Programming Embedded 1-Wire Devices as Custom Remote Sensing tools on Ethernet Networks

Prepared by Doug Warden
Spring 2006…to Jan 2007.

Mint Capstone Project

Backgrounder

When most people think of networks, they typically think of Ethernet networks; such as 10 or 100baseT connecting PC's together, since that is what they used on a daily basis. As a stretch they may recognize that Ethernet sections may be connected to each other via different types of physical networking WAN infrastructures, such as FDDI or ATM. The reality is that a network is the connection of two or more computer systems linked together in some way. This somewhat vague definition includes many types of networks and possibilities. A node on a network may be, and often is, a standard PC, but may also be something else entirely, such as a Bluetooth enabled device, a router, some sort of embedded device, or a wireless handheld PDA. For some time I have been interested in learning how to incorporate the collection of data from traditionally "non-collectable" and non-networked sources, like thermostats, lights, and appliances to gather that information and carry out some form of useful work, and bring them into the realm of the more familiar LAN.

This project will remotely gather data from sensing devices, transmit it via a 1-wire network (described later) to an embedded device which will collect this data and transmit it via standard Ethernet to a server. The application of this technology could be adapted to a wide variety of purposes, such as remote sensing of conditions in manufacturing, oil and gas production environments, or security systems. Once the data is on an Ethernet network, the technology is well established to save, store, share, and act on that data.

In this case, I was visiting a cousin on Pender Island and he asked if I could help him network the buildings on his property together. As this is seemed like a fairly straightforward implementation, I agreed to help him. I did not foresee any great difficulties since what he wanted was essentially a very large home network, with few nodes but some large distances between them. As we discussed further I learned he also wanted an external webcam and weather station which would stream data continuously to a web server, and I also agreed to this, thinking there would likely be some sort of simple off the shelf solution which I could simply plug into the network and leave. There was an off the shelf answer for the cameras, but surprisingly (to me, anyways) this turned out to not be the case for the weather station. There were many weather sensing products that could be plugged into a PC via some sort of cable, like a USB, or which came with some sort of LCD digital display which connected to the weather station using cabling or wireless connection, but I was unable to find a product which took the data, and allowed me to take the data to a centralized location, where it could be "served" via the web to any clients requiring the information. Following a number of years in industry; it was easy to think of practical, lucrative applications that would involve the remote sensing of data, and capturing it in a common and usable form in a web page. For this project I used weather as the remote sensing data, but it could be anything that you could get sensors for, humidity in a paper production environment, pressure in a pipeline, or power consumption in a deserted downtown office building.

The next step in the development would be to then reverse the flow of information, and act on the sensed device, depending on the information that was gathered.

Table of Contents

(Use Ctrl+Click to jump to that section in soft copy)

Tables and Figures

Introduction

I spent several years working for a paper packaging manufacturer in an IT capacity. It would have been extremely useful to us to be able to automatically monitor the status of things like inventory, power consumption, humidity and temperature on the factory floor and have been able to converge it to a single location to be able to do some useful work with that data. I have long wanted the opportunity to learn how to do this, when this came along, I didn't realize what it was at first - it began more as a favor to a relative, but soon turned into the challenge of collecting and sharing analog data collected on non-standard network in a digital format easily accessible through standard means.

This report will start with a discussion into the installation, configuration and considerations of the standard network installation, including equipment and challenges experienced. The report will then turn to a discussion of 1-Wire technologies, networking and communication. It will conclude with an analysis of the software framework that I used to query and save the data from the 1-Wire sensors.

Network Installation and Equipment Selection

Equipment for this project needed to meet a number of criteria. I was looking for something that was reasonably inexpensive, but ultimately flexible. I required equipment that would be more than just an off the shelf solution to the problem but at the same time also needed to be a non-obscure (supported) solution. I needed flexibility for the equipment in the project, and in terms of my learning requirements, I was looking for something which was scalable and industry supported. I was hoping to find solutions which would have a wide application. Not only did I need to build a working weather station, I needed to build something which was robust – both physically and technically. It became clear to me that much of the equipment which was marketed as tough and easy to use had never been installed by the people who sold it. The weather station itself needed to be run out over a dock to an exposed pole. Cabling, and all gear needed to be completely weather proof as well as capable of withstanding incursions of salt water.

