_DATA_INIT(CAboutDlg)
    //}}AFX_DATA_INIT
}

void CAboutDlg::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    //{{AFX_DATA_MAP(CAboutDlg)
    //}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(CAboutDlg, CDialog)
    //{{AFX_MSG_MAP(CAboutDlg)
        // No message handlers
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()
#pragma endregion

//Application Constructor
CUSBtestDlg::CUSBtestDlg(CWnd* pParent /*=NULL*/)
: CDialog(CUSBtestDlg::IDD, pParent)
{
    //{{AFX_DATA_INIT(CUSBtestDlg)
    //hDevice = NULL;
    m_Nodes = 0;
    m_Address = _T("");
    m_PortStatus = _T("");
    m_NumRecd = 0;
    m_128status = _T("");
    m_NameNmbr = _T("FTDI USB Chip.");
    m_Address = _T("");
    m_SerDesc = 0;
    m_Nodes = 0;
    m_EditSpin = _T("8");
    m_SpinValue = 8;
    //}}AFX_DATA_INIT
    // Note that LoadIcon does not require a subsequent DestroyIcon in Win32
    m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);
}

//Synchronize window values
void CUSBtestDlg::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    //{{AFX_DATA_MAP(CUSBtestDlg)
    DDX_Control(pDX, IDC_LIST1, m_Received);
    DDX_Text(pDX, IDC_EDIT_PORT_STATUS, m_PortStatus);
    DDX_Text(pDX, IDC_EDIT_NUM_RECD, m_NumRecd);
    DDX_Text(pDX, IDC_EDIT_NAME_NUMBER, m_NameNmbr);
    DDX_Text(pDX, IDC_EDIT_ADDRESS, m_Address);
    DDX_Radio(pDX, IDC_RADIO_NAME_NUM, m_SerDesc);
    DDX_Radio(pDX, IDC_RADIO_ALL_NODES, m_Nodes);
    DDX_Text(pDX, IDC_EDIT1, m_EditSpin);
    DDX_Control(pDX, IDC_SPIN, m_Spin);
    //}}AFX_DATA_MAP
}

//Message map
#pragma region Message_Map
BEGIN_MESSAGE_MAP(CUSBtestDlg, CDialog)
    //{{AFX_MSG_MAP(CUSBtestDlg)
    //ON_WM_DEVICECHANGE()
    ON_WM_SYSCOMMAND()
    ON_WM_PAINT()
    ON_WM_QUERYDRAGICON()
    ON_WM_TIMER()
    ON_WM_CLOSE()
    ON_WM_DEVICECHANGE()
    ON_WM_VSCROLL()
    ON_BN_CLICKED(IDC_BUTTON_SENDFILE, OnButtonSendFile)
    ON_BN_CLICKED(IDC_BUTTON_CLEAR, OnButtonClear)
    ON_BN_CLICKED(IDC_BUTTON_OPEN, OnButtonOpen)
    ON_BN_CLICKED(IDC_BUTTON_RESET, OnButtonReset)
    ON_BN_CLICKED(IDC_BUTTON_RS, OnButtonRs)
    ON_BN_CLICKED(IDC_BUTTON_RUN_NODES, &CUSBtestDlg::OnButtonRunNodes)
    ON_BN_CLICKED(IDC_RADIO_NAME_NUM, OnRadioNameNum)
    ON_BN_CLICKED(IDC_RADIO_SERNUM, OnRadioSernum)
    ON_BN_CLICKED(IDC_RADIO_DEVNO, OnRadioDevno)
    ON_BN_CLICKED(IDC_RADIO_ALL_NODES, OnRadioAllNodes)
    ON_BN_CLICKED(IDC_RADIO_SINGLE_NODE, OnRadioSingleNode)
    ON_BN_CLICKED(IDC_BUTTON_LIST, OnButtonList)
    ON_BN_CLICKED(IDC_BUTTON_START, OnButtonStart)
    ON_BN_CLICKED(IDC_BUTTON_STOP, OnButtonStop)
    ON_EN_CHANGE(IDC_EDIT_NAME_NUMBER, OnChangeEditNameNumber)
    ON_EN_CHANGE(IDC_EDIT_ADDRESS, OnChangeEditAddress)
    ON_LBN_SELCHANGE(IDC_LIST1, OnSelchangeList1)
    ON_NOTIFY(LVN_ITEMCHANGED, IDC_LIST2, &CUSBtestDlg::OnLvnItemchangedList2)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()
#pragma endregion
//APPLICATION MESSAGE HANDLERS
//Window Messages
BOOL CUSBtestDlg::OnInitDialog()
{
    CDialog::OnInitDialog();

    CRect rect;

    // Get size and position of the picture control
    GetDlgItem(IDC_OPENGL)->GetWindowRect(rect);

    // Convert screen coordinates to client coordinates
    ScreenToClient(rect);

    // Create OpenGL Control window
    m_oglWindow.oglCreate(rect, this);

    // Setup the OpenGL Window's timer to render
    m_oglWindow.m_unpTimer = m_oglWindow.SetTimer(1, 1, 0);

    // Add "About..." menu item to system menu.
    // IDM_ABOUTBOX must be in the system command range.
    ASSERT((IDM_ABOUTBOX & 0xFFF0) == IDM_ABOUTBOX);
    ASSERT(IDM_ABOUTBOX < 0xF000);

    CMenu* pSysMenu = GetSystemMenu(FALSE);
    if (pSysMenu != NULL)
    {
        CString strAboutMenu;
        strAboutMenu.LoadString(IDS_ABOUTBOX);
        if (!strAboutMenu.IsEmpty())
        {
            pSysMenu->AppendMenu(MF_SEPARATOR);
            pSysMenu->AppendMenu(MF_STRING, IDM_ABOUTBOX, strAboutMenu);
        }
    }

    // Set the icon for this dialog.  The framework does this automatically
    //  when the application's main window is not a dialog
```

```
    SetIcon(m_hIcon, TRUE);         // Set big icon
    SetIcon(m_hIcon, FALSE);        // Set small icon

    // TODO: Add extra initialization here
    Buttons(OFF);//turn em off till a port is active
    GetDlgItem(IDC_EDIT_ADDRESS)->EnableWindow(FALSE);

    bus_busy=0;
    board_present=0;
    SetTimer(15, 100, NULL);//check for new USB data in the buffer every 100mS
    srand((unsigned)time(0));

    //read the last settings from the registry
    m_SerDesc = AfxGetApp()->GetProfileInt("MyUSBTestAp", "SerDesc", 0);//default to      ↙
    Description

    m_Spin.SetRange(4, 8);

    if(m_SerDesc==0)
        m_NameNmbr = AfxGetApp()->GetProfileString("MyUSBTestAp", "DescString", "FTDI USB ↙
    Chip.");
    if(m_SerDesc==1)
        m_NameNmbr = AfxGetApp()->GetProfileString("MyUSBTestAp", "SerialString", "FTDI   ↙
    USB Chip.");
    if(m_SerDesc==2)
        m_NameNmbr = AfxGetApp()->GetProfileString("MyUSBTestAp", "DevNmbr", "FTDI USB    ↙
    Chip.");

