The Creative Sensor Network

A wireless sensor network capable of sending OSC Messages to creative applications

AALBORG UNIVERSITY

Department of Architecture,
Design & Media Technology
Niels Jernes Vej 14
www.create.aau.dk

Title:

The Creative Sensor Network

Description:

A wireless sensor network capable of sending OSC Messages to creative applications.

Theme:

Master's Thesis

Project period:

10th semester:
February 8th 2011 - May 31st 2011

Participant:

Tobias Thyrrestrup

Supervisors:

Dan Overholt

Esben Skouboe Poulsen

Abstract:

The goal of the project is to create a wireless sensor network capable of sending OSC Messages directly into a number of creative applications: Processing, Grasshopper, openFrameworks, Max/MSP, LuaAV, and Quartz Composer. This gives designers the option to inform their designs based on the context in which they will be placed. But also to give designers an understanding of the environment their designing for. The product developed for this report is referred to as the Creative Sensor Network. The evaluation of the product shows that there is little difficulty implementing the product in anyone of the creative applications and there is an interest in the product from people with a design background. The product has also been successfully implemented in a research project focusing on responsive architecture.

Publications: 5

Number of Pages: 138

Appendices: A - H

Finished: May 31st 2011

Preface

Formalities

Sources are referred to by ["author's surname", "year the text is written"], the literature list is drawn up with the authors' surnames in alphabetical order. If the source is a webpage for an organization or a company the name of the organization or company is used as the reference.

A presentation of the product is recorded on video, and video material is available on a CD attached Appendix H.

A digital version of the report, the product source code, and copies of Internet sources are also available on the CD attached in Appendix H.

Acknowledgments

The project has been developed in a running exchange of information with Electrotexture Lab. who has contributed with information, inspiration, and provided space and equipment for the development of the product.

A big thanks to Esben Skouboe Poulsen and Mads Brath Jensen for inspiring the project and for providing me with expertise and appropriate criticism to drive the project forward.

Also a big thanks to Dan Overholt for guidance and clarification, and to Andreas Eggertsen and Isak Worre Foged for showing an interest in the project.

Contents

1. Introduction

> *Recently we have witnessed a paradigm shift from cyberspace to pervasive computing. Instead of pulling us through the looking glass into some sterile, luminous world, digital technology now pours out beyond the screen, into our messy places, under our laws of physics; it is built into our rooms, embedded in our props and devices – everywhere* [McCullough, 2005].

This paradigm shift is interesting because it opens up a hole new world of possibilities both in interaction design and in architecture. The prospect of computers and sensors embedded everywhere could potentially produce very precise information about the spaces we inhabit and the way we interact with these same spaces. These high resolution "images" of our surroundings could future the creation of stimulating and appealing environments - perhaps even intelligent?

The more obvious facts obtained from such images could be the amount of electricity or water used in a building, wether or not the light is turned on or off in a room with no people, but it could also create an understanding of space, if one was to say that a space is interesting and fulfilling if people use the space, then factors like light intensity and color, air quality, movement in the space, and temperature could be recorded and re-used in the design process when creating new spaces.

A shift in design strategy towards using dynamic parametric design tools in the process of creating new designs has already taken place. But what if the parameters fed into these tools where directly informed by our surroundings? How is the information made available in a design context? and what are the technical challenges related to gathering the data in the first place?

These are some of the key questions if this scenario is to become part of the design process. Also a real-time interface connecting the parametric design tools to the embedded sensors is crucial. This poses the following question:

How can designers exploit the advantage of embedding computers and sensors everywhere?

2. Analysis

To tackle the question asked in Chapter 1 a few aspects need further looking into. First, ubiquitous computing is introduced and practical implementations of the concept are presented. Second, commercially available sensor networks capable of communicating sensor values back to a computer are presented. Third, creative applications capable of facilitating dynamic parametric design are presented and examined for their ability to accept input from external hardware, and last the problem statement is formulated.

2.1 Ubiquitous Computing

Ubiquitous computing also known as pervasive computing is the notion of microscopic computers embedded in everyday things, all connected on a giant network [McCullough, 2005].

Tiny computers fitted with sensors capable of data-processing constitute the network, the computers are commonly referred to as "nodes". Nodes share the information they collect with other nodes and this way produce meaningful digital representations of the environment in which they are situated. Researchers could program the network to analyze data based on specific research questions and this way produce results faster, letting the network gather and process detailed data in real-time [Butler, 2006].

The scenario of ubiquitous computing described relies on computers being extremely small, this is not yet the case. As an example the smallest microcontroller available to the general public from Atmel is a 2x2 millimeter chip [Atmel, 2011]. Although this is impressive, there is also a need for supporting circuitry and a power source, therefore it is still not small enough to be embedded everywhere. As a result a more realistic approach to ubiquitous computing needs to be taken.

The article "The Internet of Things" published by Scientific American Inc. tackles ubiquitous computing in a more practical manner. The data protocol developed during the evolution of the Internet is adopted and extended to networks of all types of devices, inter-device internetworking. This opens the possibility of connecting e.g. light bulbs, switches, alarm clocks, coffee makers, air conditioners etc. together to create "smart spaces" utilizing the Internet as the network between nodes [Gershenfeld *et al.*, 2004]. The article describes a system standardizing the communication between house hold appliances through the

Internet data protocol but more or less leaves out the aspect of sensing.

Another example where the sensing aspect is crucial is the "Japan Geigermap" where crowd-sourced radiation geiger readings are plotted on a Google map. After the disaster at the Fukushima Daiichi power plant people from all over Japan have built their own geiger sensors and upload the readings to an online real-time sensor network - Pachube. This provides everybody with a high resolution image of the radiation levels through out Japan [Zhang, 2011]. Figure 2.1 shows the sensor readings on the Japan Geiger map.

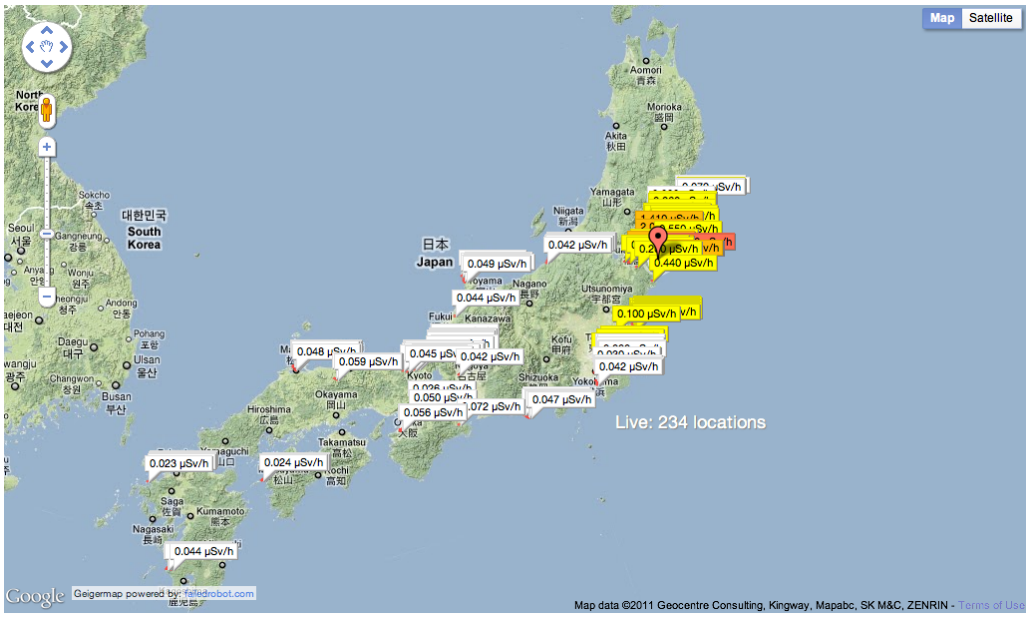


Figure 2.1: Overview of the Japan Geigermap [Zhang, 2011].

2.2 Sensor Networks

This section describes a set of commercially available wireless sensor network solutions capable of sensing the environment and relaying information back to a computer.

2.2.1 Libelium - Waspmote

The Libelium wireless sensor node - Waspmote is designed around the ATmega1281 microprocessor from Atmel. The board has 7 analog inputs and 8 digital input/output ports, 1 PWM output, 2 UART serial interfaces, 1 I^2C serial interface, and 1 USB port. The board also features 2 built-in sensors: a temperature sensor and an accelerometer. The wireless network is built around the ZigBee protocol and the XBee wireless module capable of providing different network topologies: peer-2-peer, tree and mesh. The board is ready for plug and play installation of the following sensor modules: Gases Board, Event Detection

Board, Prototyping Board, Agriculture Board, Smart Metering Board [Libelium, 2011a]. Figure 2.2 shows the Waspmote sensor module.

Figure 2.2: Libelium Waspmote [Libelium, 2011a].

A Waspmote Gateway is provided for wireless communication with the "motes" from a computer. The gateway has a USB interface for relaying the serial data from the XBee module back to a computer. Libelium also provides a networked gateway - Meshlium for logging data to a file or an internal MySQL database on the Meshlium itself or an external MySQL database. The data is then accessible to a computer capable of querying the database [Libelium, 2011b].

2.2.2 NEWPORT - Wireless Sensor System

The NEWPORT wireless sensor system is comprised of 1 - 32 End Devices communicating with a Coordinator over 2.4GHz ZigBee wireless network. The Coordinator is connected to an Ethernet network and the Internet. A number of different static sensor configurations are available for the End Devices combining the following sensors: temperature, humidity and barometric pressure. A web-based interface let users monitor sensor values without needing anything but a standard web-browser [Newport, 2011b]. Figure 2.3 shows the web-based interface, an End Device and a Coordinator.

NEWPORT offers a program for logging data to Excel or Visual Basic and an OPC Server software that integrates with popular data acquisition and automation applications. The Coordinater supports communication over TCP [Newport, 2011b].

2.2.3 MEMSIC - eKo Outdoor Wireless System

The eKo outdoor wireless system consist of one eKo Gateway, one eKo Base Radio and a number of eKo Nodes forming a wireless mesh network over the 2.4GHz ZigBee protocol.

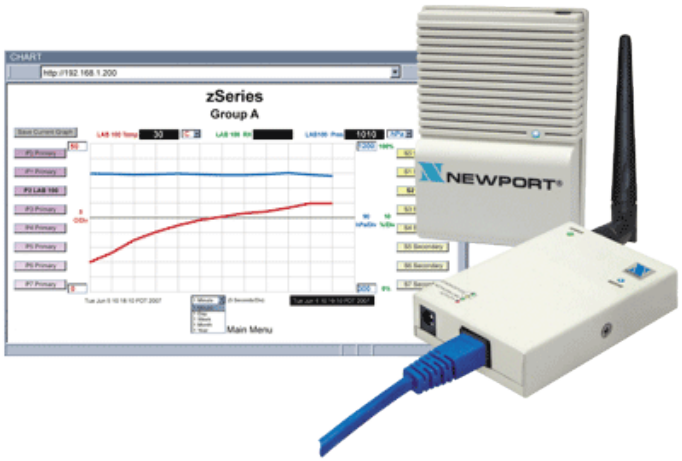


Figure 2.3: NEWPORT Wireless Sensor System [Newport, 2011a].

Each node has a battery, a solar cell for charging the battery, and 4 ports for connecting eKo compatible sensors supporting a vast number of integrated sensor devices. The eKo Gateway contains data visualization software packages, eKo View - a web-based interface and Xserve. The eKo Base Radio connects the eKo Nodes wirelessly to the eKo Gateway [Memsic, 2011]. Figure 2.4 shows the eKo Nodes, eKo Gateway and the eKo Base Radio.

Figure 2.4: MEMSIC eKo Outdoor Wireless Sensor System [CMT, 2011]

Xserve is an application running on the eKo Gateway and it makes sure that the data from the wireless sensor network can be received by a range of applications through standard XML over the network using TCP [Crossbow, 2011].

2.2.4 Summary

The sensor systems described in Section 2.2 all have a wireless interface using the 2.4 GHz ZigBee protocol. The Waspmote and the eKo Node can use a wide variety of sensors while the NEWPORT system has a fixed set of configurations only incorporating temperature, humidity and pressure. All of the systems implement different interfaces for sending data back to a computer. The eKo Gateway is using the standardized eXtensive Markup Language (XML) and the Libelium Meshlium uses a standardized database solution - MySQL whereas the NEWPORT system implements its own message format for sending packages over TCP.

2.3 Creative Applications

This section describes design tools that facilitate the creative process. A combination of ordinary integrated development environments (IDE), visual programming environments, plug-ins, libraries etc. all supporting dynamic parametric design. Furthermore the applications are examined for their ability to connect to external devices to determine if a common interface across all applications can be found.

2.3.1 Processing

Processing was initially developed to give artists and designers a tool to "sketch" ideas in code, but has since evolved into a tool for creating production-level work. Processing is based on Java and provides its users with instant feedback through its programing environment [Processing, 2011b]. Figure 2.5 shows the user interface of Processing.

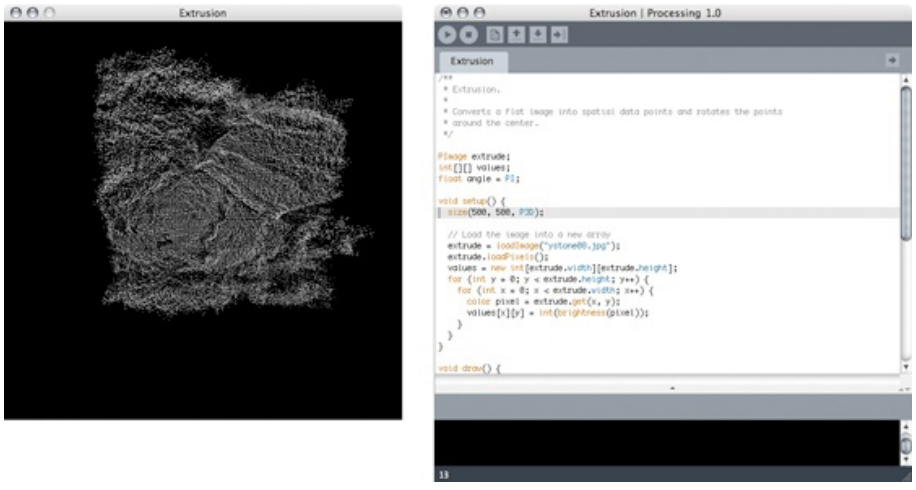


Figure 2.5: Overview of Processing IDE and the display window [Processing, 2011c].

Processing supports the following inputs from external hardware natively: Serial, and

through the use of libraries like bluetoothDesktop, UDP, oscP5, TUIOClient and proMidi: Bluetooth, UDP, OSC, TUIO, Midi [Processing, 2011a].

2.3.2 Rhinoceros - Grasshopper

Rhinoceros is a 3D-modeling tool for creating accurate models for rendering, engineering, manufacturing and construction. Rhinoceros also supports plug-ins from third-party developers [McNeel, 2011]. Grasshopper is a plug-in for Rhinoceros which provides designers with a visual programming language for experimenting with new shapes using generative algorithms. Grasshopper does not require the user to have any knowledge of programming or scripting [Grasshopper, 2011]. Figure 2.6 shows the Grasshopper user interface alongside the Rhinoceros user interface.

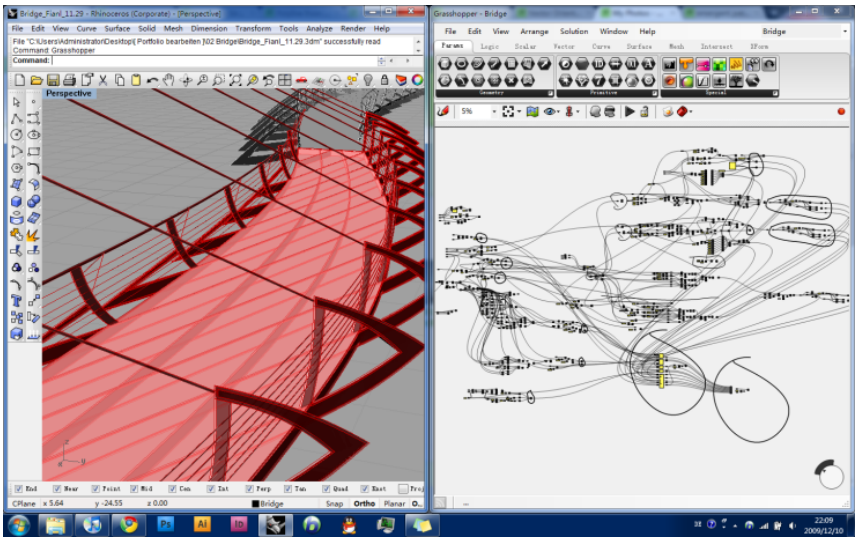


Figure 2.6: Overview of Rhinoceros using the Grasshopper plug-in [Li, 2011].

Grasshopper supports the following inputs from external hardware using the Firefly [Firefly, 2011] and gHowl [gHowl, 2011] components: Serial, UDP, OSC, TUIO.

2.3.3 openFrameworks

openFrameworks is a compilation of C++ libraries facilitating the creative and artistic process by providing a simple framework for developing audio-visual experiments. Libraries are wrapped together using a consistent interface for simplicity [openFrameworks, 2011a]. Figure 2.7 shows an example of the openFrameworks workspace.

openFrameworks support the following inputs from external hardware sources natively: Serial, and through the use of the packaged addons ofxNetwork and ofxOSC: TCP, UDP, OSC [openFrameworks, 2011b].

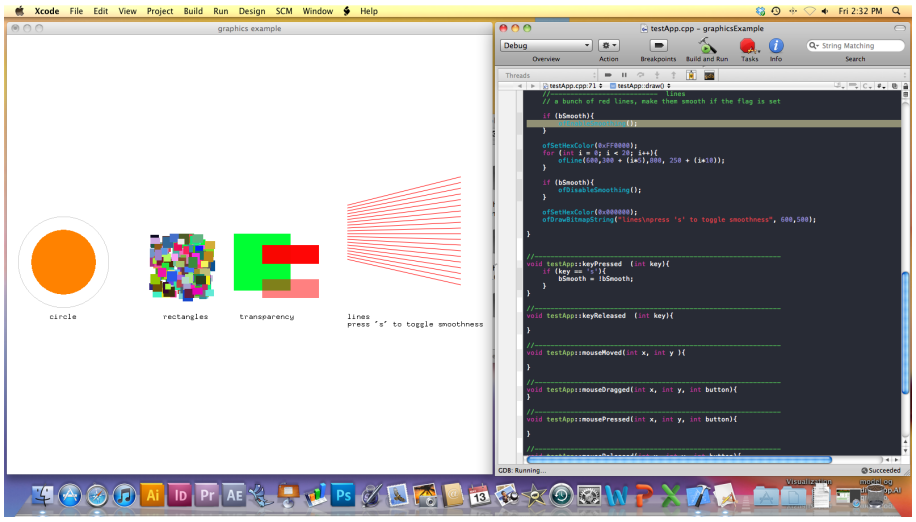


Figure 2.7: Overview of openFrameworks graphics example alongside the Xcode IDE.

2.3.4 Max/MSP

Max/MSP provides the user with a graphical programming environment for media related programming, and it is used by artists and scientists alike. *Max* provides user interface, timing and communication. *MSP* provides real-time audio synthesis and digital signal processing [Cycling74, 2011b].

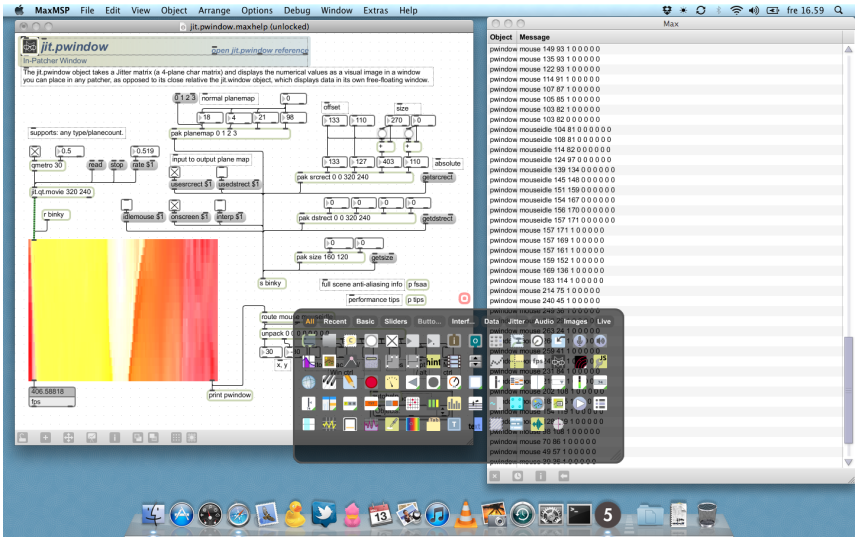


Figure 2.8: Overview of the Max/MSP visual programming environment.

Max/MSP supports the following inputs from external hardware natively: HID, Serial, OSC [Cycling74, 2011a], and through the use of the *mxj* object: TCP, UDP.

2.3.5 LuaAV

LuaAV is a real-time creative scripting environment for working with sound, image, space and time. It is based on the Lua scripting language bundled with libraries for sound graphics and media protocols. The main purpose of LuaAV is to provide the user with a tool to turn any creative thought into an experiment without thinking about the technology [LuaAV, 2011b]. Figure 2.9 shows the user interface for LuaAV.

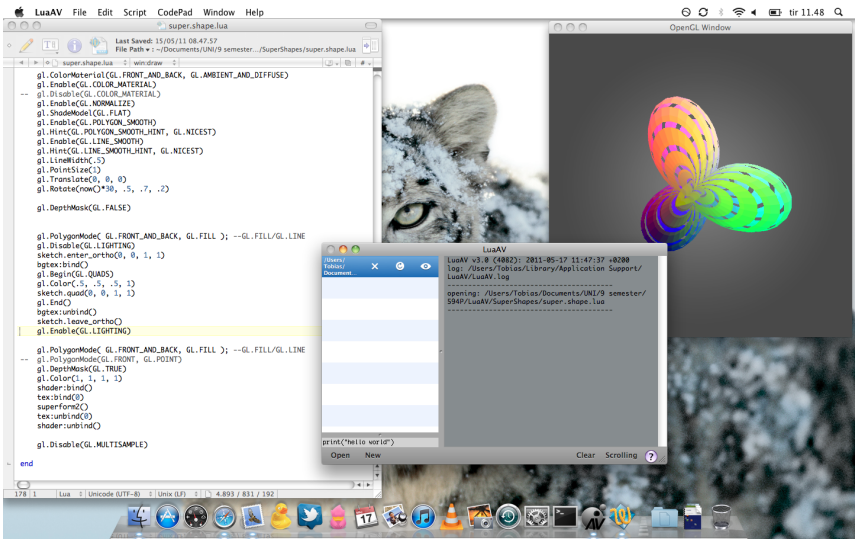


Figure 2.9: Overview of LuaAV, the OpenGL Window, and a scripting editor.

LuaAV supports the following inputs from externals hardware: Midi, OSC [LuaAV, 2011a].

2.3.6 Quartz Composer

Quartz Composer combines all the technologies of Mac OS X into a real-time visual programming environment which instantly provides the user with visual feedback. No need for time consuming rendering, greatly reducing the time spent developing [Quartz, 2011]. Figure 2.10 shows the user interface for Quartz Composer.

Quartz Composer supports the following inputs from external hardware: HID, UDP, OSC.