Weather Station

After searching around I settled on the following equipment for the weather station; A TAI8515 1-Wire Weather Instrument Kit V3.0 from aagelectronica[1] see Figure 1.

Figure 1 - disassembled TA18515-1

This weather station will measure wind speed, direction, and temperature. I also purchased two add-on products which measure barometric pressure and rainfall. The selection of this product was influenced by (and in turn influenced) the use of 1-wire networking as a means of gathering and transmitting raw data. As data is collected it is transmitted via a 1-wire network to the embedded networking device. The weather station can be seen disassembled in Figure 2.

Figure 2 - Internals of TAI 18515

Showing the top and bottom sides of the inside of the TA18515. This is a fairly simple device, consisting of a very simple circuit board holding the 1-Wire devices, which are actuated by reed switches. There are two RJ-11 connectors on the one side, and all of the circuitry on the board connects the sensing devices in a daisy chain fashion (as explained later in more detail in the 1-Wire networking section). This can be seen more clearly in Figure 3. In addition to the sensors contained in this device, I also added a rain gauge and barometric pressure sensor, which were added into the "daisy chain" using standard cat-5 cabling with RJ-11 connectors, external to the TA18515.

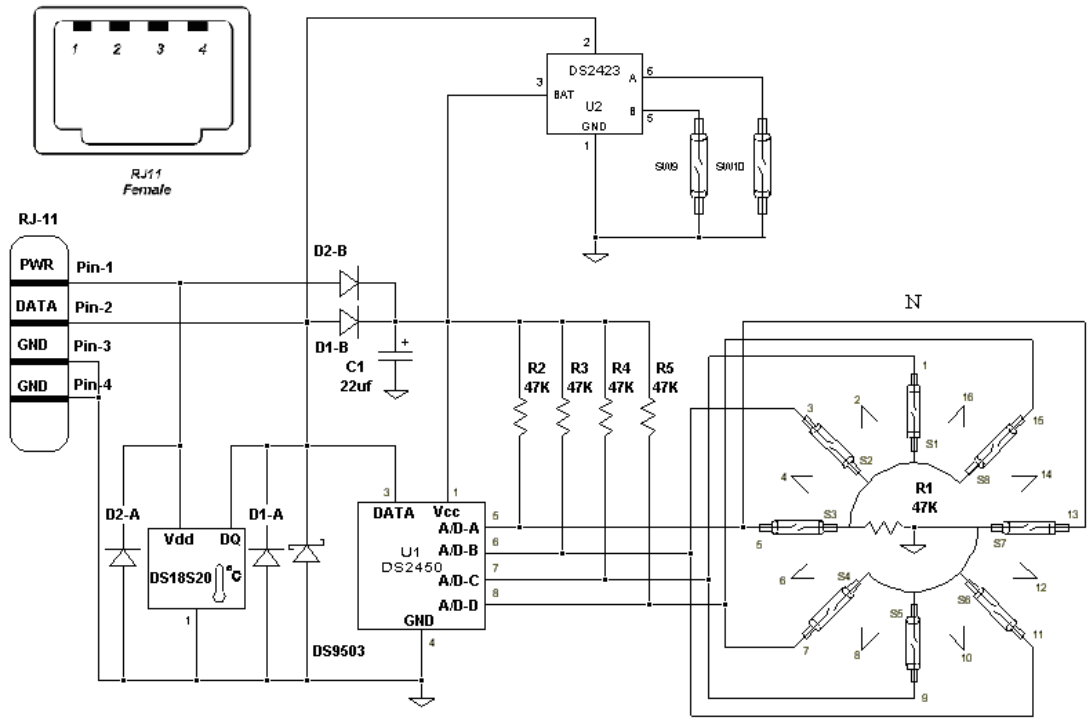

Figure 3 - TA18515 Schematic

The above schematic contains three one-wire sensors, the DS18S20 is a temperature sensor (embedded in the circuit board), the DS2450 is a 4 way analog to digital converter for measuring wind direction and uses read switches on the underside of the circuit board, while the DS2423 is for calculating wind speed (again through reed switch), based on the rotational speed on the top of the circuit board.