    UpdateData(FALSE);

    //Load Dll
    LoadDLL();

    //Register for device notification
    CUSBtestDlg::RegisterNotification(hdn);

    //List Devices
    OnButtonList();

    return TRUE;  // return TRUE  unless you set the focus to a control
}


void CUSBtestDlg::RegisterNotification(HDEVNOTIFY *hDevNotify)
{
    DEV_BROADCAST_DEVICEINTERFACE NotificationFilter={0};

    //ZeroMemory( &NotificationFilter, sizeof(NotificationFilter) );
    NotificationFilter.dbcc_size = sizeof( NotificationFilter );
    NotificationFilter.dbcc_devicetype = DBT_DEVTYP_DEVICEINTERFACE;
    NotificationFilter.dbcc_classguid = GUID_DEVINTERFACE_USB_DEVICE;
    hDevNotify = (void **)RegisterDeviceNotification(GetSafeHwnd(), &NotificationFilter,  ↙
    DEVICE_NOTIFY_WINDOW_HANDLE );

}
void CUSBtestDlg::UnregisterNotification(HDEVNOTIFY *hDevNotify)
{
    if(NULL != hDevNotify)
    {
        UnregisterDeviceNotification( hDevNotify );
        hDevNotify = NULL;
    }
}
void CUSBtestDlg::OnOK()
{
    UnregisterNotification(hdn);
    //terminate the thread
    m_sampling=0;
    Close();
    FreeLibrary(m_hmodule);

    CDialog::OnOK();
}

void CUSBtestDlg::OnSysCommand(UINT nID, LPARAM lParam)
{
    if ((nID & 0xFFF0) == IDM_ABOUTBOX)
    {
        CAboutDlg dlgAbout;
        dlgAbout.DoModal();
    }
    else
    {
        CDialog::OnSysCommand(nID, lParam);
    }
}

void CUSBtestDlg::OnPaint()
{
    if (IsIconic())
    {
        CPaintDC dc(this); // device context for painting

        SendMessage(WM_ICONERASEBKGND, (WPARAM) dc.GetSafeHdc(), 0);

        // Center icon in client rectangle
        int cxIcon = GetSystemMetrics(SM_CXICON);
        int cyIcon = GetSystemMetrics(SM_CYICON);
        CRect rect;
        GetClientRect(&rect);
        int x = (rect.Width() - cxIcon + 1) / 2;
        int y = (rect.Height() - cyIcon + 1) / 2;

        // Draw the icon
        dc.DrawIcon(x, y, m_hIcon);
    }
    else
    {
        CDialog::OnPaint();
    }
}

HCURSOR CUSBtestDlg::OnQueryDragIcon()
{
    return (HCURSOR) m_hIcon;
}

void CUSBtestDlg::OnTimer(UINT nIDEvent)
{
    unsigned char rx[600];
    DWORD ret_bytes;
    DWORD bytes_in_buf;
    FT_STATUS status;

    if(!board_present) return;

    //Don't interfere with another test
    if(bus_busy) return;

    //Is there data to read?
    status = GetQueueStatus(&bytes_in_buf);
    if(status != FT_OK)
    {
        UpdateData(TRUE);
```

```
        m_PortStatus = _T("Device not Open yet");
        Close();
        board_present=0;
        UpdateData(FALSE);
        UpdateWindow();

        bytes_in_buf = 0;
    }

    //There is stuff to read, read it!
    if(bytes_in_buf)
    {
        status = Read(rx, bytes_in_buf<600 ? bytes_in_buf : 600, &ret_bytes);
        //Read timeout
        if((status == FT_OK) || ((status == FT_IO_ERROR) && (ret_bytes > 0)) )
        {
            if(ret_bytes)
            {
                CString str;
                int lineBytes=0;
                UpdateData(TRUE);
                for(DWORD x=0; x<ret_bytes; x++)
                {
                    m_NumRecd++;
                    str.Format("%2d  0x%.2X", rx[x], rx[x]);
                    m_Received.AddString(str);
                }
            }
        }

        UpdateData(FALSE);
        UpdateWindow();
    }

    /*if (m_sampling)
    {
        for (int x=0; x<m_SpinValue; x++)
        {
            for (int y=0; y<m_SpinValue; y++)
            {
                height = 0+float(range*rand()/(RAND_MAX + 1.0));
                m_oglWindow.setHeight(x, y, height);
            }
        }
    }*/
    CDialog::OnTimer(nIDEvent);
}

void CUSBtestDlg::OnClose()
{
    UnregisterNotification(hdn);
    OnOK();
}
BOOL CUSBtestDlg::OnDeviceChange(UINT nEventType, DWORD_PTR dwData)
{
    switch(nEventType)
    {
        case DBT_DEVICEARRIVAL:
        // A device has been inserted and is now available.
            if ( ((DEV_BROADCAST_HDR *)dwData)->dbch_devicetype !=                 ↙
    DBT_DEVTYP_DEVICEINTERFACE )
                break;
            if ( ((DEV_BROADCAST_DEVICEINTERFACE *)dwData)->dbcc_classguid !=      ↙
    GUID_CLASS_USB_DEVICE )
                break;
            // set/reset connection flag
            OnButtonList();
            m_connect = TRUE;
            break;
        case DBT_DEVICEREMOVECOMPLETE:
        // Device has been removed.
        // Confirm this notify derives from the target device
            if ( ((DEV_BROADCAST_HDR *)dwData)->dbch_devicetype !=              ↙
    DBT_DEVTYP_DEVICEINTERFACE )
                break;
            if ( ((DEV_BROADCAST_DEVICEINTERFACE *)dwData)->dbcc_classguid !=   ↙
    GUID_CLASS_USB_DEVICE )
                break;
            // set/reset connection flag
            OnButtonList();
            break;
        default:
            break;
    }
    return TRUE;
}
//Control Event Message Handlers
UINT ThreadProc1(LPVOID param)//count up thread
{
    //typecast the handle to the parent window
    CUSBtestDlg *pMyHndl = (CUSBtestDlg *)param;
    unsigned char rxBuf[15*20], txBuf[3]={START_SAMPLING_INST};
    char valueBuf[10];
    DWORD ret_bytes;
    FT_STATUS status;
    CString str, writeBuf;
    DWORD bytes_in_buf=0;
    int value=0, x=0, y=0;
    int s_Add=261;