2.3.7 Summary

Table 2.1 summarizes the input capabilities of the creative applications described in Section 2.3 moreover it reveals that OSC is the only protocol which is supported by all applications. This concludes that an external device must implement the OSC protocol to be able to transmit information to anyone of the creative applications described in this section.

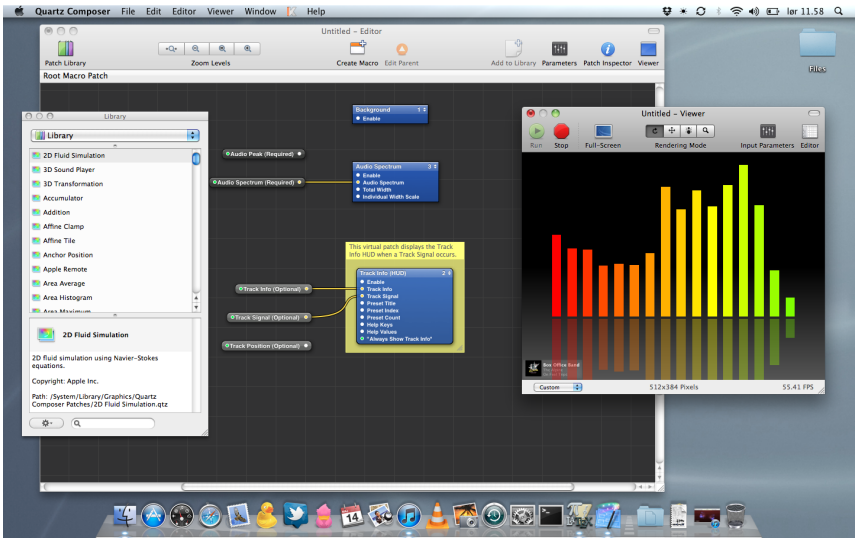


Figure 2.10: Overview of the Quartz Composer editor, viewer and objects browser.

Application	Serial	Bluetooth	HID	TCP	UDP	OSC	TUIO	Midi
Processing	x	x			x	x	x	x
Grasshopper	x				x	x	x	
openFrameworks	x			x	x	x		
Max/MSP	x		x	x	x	x		x
LuaAV						x		x
Quartz Composer			x	x	x	x		

Table 2.1: Comparison chart showing the input capabilities of each application.

2.4 Problem statement

Ubiquitous computing in its purest form with computers embedded everywhere is still a thing of the future but embedding microprocessors in everyday things and deploying sensors almost everywhere is possible as seen in the examples given in Section 2.1. Data acquisition is possible with the commercially available wireless sensor networks but they all implement different ways of communicating with a computer, some standardized and some not, on top of that extra software is needed in some cases to interpret the sensor information and relay it to a creative application. This makes it difficult to use the data directly in the creative applications as seen in the summary in Section 2.3 where only a few applications support the protocols used by the sensor systems furthermore the summary shows that all applications implements an interface for OSC communication based on this the problem statement is formulated as the following question:

How to develop and implement a wireless sensor system capable of sending OSC messages directly into creative applications?

3. Concept

This chapter presents the overall concept of the product for this report - "The Creative Sensor Network". It also presents two use cases, one where the finished product assists in the design process, and another where the product is used to inform the structure of a finished design.

3.1 The Creative Sensor Network

The concept of the product is comprised of a number of wireless sensor "Nodes" and a "Base" connected to an Ethernet network. The Nodes are capable of reading sensor values from a number of attached sensors e.g. light intensity, temperature, humidity etc. and transmit the readings to the Base. The Base handles the conversion of sensor values into a standardized OSC format. The packet is then broadcasted on the network where any creative application can receive the information directly and the user can implement the information in a design process. The user should also have the ability to send a discover message to a certain Node to make it flash its light for easy identification of Nodes. Figure 3.1 shows an overview of the Creative Sensor Network concept.

For ease of use the connection between the Nodes and the Base should be handled automatically without the need for the user to configure or setup the Nodes or the Base. The Nodes will be battery powered to maintain wireless capabilities. To save on power consumption the Nodes therefore need to implement a "sleep" state which take their power consumption to a minimum when idle.

3.1.1 Ease of Use

The setup procedure of the system is intended to be simple. First the Base is connected to the local network then power is applied to the Base. When the Light Emitting Diode (LED) turns on it signals that the device is powered and once the Base is ready to accept messages from nodes it signals the user by turning on a different LED. The sensors are then attached to the Nodes and power is applied to the Nodes. When the LED on a Node turns on it signals that the device is powered, after an initialization period where the Node connects to the Base the LED turns of letting the user know the connection has been made.

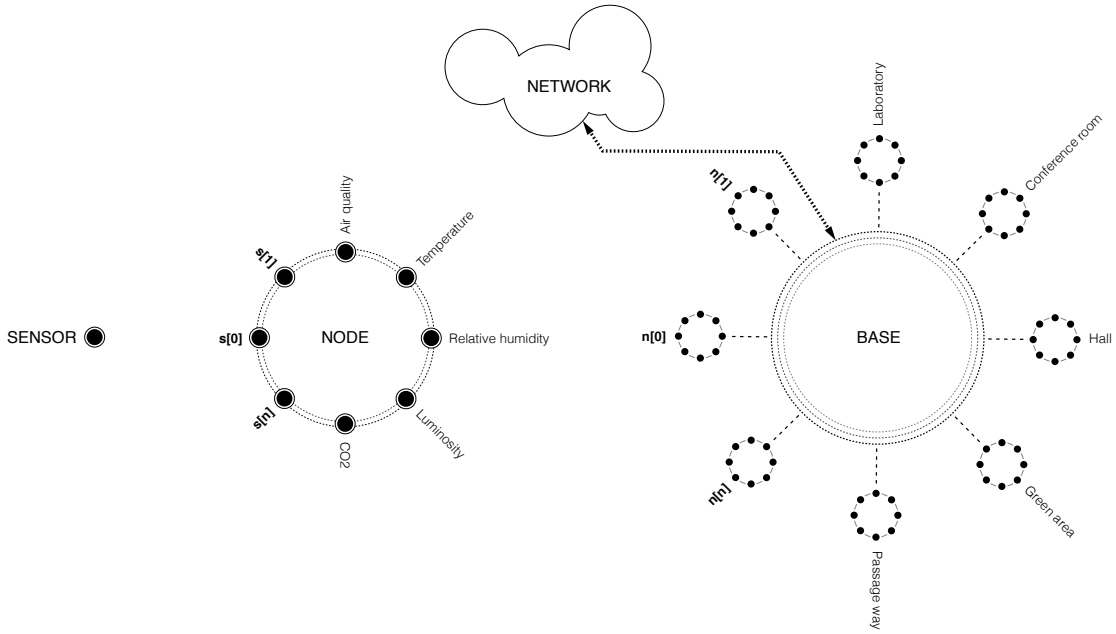


Figure 3.1: Overview of the Creative Sensor Network.

The LED on a Node blinks each time the sensors are read and the value is transmitted to the Base.

3.2 Use Cases

This section describes two use cases where the features of the Creative Sensor Network could be used: First, in a development process of a new design and second, in the everyday control of a design.

First Use Case

In the development process designers and architects look to the context of the place where their creation is to be situated e.g. looking into the history of a place, the people who use the space and so on. Incorporating the Creative Sensor Network in the design process could expand the knowledge of the context by supplying designers with information about the temperature changes, light intensities, humidity, air quality etc. through out the space. This provides designers with a high resolution "image" of the context they are about to change with their creation.

Second Use Case

A dynamic building structure is created capable of changing elements of a building based on different external factors e.g. the facade opens up and lets more light into the building if the light intensity or the temperature is to low, or a building evolves over time incorporating the changes in the environment into the building structure - like seashells where changes in the patterns on the shell is evidence of changes in the mineral composition of the water - this way the history of a building can be deciphered from the patterns of the structure.

3.3 Part Conclusion

The concept of the Creative Sensor Network has been presented and is comprised of a Base with the ability to send OSC formatted messages, and a number of Nodes capable of wirelessly sending sensor readings to the Base. The setup procedure has been outlined and is kept at a minimum for ease of use, only requiring the user to plug in sensors, connect power, and connect a network cable. Two use cases for the finished product have also been introduced, one where the product is used in the design process and another where the product is used to inform a structure.

4. Hardware Technologies

This chapter describes the different hardware technologies the Creative Sensor Network is composed of. First, the OSC protocol is examined as it needs to be supported by the hardware. Second, the microcontroller is introduced and features listed. Third, wireless networking technologies are presented. Fourth, the Ethernet networking technology is presented. Fifth, the components for the Creative Sensor Network are presented, and last the Serial Peripheral Interface (SPI) is examined as it will be the interface for connecting the components.

4.1 Open Sound Control

OSC is developed as a means of communicating between computers, sound synthesizers and other multimedia devices. OSC was originally designed as a successor for the Midi protocol using network communication instead of the more low-level approach Midi uses, but has since been implemented as a method of communicating between a range of different software and hardware applications. OSC is mainly communicating over the User Datagram Protocol (UDP) but could just as well be used over any other protocol [Noble, 2009].

OSC is a formal way of communicating over e.g. UDP as it defines how text and numbers should be transmitted such that they will be recognized by the receiving part. The following is a subset of the atomic data types defined by the OSC syntax:

- **int32** - 32-bit big-endian two's compliment integer.
- **float32** - 32-bit big-endian IEEE 754 floating point number.
- **OSC-string** - A sequence of non-null ASCII characters terminated by a null, followed by 0-3 additional null characters to make the total number of bytes a multiple of four.

An OSC Packet consist of a number of bytes containing its content, the count of bytes is always a multiple of four. The content of the OSC Packet is an OSC Message which in turn is comprised of an OSC Address Pattern, an OSC Type Tag String followed by any number of OSC Arguments [Wright, 2011].

- OSC Packet
  - OSC Message
    1. OSC Address Pattern
    2. OSC Type Tag String
    3. OSC Arguments

An OSC Address Pattern is an OSC-string starting with the character `'/'` (forward slash) followed by a sequence of characters defining the address:

```
"/osc/address"
```

An OSC Type Tag String is a OSC-string starting with the character `','` (comma) followed by a sequence of characters corresponding to the sequence of OSC Arguments in the message. Each of the characters after the comma is called a OSC Type Tag and represents the corresponding OSC Argument. Table 4.1 shows a subset of OSC Type Tags.

OSC Type Tag	**OSC Argument**
i	int32
f	float32
s	OSC-string

Table 4.1: OSC Type Tag definitions.

The OSC Type Tag String for sending two integers is as follows:

```
",ii"
```

The OSC Argument is any number of atomic data types represented in binary corresponding to the OSC Type Tag String. A simplification of a complete OSC Packet could look something like the following:

```
"/osc/address ,ii 1234 4321"
```

The correct representation of the OSC Packet is shown in Table 4.2, each byte is shown by its hex value and the bytes are separated in multiples of four:

```
0x2F (/)  0x6F (o)  0x73 (s)  0x63 (c)
0x2F (/)  0x61 (a)  0x64 (d)  0x64 (d)
0x72 (r)  0x65 (e)  0x73 (s)  0x73 (s)
0x00 ()   0x00 ()   0x00 ()   0x00 ()
0x2C (,)  0x69 (i)  0x69 (i)  0x00 ()
0x00 ()   0x00 ()   0x04 ()   0xD2 ()
0x00 ()   0x00 ()   0x10 ()   0xE1 ()
```

Table 4.2: OSC Packet represented by hex values.

4.2 Microcontrollers

A microcontroller is a miniature "computer" on a single integrated circuit or chip. It contains a processor, memory, and a number of Input/Output (I/O) pins. Digital I/O channels are the most common also called General Purpose I/O (GPIO). GPIOs are configurable by software as either digital input or digital output. In digital input mode they may be used to read the state of e.g. a button. In digital output mode they can be used to turn on or off e.g. a light or a motor. Most microcontrollers also have Analog-to-Digital Converter (ADC) input pins capable of converting an analog voltage signal into a digital representation of the signal. This enables a microcontroller to read and process values from a range of sensors e.g. vibration, acceleration etc. A serial port is available on some microcontrollers. It can be used to interface the microcontroller to a computer or another microcontroller. Specialized forms of serial interfaces include Serial Peripheral Interface (SPI) or Inter-Integrated Circuit (I^2C) these provide functionality for connecting to peripherals e.g. external memory, ethernet interfaces, wireless modules etc. [Catsoulis, 2005].

The microcontroller for the Base needs to support the following features:

- GPIO pins for controlling status leds and turning peripherals on or off.
- Serial port for sending debug messages to a computer.
- Two interfaces for connecting a wireless transceiver and an Ethernet peripheral.
- Support for 32-bit integers as specified by the OSC specification.

The microcontroller for the sensor Node needs to support the following features:

- GPIO pins for controlling status leds and turning peripherals on or off.
- Serial port for sending debug messages to a computer.
- One interface for connecting a wireless transceiver peripheral.
- A number of ADC enabled input pins for reading sensor values.

On top of the features already described it is crucial that the microcontroller for the Nodes support power management modes as the Nodes will be battery powered and good power management will increase battery life.

4.3 Wireless Networking

The microcontroller needs a wireless module to communicate wirelessly with other microcontrollers, in this case Nodes communicating sensor values to the Base.

A wireless module is comprised of a transceiver peripheral, and an antenna. The microcontroller sends messages to the transceiver over a serial interface e.g. SPI. The transceiver is then responsible for generating the radio waves for transmitting the message through the air.

A wide variety of transceiver modules are available e.g. Bluetooth, WIFI and ZigBee transceivers all communicating in the 2.4 GHz frequency band [Igoe, 2007], but also sub-GHz transceivers exists for communication in the 300 - 900 MHz frequency band. The sub-GHz transceivers often have a wider range at the same power usage because of their lower frequency, but also lower data rates than the 2.4 GHz modules.

The most common protocol for wireless sensor networks is the ZigBee protocol [Faludi, 2007], this is partly because it incorporates unique addressing of modules as-well as power saving options and security. These features are commonly implemented in the hardware layer of ZigBee compatible transceivers whereas other transceivers need these features implemented in software.

The wireless module for both the Base and the Nodes needs to support the following features:

- Unique addressing.
- Power saving options.
- Serial interface

4.4 Ethernet Networking

The microcontroller needs an Ethernet module to transmit OSC Messages to computers on the network. Ethernet is a local-area networking standard developed at Xerox PARC in the early 1970s. The Ethernet module consist of an Ethernet controller peripheral and an isolation transformer. The isolation transformer is responsible for isolating the microcontroller circuit from the rest of the Ethernet network.

Adding an Ethernet module opens up a lot of possibilities e.g. access to file servers, databases, and even the Internet. Other options include monitoring the microcontroller from afar using a web interface or have it send emails notifying about changes or errors. The most interesting option is the ability to send data to a computer at high speeds [Catsoulis, 2005].

There is a number of different options with regards to the speed of an Ethernet interface: 10 Mbps, 100 Mbps, 1Gbps. There are two reasons for choosing the 10 Mbps option - The microcontroller will not have to transfer large amounts of data rather it will transfer many small packages containing the sensor readings. Another reason is the fact that higher-speed implementations require special attention to the circuit design because of electromagnetic interference [Catsoulis, 2005].

The Ethernet module for the Base needs to support the following features:

- 10 Mbps interface.
- Serial interface

4.5 Connection diagram

Figure 4.1 shows a connection diagram of the different hardware technologies comprising the Creative Sensor Network.

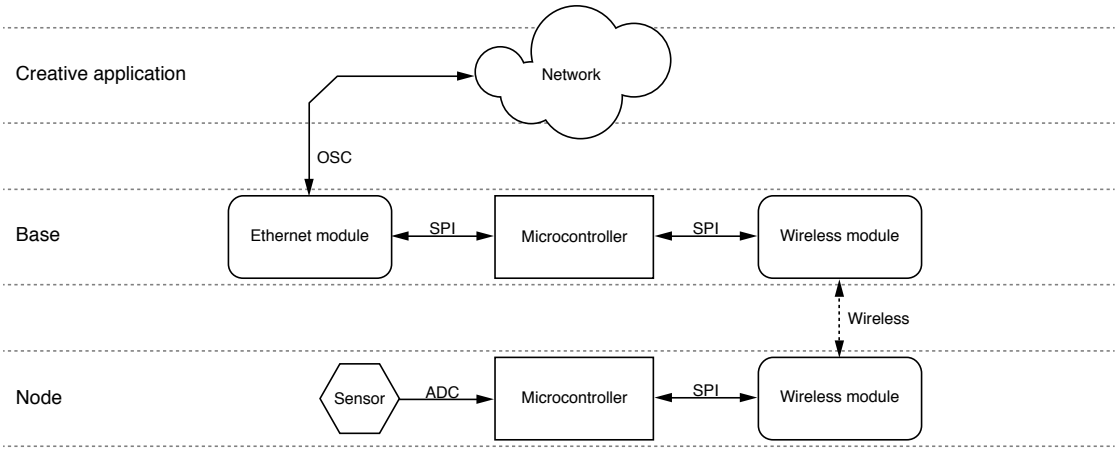


Figure 4.1: Hardware connection diagram.

First a sensor value is read through the ADC by the microcontroller on the Node. This reading is transmitted through SPI to the wireless module which transmits the value to the wireless module of the Base. The microcontroller on the Base then reads the value through SPI from the wireless module and reformats it into an OSC Message. The message is then transmitted through SPI to the Ethernet module which broadcasts the message on the network.

4.6 Components

Microchip Technologies provides the free Microchip Application Libraries (MAL) to developers when used with Microchip products. MAL is a compilation of libraries including: USB Framework, Graphics Library, Memory Disk Drive, TCP/IP Stack, mTouch Capacitive Touch Library, Smart Card Library and MiWi Development Environment [Microchip, 2011a].

The two most interesting features of MAL is the TCP/IP Stack and the MiWi Development Environment. The TCP/IP Stack support a number of protocols for connecting Microchip microcontrollers to the network and Internet, including support for UDP sockets needed to send OSC messages across the network as described in Section 4.1.

MiWi is a protocol for creating wireless networks and it is Microchips take on the ZigBee protocol incorporating some changes in the Media Access Control (MAC) layer and a smaller memory footprint [Microchip, 2011b]. Although it still runs on ZigBee compliant wireless modules.

As all the features needed for creating the product for this report are supported by the MAL. It is decided to use Microchip microcontrollers and peripherals to ease the development. The following components from Microchip are supported by the MAL and are chosen to fulfill the criteria described in Section 4.2, furthermore the microcontroller chosen for the Nodes features a very low sleep current:

- PIC32MX340F512H - 32-bit microcontroller, 2 SPI interfaces, serial port and GPIOs.
- PIC18F46K22 - 8-bit microcontroller, SPI interface, serial port, ADC and GPIOs.
- MRF24J40MA - ZigBee compatible wireless module, SPI interface.
- ENC28J60 - Ethernet Controller, 10 Mbps interface, SPI interface.

4.7 Serial Peripheral Interface

This section examines the SPI interface that will be used by the microcontroller to communicate with the peripherals. SPI was developed by Motorola to provide a simple interface for connecting peripherals to a microcontroller. SPI is a synchronous protocol where the master (microcontroller) provides a clock to the slave (peripheral) for synchronizing the data transfer between the two. Many peripherals may be connected on the same SPI interface.

SPI uses four signals: Master Out Slave In (MOSI), Master In Slave Out (MISO), Serial CLocK (SCLK), and Chip Select (CS). On some chips MOSI is labeled Serial Data In (SDI), MISO is labeled Serial Data Out (SDO), SCLK is labeled CLK. Figure 4.2 shows the connections between a microcontroller and a peripheral.

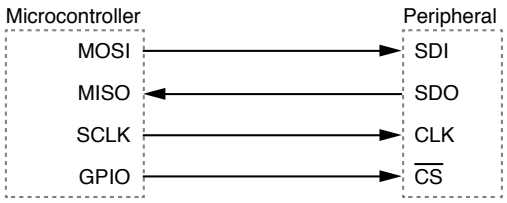


Figure 4.2: SPI connection diagram.

To initialize a connection the master pulls the specific slave's CS pin low. Then the master transmits a byte on the MOSI signal line and at the same time a byte is received from the slave on the MISO signal line. This way a simultaneous write and read operation is performed and this makes SPI communication very efficient.

SPI supports four modes of operation depending on the clock polarity. Mode 0 has a low idle clock and transitions to high when data is ready to be read. Mode 1 also has a low idle clock and transitions to low when data is ready. Mode 2 has a high idle clock and transitions to low when data is ready. Mode 3 also has a high idle clock and transitions to

high when data is ready. Figure 4.3 shows an overview of the SPI data timing for all the modes.

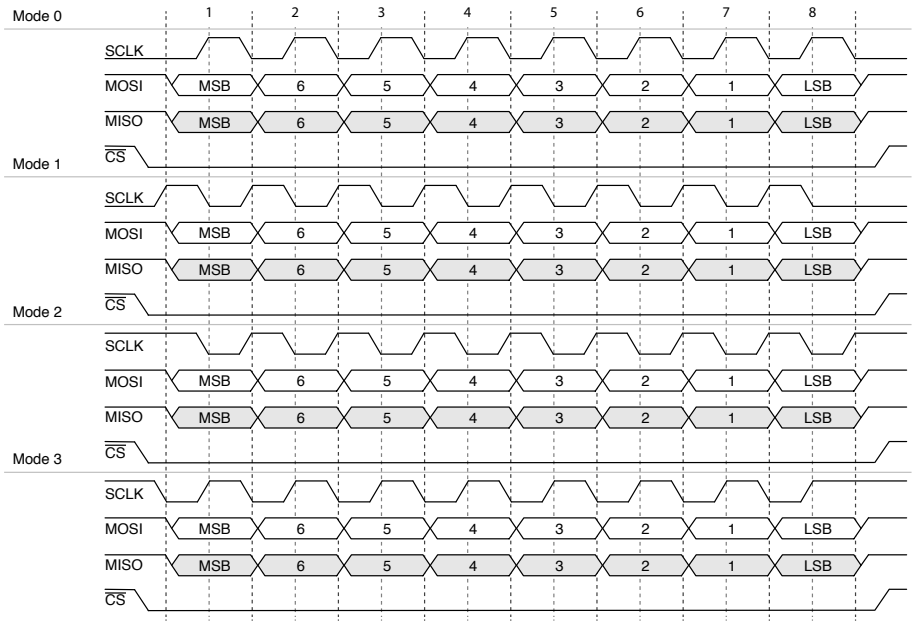


Figure 4.3: Diagram of the SPI transition modes.

4.8 Part Conclusion

OSC is examined to determined what capabilities the hardware needs to support to implement the protocol. e.g. 32-bit integers and support for the UDP communication. The microcontroller is then introduced and the features required by the Base and Nodes have been outlined. The Base need to support 32-bit integers to ease the implementation of the OSC communication, and the microcontroller for the Node needs to support low power usage, as it will be battery powered. The features of the wireless module and the Ethernet controller have been presented, common for both is the need to support a serial interface for communication. MAL is introduced as a basis framework for supporting the implementation of both wireless and Ethernet communication, especially the support for UDP is important for the OSC communication. MAL is only available for components developed by Microchip, therefore Microchip components have been chosen that fulfill the criteria set for the microcontrollers, the wireless module and the Ethernet controller. The SPI interface is examined as it will be used by the microcontroller to communicate with the peripherals.

5. Implementation

This chapter describes the implementation of the Base and the Nodes of the Creative Sensor Network. First, the hardware aspect is described. The circuit diagram connecting all the hardware components is drawn and then used to create the Printed Circuit Board (PCB). All of the components are listed in the bill of materials. Second, the software aspect is described. The OSC communication is described, as well as the events occurring during runtime using flow charts and code samples from the source code, and last the final product is presented.