Data Acquisition

The data acquisition from the TA18515 Weather station can be done in a number of ways. Connection can be done using a direct connection to a PC using a standard RS-232 9 pin serial interface (with pre-built software), a USB to PC connector, or a strictly 1-wire interface. Using 1-Wire requires the use of TINI, which stands for **T**iny **I**nter**N**et **I**nterfaces and is a produced by Dallas Semiconductors. According to the Dallas Semiconductors website, the TINI is a "microcontroller-based development platform that executes code for embedded web servers. The platform is a combination of broad-based I/O, a full TCP/IP stack, and an extensible Java runtime environment that simplifies development of network-connected equipment."[2] Dallas Semiconductor manufactures their own microcontrollers to host TINI which are high speed 8 bit microcontrollers and come with various configurations, including built in interfaces to both Ethernet LAN and 1-Wire networking. The implementation of TINI which I used was the T-stick,

constructed by Systronix, which uses the Dalsemi microcontroller (DS80C400) on their own hardware - a 72 pin SIMM stick. Interfacing this device with a TILT board (also by Systronix) allowed me to provide to install and run the TINI in the form of the T-Stick module while the TILT board provided a 1-wire, Ethernet and serial interfaces to the TINI device. Development on the TINI platform is designed to run JAVA and JAVA web servers, but can also be done in C through the use of the Kiel compiler. My original position in developing this project is that it would be easier and more network efficient to query, and save data on the TINI and then transmit this data to the Linux server via a socket program rather than hosting and trying to do all the development on the TINI server itself. Dallas Semiconductors manufactures their own version of the same controller, in a networked board, the DSTINIm400, which is interchangeable with the Systronix equipment (I have now used both, and they seem to have the same functionality and have had no problem with either of them), but I purchased the T-Stick on a recommendation, and this recommendation was seconded by the difficulty of using the Dallas Semiconductors e-commerce website.