    while(m_sampling)
    {
        if (pMyHndl->m_Nodes)
        {
            txBuf[1]=(BYTE)R_Add;
            txBuf[2]=*((char*)&R_Add+1);
        }
        else
        {
            txBuf[1]=(BYTE)s_Add;
            txBuf[2]=*((char*)&s_Add+1);
        }
        status=pMyHndl->Purge(FT_PURGE_RX || FT_PURGE_TX);
        status = pMyHndl->Write(txBuf, 3, &ret_bytes);
        while(bytes_in_buf != 5) pMyHndl->GetQueueStatus(&bytes_in_buf);
        status = pMyHndl->Read(rxBuf, 5, &ret_bytes);
        if (ret_bytes == 5)
        {
            value = rxBuf[3]+256*rxBuf[4];
            pMyHndl->m_NumRecd+=ret_bytes;
            for (int i=0; i<ret_bytes; i++)
            {
                str.Format("%2d  0x%.2X", rxBuf[i], rxBuf[i]);
                pMyHndl->m_Received.AddString(str);
            }
            itoa(value, valueBuf, 10);
            writeBuf=(CString)valueBuf+"\t";
            if(m_sampling) sampleFile.Write(writeBuf.GetBuffer(), writeBuf.GetLength());
            pMyHndl->m_oglWindow.setHeight(x,y, 20*value/4096);
            s_Add+=1;
            if ((s_Add == 273) && (!pMyHndl->m_Nodes))
            {
                s_Add=261;
                if(m_sampling) sampleFile.Write("\n", strlen("\n"));
```

```
            }
            if (pMyHndl->m_Nodes)
            {
                if (m_sampling) sampleFile.Write("\n", strlen("\n"));
            }
            if (!pMyHndl->m_Nodes)
            {
                x=(x+1)%4;
                if(x==0) y=(y+1)%3;
            }
            //Sleep(100);
        }
    }
    return 0;
}

void CUSBtestDlg::Buttons(int onoff)
{
    if(onoff == OFF)
    {
        GetDlgItem(IDC_BUTTON_RS)->EnableWindow(FALSE);
        GetDlgItem(IDC_BUTTON_RESET)->EnableWindow(FALSE);
        GetDlgItem(IDC_BUTTON_SENDFILE)->EnableWindow(FALSE);
        GetDlgItem(IDC_BUTTON_START)->EnableWindow(FALSE);
        GetDlgItem(IDC_BUTTON_STOP)->EnableWindow(FALSE);
        GetDlgItem(IDC_EDIT_NAME_NUMBER)->EnableWindow(FALSE);
        GetDlgItem(IDC_BUTTON_RUN_NODES)->EnableWindow(FALSE);
        GetDlgItem(IDC_RADIO_ALL_NODES)->EnableWindow(FALSE);
        GetDlgItem(IDC_RADIO_SINGLE_NODE)->EnableWindow(FALSE);
    }

    if(onoff == ON)
    {
        GetDlgItem(IDC_BUTTON_RS)->EnableWindow(TRUE);
        GetDlgItem(IDC_BUTTON_RUN_NODES)->EnableWindow(TRUE);
        GetDlgItem(IDC_BUTTON_RESET)->EnableWindow(TRUE);
        GetDlgItem(IDC_BUTTON_SENDFILE)->EnableWindow(TRUE);
        GetDlgItem(IDC_BUTTON_START)->EnableWindow(TRUE);
        GetDlgItem(IDC_EDIT_NAME_NUMBER)->EnableWindow(TRUE);
        GetDlgItem(IDC_RADIO_ALL_NODES)->EnableWindow(TRUE);
        GetDlgItem(IDC_RADIO_SINGLE_NODE)->EnableWindow(TRUE);
    }
}

BOOL CUSBtestDlg::OnCommand(WPARAM wParam, LPARAM lParam) //disable the ESC key
{
    if(wParam==2) return FALSE;
    return CDialog::OnCommand(wParam, lParam);
}


void CUSBtestDlg::OnButtonClear()
{
        UpdateData(TRUE);
        m_NumRecd=0;
        m_Received.ResetContent();
        UpdateData(FALSE);
        UpdateWindow();
}

void CUSBtestDlg::OnButtonOpen()
{
    unsigned char txbuf[3];
    int tries=0;
    DWORD ret_bytes=0;

    UpdateData(TRUE);
    m_PortStatus = _T("-reset-");
    UpdateData(FALSE);
    UpdateWindow();

    Close();

    //open the device
    FT_STATUS status = OpenBy();
    if(status>0)
    {
        m_PortStatus = _T("Brain not found!");
        board_present=0;
    }
    else
    {
        status=ResetDevice();
        status=Purge(FT_PURGE_RX || FT_PURGE_TX);
        status=ResetDevice();
        SetTimeouts(3000, 3000);//extend timeout while board finishes reset
        Sleep(150);

        //test for presence of board
        txbuf[0] = 0x80;
        Write(txbuf, 1, &ret_bytes);
        //Read(rxbuf, 1, &ret_bytes);
        //if(ret_bytes==0)//if no response maybe windows was busy... read it again
        //  Read(rxbuf, 1, &ret_bytes);

        m_PortStatus = _T("Brain Ready!");
        Buttons(ON);//turn em off till a port is active
        board_present=1;
    }

    SetTimeouts(300, 300);
    UpdateData(FALSE);
    UpdateWindow();
}


void CUSBtestDlg::OnButtonSendFile()
{
    DWORD ret_bytes;
    FILE *codeFile;
    CString FileName="";
    UINT lBytesRead=0, nBytes, address[2], data[16], checkSum, i, index=1;
    unsigned char fileBuf[1000];

    //Ask user for the code file
    CFileDialog FileOpenDlg(TRUE,NULL,NULL,OFN_OVERWRITEPROMPT,"MSP430 Compiled Code (*.  ↙
    a43)|*.a43|All Files (*.*)|*.*||");
    int iRet = FileOpenDlg.DoModal();
    FileName = FileOpenDlg.GetFileName();

    if(iRet == IDOK)
    {
        m_Received.ResetContent();

        //Open The File
        codeFile = fopen(FileOpenDlg.GetPathName(), "r");

        if (m_Nodes)
        {
            fileBuf[1]=(BYTE)R_Add;
            fileBuf[2]=*((char*)&R_Add+1);
            index+=2;
        }
        //Parse text file
```

```
        while (fscanf(codeFile, "%*1s %2x %2x %2x %*2x", &nBytes, &address[0],&address[1])↙
     != EOF)
        {
            if (nBytes != 0)
            {
                fileBuf[index]=(char)nBytes;
                fileBuf[index+1]=(char)address[0];
                fileBuf[index+2]=(char)address[1];
                index+=3;
                for (i=0; i < nBytes; i++)
                {
                    lBytesRead = fscanf(codeFile, "%2x", &data[i]);
                    fileBuf[index+i]=(char)data[i];
                }
                lBytesRead = fscanf(codeFile, "%2x", &checkSum);
                fileBuf[index+i]=(char)checkSum;
                index+=i+1;
            }
            else lBytesRead = fscanf(codeFile, "%*x", &nBytes);
        }
        //Close file
        fclose(codeFile);