5.1 Circuit Diagram

The circuit diagram is used to make the initial connections between the components of a circuit design. The components are visualized using simple objects, each pin of a component is labeled as described in the components data sheet. Figure 5.1 and 5.2 shows the circuit diagrams for the implementation of the electronics for the Base and Node, respectively. Full-size circuit diagrams for the Base and Node are available in Appendix A.1 and A.2.

After the circuit diagram has been drawn it is used to create the PCB layout. All the components are drawn with their appropriate land pattern or foot print, and are placed on the print, then electrical traces are routed between the components based on the connections made in the circuit diagram. Figure 5.3 and 5.4 shows the final PCB layout created based on the circuit diagrams of the Base and Node.

5.2 Bill of Materials

The Bill of Material (BOM) lists all the components needed for a specific design. The components are listed in the following manner: part number, value, device type, packaging, and a description of the component e.g.:

Part	**Value**	**Device**	**Package**	**Description**
Y2	25MHz	CRYSTAL5X3	CRYSTAL-SMD-5X3	Crystals

The BOMs for the Base and Node are available in Appendix B.1 and B.2, respectively.

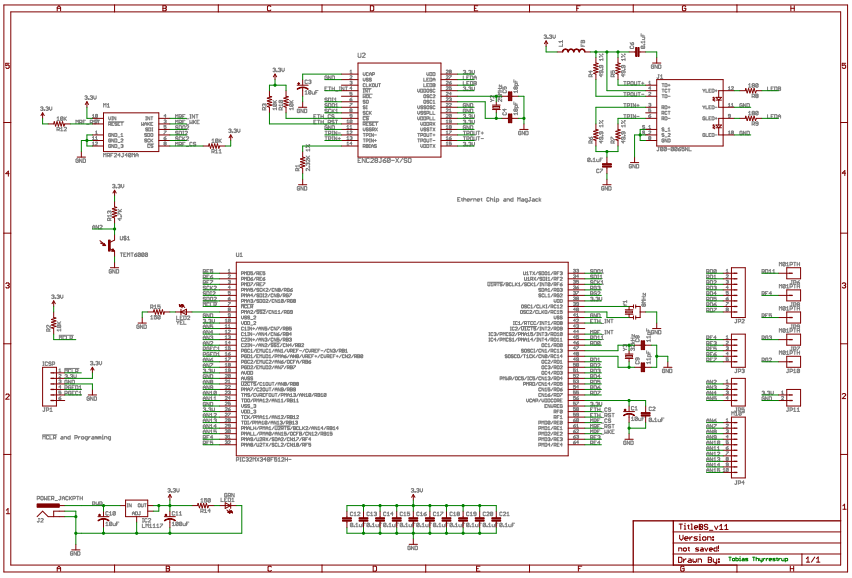


Figure 5.1: Circuit diagram of the Base.

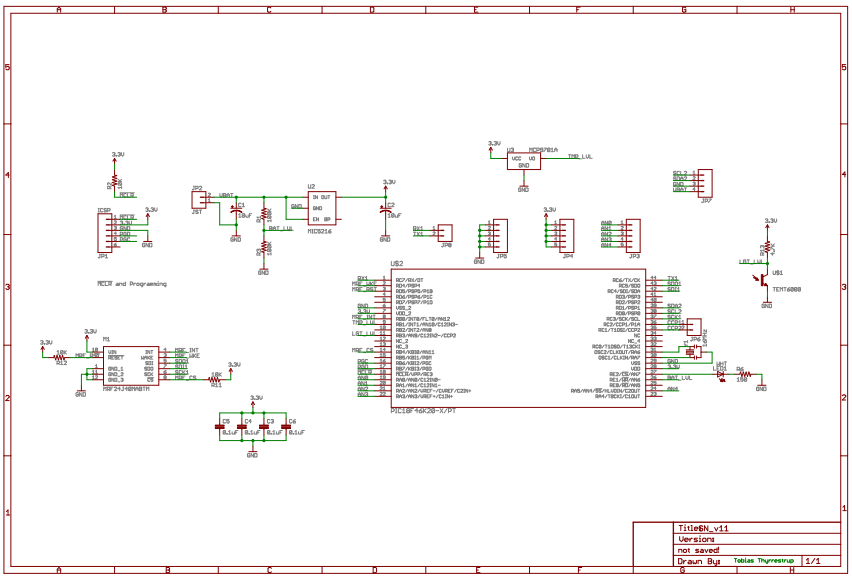


Figure 5.2: Circuit diagram of the Node.

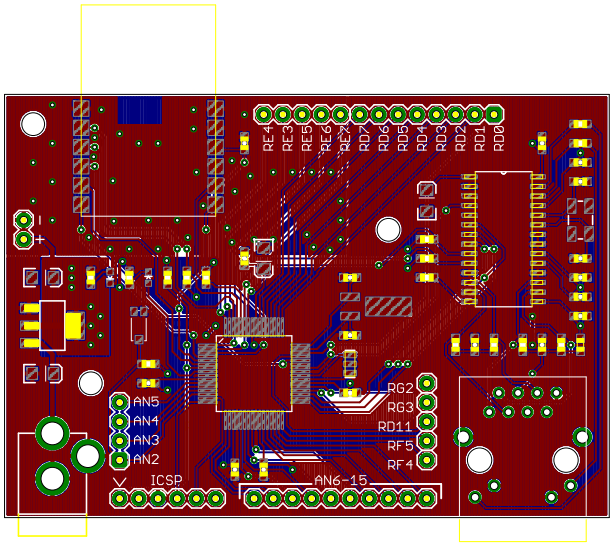


Figure 5.3: PCB layout of the Base.

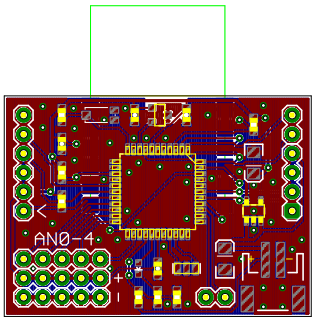


Figure 5.4: PCB layout of the Node.

5.3 Communication

This section describes the OSC Address Patterns used by the Base to communicate information back to the creative applications as well as the patterns used to send discover messages to the Nodes.

When the Base receives a message from a Node it reformats the message into an OSC Message. The message consist of a OSC Address Pattern comprised of a container and an identifier, a OSC Type Tag, and a number of OSC Arguments consistent with the number of sensors. The container in this case is "node" because the information comes from a Node and the identifier is the address of the Node. The type tag identifies that a number of integers are sent and the arguments are the actual sensor values:

```
"/node/7" ",iiii" 1023 1023 1023 1023
```

When the user wants to identify a Node a similar address pattern is sent to the Base which is then responsible for relaying the data to the Node. This time the message consist of an address, one type tag, and one argument. The address is comprised of a container, an identifier, and a function. The function in this case is "discover" and it represents the task to be executed when the message is received. The type tag identifies that only one integer is sent and the argument is either 1 or 0 to enable or disable the discover mode:

```
"/node/7/discover" ",i" 1
```

These addresses provide basic functionality for receiving sensor data from the Nodes and for discovering a specific Node.

5.4 Flow Chart

The flow chart is the diagram that represents the different software tasks of the program to run on the microcontroller. The diagram defines a starting condition for the program e.g. "Power On" and hereafter the different tasks of the program are listed e.g. initialize parameters, send or receive messages, or check whether a received message is valid or not. Figure 5.5 shows the different symbols used in the flowchart.

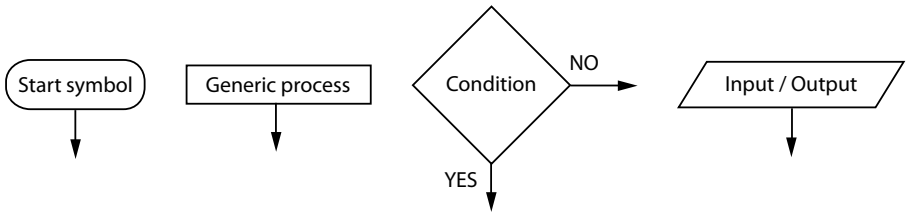


Figure 5.5: Symbols of the flow chart.

The flow chart also details what part of the tasks should be loop over and over until the device is powered off. Flow charts are available for the Base and Node in Appendix C.1 and C.2, respectively.

5.5 Code Samples

This section presents important code samples from the programs that run on the Base and Nodes. These specific samples have been chosen because they are crucial for the end product. The code samples have been simplified to preserve readability. The first code sample is from the Base. It is responsible for initializing the wireless module and creating a new network for the Nodes to join. First it scans the available radio channels for noise and then it creates a new network on the channel with the least noise, the sample is shown in Listing 5.1.

```
1  // Global variables - included to preserve readability
2  #define LED_1 LATDbits.LATD0
3
4  LED_1 = 0;
5
6  MiApp_ProtocolInit(FALSE);
7
8  MiApp_ConnectionMode(ENABLE_ALL_CONN);
9
10 Printf("Active Scanning Energy Scanning\r\n");
11
12 MiApp_StartConnection(START_CONN_ENERGY_SCN, 10, 0xFFFFFFFF);
13
14 LED_1 = 1;
```

Listing 5.1: Base - Starting a new network.

Line 2: Global definition. Line 4: Make sure LED 1 is turned off. Line 6: Initialize the wireless module and the protocol stack. Line 8: Make sure the connection is set up to enable all incoming connections. Line 10: Debug text. Line 12: Start energy scan to determine the channel with lowest noise level and then start a new network on that channel. Line 14: Turn on led to indicate that the Base is ready for connections from Nodes.

The second code sample is from the Node. It is responsible for initializing the wireless module and for joining the network created by the Base. First the Node needs to know on which radio channel the Base has created a network therefore it initializes a search for the Base and if the search returns a valid network a connection is established, if not the Node keeps searching until it finds a network, the sample is shown in Listing 5.2.

Line 2-6: Global variables. Line 8: Initialize the wireless module and the protocol stack. Line 10-23: Start endless loop. Line 12: Search for networks to join and return the number of found networks. Line 14-16: If a network is found set the channel of the Node

```
1  // Global variables - included to preserve readability
2  BYTE myChannel = 0xFF;
3  BYTE j;
4  struct {
5     BYTE Channel;
6  } *ActiveScanResults;
7
8  MiApp_ProtocolInit(FALSE);
9
10 while(1)
11 {
12    j = MiApp_SearchConnection(10, 0xFFFFFFFF);
13
14    if( j > 0 ) {
15       myChannel = ActiveScanResults[0].Channel;
16    }
17
18    if( myChannel != 0xFF ) {
19       MiApp_SetChannel(myChannel);
20       break;
21    }
22    Printf("No Suitable PAN, Rescanning...\r\n");
23 }
24
25 MiApp_ConnectionMode(DISABLE_ALL_CONN);
26
27 MiApp_EstablishConnection(0, CONN_MODE_DIRECT);
```

Listing 5.2: Node - Joining a network.

to the channel of the first search result. Line 18-21: If the channel has been changed send the channel to the wireless module and break the loop. Line 22: Debug text. Line 25: Disable all incoming connection. Line 27: Connect to the first network returned by the network search.

The third code sample is from the Base. It is responsible for converting incoming messages from Nodes into an OSC Message. The message received from the Node contains the sensor readings, the Node address and the signal strength of the received message. This information is reformatted into an OSC Message ready with an OSC Address Pattern that matches the specific Node.

```
1  // Global variables - included to preserve readability
2  int tmp[3];
3  char nodeID[10];
4  struct {
5     BYTE *SourceAddress;
6     BYTE PacketRSSI;
7  } rxMessage;
8
9  sprintf(nodeID, "/node/%u", rxMessage.SourceAddress[0]);
10
11 OSCCreateMessage(nodeID, ",iiii", tmp[0], tmp[1], tmp[2], rxMessage.PacketRSSI);
```

Listing 5.3: Base - Creating a new OSC Message.

Line 2-7: Global variables. Line 9: Create a string containing the OSC Address Pattern

"/node/" appended with the lowest byte of the Node address of the message from the Node. Line 11: Create an OSC Message with the specific OSC Address Pattern containing four integers - three sensor values and the signal strength from the Node. The full source code for the Base is available in Appendix D.1. The full source code for the Node is available in Appendix D.2. The source is included without the MAL libraries. The source code for these libraries is available for download from: www.microchip.com/mal/

5.6 Product

The finished product consist of a number of battery powered Nodes capable of reading sensors and communicating the sensor values wirelessly back to the Base. The Base is connected to the network through an Ethernet connector and is capable of sending OSC Messages to all devices on the network. The Nodes have been fitted with a temperature sensor and a light intensity sensor. Figure 5.6 shows a photo of the finished product.

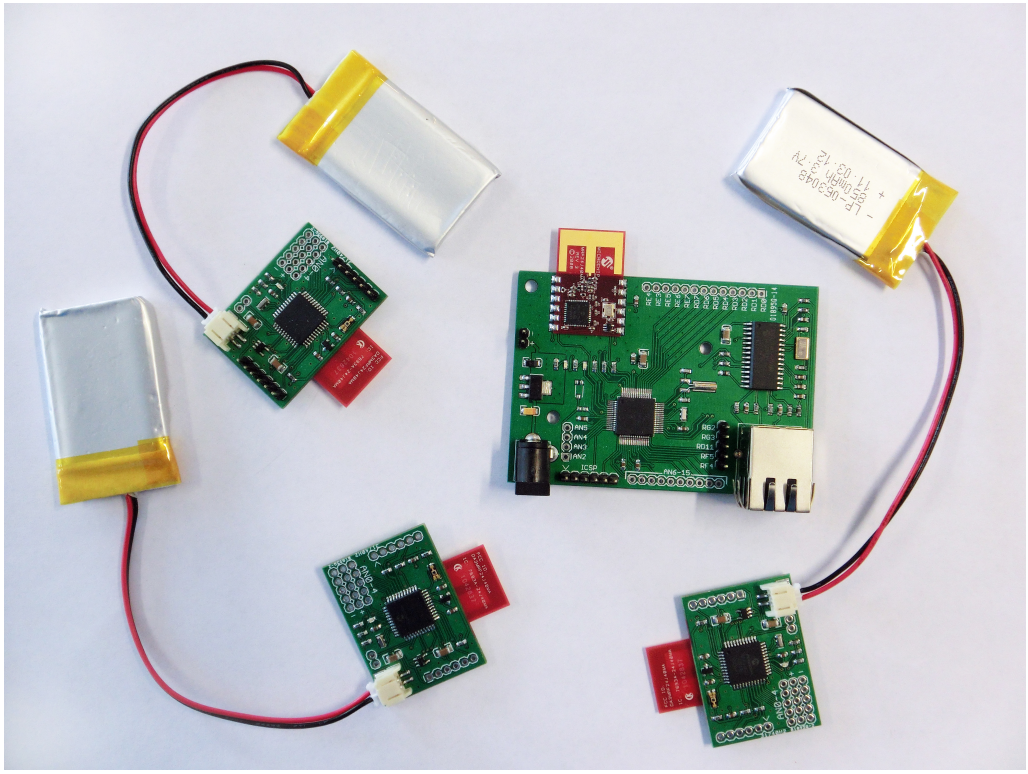


Figure 5.6: Photo of the Creative Sensor Network.

5.7 Part Conclusion

The circuit diagram and PCB layout for the Creative Sensor Network has been presented together with a BOM for one Base and one Node. The OSC Address Patterns for the communication of sensor values to the creative applications and the process of discovering a Node have been outlined. Moreover flow charts of the programs running on the Base and the Node have been presented. The process of initializing the wireless network connection and the process of creating an OSC Messages have been presented using code samples. The final product has been presented with a photo of the assembled product.

6. Evaluation

This chapter will evaluate the product of this report - The Creative Sensor Network. First, showing how only a few lines of code or a few objects are needed to directly receive sensor values in the different creative applications described in Section 2.3. Second, potential users with a design background have commented on the product and its use in the design process. Third, a case where the product has been used in a design process is presented.

6.1 Creative Applications

This section presents examples of how easy it is to implement the Creative Sensor Network in anyone of the creative applications. For applications based on code, code examples are shown in listings and for applications based on visual programming the number of objects needed to receive the information are listed and a screenshot of the connection of the objects is available in Appendix E.

Each of the examples show how to setup the application to receive OSC Messages on port 12345 and how to identify a specific OSC Address Pattern e.g. "/node/7". Furthermore the OSC Arguments (sensor values) from that specific address are extracted from the OSC Message, this can then be replicated to extract all of the sensor values from all of the Nodes, the only modification needed is changing the OSC Address Pattern to match the Node of interest.

6.1.1 Processing

The code example for Processing is using the **oscP5** library. First the library is imported, then the OSC receiver is initialized and the callback is established. When the program runs all OSC Messages are sent to the callback where specific messages can be identified and used in anyway the designer chooses. Listing 6.1 shows the code.

Line 1-2: Import the necessary libraries. Line 4-5: Create OSC handler and an array for storing sensor values. Line 7-9: Processing setup function - Initialize the OSC handler and receive port. Line 11-12: Processing draw loop. Line 14-21: OSC callback function. Line 15: Print all OSC Messages. Line 16: Identify "/node/7" for further processing. Line 17-19: Store the sensor values from Node 7 in the array.

```
1  import oscP5.*;
2  import netP5.*;
3
4  OscP5 oscP5;
5  int [] s = new int[4];
6
7  void setup() {
8     oscP5 = new OscP5(this, 12345);
9  }
10
11 void draw() {
12 }
13
14 void oscEvent(OscMessage msg) {
15    msg.print();
16    if(msg.checkAddrPattern("/node/7")) {
17       for(int i = 0; i < 4; i++) {
18          s[i] = msg.get(i).intValue();
19       }
20    }
21 }
```

Listing 6.1: Receiving sensor values in Processing

6.1.2 Grasshopper

Grasshopper needs the **gHowl** component to support OSC. After this component has been installed the example queries for at valid network connection on which to receive OSC Messages. The received messages are filtered based on their OSC Address Pattern and the OSC Arguments of all valid addresses are printed in a text panel. Then the sensor values are extracted from on of the messages. A screenshot of the example is available in Appendix E.1.

The following objects are used to receive OSC Messages and extract the OSC Arguments from messages with a specific OSC Address Pattern:

- **NetSource**
- **>UDP<**
- **OSC_D**
- **Item**
- **Split**

The NetSource object registers if there is a valid network to receive messages from. The >UDP< object is setup to receive OSC Messages on port 12345. The OSC_D object stores a list with the OSC Arguments of messages with OSC Address Patterns of interest. The Item object extracts Node 7 from the list. The Split object is used to extract the sensor values from a list of arguments.

6.1.3 openFrameworks

openFrameworks support OSC through the **ofxOsc** addon. The addon is included in the header file for the program. The program is setup to print the OSC Address Pattern of incoming OSC Messages to the console. A message with a specific address is filtered out and the OSC Arguments of the message are stored. To minimize the space needed openFrameworks specific code has been left out. The full source code is available in Appendix F. Listing 6.2 shows the code.

```
1  // testApp.h
2  #include "ofxOsc.h"
3
4  ofxOscReceiver receiver;
5  int    s[4];
6
7  //testApp.c
8  void testApp::setup() {
9     receiver.setup( 12345 );
10 }
11
12 void testApp::update(){
13    while( receiver.hasWaitingMessages() ) {
14       ofxOscMessage m;
15       receiver.getNextMessage( &m );
16       cout << m.getAddress() << endl;
17       if ( m.getAddress() == "/node/7" ) {
18          for (int i = 0; i < m.getNumArgs(); i++) {
19             s[i] = m.getArgAsInt32( i );
20          }
21       }
22    }
23 }
```

Listing 6.2: Receiving sensor values in openFrameworks

Line 2: Include addon. Line 4-5: Create OSC handler and array for storing sensor values. Line 8-10: Setup OSC handler to receive on port 12345. Line 12-23: Update loop. Line 13-22: Receive all OSC Messages. Line 14-15: Store current message. Line 16: Print the OSC Address Pattern to the console. Line 17-21: check if the address match "/node/7". Line 18-20. Loop through the OSC Arguments of the message. Line 19: Extract the sensor values and store them.

6.1.4 Max/MSP

Fortunately the internal message structure of Max/MSP is based on OSC Messages therefore there is no need to import extra libraries as OSC is natively supported in Max/MSP. The example is setup to print all OSC Messages to the Max window for debugging. A specific OSC Address Pattern is routed to an object that unpacks the OSC Arguments and shows each of the integer values in a number box. A screenshot of the example is available in Appendix E.2.

The following objects are used to receive and extract the sensor values from OSC Messages in Max/MSP:

- **udpreceive**
- **print**
- **route**
- **unpack**

The udpreceive object is setup to receive messages on port 12345. All messages are printed to the Max window using the print object. The route object identify OSC Messages with the OSC Address Pattern "/node/7" and the unpack object extracts the OSC Arguments.

6.1.5 LuaAV

Although LuaAV supports OSC natively the OSC module still have to be required at the beginning of the script. An OSC handler is setup to print all of the incoming OSC Messages to the LuaAV console. It also filters out messages with a specific OSC Address Pattern and stores the OSC Arguments from these messages. Furthermore the OSC handler is setup to run simultaneous with the rest of the script. Listing 6.3 shows the code.

```
1  local osc = require("osc")
2  local oscin = osc.Recv(12345)
3  local s = {}
4
5  function get_osc()
6     for msg in oscin:recv() do
7        print(msg.addr, msg.types, unpack(msg))
8        if msg.addr == "/node/7" then
9           for i = 1, #msg, 1 do
10             s[i] = msg[i]
11          end
12       end
13    end
14 end
15
16 go(function()
17    while(true) do
18       get_osc()
19       wait(1/50)
20    end
21 end)
```

Listing 6.3: Receiving sensor values in LuaAV

Line 1: Require OSC. Line 2: Initialize OSC to receive messages on port 12345. Line 3: Create table to hold sensor values. Line 5-14: Function for receiving OSC Messages. Line 6-13: Loop through all received messages. Line 7: Print messages to the console. Line 8-12: Check if OSC Address Pattern matches "/node/7". Line 9-11. Loop through

all OSC Arguments and store them. Line 16-21: Create a simultaneous process that calls the OSC handler once every 20 milliseconds.

6.1.6 Quartz Composer

Quartz Composer also support OSC natively and it is simple to setup. The example shows how OSC Messages are filtered based on their OSC Address Pattern so only the messages of interest enter the program, the rest are discarded. The example then extracts the OSC Arguments from a message with a specific OSC Address Pattern. Moreover it is shown how to print the arguments to the viewer, although this is not needed. A screenshot of the example is available in Appendix E.3.

Only two different objects are used to extract the OSC Arguments in Quartz Composer:

- **OSC Receiver**
- **Structure Index Member**

The OSC Receiver object is setup to receive messages on port 12345 and then the OSC Address Patterns of interest are added to the object. The Structure Index Member object handles the extraction of the OSC Arguments by setting the index of the argument to extract.

6.1.7 Summary

As seen in the examples above it only takes a few lines of code or a few objects to setup anyone of the creative applications to receive sensor values from the Creative Sensor Network. This makes it very easy for designers familiar with these applications to get up and running and start exploring the possibilities of this new design tool, rather than spending time wondering about different protocols or writing data parsers or proxies.