        //Configure File Buffer
        if(m_Nodes) fileBuf[0]=REFLASH_DEV_INST;
        else fileBuf[0]=REFLASH_ALL_INST;

        //Flush Buffer
        FT_STATUS status=Purge(FT_PURGE_RX || FT_PURGE_TX);

        //Send frame
        Write(fileBuf, index, &ret_bytes);
    }
}
void CUSBtestDlg::OnButtonReset()
{
    DWORD ret_bytes;
    unsigned char txBuf=RESET_BRAIN_INST;
    unsigned char* instnBuf=&txBuf;

    Purge(FT_PURGE_RX || FT_PURGE_TX);
    //Send Reset Instruction
    FT_STATUS status = Write(instnBuf, 1, &ret_bytes);
    if (status != FT_OK) MessageBox("Error Sending Instruction");
}
void CUSBtestDlg::OnButtonRs()
{
    DWORD ret_bytes;
    unsigned char txBuf[3];
    int txBytes=1;

    Purge(FT_PURGE_RX || FT_PURGE_TX);
    //Send Reset Instruction
    //if (m_Nodes)
    //{
    //  txBuf[1]=(BYTE)R_Add;
    //  txBuf[2]=*((char*)&R_Add+1);
    //  txBuf[0]=RESET_DEV_INST;
    //  txBytes = 3;
    //}
    //else
    txBuf[0]=RESET_SKIN_INST;
    FT_STATUS status = Write(txBuf, txBytes, &ret_bytes);
    if (status != FT_OK) MessageBox("Error Sending Instruction");
    // TODO: Add your control notification handler code here
}
```

```
void CUSBtestDlg::OnButtonRunNodes()
{
    DWORD ret_bytes;
    unsigned char txBuf=EXECUTE_CODE_INST;
    unsigned char* instnBuf=&txBuf;

    Purge(FT_PURGE_RX || FT_PURGE_TX);
    //Send Reset Instruction
    FT_STATUS status = Write(instnBuf, 1, &ret_bytes);
    if (status != FT_OK) MessageBox("Error Sending Instruction: 0xA4");
}

void CUSBtestDlg::OnButtonStart()
{
    unsigned char txBuf[3]={START_SAMPLING_INST};
    int nBytes=1;
    CString FileName="";

    if(board_present)
    {
        //Open a file to write data read from nodes
        CFileDialog FileOpenDlg(FALSE,NULL,NULL,OFN_OVERWRITEPROMPT,"Skin Sensor Data (*. ↙
    xls)|*.xls|All Files (*.*)|*.*||");
        int iRet = FileOpenDlg.DoModal();
        FileName = FileOpenDlg.GetFileName();
        if (FileName.Find(".xls") == -1) FileName += ".xls";

        if ((iRet == IDOK) && (FileName != ""))
        {
            //Open The File
            m_sampling = true;
            bool file = sampleFile.Open(FileName, CFile::modeCreate|CFile::modeWrite);
            if (file)
            {
                Buttons(OFF);
                GetDlgItem(IDC_BUTTON_LIST)->EnableWindow(FALSE);
                GetDlgItem(IDC_BUTTON_OPEN)->EnableWindow(FALSE);

                GetDlgItem(IDC_BUTTON_STOP)->EnableWindow(TRUE);
                Purge(FT_PURGE_RX || FT_PURGE_TX);
                //Start thread
                KillTimer(15);
                OnButtonRunNodes();
                pWThread = AfxBeginThread(ThreadProc1, this);
                //else MessageBox("Error Sending Instruction");
                OnButtonClear();
            }
            else MessageBox("Error: file could not be created");
        }
    }
}
void CUSBtestDlg::OnButtonStop()
{
    DWORD ret_bytes;
    unsigned char txBuf[3]={STOP_SAMPLING_INST};
    Purge(FT_PURGE_TX);

    m_sampling = false;

    //Send instruction to stop sampling
    FT_STATUS status = Write(txBuf, 1, &ret_bytes);

    //Close file
    sampleFile.Close();
    Buttons(ON);
    GetDlgItem(IDC_BUTTON_LIST)->EnableWindow(TRUE);
    GetDlgItem(IDC_BUTTON_OPEN)->EnableWindow(TRUE);
```

```
    GetDlgItem(IDC_BUTTON_STOP)->EnableWindow(FALSE);
}
void CUSBtestDlg::OnRadioNameNum()
{
    CWinApp* pApp = AfxGetApp();
    //write the new setup to the registry
    pApp->WriteProfileInt("MyUSBTestAp", "SerDesc", 0);

    UpdateData(TRUE);
    m_NameNmbr = AfxGetApp()->GetProfileString("MyUSBTestAp", "DescString", "FTDI USB Chip↙
    .");
    UpdateData(FALSE);
    UpdateWindow();
}

void CUSBtestDlg::OnRadioSernum()
{
    CWinApp* pApp = AfxGetApp();
    //write the new setup to the registry
    pApp->WriteProfileInt("MyUSBTestAp", "SerDesc", 1);

    UpdateData(TRUE);
    m_NameNmbr = AfxGetApp()->GetProfileString("MyUSBTestAp", "SerialString", "FTDI USB   ↙
    Chip.");
    UpdateData(FALSE);
    UpdateWindow();
}

void CUSBtestDlg::OnRadioDevno()
{
    CWinApp* pApp = AfxGetApp();
    //write the new setup to the registry
    pApp->WriteProfileInt("MyUSBTestAp", "SerDesc", 2);

    UpdateData(TRUE);
    m_NameNmbr = AfxGetApp()->GetProfileString("MyUSBTestAp", "DevNmbr", "FTDI USB Chip.")↙
    ;
    UpdateData(FALSE);
    UpdateWindow();
}


void CUSBtestDlg::OnRadioAllNodes()
{
    if(board_present)
    {
        GetDlgItem(IDC_BUTTON_START)->EnableWindow(TRUE);
        GetDlgItem(IDC_BUTTON_SENDFILE)->EnableWindow(TRUE);
        GetDlgItem(IDC_EDIT_ADDRESS)->EnableWindow(FALSE);
    }
}

void CUSBtestDlg::OnRadioSingleNode()
{

    if ((R_Add >= 1) && (R_Add <= 1023))
    {
        GetDlgItem(IDC_BUTTON_SENDFILE)->EnableWindow(TRUE);
        GetDlgItem(IDC_BUTTON_START)->EnableWindow(TRUE);
    }
    else
    {
        GetDlgItem(IDC_BUTTON_SENDFILE)->EnableWindow(FALSE);
        GetDlgItem(IDC_BUTTON_START)->EnableWindow(FALSE);