6.2 Comments

This sections presents two comments from potential users of the Creative Sensor Network. The first comment is from Andreas Eggertsen - Architect Cand. polyt. who is working at Snøhetta studio in Norway. He had the following to say about the uses of the design tool:

> *The telling of the story of condensed experience meaningful and relevant contains the relationship to the environment in a specific time. The information can form the basis for an understanding with real world data developed in interaction with the context, with a feed-back loop in the design process the designer can develop the precision of design response and develop the understanding of how to unfold the potentials of the specific situation.*

Andreas Eggertsen
24.05.2011

The second comment is from Isak Worre Foged - M.Arch, partner in studio AREA, and research assistant at Architecture and Design, Aalborg University. He has commented on aspects of architectural research that could benefit from using a tool like the Creative Sensor Network:

> *A part of architecture is to create environments. In the current conducted research work, architecture is suggested to construct new types of spaces that take greater emphasis on changing conditions, provided by daily to yearly altering climates and rapid changes in occupancy intensity and activity. Buildings call for constructs that facilitate a higher level of responsive adaptability to meet new conditions - instantly. In this pursuit lies a common understanding of sensing, decision taking and actuating, to which an improved level of all three aspects could push towards novel environmental conditions in architecture.*

Isak Worre Foged
16.05.2011

6.3 Case Using Grasshopper

After learning about the product Isak Worre Foged agreed to incorporate it in a research project currently being conducted. In the following paragraph the focus of the project is described and also how the Creative Sensor Network is used in the process:

> ***Extended Sensitivity - Towards Novel Awareness in Architecture***
> *Most current research in architecture deals with a low level sensory field of one to three sensors, situated locally. To instead situate sensors locally and regionally in an infrastructure of contextual awareness opens to new ways of responding to altering conditions and to the basis of how we understand spatial sensitivity in architecture. Within the conducted research work is a physical prototype constructed which mediates between internal and external environments of a building, enabled through an extended sensory field to enhance the contextual awareness beyond our human sensing capacities and furthermore, perhaps more importantly, to install logics that predict adaptation from the regional climatic patterns compared to the local climatic and occupancy driven patterns. The raw contextual information is captured through the Creative Sensor Network system, and translated through the parametric bundle of softwares Rhinoceros + Grasshopper + gHowl + Firefly to physical actuation within the architectural*

prototype, adapting and changing the local milieu and thereby returning a new set of local environmental patterns to be evaluated against the regional patterns.

The visual programming for the project done in Grasshopper is shown as a screenshot in Figure 6.1. A full-size version is available in Appendix G.

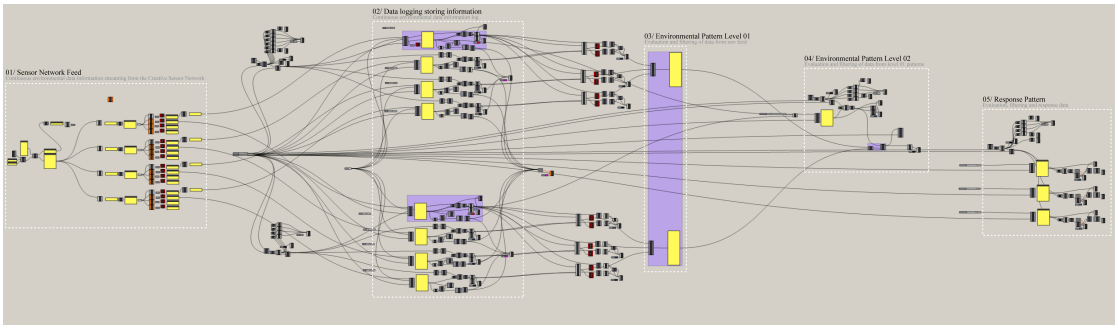


Figure 6.1: Grasshopper program for the research project.

The objects in the area furthest to the left are responsible for receiving sensor values from the Creative Sensor Network. The rest of the objects are responsible for logging and filtering the received data as well as visualizing the information through the use of geometry objects. Again it can be seen that the number of objects needed for receiving the sensor readings is a small part of the complete program. The output of the program can be seen in Figure 6.2. A full-size version is available in Appendix G.

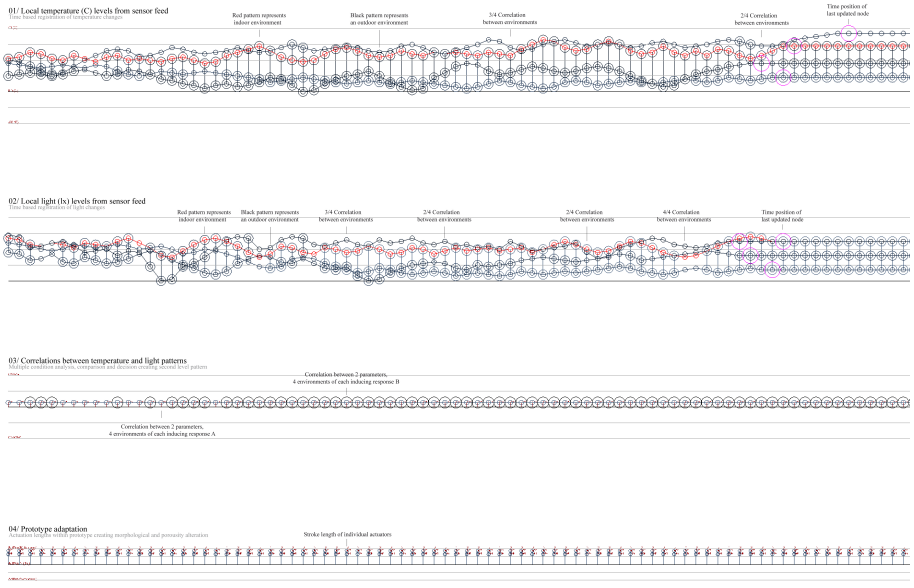


Figure 6.2: Grasshopper visualization for the research project.

The first section is visualizing the relations between temperatures from different Nodes. The second section is visualizing the relations between incoming light intensities from different Nodes. The third section visualizes the correlation between light intensities and

temperatures. The fourth section is visualizing the resulting actuation lengths for controlling a structure.

6.4 Part Conclusion

Only a few lines of code or few objects are needed to implement the Creative Sensor Network in anyone of the creative applications. This makes it very easy for designers already familiar with one or more of the applications to get up and running and start using the sensor readings in a design context. Furthermore two potential users with a background in design have commented on scenarios where the product can be used in the creation of new environments. The Creative Sensor Network has also been successfully implemented in a current research project conducted by Isak Worre Foged focusing on architecture which reacts to changes its surroundings.

7. Discussion

The Creative Sensor Network differentiate itself from other wireless sensor networks in the way it provides the user with sensor values. The interface is directly compatible with anyone of the creative applications described in Section 2.3 because the OSC communication protocol is implemented in the hardware. This compatibility could presumably have been accomplished using one of the sensor networks described in Section 2.2 accompanied by a piece of software capable of translating the received data from the sensors into OSC Messages. However this would create another hurdle for designers as they get pulled away from their preferred creative applications and this is not the intension as this product is thought to be a transparent extension to the design tools that lets the designer focus on the creative process rather that the technical aspects.

One could argue that a wireless sensor network has nothing to do with the design process and this is the reason why none of the sensor systems implement an interface suitable for communicating directly with creative applications. However this is improbable based on the comments from Andreas Eggertsen and Isak Worre Foged. As they both describe aspects of the architectural discipline where a deeper knowledge and understanding of the spaces we inhabit could be beneficial in the design process. Furthermore the fact that the Creative Sensor Network was interesting enough to be implemented in the research project conducted by Isak Worre Foged is evidence of an interest in a product that facilitates an ease of use real-time interface between the real world and the creative applications.

The Creative Sensor Network is not thought as a replacement for the established design process or the architectural models concerned with the understanding of spaces which have been developed through many years. Instead it is imagined as a means to discover these aspects and make them more visible to the designer and this way extend the design process. It can be seen as a real-time "image" of a space to either guide the design process or inform a design once it has been constructed, or perhaps both.

8. Conclusion

Ubiquitous computing has been introduced as a way of gathering precise readings of our surroundings. It has been proposed that the control parameters in a design process can be regulated by these readings. A practical approach to ubiquitous computing as of today has been to use wireless sensor networks to collect information about our environment. A number of commercially available sensor networks have been examined for their ability to communicate information back to a computer. Also a number of creative applications have been examined for their ability to directly receive information from external devices and the OSC protocol is found to be the only one supported by all of the creative applications. This particular protocol is however not supported by any of the sensor networks and therefore it is cumbersome at the very least to setup a communication interface between the creative applications and the sensor networks.

A concept is described consisting of a Base and a number of Nodes. The Base implements the ability to send OSC Messages over an Ethernet network. These messages can then be received by any of the creative applications. The Base can also receive messages wirelessly from a number of Nodes. Each of the Nodes can be fitted with sensors to give readings of their surroundings and transmit this information back to the Base.

Hardware technologies like microcontrollers, wireless modules and Ethernet modules have been briefly introduced. And the communication aspect relaying information from the Base to the creative applications through OSC has been examined. The communication between hardware components through SPI has also been examined.

A product has been created capable of sensing the environment in which it is installed. The sensory information is communicated back to a central unit capable of serving the data directly into a number of creative applications. It has been shown that the programming impact of implementing the Creative Sensor Network in anyone of the described creative applications can be neglected. The product has also been successfully applied in an ongoing research project.

9. Perspective

In perspective the Creative Sensor Network could be implemented with a variety of functionality. The Base already features an Ethernet controller capable of connecting it to the Internet and not only the local network. This way multiple devices could communicate their information over the Internet to connect into larger "smart grids". Designers could then tap into these grids to find information not only about our surroundings on a local scale but also on a global scale.

Another option is to use the GPIO pins on the Nodes to control actuators making the Nodes capable not only of reading sensor values but also of making actuations based on the readings. And as the sensor readings are fed to a computer and processed further the computer could again send global actuations across all the Nodes through the Base. This complete feedback system could create interesting levels of actuation as Nodes start to process the sensor data and maybe create local subtle changes to a structure. Then when the information has been processed on the computer and correlated to sensor readings received from other Nodes, a change is made to the structure on a global level.

A plug and play sensor system could also be developed to ease the installation of new sensors on the Nodes. The current version of the Creative Sensor Network requires that the user has some understanding of electronics to install a new sensor as the sensor inputs are comprised of a ground pin, a positive voltage, and an analog input pin. Although it is common to see sensors with this configuration there is still a lot of sensors which come in different configurations. Some newer sensors don't even provide an analog output as they use a digital interface like SPI or I^2C with an ADC built into the sensor. This option could also be implemented to support a wider range of sensors.

Bibliography

[Atmel, 2011] Atmel (2011). *TinyAVR - Unmatched Performance and Efficiency in a Small Package*. Atmel Corporation. http://www.atmel.com/dyn/products/devices.asp?category_id=163&family_id=607&subfamily_id=791 [date: 05.13.2011].

[Butler, 2006] Butler, Declan (2006). *Everything, Everywhere*. Nature Publishing Group.

[Catsoulis, 2005] Catsoulis, John (2005). *Designing Embedded Hardware*, p. 26–29. O'Reilly.

[CMT, 2011] CMT (2011). *Wireless Sensor Networks*. CMT. http://www.cmt-gmbh.de/1/Produkte/Wireless_Sensor_Networks/Wireless_Sensor_Networks.html [date: 05.15.2011].

[Crossbow, 2011] Crossbow (2011). *Xserve Users Manual*. Crossbow. http://www.memsic.com/support/documentation/wireless-sensor-networks/category/6-user-manuals.html?download=96%3Axserve-user-s-manual [date: 05.15.2011].

[Cycling74, 2011a] Cycling74 (2011a). *A Functional Listing of all Max Objects*. Cycling 74. http://cycling74.com/docs/max5/vignettes/core/max_functional.html [date: 05.13.2011].

[Cycling74, 2011b] Cycling74 (2011b). *Products - Cycling 74*. Cycling 74. http://cycling74.com/products/ [date: 05.13.2011].

[Faludi, 2007] Faludi, Robert (2007). *Building Wireless Sensor Networks*. O'Reilly.

[Firefly, 2011] Firefly (2011). *Firefly - Tools*. Firefly Experiments. http://www.fireflyexperiments.com/tools/ [date: 05.13.2011].

[Gershenfeld *et al.*, 2004] Gershenfeld, Neil, Krikorian, Raffi, & Cohen, Danny (2004). *The Internet of Things*. Scientific American Inc.

[gHowl, 2011] gHowl (2011). *gHowl - Grasshopper*. gHowl. http://www.grasshopper3d.com/group/ghowl [date: 05.13.2011].

[Grasshopper, 2011] Grasshopper (2011). *Grasshopper - Generative modeling for Rhino*. grasshopper3d.com. http://www.grasshopper3d.com/ [date: 05.13.2011].

[Igoe, 2007] Igoe, Tom (2007). *Making Things Talk*, p. 178–180. O'Reilly.

[Li, 2011] Li, Bo (2011). *Walk Bridge Modelled with Grasshopper*. grasshopper3d.com. http://www.grasshopper3d.com/photo/walk-bridge-modelled-with-7 [date: 05.13.2011].

[Libelium, 2011a] Libelium (2011a). *Waspmote - Datasheet*. Libelium. http://www.libelium.com/documentation/waspmote/waspmote-datasheet_eng.pdf [date: 05.15.2011].

[Libelium, 2011b] Libelium (2011b). *Wireless Sensor Networks with Waspmote and Meshlium*. Libelium. http://www.libelium.com/documentation/mesh_extreme/wsn-waspmote_and_meshlium_eng.pdf [date: 05.15.2011].

[LuaAV, 2011a] LuaAV (2011a). *LuaAV - Documentation*. AlloSphere Research Group. http://lua-av.mat.ucsb.edu/doc/index.html [date: 05.14.2011].

[LuaAV, 2011b] LuaAV (2011b). *LuaAV - Real-Time Audio Visual Scripting*. AlloSphere Research Group. http://lua-av.mat.ucsb.edu/blog/ [date: 05.14.2011].

[McCullough, 2005] McCullough, Malcolm (2005). *Digital Ground - Architecture, Pervasive Computing, and Environmental Knowing*, p. 7, 9, 11, 12, 14, 19, 21. The MIT Press.

[McNeel, 2011] McNeel (2011). *Rhinoceros - Modeling tools for designers*. McNeel. http://www.rhino3d.com/ [date: 05.13.2011].

[Memsic, 2011] Memsic (2011). *eKo Outdoor Wireless System*. MEMSIC. http://www.memsic.com/support/documentation/eko/category/15-datasheets.html?download=156%3Aeko-base-station [date: 05.15.2011].

[Microchip, 2011a] Microchip (2011a). *Microchip Application Libraries*. http://www.microchip.com/mal/ [date: 05.22.2011].

[Microchip, 2011b] Microchip (2011b). *Microchip MiWi P2P Wireless Protocol*. http://ww1.microchip.com/downloads/en/AppNotes/01204B.pdf [date: 05.22.2011].

[Newport, 2011a] Newport (2011a). *NEWPORT - Wireless Sensor System*. Newport. http://www.newportelect.com/ppt/ZSERIES.html [date: 05.15.2011].

[Newport, 2011b] Newport (2011b). *NEWPORT - Wireless Sensor System - Specifications*. Newport. http://www.newportelect.com/PDFspecs/zSeries_n_lo.pdf [date: 05.15.2011].

[Noble, 2009] Noble, Joshua (2009). *Programming Interactivity*, p. 614–615. O'Reilly.

[openFrameworks, 2011a] openFrameworks (2011a). *openFrameworks - About*. openFrameworks. http://www.openframeworks.cc/about/ [date: 05.13.2011].

[openFrameworks, 2011b] openFrameworks (2011b). *openFrameworks - Addons*. openFrameworks. http://www.openframeworks.cc/addons/ [date: 05.13.2011].

[Processing, 2011a] Processing (2011a). *Libraries Processing*. processing.org. http://processing.org/reference/libraries/ [date: 05.13.2011].

[Processing, 2011b] Processing (2011b). *Overview Processing*. processing.org. http://processing.org/learning/overview/ [date: 05.13.2011].

[Processing, 2011c] Processing (2011c). *Overview Processing*. processing.org. http://processing.org/about/ [date: 05.13.2011].

[Quartz, 2011] Quartz (2011). *Working with Quartz Composer in Leopard*. Apple Inc. https://developer.apple.com/library/mac/#featuredarticles/WorkingWithQuartzComposer/_index.html [date: 05.14.2011].

[Wright, 2011] Wright, Matt (2011). *The Open Sound Control 1.0 Specification*. CNMAT. http://opensoundcontrol.org/spec-1_0/ [date: 05.14.2011].

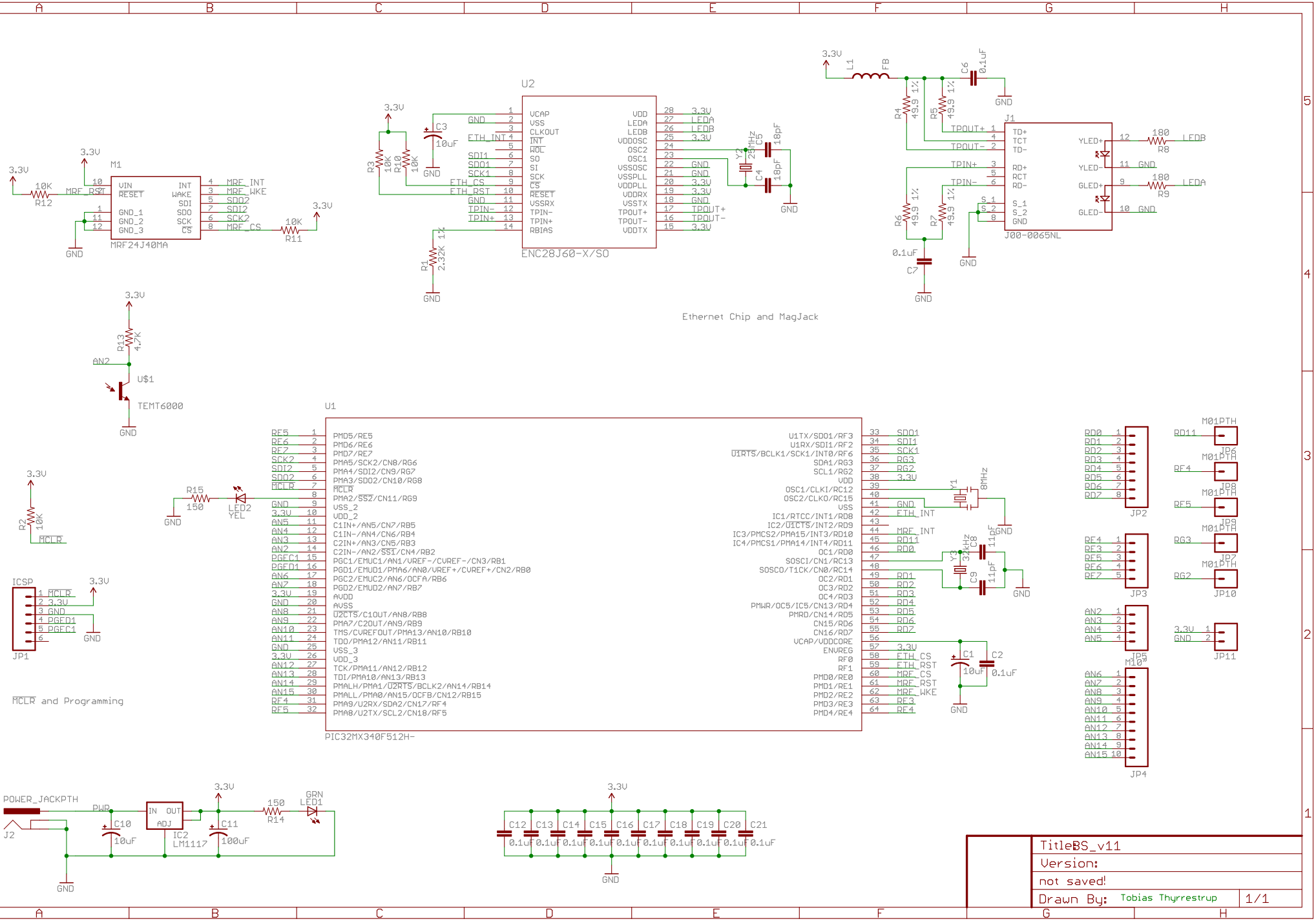
[Zhang, 2011] Zhang, Haiyan (2011). *Japan Geigermap*. http://japan.failedrobot.com/ [date: 05.16.2011].

List of Figures

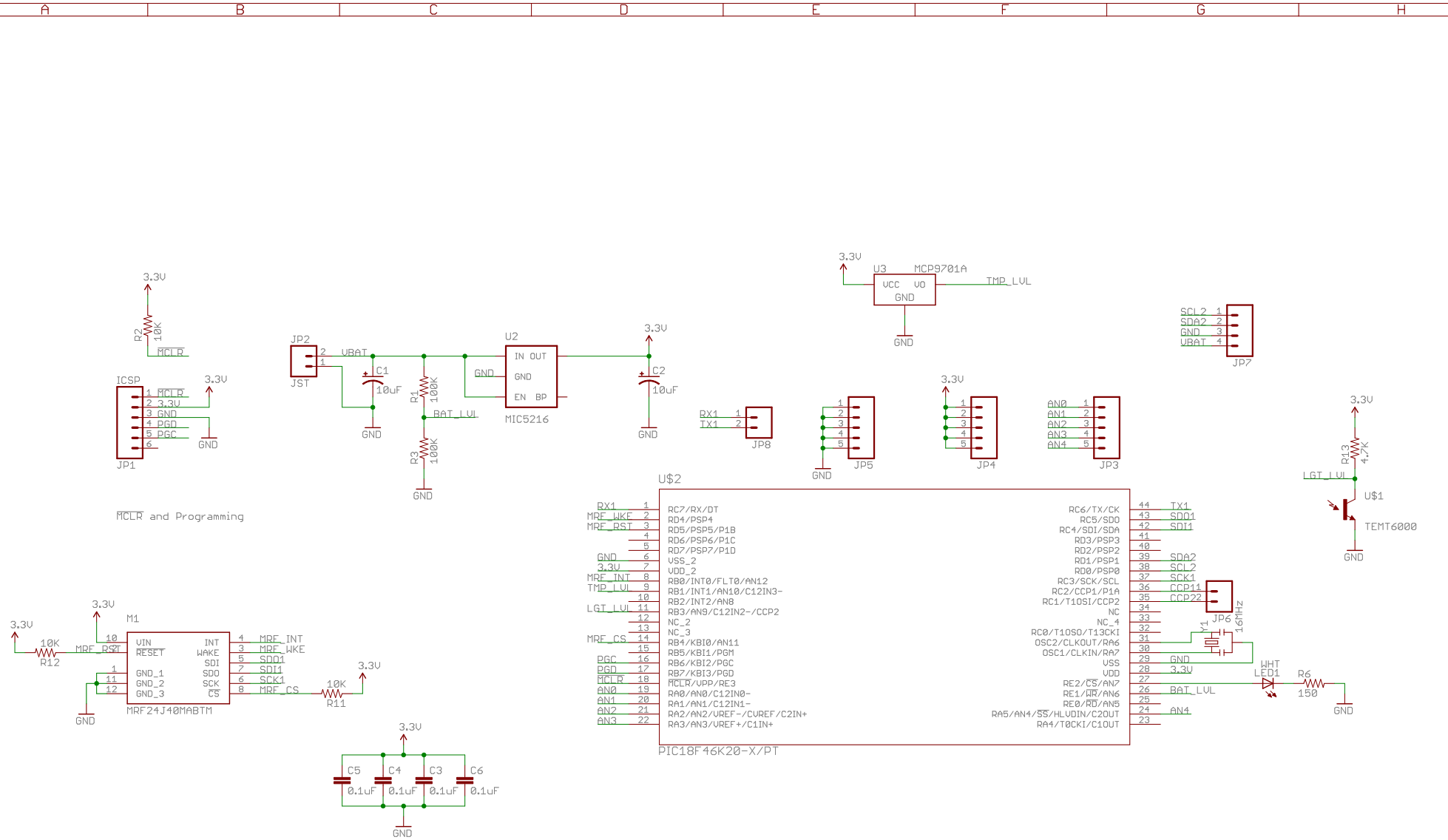
Appendices

A. Schematics

A.1 Base


U2
ENC28J60-X/SO
M1
MRF24J40MA
J1
J00-0065NL
Ethernet Chip and MagJack
U$1
TEMT6000
U1
PIC32MX340F512H-
JP1
JP2
JP3
JP4
JP5
JP6
JP7
JP8
JP9
JP10
JP11
MCLR and Programming
POWER_JACKPTH
J2
IC2
LM1117
LED1
LED2
Title: BS_v11
Version:
not saved!
Drawn By: Tobias Thyrrestrup
1/1

A.2 Node


3.3V
R2
10K
MCLR
ICSP
1 MCLR
2 3.3V
3 GND
4 PGD
5 PGC
6
JP1
GND
MCLR and Programming
JP2
JST
VBAT
C1
10uF
R1
100K
BAT_LVL
R3
100K
GND
U2
IN OUT
GND
EN BP
MIC5216
C2
10uF
U3
MCP9701A
VCC VO
GND
TMP_LVL
SCL2 1
SDA2 2
GND 3
VBAT 4
JP7
RX1 1
TX1 2
JP8
JP5
JP4
AN0 1
AN1 2
AN2 3
AN3 4
AN4 5
JP3
R13
4.7K
LGT_LVL
U$1
TEMT6000
M1
VIN
RESET
GND_1
GND_2
GND_3
INT
WAKE
SDI
SDO
SCK
CS
MRF_INT
MRF_WKE
SDO1
SDI1
SCK1
MRF_CS
MRF24J40MABTM
10K
R12
MRF_RST
10K
R11
U$2
RX1
MRF_WKE
MRF_RST
GND
3.3V
MRF_INT
TMP_LVL
LGT_LVL
MRF_CS
PGC
PGD
MCLR
AN0
AN1
AN2
AN3
RC7/RX/DT
RD4/PSP4
RD5/PSP5/P1B
RD6/PSP6/P1C
RD7/PSP7/P1D
VSS_2
VDD_2
RB0/INT0/FLT0/AN12
RB1/INT1/AN10/C12IN3-
RB2/INT2/AN8
RB3/AN9/C12IN2-/CCP2
NC_2
NC_3
RB4/KBI0/AN11
RB5/KBI1/PGM
RB6/KBI2/PGC
RB7/KBI3/PGD
MCLR/VPP/RE3
RA0/AN0/C12IN0-
RA1/AN1/C12IN1-
RA2/AN2/VREF-/CVREF/C2IN+
RA3/AN3/VREF+/C1IN+
RC6/TX/CK
RC5/SDO
RC4/SDI/SDA
RD3/PSP3
RD2/PSP2
RD1/PSP1
RD0/PSP0
RC3/SCK/SCL
RC2/CCP1/P1A
RC1/T1OSI/CCP2
NC
NC_4
RC0/T1OSO/T13CKI
OSC2/CLKOUT/RA6
OSC1/CLKIN/RA7
VSS
VDD
RE2/CS/AN7
RE1/WR/AN6
RE0/RD/AN5
RA5/AN4/SS/HLVDIN/C2OUT
RA4/T0CKI/C1OUT
TX1
SDO1
SDI1
SDA2
SCL2
SCK1
CCP11
CCP22
AN4
PIC18F46K20-X/PT
JP6
Y1
16MHz
WHT
LED1
R6
150
C5
C4
C3
C6
0.1uF
Title$N_v11
Version:
not saved!
Drawn By: Tobias Thyrrestrup
1/1

B. Bill of Materials

B.1 Base

Partlist for the Base

Part	Value	Device	Package	Description
C1	10uF	CAP_POL1206	EIA3216	Capacitor Polarized
C2	0.1uF	CAP0603-CAP	0603-CAP	Capacitor
C3	10uF	CAP_POL1206	EIA3216	Capacitor Polarized
C4	18pF	CAP0603-CAP	0603-CAP	Capacitor
C5	18pF	CAP0603-CAP	0603-CAP	Capacitor
C6	0.1uF	CAP0603-CAP	0603-CAP	Capacitor
C7	0.1uF	CAP0603-CAP	0603-CAP	Capacitor
C8	11pF	CAP0603-CAP	0603-CAP	Capacitor
C9	11pF	CAP0603-CAP	0603-CAP	Capacitor
C10	10uF	CAP_POL1206	EIA3216	Capacitor Polarized
C11	100uF	CAP_POL1206	EIA3216	Capacitor Polarized
C12	0.1uF	CAP0603-CAP	0603-CAP	Capacitor
C13	0.1uF	CAP0603-CAP	0603-CAP	Capacitor
C14	0.1uF	CAP0603-CAP	0603-CAP	Capacitor
C15	0.1uF	CAP0603-CAP	0603-CAP	Capacitor
C16	0.1uF	CAP0603-CAP	0603-CAP	Capacitor
C17	0.1uF	CAP0603-CAP	0603-CAP	Capacitor
C18	0.1uF	CAP0603-CAP	0603-CAP	Capacitor
C19	0.1uF	CAP0603-CAP	0603-CAP	Capacitor
C20	0.1uF	CAP0603-CAP	0603-CAP	Capacitor
C21	0.1uF	CAP0603-CAP	0603-CAP	Capacitor
IC2	LM1117	V_REG_LM1117SOT223	SOT223	Voltage Regulator
J1	J00-0065NL	J00-0065NL	J00-0065NL	MagJack 10/100 Ethernet Jack
J2	POWER_JACKPTH	POWER_JACKPTH	POWER_JACK_PTH	Power Jack
L1	FB	INDUCTOR0603	0603	Inductors Ferrite Bead
LED1	GRN	LED0603	LED-0603	LEDs
LED2	YEL	LED0603	LED-0603	LEDs
M1	MRF24J40MA	MRF24J40MA	MRF24J40MA	Microchip Wireless 2.4GHz module
R1	2.32K 1%	RESISTOR0603-RES	0603-RES	Resistor
R2	10K	RESISTOR0603-RES	0603-RES	Resistor
R3	10K	RESISTOR0603-RES	0603-RES	Resistor
R4	49.9 1%	RESISTOR0603-RES	0603-RES	Resistor
R5	49.9 1%	RESISTOR0603-RES	0603-RES	Resistor
R6	49.9 1%	RESISTOR0603-RES	0603-RES	Resistor
R7	49.9 1%	RESISTOR0603-RES	0603-RES	Resistor
R8	180	RESISTOR0603-RES	0603-RES	Resistor
R9	180	RESISTOR0603-RES	0603-RES	Resistor
R10	10K	RESISTOR0603-RES	0603-RES	Resistor
R11	10K	RESISTOR0603-RES	0603-RES	Resistor
R12	10K	RESISTOR0603-RES	0603-RES	Resistor
R13	4.7K	RESISTOR0603-RES	0603-RES	Resistor
R14	150	RESISTOR0603-RES	0603-RES	Resistor
R15	150	RESISTOR0603-RES	0603-RES	Resistor
U$1	TEMT6000	TEMT6000	TEMT6000-SEN	Ambient Light Sensor
U1	PIC32MX340F512H-	PIC32MX340F512H-	TQFP64_10X10MC	Microcontroller
U2	ENC28J60-X/SO	ENC28J60-X/SO	SOIC28-W_MC	Ethernet Controller
Y1	8MHz	RESONATORSMD	RESONATOR-SMD	Resonator
Y2	25MHz	CRYSTAL5X3	CRYSTAL-SMD-5X3	Crystals
Y3	32kHz	CRYSTAL32-SMD	CRYSTAL-32KHZ-SMD	Crystals

B.2 Node

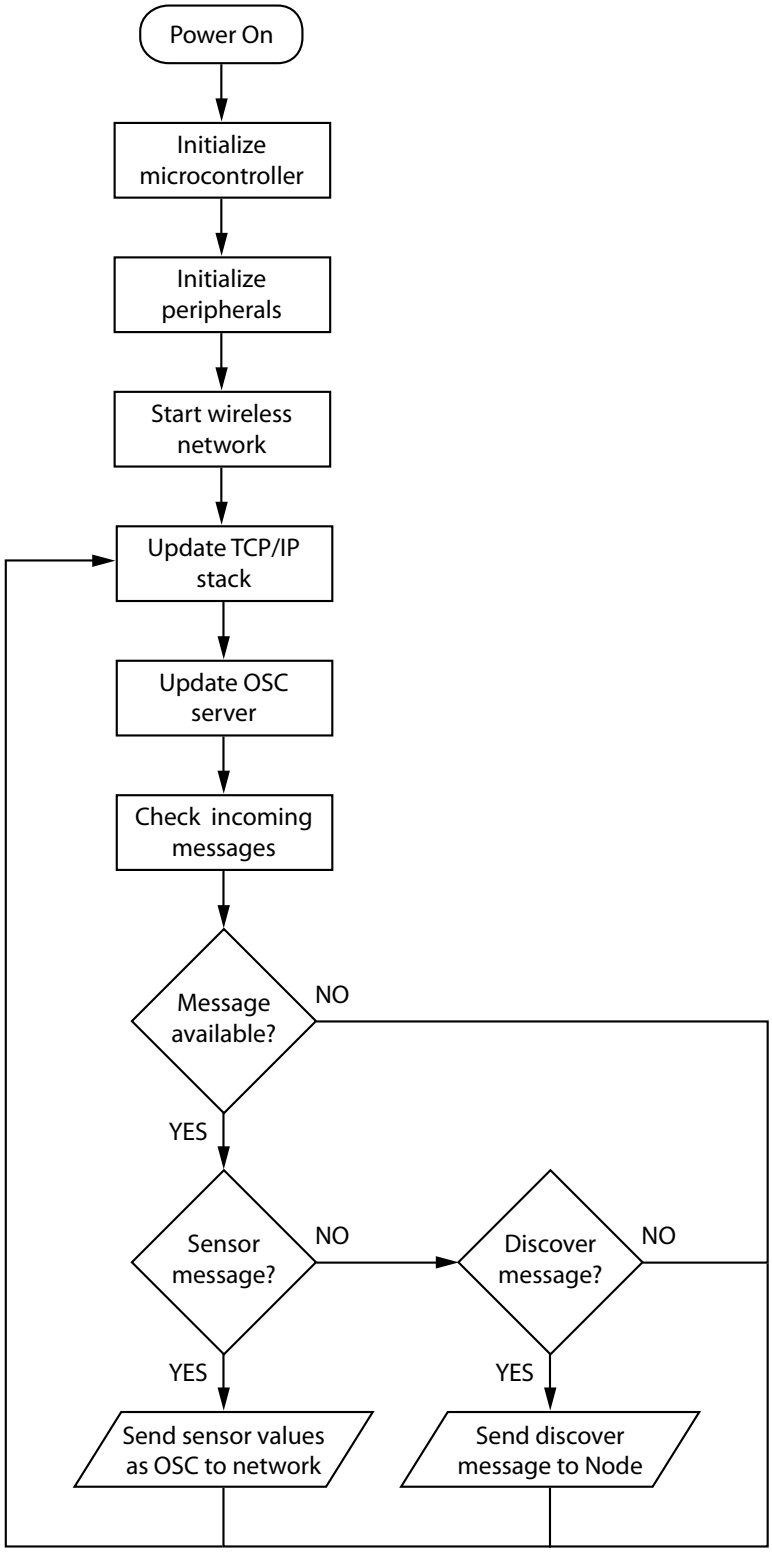
Partlist for the Node

Part	Value	Device	Package	Description
C1	10uF	CAP_POL1206	EIA3216	Capacitor Polarized
C2	10uF	CAP_POL1206	EIA3216	Capacitor Polarized
C3	0.1uF	CAP0603-CAP	0603-CAP	Capacitor
C4	0.1uF	CAP0603-CAP	0603-CAP	Capacitor
C5	0.1uF	CAP0603-CAP	0603-CAP	Capacitor
C6	0.1uF	CAP0603-CAP	0603-CAP	Capacitor
LED1	WHT	LED0603	LED-0603	LEDs
M1	MRF24J40MABTM	MRF24J40MABTM	MRF24J40MA-BTM	Microchip Wireless 2.4GHz module
R1	100K	RESISTOR0603-RES	0603-RES	Resistor
R2	10K	RESISTOR0603-RES	0603-RES	Resistor
R3	100K	RESISTOR0603-RES	0603-RES	Resistor
R6	150	RESISTOR0603-RES	0603-RES	Resistor
R11	10K	RESISTOR0603-RES	0603-RES	Resistor
R12	10K	RESISTOR0603-RES	0603-RES	Resistor
R13	4.7K	RESISTOR0603-RES	0603-RES	Resistor
U$1	TEMT6000	TEMT6000	TEMT6000-SEN	Ambient Light Sensor
U$2	PIC18F46K20-X/PT	PIC18F46K20-X/PT	TQFP44_MC	Microcontroller
U2	MIC5216	V_REG_LDOSMD	SOT23-5	Voltage Regulator LDO
U3	MCP9701A	MCP9700SMD	SOT23-3	Temperature Sensor
Y1	16MHz	RESONATORSMD	RESONATOR-SMD	Resonator

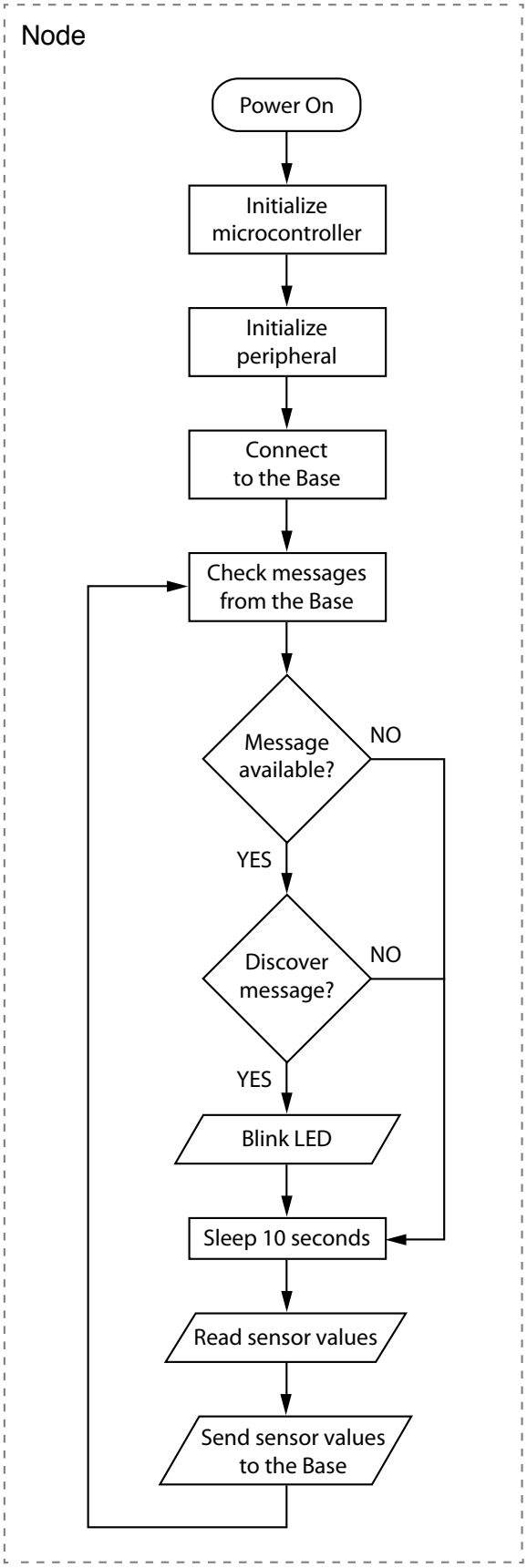
C. Flow Charts

C.1 Base

Base


Power On
Initialize microcontroller
Initialize peripherals
Start wireless network
Update TCP/IP stack
Update OSC server
Check incoming messages
Message available?
NO
YES
Sensor message?
NO
Discover message?
NO
YES
YES
Send sensor values as OSC to network
Send discover message to Node

C.2 Node


Node
Power On
Initialize microcontroller
Initialize peripheral
Connect to the Base
Check messages from the Base
Message available?
NO
YES
Discover message?
NO
YES
Blink LED
Sleep 10 seconds
Read sensor values
Send sensor values to the Base