    }
    GetDlgItem(IDC_EDIT_ADDRESS)->EnableWindow(TRUE);
```

```
}

void CUSBtestDlg::OnButtonList()
{
    //search for Descriptions or Serial Numbers depending on the current mode
    FT_STATUS ftStatus;


    Close();//must be closed to perform the ListDevices() function
    UpdateData(TRUE);
    m_PortStatus = _T("Devices Closed.");
    m_NumRecd=0;
    m_NameNmbr = _T("");
    m_Received.ResetContent();
    UpdateData(FALSE);
    UpdateWindow();

    ftStatus = ListDevices(&numDevs, NULL, FT_LIST_NUMBER_ONLY);
    if(ftStatus == FT_OK)
    {
        // FT_ListDevices OK, show number of devices connected in list box
        CString str;
        str.Format("%d device(s) attached:", (int)numDevs);
        m_Received.AddString(str);


        //if current mode is open "by device #" then list device numbers
        if((m_SerDesc==2) && (numDevs>0))
        {
            for(DWORD d=1; d<=numDevs; d++)
            {
                str.Format("%d", d);
                m_Received.AddString(str);
            }

        }

        //if current mode is open "by description" then list descriptions of all connected↙
     devices
        if((m_SerDesc==0) && (numDevs>0))
        {
            ftStatus = ListDevices(&numDevs, NULL, FT_LIST_NUMBER_ONLY);
            if(ftStatus == FT_OK)
            {
                char *BufPtrs[64]; // pointer to array of 64 pointer
                DWORD d=0;

                for(d=0; d<numDevs; d++){
                    BufPtrs[d] = new char[64];
                }
                BufPtrs[d] = NULL;

                ftStatus = ListDevices(BufPtrs, &numDevs, FT_LIST_ALL|                    ↙
    FT_OPEN_BY_DESCRIPTION);
                if (FT_SUCCESS(ftStatus))
                {
                    for(DWORD u=0; u<numDevs; u++)
                    {
                        str.Format("%s", BufPtrs[u]);
                        m_Received.AddString(str);
                    }
                }
                else
                {
                    str.Format("ListDevices failed");
                    m_Received.AddString(str);
                }
```

```
                //free ram to avoid memory leaks
                for(d=0; d<numDevs; d++)
                {
                    delete BufPtrs[d];
                }

            }
        }

        //if current mode is open "by serial number" the list descriptions
        //of all connected devices
        if((m_SerDesc==1) && (numDevs>0))
        {
            //AfxMessageBox("by serial");
            ftStatus = ListDevices(&numDevs, NULL, FT_LIST_NUMBER_ONLY);
            if(ftStatus == FT_OK)
            {
                DWORD d=0;
                char *BufPtrs[64]; // pointer to array of 64 pointers
                for(d=0; d<numDevs; d++){
                    BufPtrs[d] = new char[64];
                }
                BufPtrs[d] = NULL;

                ftStatus = ListDevices(BufPtrs, &numDevs, FT_LIST_ALL|                    ↙
    FT_OPEN_BY_SERIAL_NUMBER);
                if (FT_SUCCESS(ftStatus))
                {
                    for(DWORD u=0; u<numDevs; u++)
                    {
                        str.Format("%s", BufPtrs[u]);
                        m_Received.AddString(str);
                    }
                }
                else
                {
                    str.Format("ListDevices failed");
                    m_Received.AddString(str);
                }

                //free ram to avoid memory leaks
                for(d=0; d<numDevs; d++)
                    delete BufPtrs[d];
            }
        }

        if (numDevs == 1)
        {
            AfxGetApp()->WriteProfileString("MyUSBTestAp", "DescString", str);
            OnButtonOpen();
        }
        if (numDevs == 0)
        {
            Buttons(OFF);
            m_connect = FALSE;
        }
        else
        {
            Buttons(ON);
        }
    }
    else
    {
        // FT_ListDevices failed
        AfxMessageBox("FT_ListDevices failed");
    }
```

```
}


void CUSBtestDlg::OnChangeEditNameNumber()
{
    UpdateData(TRUE);

    //write the new setup to the registry
    if(m_SerDesc==0)
        AfxGetApp()->WriteProfileString("MyUSBTestAp", "DescString", m_NameNmbr);
    if(m_SerDesc==1)
        AfxGetApp()->WriteProfileString("MyUSBTestAp", "SerialString", m_NameNmbr);
    if(m_SerDesc==2)
        AfxGetApp()->WriteProfileString("MyUSBTestAp", "DevNmbr", m_NameNmbr);

    if(m_NameNmbr.GetLength() < 1) m_SerDesc = 2;
    UpdateData(FALSE);
    UpdateWindow();
}


void CUSBtestDlg::OnChangeEditAddress()
{
    UpdateData(TRUE);
    R_Add = atoi(m_Address);
    if (board_present)
    {
        if ((R_Add >= 1) && (R_Add <= 1023))
        {
            GetDlgItem(IDC_BUTTON_SENDFILE)->EnableWindow(TRUE);
            GetDlgItem(IDC_BUTTON_START)->EnableWindow(TRUE);
        }
        else
        {
            GetDlgItem(IDC_BUTTON_SENDFILE)->EnableWindow(FALSE);
            GetDlgItem(IDC_BUTTON_START)->EnableWindow(FALSE);

        }
    }
    UpdateData(FALSE);
}
void CUSBtestDlg::LoadDLL()
{
    m_hmodule = LoadLibrary("Ftd2xx.dll");
    if(m_hmodule == NULL)
    {
        AfxMessageBox("Error: Can't Load ft8u245.dll");
        return;
    }

    m_pWrite = (PtrToWrite)GetProcAddress(m_hmodule, "FT_Write");
    if (m_pWrite == NULL)
    {
        AfxMessageBox("Error: Can't Find FT_Write");
        return;
    }

    m_pRead = (PtrToRead)GetProcAddress(m_hmodule, "FT_Read");
    if (m_pRead == NULL)
    {
        AfxMessageBox("Error: Can't Find FT_Read");
        return;
    }

    m_pOpen = (PtrToOpen)GetProcAddress(m_hmodule, "FT_Open");
    if (m_pOpen == NULL)
```