D. Source Code

D.1 Base

```
/*
 HardwareProfile.h
 */

/********************************************************************
 *
 *  Hardware specific definitions for:
 *    - PIC32 Starter Kit
 *    - PIC32MX360F512L
 *    - Ethernet PICtail Plus (ENC28J60)
 *
 ********************************************************************
 * FileName:        HardwareProfile.h
 * Dependencies:    Compiler.h
 * Processor:       PIC32
 * Compiler:        Microchip C32 v1.11 or higher
 * Company:         Microchip Technology, Inc.
 *
 * Software License Agreement
 *
 * Copyright (C) 2002-2010 Microchip Technology Inc.  All rights
 * reserved.
 *
 * Microchip licenses to you the right to use, modify, copy, and
 * distribute:
 * (i)  the Software when embedded on a Microchip microcontroller or
 *      digital signal controller product ("Device") which is
 *      integrated into Licensee's product; or
 * (ii) ONLY the Software driver source files ENC28J60.c, ENC28J60.h,
 *      ENCX24J600.c and ENCX24J600.h ported to a non-Microchip device
 *      used in conjunction with a Microchip ethernet controller for
 *      the sole purpose of interfacing with the ethernet controller.
 *
 * You should refer to the license agreement accompanying this
 * Software for additional information regarding your rights and
 * obligations.
 *
 * THE SOFTWARE AND DOCUMENTATION ARE PROVIDED "AS IS" WITHOUT
 * WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING WITHOUT
 * LIMITATION, ANY WARRANTY OF MERCHANTABILITY, FITNESS FOR A
 * PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT. IN NO EVENT SHALL
 * MICROCHIP BE LIABLE FOR ANY INCIDENTAL, SPECIAL, INDIRECT OR
 * CONSEQUENTIAL DAMAGES, LOST PROFITS OR LOST DATA, COST OF
 * PROCUREMENT OF SUBSTITUTE GOODS, TECHNOLOGY OR SERVICES, ANY CLAIMS
 * BY THIRD PARTIES (INCLUDING BUT NOT LIMITED TO ANY DEFENSE
 * THEREOF), ANY CLAIMS FOR INDEMNITY OR CONTRIBUTION, OR OTHER
 * SIMILAR COSTS, WHETHER ASSERTED ON THE BASIS OF CONTRACT, TORT
 * (INCLUDING NEGLIGENCE), BREACH OF WARRANTY, OR OTHERWISE.
 *
 *
 * Author               Date        Comment
 *~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 * Howard Schlunder     09/16/2010  Regenerated for specific boards
 ********************************************************************/
#ifndef HARDWARE_PROFILE_H
#define HARDWARE_PROFILE_H

#include "Compiler.h"

// Set configuration fuses (but only in MainDemo.c where THIS_IS_STACK_APPLICATION is defined)
#if defined(THIS_IS_STACK_APPLICATION)
    #pragma config FPLLODIV = DIV_1, FPLLMUL = MUL_20, FPLLIDIV = DIV_2, FWDTEN = OFF
    #pragma config FPBDIV = DIV_2, POSCMOD = HS, FNOSC = PRIPLL, CP = OFF, BWP = OFF, DEBUG = ON, ICESEL = ICS_PGx1
#endif

#if defined(__PIC32MX__)

    #define PIC32MX_SPI2_SDO_SCK_MASK_VALUE      (0x00000140)
    #define PIC32MX_SPI2_SDI_MASK_VALUE          (0x00000080)
    #define PIC32MX_INT3_MASK_VALUE              (0x00000400)
    //#define PIC32MX_INT2_MASK_VALUE              (0x00000010)
    //#define PIC32MX_INT1_MASK_VALUE              (0x00000008)
    //#define PIC32MX_INT1_MASK_VALUE              (0x00000100)
    /* MAX SPI CLOCK FREQ SUPPORTED FOR MIWI TRANSCIEVER */
    #define MAX_SPI_CLK_FREQ_FOR_P2P             (1000000)

#endif

// Clock frequency values
// These directly influence timed events using the Tick module.  They also are used for UART and SPI baud rate generation.
#define GetSystemClock()        (80000000ul)            // Hz
#define GetInstructionClock()   (GetSystemClock()/1)    // Normally GetSystemClock()/4 for PIC18, GetSystemClock()/2 for PIC24/
dsPIC, and GetSystemClock()/1 for PIC32.  Might need changing if using Doze modes.
#define GetPeripheralClock()    (GetSystemClock()/4)    // Normally GetSystemClock()/4 for PIC18, GetSystemClock()/2 for PIC24/
dsPIC, and GetSystemClock()/1 for PIC32.  Divisor may be different if using a PIC32 since it's configurable.

// Hardware I/O pin mappings

#define CLOCK_FREQ      80000000
#define USE_DATA_EEPROM

// Transceiver Configuration
#define RFIF                IFS0bits.INT3IF
#define RFIE                IEC0bits.INT3IE

#define PHY_CS              LATEbits.LATE0
#define PHY_CS_TRIS         TRISEbits.TRISE0

#define PHY_RESETn          LATEbits.LATE1
#define PHY_RESETn_TRIS     TRISEbits.TRISE1

#define PHY_WAKE            LATEbits.LATE2
#define PHY_WAKE_TRIS       TRISEbits.TRISE2

#define RF_INT_PIN          PORTDbits.RD10
#define RF_INT_TRIS         TRISDbits.TRISD10
```

```
#define SPI_SDI             PORTGbits.RG7
#define SDI_TRIS            TRISGbits.TRISG7

#define SPI_SDO             LATGbits.LATG8
#define SDO_TRIS            TRISGbits.TRISG8

#define SPI_SCK             LATGbits.LATG6
#define SCK_TRIS            TRISGbits.TRISG6

// Misc
#define LED_1               LATGbits.LATG9
#define LED_1_TRIS          TRISGbits.TRISG9

#define TMRL                TMR2

// UART configuration (not too important since we don't have a UART
// connector attached normally, but needed to compile if the STACK_USE_UART
// or STACK_USE_UART2TCP_BRIDGE features are enabled.
#define UARTTX_TRIS         (TRISFbits.TRISF5)
#define UARTRX_TRIS         (TRISFbits.TRISF4)

// ENC28J60 I/O pins
#define ENC_CS_TRIS         (TRISFbits.TRISF0)
#define ENC_CS_IO           (LATFbits.LATF0)
#define ENC_RST_TRIS        (TRISFbits.TRISF1)  // Not connected by default.  It is okay to leave this pin completely
unconnected, in which case this macro should simply be left undefined.
#define ENC_RST_IO          (LATFbits.LATF1)

// SPI SCK, SDI, SDO pins are automatically controlled by the
#define ENC_SPI_IF          (IFS0bits.SPI1RXIF)
#define ENC_SSPBUF          (SPI1BUF)
#define ENC_SPICON1         (SPI1CON)
#define ENC_SPICON1bits     (SPI1CONbits)
#define ENC_SPIBRG          (SPI1BRG)
#define ENC_SPISTATbits     (SPI1STATbits)

#define SaveAppConfig(a)

#endif // #ifndef HARDWARE_PROFILE_H
```

```
/*
 OSC.h - OSC implementation for the Microchip TCP/IP Stack
 */

#ifndef __OSC_H
#define __OSC_H

#include <string.h>
#include <stdarg.h>
#include <stdlib.h>
#include "GenericTypeDefs.h"

#define OSC_PORT 12345

#define OSC_SUCCESS 0
#define OSC_ERROR_BAD_DATA 1
#define OSC_ERROR_BAD_BUFFER 2
#define OSC_ERROR_BAD_INDEX 3
#define OSC_ERROR_BAD_PROPERTY 4
#define OSC_ERROR_BAD_TAG 5

#define OSC_MAX_MESSAGE_IN 300
#define OSC_MAX_MESSAGE_OUT 300

#define OSC_TEXT_BUFFER 50

#define OSC_SUB_SYSTEM_SIZE 10

void OSCServerTask(void);
void OSCCloseSocket(void);
int OSCisActive(void);
int OSCCreateMessage(char* address, char* format, ... );
int OSCRegSubSystem(const char* name, int (*subReceiveMessage)(char* bp, int len));
int OSCIntReceiverHelper(char *bp, int len, const char* subSystemName,
                         int (*propertySet)(int property, int value),
                         int(*propertyGet)(int property),
                         char* propertyNames[]);
int OSCIndexIntReceiverHelper(char *bp, int len, int indexCount, const char* subSystemName,
                         int (*propertySet)(int index, int property, int value),
                         int(*propertyGet)(int index, int property),
                         char* propertyNames[]);

#endif
```

```c
/*
 OSC.c - OSC implementation for the Microchip TCP/IP Stack

 Based on code from MakingThings.
 The code has been rewritten to work with the Microchip TCP/IP Stack:
 http://www.makingthings.com/ref/firmware/html/osc_8c-source.html
 */

#include <ctype.h>
#include "TCPIP Stack/TCPIP.h"
#include "OSC/OSC.h"

typedef struct _OSCSubSystem
{
    const char* name;
    int (*receiveMessage)(char* bp, int len);
} OSCSubSystem;

static enum _OSCState
{
    OSC_INIT = 0,
    OSC_SOCKET,
    OSC_LISTEN,
    OSC_CLOSE,
} OSCState = OSC_INIT;

typedef struct _OSCStruct
{
    char buffer[OSC_MAX_MESSAGE_OUT];
    char recvBuffer[OSC_MAX_MESSAGE_IN];
    char* bufferPtr;
    int bufferRemaining;
    int messages;
    char textBuf[OSC_TEXT_BUFFER];
    int regSubSystems;
    OSCSubSystem subSystem[OSC_SUB_SYSTEM_SIZE];
    UDP_SOCKET socket;
} OSCStruct;

OSCStruct OSC;

void OSCResetBuffer(void)
{
    OSC.bufferPtr = OSC.buffer;
    OSC.bufferRemaining = OSC_MAX_MESSAGE_OUT;
    OSC.messages = 0;
}

int OSCEndianSwap(int a)
{
    return ((a & 0x000000FF) << 24 ) |
           ((a & 0x0000FF00) << 8 ) |
           ((a & 0x00FF0000) >> 8 ) |
           ((a & 0xFF000000) >> 24 );
}

void OSCCloseSocket(void)
{
    OSCState = OSC_CLOSE;
}

int OSCisActive(void)
{
    if(OSC.socket != INVALID_UDP_SOCKET)
    {
        return 1;
    }
    return 0;
}

int OSCRegSubSystem(const char* name, int (*subReceiveMessage)(char* bp, int len))
{
    int subSystem = OSC.regSubSystems++;

    if(OSC.regSubSystems > OSC_SUB_SYSTEM_SIZE)
    {
        return OSC_ERROR_BAD_INDEX;
    }

    OSCSubSystem *sub = &OSC.subSystem[subSystem];
    sub->name = name;
    sub->receiveMessage = subReceiveMessage;

    return OSC_SUCCESS;
}

int OSCPropertyLookup(char** properties, char* property)
{
    char** p = properties;
    int index = 0;
    while(*p != NULL)
    {
        if(strcmp(property, *p++) == 0)
        {
            return index;
        }
        index++;
    }
    return -1;
}

char* OSCFindDataTag(char* bp, int len)
{
    while(*bp != ',' && len-- > 0)
    {
        bp++;
    }
```

```
    if(len <= 0)
    {
        return NULL;
    }
    else
    {
        return bp;
    }
}

int OSCReadInt(char*bp)
{
    int v = *((int*)bp);
    v = OSCEndianSwap(v);
    return v;
}

int OSCExtractData(char* bp, char* format, ...)
{
    va_list args;
    va_start(args, format);
    int count = 0;
    BOOL cont = TRUE;

    int strLen = strlen(bp);
    int pad = 4 - (strLen % 4);

    char* data = bp + (strLen + pad);

    char* fp;
    char* tp = bp + 1; // Skip comma ','
    for(fp = format; cont == TRUE; fp++)
    {
        switch(*fp)
        {
            case 'i':
                if(*tp == 'i')
                {
                    *(va_arg(args, int*)) = OSCReadInt(data);
                    data += 4;
                    count++;
                }
                else
                {
                    cont = FALSE;
                }
                break;
            default:
                cont = FALSE;
                break;
        }
        tp++;
    }
    va_end(args);

    return count;
}

int OSCNumberMatch(int count, char* bp, int* bits)
{
    int n = 0;
    int digits = 0;
    while(isdigit(*bp))
    {
        digits++;
        n = n * 10 + (*bp++ - '0');
    }

    *bits = -1;

    if(digits == 0)
    {
        return -1;
    }
    return n;
}

int OSCIntReceiverHelper(char *bp, int len, const char* subSystemName,
                         int (*propertySet)(int property, int value),
                         int(*propertyGet)(int property),
                         char* propertyNames[])
{
    if(*bp == '\0' || *bp == ' ')
    {
        return OSC_SUCCESS;
    }

    int propertyIndex = OSCPropertyLookup(propertyNames, bp);
    if(propertyIndex == -1)
    {
        return OSC_ERROR_BAD_PROPERTY;
    }

    char* type = OSCFindDataTag(bp, len);
    if(type == NULL)
    {
        return OSC_ERROR_BAD_TAG;
    }

    int value;

    if(type[1] == 'i')
    {
        int count = OSCExtractData(type, "i", &value);

        if(count != 1)
        {
```

```
            return OSC_ERROR_BAD_DATA;
        }

        (*propertySet)(propertyIndex, value);
    }
    else
    {
        value = (*propertyGet)(propertyIndex);
        sprintf(OSC.textBuf, "/%s/%s", subSystemName, propertyNames[propertyIndex]);
        OSCCreateMessage(OSC.textBuf, ",i", value);
    }

    return OSC_SUCCESS;
}

int OSCIndexIntReceiverHelper(char *bp, int len, int indexCount, const char* subSystemName,
                        int (*propertySet)(int index, int property, int value),
                        int(*propertyGet)(int index, int property),
                        char* propertyNames[])
{
    int i;
    if(*bp == '\0' || *bp == ' ')
    {
        return OSC_SUCCESS;
    }

    char* prop = strchr(bp, '/');
    char* propHelp = NULL;
    if(prop == NULL)
    {
        propHelp = bp + strlen(bp);
    }

    *prop = 0;

    int bits;
    int number = OSCNumberMatch(indexCount, bp, &bits);
    if(number == -1 && bits == -1)
    {
        return OSC_ERROR_BAD_INDEX;
    }

    *prop = '/';

    if(propHelp != NULL && (*propHelp == '\0' || *propHelp == ' '))
    {
        return OSC_SUCCESS;
    }

    int propertyIndex = OSCPropertyLookup(propertyNames, prop + 1);
    if(propertyIndex == -1)
    {
        return OSC_ERROR_BAD_PROPERTY;
    }

    char* type = OSCFindDataTag(bp, len);
    if(type == NULL)
    {
        return OSC_ERROR_BAD_TAG;
    }

    if(type[1] == 'i' || type[1] == 'f')
    {
        int value;
        int count = OSCExtractData(type, "i", &value);
        if(count != 1)
        {
            return OSC_ERROR_BAD_DATA;
        }

        if(number != -1)
        {
            (*propertySet)(number, propertyIndex, value);
        }
        else
        {
            int index = 0;
            while(bits > 0 && index < indexCount)
            {
                if(bits & 1)
                {
                    (*propertySet)(index, propertyIndex, value);
                    bits >>= 1;
                    index++;
                }
            }
        }
    }
    else
    {
        if(number != -1)
        {
            int value = (*propertyGet)(number, propertyIndex);
            sprintf(OSC.textBuf, "/%s/%d/%s", subSystemName, number, propertyNames[propertyIndex]);
            OSCCreateMessage(OSC.textBuf, ",i", value);
        }
        else
        {
            int index = 0;
            while(bits > 0 && index < indexCount)
            {
                if(bits & 1)
                {
                    int value = (*propertyGet)(index, propertyIndex);
                    sprintf(OSC.textBuf, "/%s/%d/%s", subSystemName, index, propertyNames[propertyIndex]);
                    OSCCreateMessage(OSC.textBuf, ",i", value);
                }
```

```
                bits >>= 1;
                index++;
            }
        }
    }

    return OSC_SUCCESS;
}

int OSCSendPacket(void)
{
    char* bp;
    int len;
    int maxPut;

    if(OSC.messages == 0)
    {
        return OSC_SUCCESS;
    }

    bp = OSC.buffer;
    len = OSC_MAX_MESSAGE_OUT - OSC.bufferRemaining;

    if(OSC.messages == 1)
    {
        bp += 20;
        len -= 20;
    }

    if(UDPIsPutReady(OSC.socket))   // Get UDP TX FIFO free space
    {
        UDPPutArray(bp, len);
        UDPFlush();
    }
    OSCResetBuffer();
    return OSC_SUCCESS;
}

int OSCReceiveMessage(char* bp, int len)
{
    int i;
    if(*bp == '/')
    {
        if(strlen(bp) > (unsigned int)len)
        {
            return OSC_ERROR_BAD_DATA;
        }

        char* nextChar = bp + 1;
        if(*nextChar == '\0' || *nextChar == ' ')
        {
            return OSC_SUCCESS;
        }

        char *nextSlash = strchr(nextChar, '/');
        if(nextSlash != NULL)
        {
            *nextSlash = 0;
        }

        int count = 0;
        for(i = 0; i < OSC.regSubSystems; i++)
        {
            OSCSubSystem *sub = &OSC.subSystem[i];
            int match = strcmp(nextChar, sub->name);
            if(match == 0)
            {
                if(nextSlash)
                {
                    (sub->receiveMessage)(nextSlash + 1, len - (nextSlash - bp) - 1);
                }
                else
                {
                    char* noNextSlash = bp + strlen(bp);
                    (sub->receiveMessage)(noNextSlash, 0);
                }
            }
        }
    }
    else
    {
        return OSC_ERROR_BAD_DATA;
    }
    return OSC_SUCCESS;
}

int OSCReceivePacket(char* bp, int len)
{
    int status = -1;

    if(len > 0)
    {
        char* bp = OSC.recvBuffer;
        switch(*bp)
        {
            case '/':
                status = OSCReceiveMessage(bp, len);
                break;
            case '#':
                if(strcmp(bp, "#bundle") == 0)
                {
                    bp += 16;
                    len -= 16;
                    while(len > 0)
                    {
                        int messageLen = OSCEndianSwap(*((int*)bp));
                        bp += 4;
```

```
                        len -= 4;
                        if(messageLen <= len)
                        {
                            OSCReceivePacket(bp, messageLen);
                        }
                        bp += messageLen;
                        len -= messageLen;
                    }
                }
                break;
            default:
                break;
        }
    }
    UDPDiscard();
    return status;
}

void OSCServerTask(void)
{
    char* bp;
    int len;
    int maxGet, maxPut;

    switch(OSCState)
    {
        case OSC_INIT:
            OSC.socket = INVALID_UDP_SOCKET;
            OSCResetBuffer();
            OSCState = OSC_SOCKET;
            break;
        case OSC_SOCKET:
            OSC.socket = UDPOpen(OSC_PORT, NULL, OSC_PORT);
            if(OSC.socket == INVALID_UDP_SOCKET)
            {
                break;
            }
            OSCState = OSC_LISTEN;
            break;
        case OSC_LISTEN:
            // Figure out how many bytes have been received and how many we can transmit.
            if(UDPIsGetReady(OSC.socket))   // Get UDP RX FIFO byte count
            {
                int len = UDPGetArray(OSC.recvBuffer, OSC_MAX_MESSAGE_IN);
                OSCReceivePacket(OSC.recvBuffer, len);
                UDPDiscard();
            UDPSocketInfo[OSC.socket].remotePort = OSC_PORT;

            }
            OSCSendPacket();
            break;
        case OSC_CLOSE:
            if(OSC.socket != INVALID_UDP_SOCKET)
            {
                UDPClose(OSC.socket);
                OSCState = OSC_INIT;
            }
            break;
    }
}

char* OSCWriteString(char* bp, int* len, char* string)
{
    int i;
    int strLen = strlen(string);
    int pad = 4 - (strLen % 4);

    *len -= strLen + pad;

    strcpy(bp, string);
    bp += strLen;

    for(i = 0; i < pad; i++)
    {
        *(bp++) = 0;
    }

    return bp;
}

char* OSCCreateInternalMessage(char* bp, int* len, char* address, char* format, va_list args)
{
    char* fp;
    BOOL cont = TRUE;
    int val;

    // Write address as OSC string
    bp = OSCWriteString(bp, len, address);
    // Write format as OSC string
    bp = OSCWriteString(bp, len, format);

    for(fp = format + 1; cont == TRUE; fp++)
    {
        switch(*fp)
        {
            case 'i':
                *len -= 4;
                if(*len >= 0)
                {
                    val = va_arg(args, int);
                    val = OSCEndianSwap(val);
                    *((int*)bp) = val;
                    bp += 4;
                }
                else
                {
                    cont = FALSE;
```

```
                }
                break;
            default:
                cont = FALSE;
                break;
        }
    }

    return cont ? NULL : bp;
}

char* OSCWriteTimeTag(char* bp, int* len, int a, int b)
{
    if(*len < 8)
    {
        return NULL;
    }

    *((int*)bp) = OSCEndianSwap(a);
    *((int*)bp) = OSCEndianSwap(b);
    bp += 8;
    *len -= 8;

    return bp;
}

char* OSCCreateBundle(char* bp, int* len, int a, int b)
{
    char* bp2 = bp;

    bp2 = OSCWriteString(bp2, len, "#bundle");
    if(bp2 == NULL)
    {
        return NULL;
    }

    bp2 = OSCWriteTimeTag(bp2, len, a, b);

    if(bp2 == NULL)
    {
        return NULL;
    }

    return bp2;
}

int OSCCreateMessage(char* address, char* format, ...)
{
    char* bp;
    int len;
    int count = 0;

    if(address == NULL || format == NULL || *format != ',')
    {
        return OSC_ERROR_BAD_DATA;
    }

    if( OSC.bufferPtr == NULL)
    {
        OSCResetBuffer();
    }

    bp = OSC.bufferPtr;
    len = OSC.bufferRemaining;

    if(bp == OSC.buffer)
    {
        bp = OSCCreateBundle(bp, &len, 0, 0);
        if(bp == NULL)
        {
            return OSC_ERROR_BAD_BUFFER;
        }
    }

    int* lp = (int*)bp;
    bp += 4;
    len -=4;

    char* mp = bp;

    if(len > 0)
    {
        va_list args;
        va_start(args, format);

        bp = OSCCreateInternalMessage(bp, &len, address, format, args);

        va_end(args);
    }
    else
    {
        bp = NULL;
    }

    if(bp != NULL)
    {
        *lp = OSCEndianSwap(bp - mp);

        OSC.bufferPtr = bp;
        OSC.bufferRemaining = len;
        OSC.messages++;
    }
    return ((bp != NULL) ? OSC_SUCCESS : OSC_ERROR_BAD_BUFFER);
}
```

```c
/*
 BaseHandler.h
 */
#ifndef __BASE_HANDLER_H_
#define __BASE_HANDLER_H_

int baseReceiveMessage(char* bp, int len);

#endif

/*
 BaseHandler.c
 */
#include "OSC\OSC.h"
#include "HardwareProfile.h"

const char baseSubSystemName[] = "base";

const char* baseGetSubSystemName(void)
{
    return baseSubSystemName;
}

char* basePropertyNames[] = {"led", "active", 0};

int basePropertySet(int property, int value)
{
    switch(property)
    {
        case 0:
            LED_1 = (BYTE)value;
            break;
        case 1:
            OSCCloseSocket();
            break;
    }
    return OSC_SUCCESS;
}

int basePropertyGet(int property)
{
    int value;
    switch(property)
    {
        case 0:
            value = LED_1;
            break;
        case 1:
            value = OSCisActive();
            break;
    }
    return value;
}

int baseReceiveMessage(char* bp, int len)
{
    return OSCIntReceiverHelper(bp, len, baseSubSystemName,
                        basePropertySet,
                        basePropertyGet,
                        basePropertyNames);
}
```

```
/*
 NodeHandler.h
 */
#ifndef __NODE_HANDLER_H_
#define __NODE_HANDLER_H_

int nodeReceiveMessage(char* bp, int len);

#endif

/*
 NodeHandler.c
 */
#include "OSC\OSC.h"
#include "HardwareProfile.h"
#include "WirelessProtocols/MCHP_API.h"

BYTE nodeAddressSuffix[8] = {EUI_0, EUI_1, EUI_2, EUI_3, EUI_4, EUI_5, EUI_6, EUI_7};

const char nodeSubSystemName[] = "node";

const char* nodeGetSubSystemName(void)
{
    return nodeSubSystemName;
}

char* nodePropertyNames[] = {"discover", 0};

int nodePropertySet(int index, int property, int value)
{
    switch(property)
    {
        case 0:
            if(value == 1 || value == 0)
            {
                MiApp_FlushTx();
                MiApp_WriteData(0x30 | (BYTE)value);
                nodeAddressSuffix[0] = (BYTE)index;
                MiApp_UnicastAddress(nodeAddressSuffix, TRUE, FALSE);
            }
            break;
    }
    return OSC_SUCCESS;
}

int nodePropertyGet(int index, int property)
{
    int value;
    switch(property)
    {
        case 0:
            value = 0;
            break;
    }
    return value;
}

int nodeReceiveMessage(char* bp, int len)
{
    return OSCIndexIntReceiverHelper(bp, len, 1, nodeSubSystemName,
                            nodePropertySet,
                            nodePropertyGet,
                            nodePropertyNames);
}
```

```c
/*
 main.c - main loop for the Base
 */

#define THIS_IS_STACK_APPLICATION

#include "TCPIP Stack/TCPIP.h"
#include "WirelessProtocols/Console.h"
#include "ConfigApp.h"
#include "HardwareProfile.h"
#include "WirelessProtocols/MCHP_API.h"
#include "OSC/OSC.h"
#include "BaseHandler.h"
#include "NodeHandler.h"

#define BASE    0x01
#define NODE    0x02

#if ADDITIONAL_NODE_ID_SIZE > 0
    BYTE AdditionalNodeID[ADDITIONAL_NODE_ID_SIZE] = {BASE};
#endif

BYTE myChannel = 11;
APP_CONFIG AppConfig;
static unsigned short wOriginalAppConfigChecksum;   // Checksum of the ROM defaults for AppConfig
char nodeID[10];
extern char baseSubSystemName[];
extern char nodeSubSystemName[];

static void InitAppConfig(void);
static void InitBoard(void);

#if defined(__C32__)
    void _general_exception_handler(unsigned cause, unsigned status)
    {
        Nop();
        Nop();
    }
#endif

int main(void)
{
    BYTE count = 0;
    BYTE i;
    BYTE buffer[15];
    WORD tmp[3];
    char textBuf[60];

    /*********************************************************************/
    // Initialize the system
    /*********************************************************************/
    ConsoleInit();
    InitBoard();
    TickInit();
    InitAppConfig();
    StackInit();

    #if defined(PROTOCOL_P2P)
        Printf("\r\n\r\nStarting MiWi(TM) P2P Stack ...");
    #endif
    #if defined(PROTOCOL_MIWI)
        Printf("\r\n\r\nStarting MiWi(TM) Stack ...");
    #endif

    #if defined(MRF24J40)
        Printf("\r\n     RF Transceiver: MRF24J40\r\n");
    #elif defined(MRF49XA)
        Printf("\r\n     RF Transceiver: MRF49XA\r\n");
    #elif defined(MRF89XA)
        Printf("\r\n     RF Transceiver: MRF89XA\r\n");
    #endif

    LED_1 = 0;

    /*********************************************************************/
    // Initialize Microchip proprietary protocol. Which protocol to use
    // depends on the configuration in ConfigApp.h
    /*********************************************************************/
    MiApp_ProtocolInit(FALSE);

    #ifndef ENABLE_ED_SCAN
        MiApp_SetChannel(myChannel);
    #endif

    MiApp_ConnectionMode(ENABLE_ALL_CONN);

    #ifdef ENABLE_ED_SCAN
        Printf("\r\nActive Scanning Energy Scanning");
        MiApp_StartConnection(START_CONN_ENERGY_SCN, 10, 0xFFFFFFFF);
    #else
        MiApp_StartConnection(START_CONN_DIRECT, 10, 0);
    #endif

    // Turn on LED 1 to indicate ready to accept new connections
    LED_1 = 1;

    // OSC HANDLERS
    OSCRegSubSystem(baseSubSystemName, baseReceiveMessage);
    OSCRegSubSystem(nodeSubSystemName, nodeReceiveMessage);

    while(1)
    {
        static DWORD t = 0;

        StackTask();
        StackApplications();
        OSCServerTask();
```

```
        if( MiApp_MessageAvailable() )
        {
            for(i = 0; i < rxMessage.PayloadSize; i++)
            {
                buffer[i] = rxMessage.Payload[i];
            }

            for(i = 0; i < 3; i++)
            {
                tmp[i] = ((WORD)buffer[i * 2] << 8) & 0xFF00;
                tmp[i] |= buffer[i * 2 + 1];
            }

            #ifdef ENABLE_CONSOLE
            DWORD time = SNTPGetUTCSeconds();
            sprintf(textBuf, "%u, %u, %u, %u, %u, %u\r\n", (BYTE)rxMessage.SourceAddress[0], time, tmp[0], tmp[1], tmp[2],
(BYTE)rxMessage.PacketRSSI);
            #endif
            Printf(textBuf);
            sprintf(nodeID, "/node/%u", rxMessage.SourceAddress[0]);
            OSCCreateMessage(nodeID, ",iiii", tmp[0], tmp[1], tmp[2], (BYTE)rxMessage.PacketRSSI);

            MiApp_DiscardMessage();
        }
    }
}

static void InitBoard(void)
{
    // LEDs
    LED_1_TRIS = 0;
    LED_1 = 0;

    PHY_CS_TRIS = 0;
    PHY_CS = 1;
    PHY_RESETn_TRIS = 0;
    PHY_RESETn = 1;

    RF_INT_TRIS = 1;

    SDI_TRIS = 1;
    SDO_TRIS = 0;
    SCK_TRIS = 0;
    SPI_SDO = 0;
    SPI_SCK = 0;

    PHY_WAKE_TRIS = 0;
    PHY_WAKE = 1;

    #ifdef __PIC32MX__

        // Enable optimal performance
        SYSTEMConfigPerformance(GetSystemClock());
        mOSCSetPBDIV(OSC_PB_DIV_2);             // Use 1:2 CPU Core:Peripheral clocks

        #ifdef HARDWARE_SPI


        SPI2CON = 0x00008120;

        mSpiChnSetBrg(2,19);

        #endif

        /* Set the Port Directions of SDO, SDI, Clock & Slave Select Signal */
        mPORTGSetPinsDigitalOut(PIC32MX_SPI2_SDO_SCK_MASK_VALUE);
        mPORTGSetPinsDigitalIn(PIC32MX_SPI2_SDI_MASK_VALUE);

        /* Set the INT3 port pin to input */
        mPORTDSetPinsDigitalIn(BIT_10);

        // Set CS, RST, WKE to outputs
        mPORTESetPinsDigitalOut(BIT_0 | BIT_1 | BIT_2);

        /* Set the Interrupt Priority */
        mINT3SetIntPriority(4);

        /* Set Interrupt Subpriority Bits for INT1 */
        mINT3SetIntSubPriority(2);

        /* Set INT1 to falling edge */
        mINT3SetEdgeMode(0);

        /* Enable INT1 */
        mINT3IntEnable(1);
        /* Enable Multi Vectored Interrupts */
        INTEnableSystemMultiVectoredInt();

        // Disable JTAG port so we get our I/O pins back, but first
        // wait 50ms so if you want to reprogram the part with
        // JTAG
        DelayMs(50);
        #if !defined(__MPLAB_DEBUGGER_PIC32MXSK) && !defined(__MPLAB_DEBUGGER_FS2)
            DDPCONbits.JTAGEN = 0;
        #endif

    #endif

    RFIF = 0;

    if( RF_INT_PIN == 0 )
    {
        RFIF = 1;
    }

    // Deassert all chip select lines so there isn't any problem with
```

```
    // initialization order.

    #if defined(ENC_CS_TRIS)
        ENC_CS_IO = 1;
        ENC_CS_TRIS = 0;
    #endif
}

static ROM BYTE SerializedMACAddress[6] = {MY_DEFAULT_MAC_BYTE1, MY_DEFAULT_MAC_BYTE2, MY_DEFAULT_MAC_BYTE3,
MY_DEFAULT_MAC_BYTE4, MY_DEFAULT_MAC_BYTE5, MY_DEFAULT_MAC_BYTE6};

static void InitAppConfig(void)
{
    while(1)
    {
        // Start out zeroing all AppConfig bytes to ensure all fields are
        // deterministic for checksum generation
        memset((void*)&AppConfig, 0x00, sizeof(AppConfig));

        AppConfig.Flags.bIsDHCPEnabled = TRUE;
        AppConfig.Flags.bInConfigMode = TRUE;
        memcpypgm2ram((void*)&AppConfig.MyMACAddr, (ROM void*)SerializedMACAddress, sizeof(AppConfig.MyMACAddr));

        AppConfig.MyIPAddr.Val = MY_DEFAULT_IP_ADDR_BYTE1 | MY_DEFAULT_IP_ADDR_BYTE2<<8ul | MY_DEFAULT_IP_ADDR_BYTE3<<16ul |
MY_DEFAULT_IP_ADDR_BYTE4<<24ul;
        AppConfig.DefaultIPAddr.Val = AppConfig.MyIPAddr.Val;
        AppConfig.MyMask.Val = MY_DEFAULT_MASK_BYTE1 | MY_DEFAULT_MASK_BYTE2<<8ul | MY_DEFAULT_MASK_BYTE3<<16ul |
MY_DEFAULT_MASK_BYTE4<<24ul;
        AppConfig.DefaultMask.Val = AppConfig.MyMask.Val;
        AppConfig.MyGateway.Val = MY_DEFAULT_GATE_BYTE1 | MY_DEFAULT_GATE_BYTE2<<8ul | MY_DEFAULT_GATE_BYTE3<<16ul |
MY_DEFAULT_GATE_BYTE4<<24ul;
        AppConfig.PrimaryDNSServer.Val = MY_DEFAULT_PRIMARY_DNS_BYTE1 | MY_DEFAULT_PRIMARY_DNS_BYTE2<<8ul  |
MY_DEFAULT_PRIMARY_DNS_BYTE3<<16ul  | MY_DEFAULT_PRIMARY_DNS_BYTE4<<24ul;
        AppConfig.SecondaryDNSServer.Val = MY_DEFAULT_SECONDARY_DNS_BYTE1 | MY_DEFAULT_SECONDARY_DNS_BYTE2<<8ul  |
MY_DEFAULT_SECONDARY_DNS_BYTE3<<16ul  | MY_DEFAULT_SECONDARY_DNS_BYTE4<<24ul;

        // Load the default NetBIOS Host Name
        memcpypgm2ram(AppConfig.NetBIOSName, (ROM void*)MY_DEFAULT_HOST_NAME, 16);
        FormatNetBIOSName(AppConfig.NetBIOSName);

        // Compute the checksum of the AppConfig defaults as loaded from ROM
        wOriginalAppConfigChecksum = CalcIPChecksum((BYTE*)&AppConfig, sizeof(AppConfig));
        break;
    }
}
```

D.2 Node

```
/*
 HardwareProfile.h
 */

#ifndef _HARDWARE_PROFILE_H
#define _HARDWARE_PROFILE_H

    #include "GenericTypeDefs.h"
    #include "ConfigApp.h"


    #define SENSOR_NODE


    #define CLOCK_FREQ          16000000
    #define USE_DATA_EEPROM

    // Transceiver Configuration
    #define RFIF                INTCONbits.INT0IF
    #define RFIE                INTCONbits.INT0IE

    #define PHY_CS              LATBbits.LATB4
    #define PHY_CS_TRIS         TRISBbits.TRISB4
    #define PHY_CS_ANS          ANSELBbits.ANSB4

    #define PHY_RESETn          LATDbits.LATD5
    #define PHY_RESETn_TRIS     TRISDbits.TRISD5
    #define PHY_RESETn_ANS      ANSELDbits.ANSD5

    #define PHY_WAKE            LATDbits.LATD4
    #define PHY_WAKE_TRIS       TRISDbits.TRISD4
    #define PHY_WAKE_ANS        ANSELDbits.ANSD4

    #define RF_INT_PIN          PORTBbits.RB0
    #define RF_INT_TRIS         TRISBbits.TRISB0
    #define RF_INT_ANS          ANSELBbits.ANSB0

    #define SPI_SDI             PORTCbits.RC4
    #define SDI_TRIS            TRISCbits.TRISC4
    #define SDI_ANS             ANSELCbits.ANSC4

    #define SPI_SDO             LATCbits.LATC5
    #define SDO_TRIS            TRISCbits.TRISC5
    #define SDO_ANS             ANSELCbits.ANSC5

    #define SPI_SCK             LATCbits.LATC3
    #define SCK_TRIS            TRISCbits.TRISC3
    #define SCK_ANS             ANSELCbits.ANSC3

// ADC - Channels

    #define LGT_TRIS            TRISBbits.TRISB3
    #define LGT_ANS             ANSELBbits.ANSB3

    #define TMP_TRIS            TRISBbits.TRISB1
    #define TMP_ANS             ANSELBbits.ANSB1

    #define BAT_TRIS            TRISEbits.TRISE1
    #define BAT_ANS             ANSELEbits.ANSE1

    #define AN0_TRIS            TRISAbits.TRISA0
    #define AN0_ANS             ANSELAbits.ANSA0
    #define AN1_TRIS            TRISAbits.TRISA1
    #define AN1_ANS             ANSELAbits.ANSA1
    #define AN2_TRIS            TRISAbits.TRISA2
    #define AN2_ANS             ANSELAbits.ANSA2
    #define AN3_TRIS            TRISAbits.TRISA3
    #define AN3_ANS             ANSELAbits.ANSA3
    #define AN4_TRIS            TRISAbits.TRISA5
    #define AN4_ANS             ANSELAbits.ANSA5

// Misc
    #define LED_1               LATEbits.LATE2
    #define LED_1_TRIS          TRISEbits.TRISE2
    #define LED_1_ANS           ANSELEbits.ANSE2

    #define TMRL                TMR0L

    // Following definition is for delay functionality
    #if defined(__18CXX)
        #define GetInstructionClock()   (CLOCK_FREQ/4)
    #endif

    void BoardInit(void);

#endif
```

```c
/*
 HardwareProfile.c
 */

#include "SystemProfile.h"
#include "Compiler.h"
#include "WirelessProtocols/Console.h"
#include "TimeDelay.h"
#include "HardwareProfile.h"
#include "WirelessProtocols/SymbolTime.h"

void BoardInit(void)
{
    #if defined(SENSOR_NODE)
        WDTCONbits.SWDTEN = 0; //disable WDT

// Transceiver configuration
        PHY_CS = 1;
        PHY_CS_TRIS = 0;
        PHY_CS_ANS = 0;

        PHY_RESETn = 1;
        PHY_RESETn_TRIS = 0;
        PHY_WAKE = 0;
        PHY_WAKE_TRIS = 0;

        RF_INT_TRIS = 1;
        RF_INT_ANS = 0;

        RFIF = 0;               //clear the interrupt flag

// SPI configuration
        SPI_SDI = 0;
        SDI_TRIS = 1;
        SDI_ANS = 0;

        SPI_SDO = 0;
        SDO_TRIS = 0;
        SDO_ANS = 0;

        SPI_SCK = 0;
        SCK_TRIS = 0;
        SCK_ANS = 0;


// ADC - Channels
        LGT_TRIS = 1;
        LGT_ANS = 1;

        TMP_TRIS = 1;
        TMP_ANS = 1;

        BAT_TRIS = 1;
        BAT_ANS = 1;

// LED configuration
        LED_1 = 1;
        LED_1_TRIS = 1;
        LED_1_ANS = 0;

        CCPTMRS1 = 0x08;
        PR6 = 0x63;
        CCP5CON = 0x0C;
        CCPR5L = 0x00;

        PIR5bits.TMR6IF = 0;
        T6CON = 0x04;

        while(!PIR5bits.TMR6IF);

        LED_1_TRIS = 0;

        // The MRF24J40 is using INT0 for interrupts
        INTCON = 0x00; // Is this needed?
        INTCON2 = 0xC0; // Disable PORTB pull-ups and set INT0 interrupt on rising edge
        INTCON3 = 0x00; // Disable INT1, INT2

        // Set the SPI module
        #if defined(HARDWARE_SPI)
            SSP1STAT = 0xC0; // Mode 0,0
            SSP1CON1 = 0x21; // FOSC/16
        #endif

        // Set the ADC module
        ADCON0 = 0x01;                  // AD Control Register 1: Enable (turn on ADC)
        ADCON1 = 0x00;                  // Set VREF+ to VDD and VREF- to VSS;
        ADCON2 = 0xBD;                  // AD Control Register 2: 20 TAD (accuracy), FOSC_16 (freq/16), right justified

        PMD0 = 0xBF;                    // Power ON USART1 Peripheral only
        PMD1 = 0xBF;                    // Power ON MSSP1 Peripheral only
        PMD2 = 0xFE;                    // Power ON ADC Peripheral only

        RCONbits.IPEN = 1;              // Enable interrupt priority
        INTCONbits.GIEH = 1;            // Enable high priority interrupts
    #else
        #error "Unknown demo board.  Please properly initialize the part for the board."
    #endif
}

#if defined(__18CXX)
    void UserInterruptHandler(void) {
        if(INTCONbits.INT0IE && INTCONbits.INT0IF ) {
            INTCONbits.INT0IF = 0;
        }
    }
#endif
```

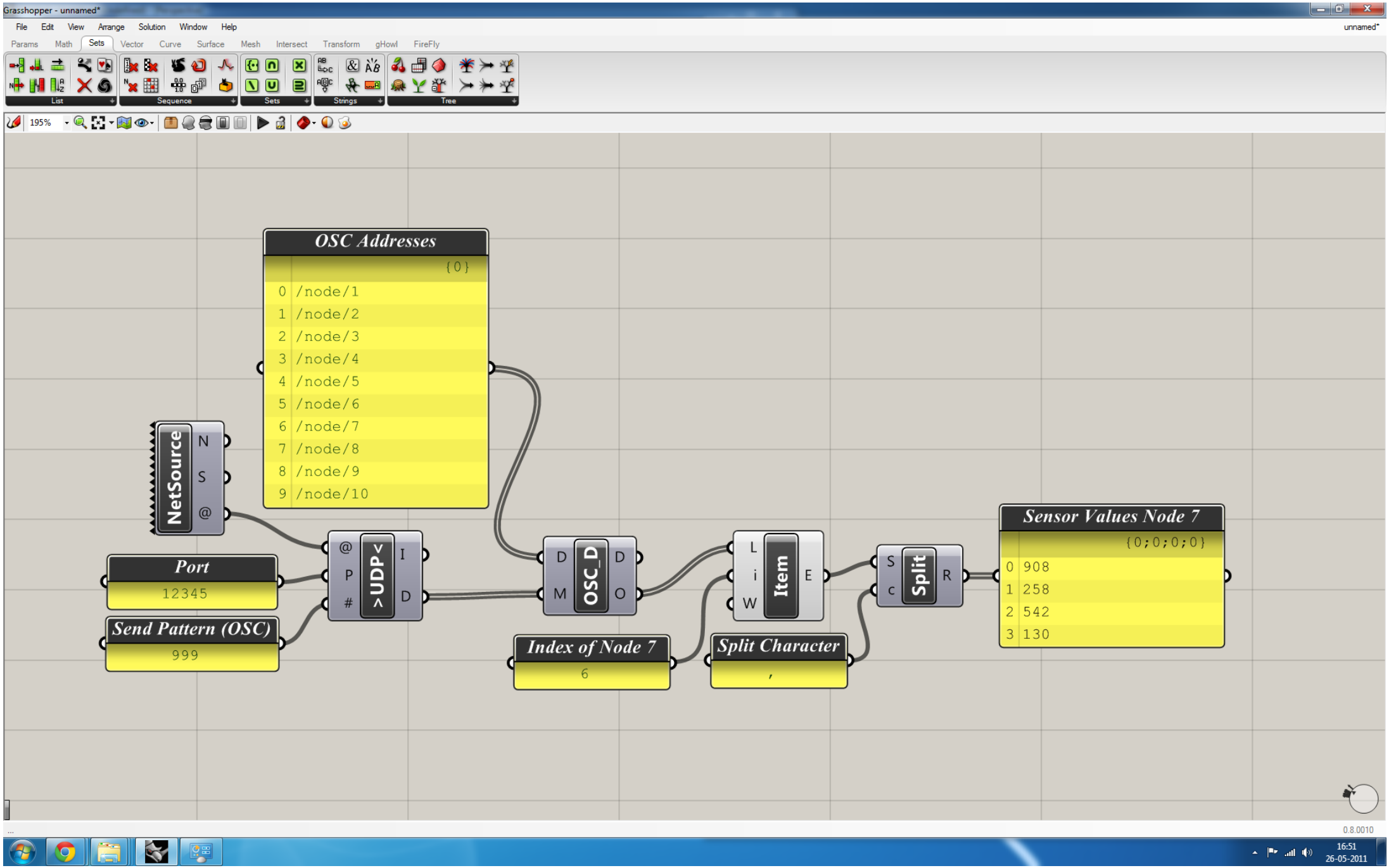
```c
/*
 main.c - main loop for the Node
 */

/************************ HEADERS ****************************************/
#include "WirelessProtocols/Console.h"
#include "ConfigApp.h"
#include "HardwareProfile.h"
#include "WirelessProtocols/MCHP_API.h"
#include "pwm.h"
#include "delays.h"

/************************ VARIABLES ********************************/
#define BASE    0x01
#define NODE    0x02

#if ADDITIONAL_NODE_ID_SIZE > 0
    BYTE AdditionalNodeID[ADDITIONAL_NODE_ID_SIZE] = {NODE};
#endif

BYTE myChannel = 0xFF;
BYTE TxErrors = 0;
BOOL Sleeping = FALSE;

void getAD(BYTE channel, WORD * buffer)
{
    ADCON0 &= 0b10000011;               // Clear CHS4:CHS0
    ADCON0 |= channel << 2;             // Set channel

    ADCON0bits.GO = 1;

    while (ADCON0bits.NOT_DONE);        // Wait for conversion

    *buffer = ((WORD)ADRESH << 8) & 0xFF00;
    *buffer |= ADRESL;
}

void main(void)
{
    BYTE i, j;
    WORD val[9][3];
    BYTE chan[3];

    static BYTE duty = 0;
    static BYTE sign = 1;
    static enum _State
    {
        SLEEP = 0,
        BLINK,
    } State = SLEEP;

    BoardInit();
    ConsoleInit();

    chan[0] = 9;        // ADC Channel for LIGHT sensor
    chan[1] = 10;       // ADC Channel for TEMP sensor
    chan[2] = 6;        // ADC Channel for BATTERY sensor

    #if defined(PROTOCOL_P2P)
        Printf("\r\n\r\nStarting MiWi(TM) P2P Stack ...");
    #endif
    #if defined(PROTOCOL_MIWI)
        Printf("\r\n\r\nStarting MiWi(TM) Stack ...");
    #endif

    #if defined(MRF24J40)
        Printf("\r\n     RF Transceiver: MRF24J40\r\n");
    #elif defined(MRF49XA)
        Printf("\r\n     RF Transceiver: MRF49XA\r\n");
    #elif defined(MRF89XA)
        Printf("\r\n     RF Transceiver: MRF89XA\r\n");
    #endif

    LED_1 = 1; // Turn on LED 1 to indicate on

    /*******************************************************************/
    // Initialize Microchip proprietary protocol. Which protocol to use
    // depends on the configuration in ConfigApp.h
    /*******************************************************************/
    /*******************************************************************/
    // Function MiApp_ProtocolInit initialize the protocol stack. The
    // only input parameter indicates if previous network configuration
    // should be restored. In this simple example, we assume that the
    // network starts from scratch.
    /*******************************************************************/
    MiApp_ProtocolInit(FALSE);

    #ifdef ENABLE_ACTIVE_SCAN

        ConsolePutROMString((ROM char*)"\r\nStarting Active Scan...");

        while(1)
        {
            j = MiApp_SearchConnection(10, 0xFFFFFFFF);

            if( j > 0 )
            {
                // now print out the scan result.
                Printf("\r\nActive Scan Results: \r\n");
                for(i = 0; i < j; i++)
                {
                    Printf("Channel: ");
                    PrintDec(ActiveScanResults[i].Channel);
                    Printf("   RSSI: ");
                    PrintChar(ActiveScanResults[i].RSSIValue);
                    #if defined(IEEE_802_15_4)
                        #if ADDITIONAL_NODE_ID_SIZE > 0
```

```
                        Printf("   PeerInfo: ");
                        for(j = 0; j < ADDITIONAL_NODE_ID_SIZE; j++)
                        {
                            PrintChar(ActiveScanResults[i].PeerInfo[j]);
                        }
                    #endif
                    Printf("    PANID: ");
                    PrintChar(ActiveScanResults[i].PANID.v[1]);
                    PrintChar(ActiveScanResults[i].PANID.v[0]);
                    Printf("\r\n");
                #endif
                myChannel = ActiveScanResults[i].Channel;
            }
        }

        if( myChannel != 0xFF )
        {
            MiApp_SetChannel(myChannel);
            break;
        }

        Printf("\r\nNo Suitable PAN, Rescanning...");
    }
#else
    MiApp_SetChannel(myChannel);
#endif

MiApp_ConnectionMode(DISABLE_ALL_CONN);

#ifdef ENABLE_HAND_SHAKE
    i = MiApp_EstablishConnection(0, CONN_MODE_DIRECT);
#endif

while(1)
{
    if( MiApp_MessageAvailable() )
    {
        if(((BYTE)rxMessage.Payload[0] & 0x30) == 0x30)
        {
            BYTE state = ((BYTE)rxMessage.Payload[0] & 0x01);
            if(state == 1)
            {
                State = BLINK;
            }
            else
            {
                State = SLEEP;
            }
        }
        MiApp_DiscardMessage();
    }
    else
    {
        switch(State)
        {
            case SLEEP:
                #ifdef ENABLE_FREQUENCY_AGILITY

                    if(TxErrors > 3)
                    {
                        TxErrors = 0;
                        MiApp_TransceiverPowerState(POWER_STATE_WAKEUP);
                        Sleeping = FALSE;
                        MiApp_ResyncConnection(0, 0xFFFFFFFF);
                    }
                    else
                    {
                        Sleeping = TRUE;
                    }

                #endif
                #ifdef ENABLE_SLEEP
                    if(Sleeping == TRUE)
                    {
                        PMD0 |= 0x20;                   // Power OFF Timer6
                        PMD1 |= 0x10;                   // Power OFF CCP5

                        MiApp_TransceiverPowerState(POWER_STATE_SLEEP);
                        while(ConsoleIsPutReady() == 0);

                        ClrWdt();

                        // Turn off LED 1 to indicate sleep
                        LED_1 = 0;

                        WDTCONbits.SWDTEN = 1;      // enable watch dog timer

                        Sleep();

                        WDTCONbits.SWDTEN = 0;      // disable watch dog timer

                        // Turn on LED 1 to indicate wake-up
                        LED_1 = 1;

                        for(i = 0; i < 3; i++)
                        {
                            for(j = 0; j < 8; j++)
                            {
                                getAD(chan[i], &val[j][i]);
                            }
                        }

                        for(i = 0; i < 3; i++)
                        {
                            val[8][i] = 0;
                            for(j = 0; j < 8; j++)
                            {
```