```
    {
        AfxMessageBox("Error: Can't Find FT_Open");
        return;
    }

    m_pOpenEx = (PtrToOpenEx)GetProcAddress(m_hmodule, "FT_OpenEx");
    if (m_pOpenEx == NULL)
    {
        AfxMessageBox("Error: Can't Find FT_OpenEx");
        return;
    }

    m_pListDevices = (PtrToListDevices)GetProcAddress(m_hmodule, "FT_ListDevices");
    if(m_pListDevices == NULL)
        {
            AfxMessageBox("Error: Can't Find FT_ListDevices");
            return;
        }

    m_pClose = (PtrToClose)GetProcAddress(m_hmodule, "FT_Close");
    if (m_pClose == NULL)
    {
        AfxMessageBox("Error: Can't Find FT_Close");
        return;
    }

    m_pResetDevice = (PtrToResetDevice)GetProcAddress(m_hmodule, "FT_ResetDevice");
    if (m_pResetDevice == NULL)
    {
        AfxMessageBox("Error: Can't Find FT_ResetDevice");
        return;
    }

    m_pPurge = (PtrToPurge)GetProcAddress(m_hmodule, "FT_Purge");
    if (m_pPurge == NULL)
    {
        AfxMessageBox("Error: Can't Find FT_Purge");
        return;
    }

    m_pSetTimeouts = (PtrToSetTimeouts)GetProcAddress(m_hmodule, "FT_SetTimeouts");
    if (m_pSetTimeouts == NULL)
    {
        AfxMessageBox("Error: Can't Find FT_SetTimeouts");
        return;
    }

    m_pGetQueueStatus = (PtrToGetQueueStatus)GetProcAddress(m_hmodule, "FT_GetQueueStatus"↙
    );
    if (m_pGetQueueStatus == NULL)
    {
        AfxMessageBox("Error: Can't Find FT_GetQueueStatus");
        return;
    }
}
void CUSBtestDlg::OnSelchangeList1()
{
    UpdateData(TRUE);

    int pos = m_Received.GetCurSel();
    m_Received.GetText(pos, m_NameNmbr);

    UpdateData(FALSE);
    UpdateWindow();

    if(m_SerDesc==0)
        AfxGetApp()->WriteProfileString("MyUSBTestAp", "DescString", m_NameNmbr);
    if(m_SerDesc==1)
        AfxGetApp()->WriteProfileString("MyUSBTestAp", "SerialString", m_NameNmbr);
    if(m_SerDesc==2)
        AfxGetApp()->WriteProfileString("MyUSBTestAp", "DevNmbr", m_NameNmbr);
}




void CUSBtestDlg::OnLvnItemchangedList2(NMHDR *pNMHDR, LRESULT *pResult)
{
    LPNMLISTVIEW pNMLV = reinterpret_cast<LPNMLISTVIEW>(pNMHDR);
    // TODO: Add your control notification handler code here
    *pResult = 0;
}

void CUSBtestDlg::OnVScroll(UINT nSBCode, UINT nPos, CScrollBar* pScrollBar){
    // TODO: Add your control notification handler code here
    m_SpinValue = nPos;
    m_oglWindow.setMatrixSize(nPos, nPos);
    //OnVScroll(nSBCode, nPos, pScrollBar);
}
#pragma region USB_DLL_Functions
//USB Driver Function Definitions
FT_STATUS CUSBtestDlg::Read(LPVOID lpvBuffer, DWORD dwBuffSize, LPDWORD lpdwBytesRead)
{
    if (!m_pRead)
    {
        AfxMessageBox("FT_Read is not valid!");
        return FT_INVALID_HANDLE;
    }

    return (*m_pRead)(m_ftHandle, lpvBuffer, dwBuffSize, lpdwBytesRead);
}


FT_STATUS CUSBtestDlg::Write(LPVOID lpvBuffer, DWORD dwBuffSize, LPDWORD lpdwBytes)
{
    if (!m_pWrite)
    {
        AfxMessageBox("FT_Write is not valid!");
        return FT_INVALID_HANDLE;
    }

    return (*m_pWrite)(m_ftHandle, lpvBuffer, dwBuffSize, lpdwBytes);
}

FT_STATUS CUSBtestDlg::Open(PVOID pvDevice)
{
    if (!m_pOpen)
    {
        AfxMessageBox("FT_Open is not valid!");
        return FT_INVALID_HANDLE;
    }

    return (*m_pOpen)(pvDevice, &m_ftHandle );
}

FT_STATUS CUSBtestDlg::OpenEx(PVOID pArg1, DWORD dwFlags)
{
    if (!m_pOpenEx)
    {
        AfxMessageBox("FT_OpenEx is not valid!");
        return FT_INVALID_HANDLE;
    }
```

```
    return (*m_pOpenEx)(pArg1, dwFlags, &m_ftHandle);
}

FT_STATUS CUSBtestDlg::ListDevices(PVOID pArg1, PVOID pArg2, DWORD dwFlags)
{
    if (!m_pListDevices)
    {
        AfxMessageBox("FT_ListDevices is not valid!");
        return FT_INVALID_HANDLE;
    }

    return (*m_pListDevices)(pArg1, pArg2, dwFlags);
}

FT_STATUS CUSBtestDlg::Close()
{
    if (!m_pClose)
    {
        AfxMessageBox("FT_Close is not valid!");
        return FT_INVALID_HANDLE;
    }

    return (*m_pClose)(m_ftHandle);
}

FT_STATUS CUSBtestDlg::ResetDevice()
{
    if (!m_pResetDevice)
    {
        AfxMessageBox("FT_ResetDevice is not valid!");
        return FT_INVALID_HANDLE;
    }

    return (*m_pResetDevice)(m_ftHandle);
}



FT_STATUS CUSBtestDlg::Purge(ULONG dwMask)
{
    if (!m_pPurge)
    {
        AfxMessageBox("FT_Purge is not valid!");
        return FT_INVALID_HANDLE;
    }

    return (*m_pPurge)(m_ftHandle, dwMask);
}



FT_STATUS CUSBtestDlg::SetTimeouts(ULONG dwReadTimeout, ULONG dwWriteTimeout)
{
    if (!m_pSetTimeouts)
    {
        AfxMessageBox("FT_SetTimeouts is not valid!");
        return FT_INVALID_HANDLE;
    }

    return (*m_pSetTimeouts)(m_ftHandle, dwReadTimeout, dwWriteTimeout);
}


FT_STATUS CUSBtestDlg::GetQueueStatus(LPDWORD lpdwAmountInRxQueue)
{
    if (!m_pGetQueueStatus)
    {
        AfxMessageBox("FT_GetQueueStatus is not valid!");
        return FT_INVALID_HANDLE;
    }

    return (*m_pGetQueueStatus)(m_ftHandle, lpdwAmountInRxQueue);
}
#pragma endregion
FT_STATUS CUSBtestDlg::OpenBy()
{
    UpdateData(TRUE);

    if(m_SerDesc==0)
        m_NameNmbr = AfxGetApp()->GetProfileString("MyUSBTestAp", "DescString", "FTDI USB ↙
    Chip.");
    if(m_SerDesc==1)
        m_NameNmbr = AfxGetApp()->GetProfileString("MyUSBTestAp", "SerialString", "FTDI   ↙
    USB Chip.");
    if(m_SerDesc==2)
        m_NameNmbr = AfxGetApp()->GetProfileString("MyUSBTestAp", "DevNmbr", "FTDI USB    ↙
    Chip.");

    if(m_NameNmbr == "")
    {
        //highlight the descriptor/serial number to draw attention...
        CEdit *pEdt;
        pEdt = (CEdit *)GetDlgItem(IDC_EDIT_NAME_NUMBER);
        pEdt->SetFocus();
        pEdt->SetSel(0, -1, FALSE);
        AfxMessageBox("You must enter a valid Brain Description or Serial Number.");
        return FT_DEVICE_NOT_OPENED;
    }

    FT_STATUS status;
    ULONG x=0;
    if(m_SerDesc==0)
        status = OpenEx((PVOID)(LPCTSTR)m_NameNmbr, FT_OPEN_BY_DESCRIPTION);
    if(m_SerDesc==1)
        status = OpenEx((PVOID)(LPCTSTR)m_NameNmbr, FT_OPEN_BY_SERIAL_NUMBER);
    if((m_SerDesc==2) || (m_SerDesc==-1))//if open by device OR no method was selected
    {
        if(m_NameNmbr.GetLength() < 1)//nothing entered - open default device 0
        {
            status = Open((PVOID)x);//load default device 0
        }
       else
       {
            if(m_NameNmbr.GetLength() > 2)//no Open() method selected...
            {
                AfxMessageBox("You must enter a valid Brain Description or Serial Number. ↙
    ");
                return FT_DEVICE_NOT_OPENED;
            }

            //convert string to decimal number and open(x)
            char str[3];
            strcpy(str, (LPCTSTR)m_NameNmbr);
            x = atoi(str);
            if((x<0) || (x>64))
            {
                AfxMessageBox("You must enter a valid Brain Description or Serial Number. ↙
    ");
                return FT_DEVICE_NOT_OPENED;
            }
            status = Open((PVOID)x);
       }
    }
```