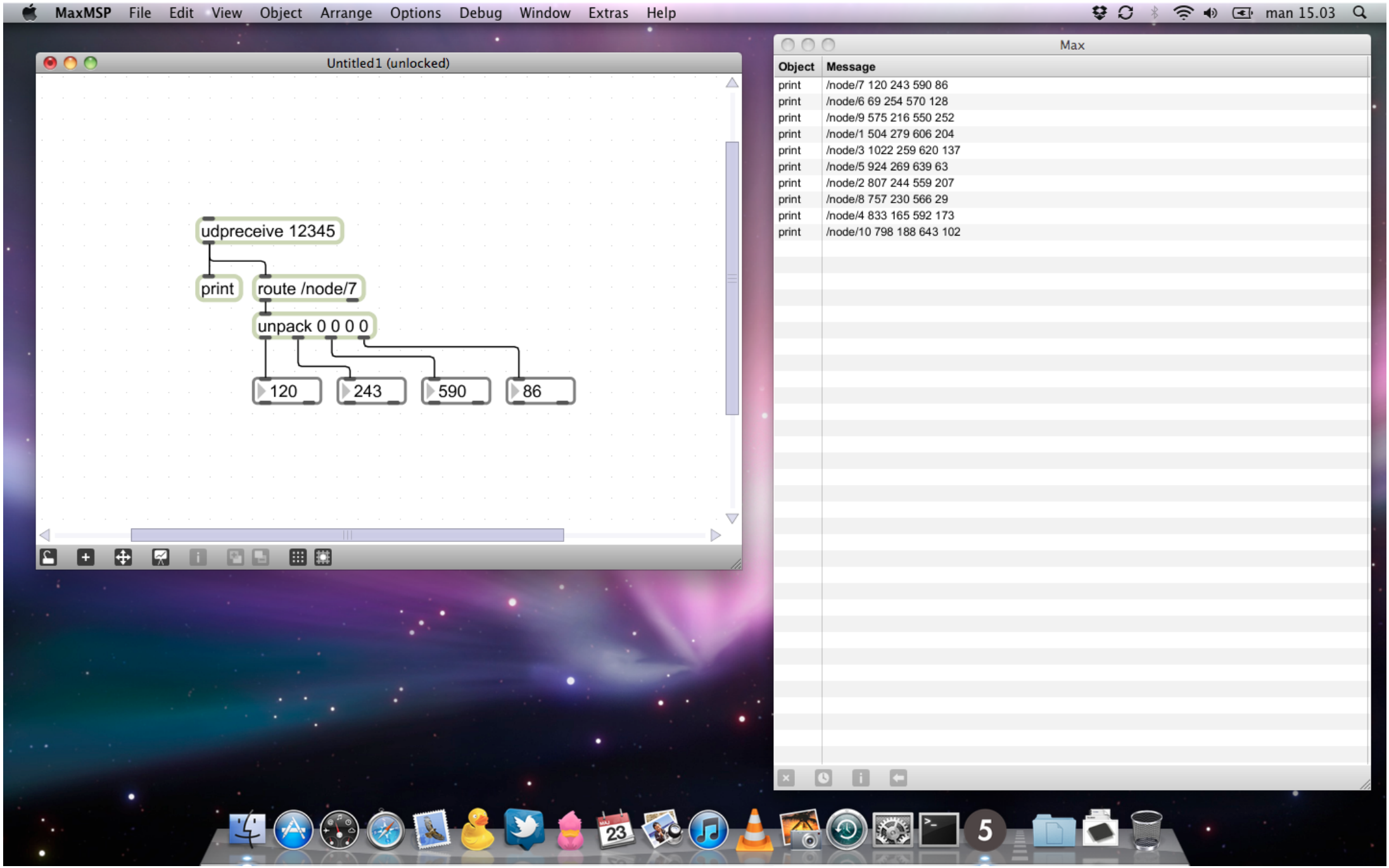
```
                            val[8][i] += val[j][i];
                        }
                        val[8][i] >>= 3;
                    }

                    MiApp_FlushTx();

                    for(i = 0; i < 3; i++)
                    {
                         MiApp_WriteData((BYTE)(val[8][i] >> 8));
                         MiApp_WriteData((BYTE)val[8][i]);
                    }

                    if( MiApp_UnicastConnection(0, FALSE) == FALSE )
                    {
                        Printf("\r\nUnicast Failed\r\n");
                    }

                    if(MiApp_TransceiverPowerState(POWER_STATE_WAKEUP_DR) > SUCCESS)
                    {
                        TxErrors++;
                    }
                    else
                    {
                        TxErrors = 0;
                    }
                }
            #endif
            break;
        case BLINK:

            MiApp_TransceiverPowerState(POWER_STATE_SLEEP);
            while(ConsoleIsPutReady() == 0);

            PMD0 &= 0xDF;                   // Power ON Timer6
            PMD1 &= 0xEF;                   // Power ON CCP5

            LED_1_TRIS = 1;

            CCP5CON = 0x0C;
            T6CON = 0x04;

            while(!PIR5bits.TMR6IF);

            LED_1_TRIS = 0;

            for(i = 0; i < 0xFF; i++)
            {
                Delay10KTCYx(4);
                SetDCPWM5(duty += sign);
            }
            Delay10KTCYx(0);
            Delay10KTCYx(0);

            sign *= -1;
            // Turn on LED 1 to indicate wake-up

            for(i = 0; i < 3; i++)
            {
                for(j = 0; j < 8; j++)
                {
                    getAD(chan[i], &val[j][i]);
                }
            }

            for(i = 0; i < 3; i++)
            {
                val[8][i] = 0;
                for(j = 0; j < 8; j++)
                {
                    val[8][i] += val[j][i];
                }
                val[8][i] >>= 3;
            }

            MiApp_FlushTx();

            for(i = 0; i < 3; i++)
            {
                 MiApp_WriteData((BYTE)(val[8][i] >> 8));
                 MiApp_WriteData((BYTE)val[8][i]);
            }

            if( MiApp_UnicastConnection(0, FALSE) == FALSE)
            {
                Printf("\r\nUnicast Failed\r\n");
            }

            if(MiApp_TransceiverPowerState(POWER_STATE_WAKEUP_DR) > SUCCESS)
            {
                TxErrors++;
            }
            else
            {
                TxErrors = 0;
            }
            break;
        }
    }
  }
}
```

E. Visual Programming

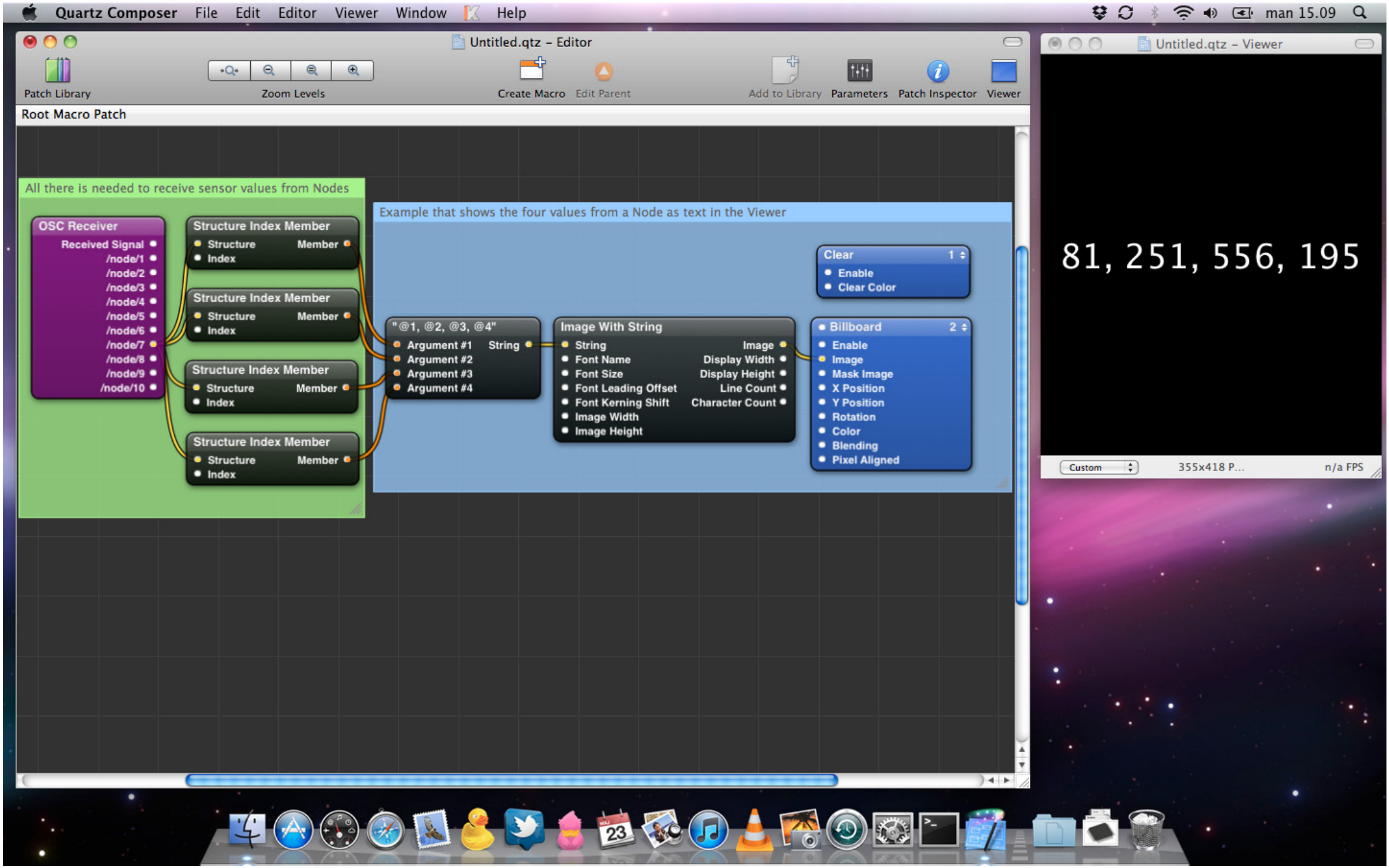
E.1 Grasshopper


Grasshopper - unnamed*
File Edit View Arrange Solution Window Help
Params Math Sets Vector Curve Surface Mesh Intersect Transform gHowl FireFly
List Sequence Sets Strings Tree
195%
OSC Addresses
{0}
0 /node/1
1 /node/2
2 /node/3
3 /node/4
4 /node/5
5 /node/6
6 /node/7
7 /node/8
8 /node/9
9 /node/10
NetSource
N
S
@
Port
12345
Send Pattern (OSC)
999
@
P
#
UDP
I
D
D
M
OSC_D
D
O
Index of Node 7
6
L
i
W
Item
E
Split Character
,
S
c
Split
R
Sensor Values Node 7
{0;0;0;0}
0 908
1 258
2 542
3 130
0.8.0010
16:51
26-05-2011

E.2 Max/MSP


MaxMSP File Edit View Object Arrange Options Debug Window Extras Help
man 15.03
Untitled1 (unlocked)
udpreceive 12345
print
route /node/7
unpack 0 0 0 0
120
243
590
86
Max
Object Message
print /node/7 120 243 590 86
print /node/6 69 254 570 128
print /node/9 575 216 550 252
print /node/1 504 279 606 204
print /node/3 1022 259 620 137
print /node/5 924 269 639 63
print /node/2 807 244 559 207
print /node/8 757 230 566 29
print /node/4 833 165 592 173
print /node/10 798 188 643 102

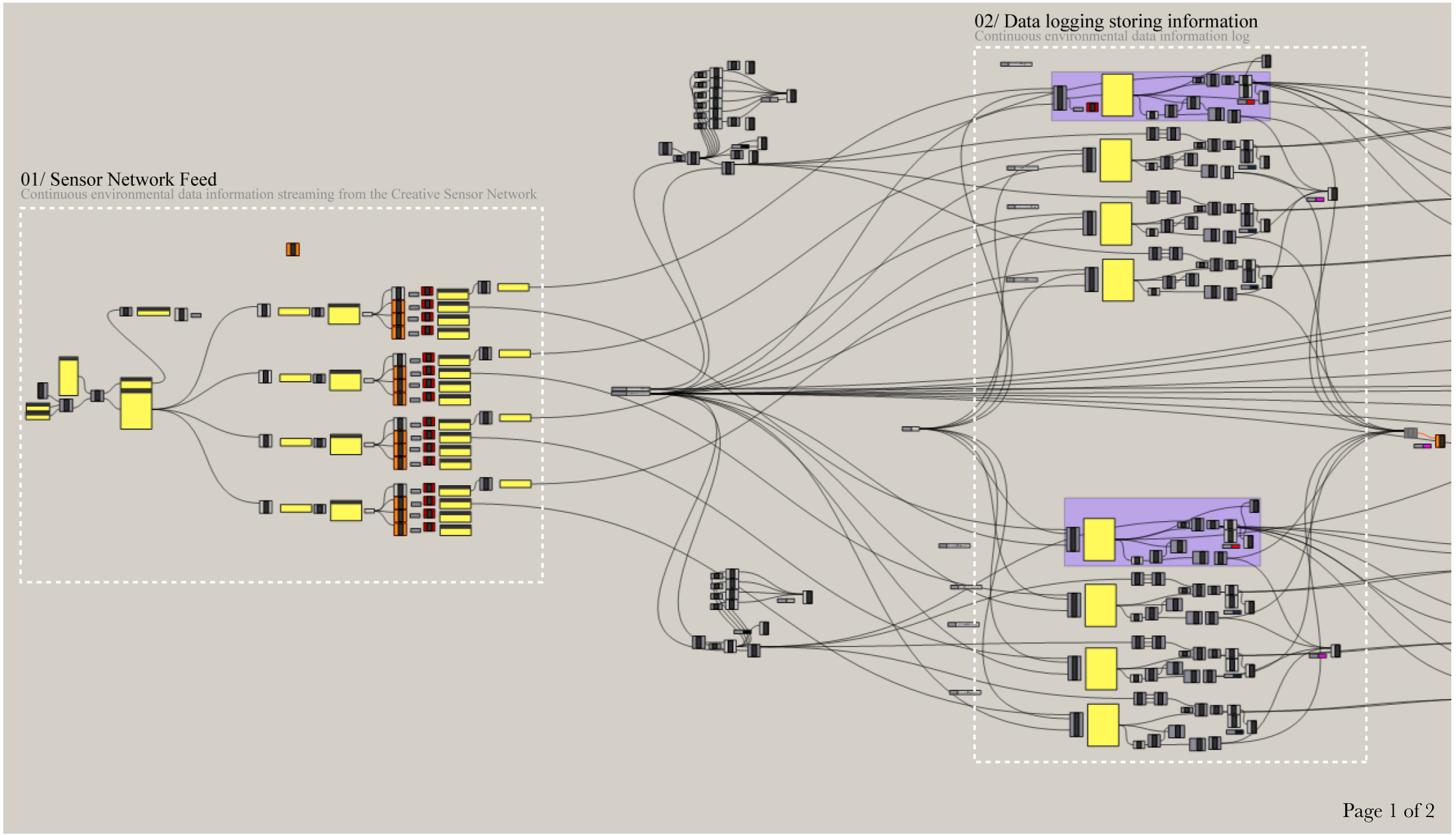
E.3 Quartz Composer

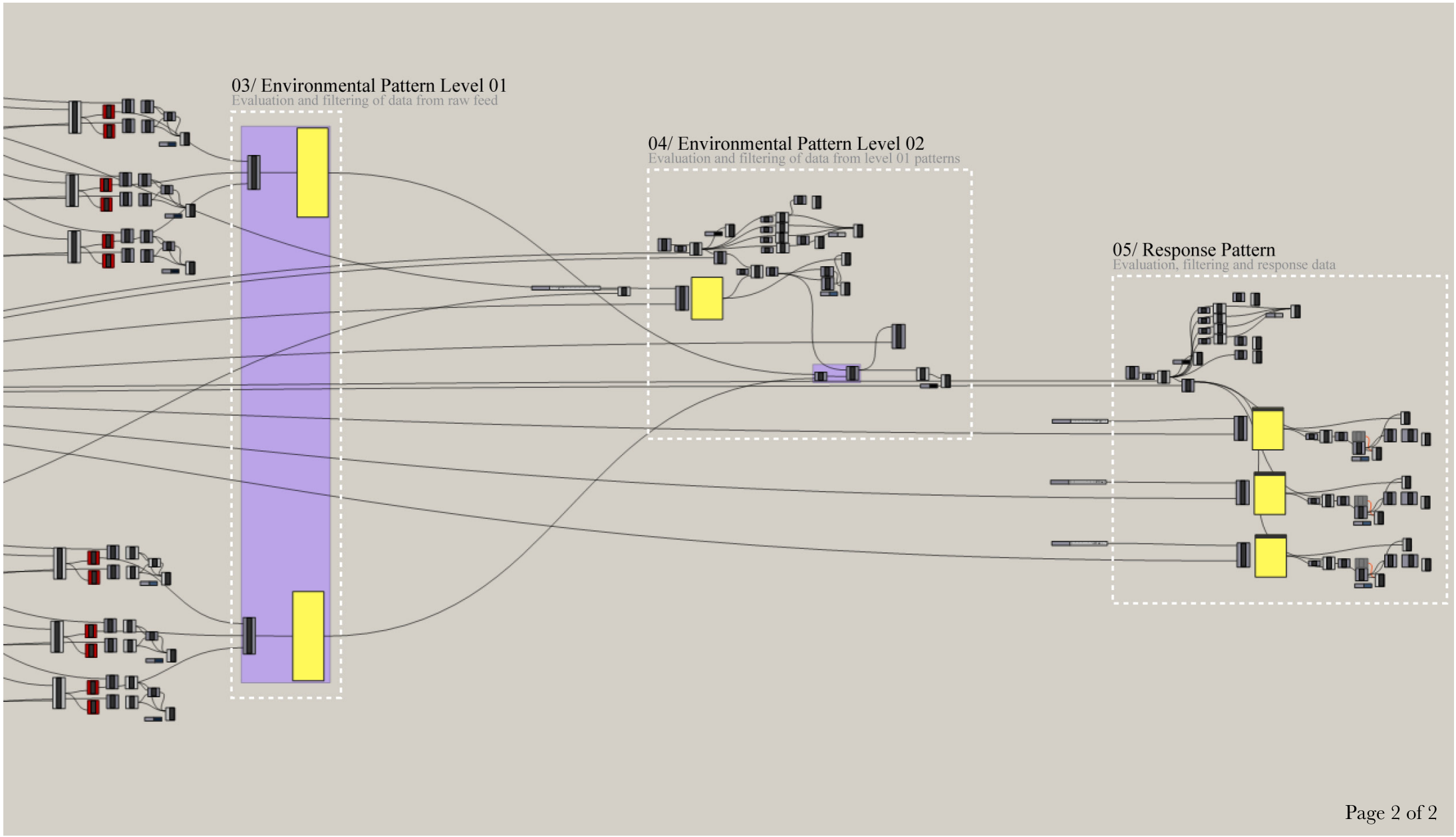

Quartz Composer File Edit Editor Viewer Window Help man 15.09
Untitled.qtz – Editor
Patch Library
Zoom Levels
Create Macro
Edit Parent
Add to Library
Parameters
Patch Inspector
Viewer
Root Macro Patch
All there is needed to receive sensor values from Nodes
OSC Receiver
Received Signal
/node/1
/node/2
/node/3
/node/4
/node/5
/node/6
/node/7
/node/8
/node/9
/node/10
Structure Index Member
Structure
Index
Member
Example that shows the four values from a Node as text in the Viewer
"@1, @2, @3, @4"
Argument #1
Argument #2
Argument #3
Argument #4
String
Image With String
String
Font Name
Font Size
Font Leading Offset
Font Kerning Shift
Image Width
Image Height
Image
Display Width
Display Height
Line Count
Character Count
Clear
Enable
Clear Color
Billboard
Enable
Image
Mask Image
X Position
Y Position
Rotation
Color
Blending
Pixel Aligned
Untitled.qtz – Viewer
81, 251, 556, 195
Custom
355x418 P...
n/a FPS

F. openFrameworks - Example

```
/*
 testApp.h
 */
#pragma once

#include "ofMain.h"
#include "ofxOsc.h"

class testApp : public ofBaseApp{
    public:

        void setup();
        void update();
        void draw();

        void keyPressed(int key);
        void keyReleased(int key);
        void mouseMoved(int x, int y );
        void mouseDragged(int x, int y, int button);
        void mousePressed(int x, int y, int button);
        void mouseReleased(int x, int y, int button);
        void windowResized(int w, int h);

        ofxOscReceiver receiver;
        int s[4];
};

/*
 testApp.c
 */
#include "testApp.h"

//--------------------------------------------------------------
void testApp::setup(){
    receiver.setup( 12345 );
}

//--------------------------------------------------------------
void testApp::update(){
    while( receiver.hasWaitingMessages() ) {

        ofxOscMessage m;
        receiver.getNextMessage( &m );

        cout << m.getAddress() << endl;

        if ( m.getAddress() == "/node/7" )
        {
            for (int i = 0; i < m.getNumArgs(); i++) {
                s[i] = m.getArgAsInt32(i);
            }
        }
    }
}

//--------------------------------------------------------------
void testApp::draw(){
}

//--------------------------------------------------------------
void testApp::keyPressed  (int key){

}

//--------------------------------------------------------------
void testApp::keyReleased(int key){

}

//--------------------------------------------------------------
void testApp::mouseMoved(int x, int y ){

}

//--------------------------------------------------------------
void testApp::mouseDragged(int x, int y, int button){

}

//--------------------------------------------------------------
void testApp::mousePressed(int x, int y, int button){
}

//--------------------------------------------------------------
void testApp::mouseReleased(int x, int y, int button){

}

//--------------------------------------------------------------
void testApp::windowResized(int w, int h){

}
```

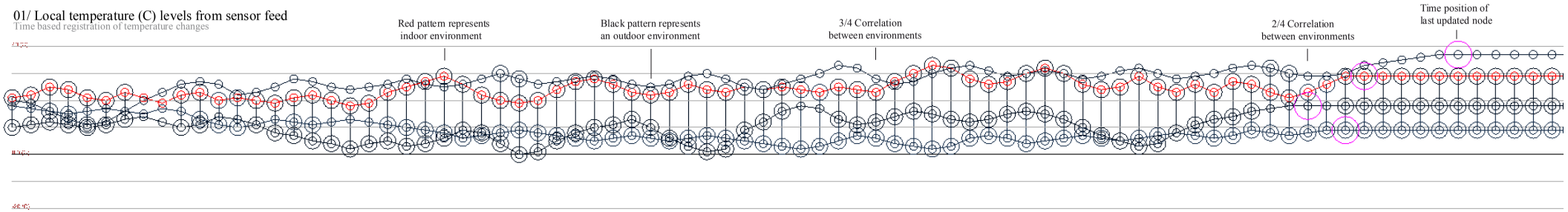
G. Case Using Grasshopper


02/ Data logging storing information
Continuous environmental data information log
01/ Sensor Network Feed
Continuous environmental data information streaming from the Creative Sensor Network


03/ Environmental Pattern Level 01
Evaluation and filtering of data from raw feed
04/ Environmental Pattern Level 02
Evaluation and filtering of data from level 01 patterns
05/ Response Pattern
Evaluation, filtering and response data

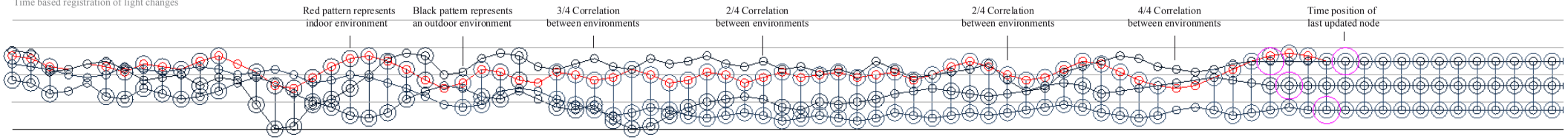
01/ Local temperature (C) levels from sensor feed

Time based registration of temperature changes



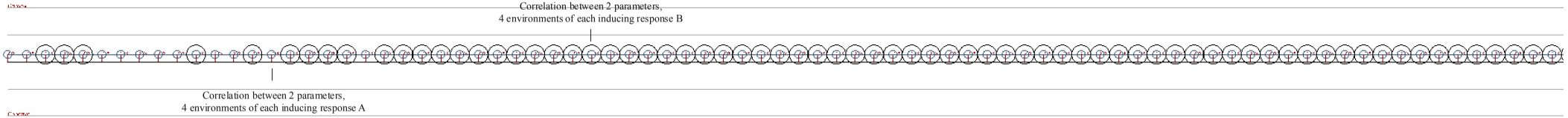
02/ Local light (lx) levels from sensor feed

Time based registration of light changes



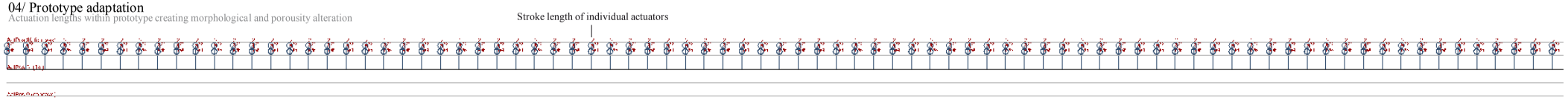
03/ Correlations between temperature and light patterns

Multiple condition analysis, comparison and decision creating second level pattern



04/ Prototype adaptation

Actuation lengths within prototype creating morphological and porousity alteration



H. CD