```
    return status;
}
```

Bibliography

[1] John C. Ansel. Skin – nervous system interactions. *Dermatology Foundation*, 1:198–204, 1996.

[2] Jessica Banks. Design and control of an anthropomorphic robotic finger with multi-point tactile sensation. Master's thesis, Massachusetts Institute of Technology, 2001. 60

[3] Richard Barry. Freertos modules, 2006. 80

[4] Shekar Bhansali, H. Thurman Henderson, and Steven B. Hoath. Probing human skin as an information-rich smart biological interface using mems-based informatics. In SPIE, editor, *Proceedings of SPIE–the international society for optical engineering*, volume 4235. SPIE, 2001.

[5] Athanassios Boulis. Design and implementation of a framework for efficient and programmable sensor networks. *Proceedings of the 1st international conference on Mobile systems, applications and services*, 1:187–200, 2003. 80

[6] William Butera. *Programming a Paintable Computer.* PhD thesis, Massachusetts Institute of Technology, 2002. 24

[7] E. Catterall, K. Van Laerhoven, and M. Strohbach. Self organization in ad hoc sensor networks: An empirical study, 2002.

[8] Loren P. Clare. Multipoint sensing by intelligent collectives. In *The First Annual Symposium on Autonomous Intelligent Networks and Systems*. AINS, May 8-9 2002.

[9] Adam Dunkels. The contiki operating system 2.x, 2006. 80

[10] EFunda. Strain gage: Theoretical background. 61

[11] Jonathan Engel, Jack Chen, and Chang Liu. Development of a multi-modal, flexible tactile sensing skin using polymer micromachining. In *Proc. of The 12th International Conference on Solid State Sensors, Actuatorsand Microsystems*. IEEE, June 2003. 30, 37

[12] FTDI. Ft245bm designers guide version 2.0, 2003. 76

[13] Ken Gilleo. *Handbook of Flexible Cicuits*. Van Nostrand Reinhold, 1992. 32, 44

[14] Franz Graf. Features of the msp430 bootstrap loader, January 2003. 81

[15] Vishay Measurements Group. Shunt calibration of strain gage instrumentation. 110

[16] Vishay Measurements Group. Strain gage thermal output and gage factor variation with temperature. 114

[17] Mitsuhiro HAKOZAKI, Atsushi HATORI, and Hiroyuki SHINODA. A sensitive skin using wireless tactile sensing elements. In *TECHNICAL DIGEST OF THE 18TH SENSOR SYMPOSIUM*, 2001. 28

[18] M.B. Haris. New design of robotic tactile sensor using dynamic wafer technology based on vlsi technique. In *Biomedical Research in 2001*, 2001. 44

[19] Charles A. Harper. *High Performance Printed Circuit Boards*. McGraw-Hill, 1999. 44

[20] Falke M. Hendriks. *Mechanical behaviour of human epidermal and dermal layers in vivo*. Technische Universiteit Eindhoven,, 2005. 17

[21] Robert D. Howe. Tactile sensing and control of robotic manipulation. *Journal of Robotic Manipulation*, 8:245–61, 1994.

[22] Dimitris Hristu. The performance of a deformable-membrane tactile-sensor: basic results on geometrically-defined tasks. *IEEE Internationall Conference on Robotics and Automation*, 1:508–513, 2000. 28, 60

[23] Vishay Inc. Strain gage selection. 45

[24] Vishay Inc. Strain gage selection criteria procedures recommendations. 63

[25] National Instruments. Lm20 temperature sensor datasheet, October 2005. 125

[26] Texas Instruments. MSP430F1611 Datasheet. 56

[27] Texas Instruments. *MSP430Fx1xx User's Guide*, 2006. 56

[28] Future Technology Devices International. D2xx programmer's guide, 2006. 93, 104

[29] IPC. Adhesive coated dielectric films for use as cover sheets for flexible printed circuitry and flexible adhesive bonding films. Technical report, IPC, 2002. 46

[30] IPC. Flexible metal-clad dielectrics for use in fabrication of flexible printed circuitry. Technical report, IPC, 2002. 46

[31] IPC. Sectional design standard for rigid organic printed boards. Technical report, IPC, 2002. 51

[32] T.J.C. Jacob, C.S. Fraser, L. Wang, V.E. Walker, and S. O'Connor. Psychophysical evaluation of responses to pleasant and mal-odour stimulation in human subjects; adaptation, dose response and gender differences. *International Journal of Psychophysiology*, 48:67–80, 2003. 42

[33] Joseph Jacobson, B Comiskey, C Turner, J Albert, and P Tsao. The last book. *IBM Systems Journal*, 36:457–463, 1997. 34

[34] Rodolphe Koehly, Denis Curtil, and Marcelo M. Wanderley. Paper fsrs and latex/fabric traction sensors: Methods for the development of home.made touch sensors. In *Proceedings of the 2006 International Conference on New Interfaces for Musical Expression*, 2006. 61

[35] Stéphanie P. Lacour. Design and performance of thin metal film interconnects for skin-like electronic circuits. *IEEE Electron Device Letters*, 25:179–181, 2004. 34, 44

[36] PhD Laura E. Edsberg, PhD Robert E. Mates, PhD Robert E. Baier, and ME Mark Lauren. Mechanical characteristics of human skin subjected to static versus cyclic normal pressures. *Journal of Rehabilitation Research and Development*, 36:1–5, 1999. 17

[37] M.H. Lee and H.R. Nicholls. Tactile sensing for mechatronics – a state of the art survey. *Mechatronics*, 9:1–31, 1999. 28

[38] Philip Levis. Tinyos 2.0 overview. 80

[39] Joshua H. Lifton. Pushpin computing: a platform for distributed sensor networks. Master's thesis, Massachusetts Institute of Technology, 2002. 26, 70

[40] Joshua H. Lifton. Pushpin computing system overview: A platform for distributed, embedded, ubiquitous sensor networks. *Proceedings of the First International Conference on Pervasive Computing*, 1:139–151, 2002. 26, 80, 90

[41] Joshua H. Lifton. Distributed sensor networks as sensate skin. In *Sensors, 2003. Proceedings of IEEE*, 2003. 25, 69

[42] D. Um; V. Lumelsky. Fault tolerance via component redundancy for a modularized sensitive skin. In *Proceedings of the 1999 IEEE International Conference on Robotics & Automation*, 1999.

[43] Vladimir J. Lumelsky. Sensitive skin. *IEEE Sensors Journal*, 1:41–51, 2001. 18, 28, 37

[44] Sergio Maximilian. A textile based capacitive pressure sensor. *Sensor Letters*, 2:153–160, 2004. 30, 60

[45] Claudio Melchiorri. Tactile sensing for robotic manipulation. *The International Journal of Robotics Research*, 0:25–36, 1990. 28

[46] Minco. Flex-circuit design guide. 11, 34, 47, 49

[47] Minco. Flex circuits. web, December 2002. 44

[48] Omega. The strain gage. 62

[49] T.V. Papakostas. A large area force sensor for smart skin applications. In *Proceedings of the 1st IEEE Sensors Conference*, 2002. 30, 44

[50] Joseph A. Paradiso, Joshua Lifton, and Michael Broxton. Sensate media multimodal electronic skins as dense sensor networks. *BT Technology Journal*, 22:32–44, 2004. 18

[51] Joseph A. Paradiso, Joshua Lifton, and Michael Broxton. Sensate media: Multimodal electronic skins as dense sensor networks. *BT Technology Journal*, 22:32–44, 2004.

[52] Peratech. 11, 66

[53] Peratech. Integration guide. 67

[54] Philips. The i2c-bus specification. Technical report, Philips, January 2000. 40

[55] Philips. The i2c-bus and how to use it (including specifications), April 2001. 134

[56] Seppo Pohja. Survey of studies on tactile senses. Technical report, European Research Consortium for Informatics and Mathematics at SICS, 2001. 18, 59

[57] Niels Reijers. Efficient code distribution in wireless sensor networks. *Proceedings of the 2nd ACM international conference on Wireless sensor networks and applications*, 2:60–67, 2003. 80

[58] Jun Rekimoto. Smartskin: an infrastructure for freehand manipulation on interactive surfaces. In *Proceedings of the SIGCHI conference on Human factors in computing systems: Changing our world, changing ourselves*, 2002. 28

[59] Bruce Richardson, Krispin Leydon, Mikael Fernstrom, and Joseph A. Paradiso. Z-tiles: building blocks for modular, pressure-sensing floorspaces. In *CHI '04: CHI '04 extended abstracts on Human factors in computing systems*, 2004. 132

[60] Doug Richardson. Smart skins and structures. *Armada International*, 22:50–56, 1998.

[61] D. De Rossi. Electroactive fabrics for distributed, conformable and interactive systems. *Sensors, 2002. Proceedings of IEEE*, 2:1608–1613, 2002. 60

[62] Takao Someya. Integration of organic field-effect transistors and rubbery pressure sensors for artificial skin applications. In *Electron Devices Meeting IEDM '03 Technical Digest.*, 2003. 30

[63] Measurement Specialties. Piezo film sensors technical manual, 1999. 70

[64] Thomas H. Stearns. *Flexible Printed Circuitry*. McGraw-Hill, 1995. 44

[65] W.D. Stiehl, Lalla L., and Cynthia. Breazeal. A "somatic alphabet" approach to "sensitive skin". In *Robotics and Automation, 2004. Proceedings. ICRA '04. 2004 IEEE International Conference on*, 2004. 26

[66] Johan Tegin and Jan Wikander. Tactile sensing in intelligent robotic manipulation. *The Industrial robot*, 32:64–70, 2005.

[67] Toshiba. Tps851 datasheet, 2004. 71

[68] Vilis Tutis. The physiology of the senses transformations for perception and action. web. 42

[69] D. Um, B. Stankovic, K. Giles, T. Hammond, and V. Lumelsky. A modularized sensitive skin for motion planning in uncertain environments. In *Proceedings of the 1998 IEEE International Conference on Robotics & Automation*, 1998. 28, 37, 49

[70] Shawna Vogel. Smart skin. *Discover*, 11:26, 1990.

[71] Ravindra Wijesiriwardana. Inductive fiber-meshed strain and displacement transducers for respiratory measuring systems and motion capturing systems. *IEEE SENSORS JOURNAL,*, 6:571–579, 2006. 60

[72] Adrian C Williams. *Transdermal and Topical Drug Delivery*. Pharmaceutical Press, 2003. 17

[73] Tom Woznicki. Flex circuits news. 46

[74] Yong Xu, Yu-Chong Tai, Adam Huang, and Chih-Ming Ho. Ic-integrated flexible shear-stress sensor skin. In *Solid-State Sensor, Actuator and Microsystems Workshop*, 2002. 30

[75] Kouichi Yamada, Kenji Goto, Yoshiki Nakajima, and Nobuyoshi Koshida. A sensor skin using wire-free tactile sensing elements based on optical connection. *SICE -ANNUAL CONFERENCE*, 41:131–134, 1999. 28, 60

[76] Yamada Yoji, Takashi Maeno, Isao Fujimoto, Tetsuya Morizono, and Yoji Umetani. Identification of incipient slip phenomena based on the circuit output signals of pvdf film strips embedded in artificial finger ridges. *TRANSACTIONS - Society of Instruments and Control Engineers*, 40:648–655, 2004. 60