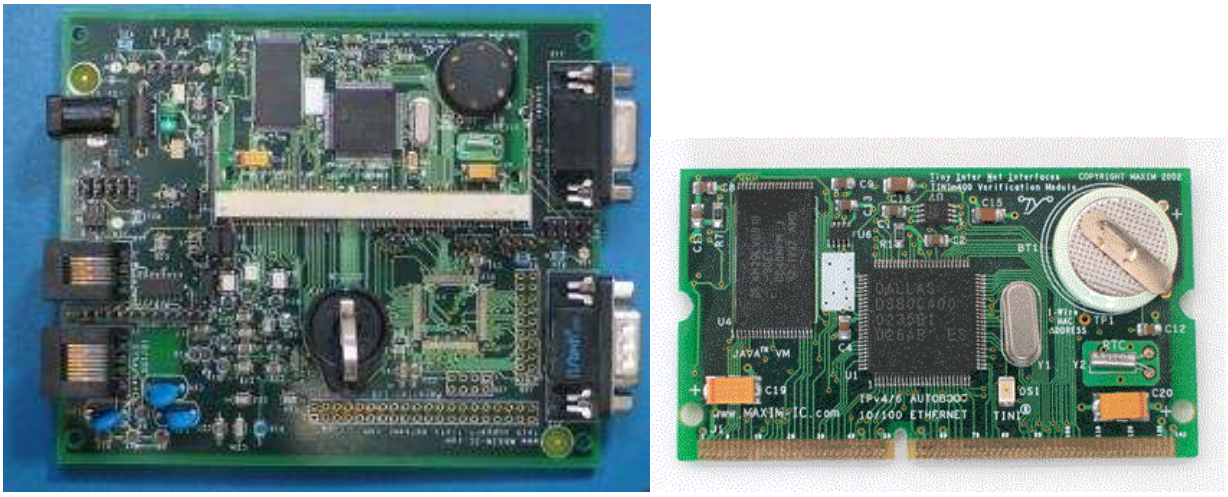

Figure 4 - DSTINIm400 and DSTINIs400

TINI Hardware

The TINI device requires an operating system be manually loaded in order to operate and any further programs uploaded. Loading the OS turned out to be somewhat problematic in ways I had not anticipated. The Tilt board, see Figure 5, provides numerous interfaces to allow information to reach the server, including a RJ-45 connector, male and female serial connectors, and a 4 pin RJ-11, which is a 1-wire connector.

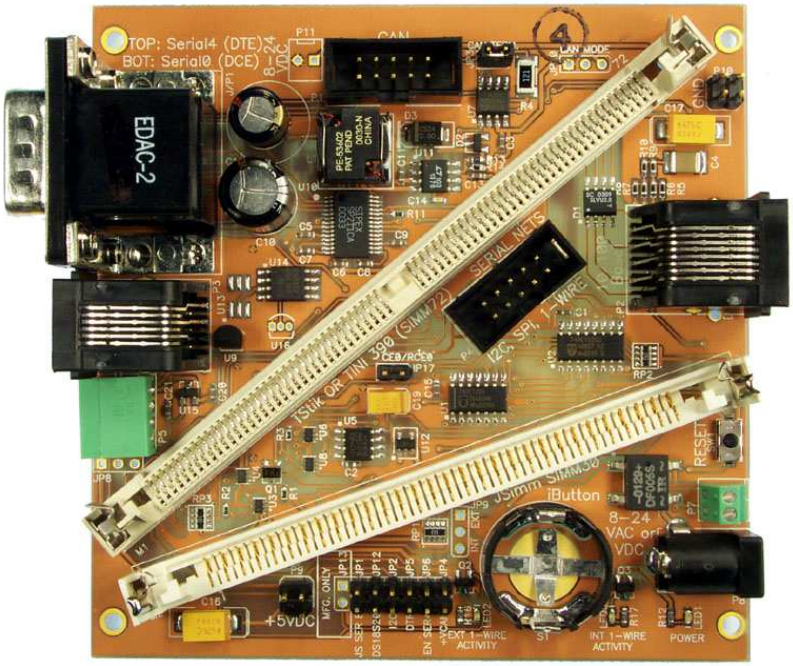

Figure 5 - TILT board – 10 cm2

The actual DS80C400 processor is placed by Systronix on a "T-Stik", which is a good, old fashioned, 72 pin SIMM, and can be seen in Figure 6.

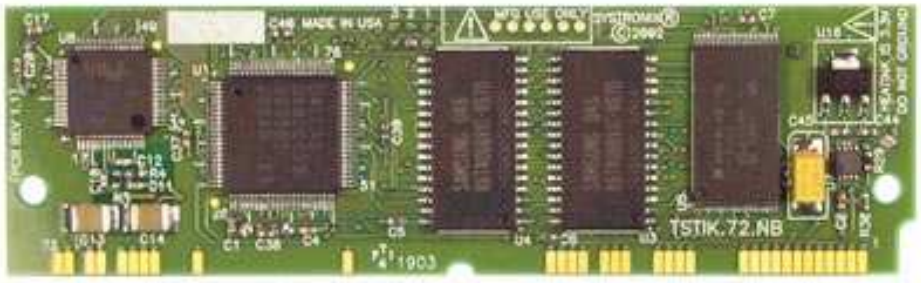

Figure 6 - T-Stik with DS80C400 chip – standard 72 pin DIMM

TINI O/S Installation and configuration

The standard method for loading an OS involves connecting to the serial connector and uploading the O/S[3]. Unfortunately, most laptops no longer have serial interfaces, and to circumvent this problem, I was required to get a USB to serial adapter and load the appropriate drivers for this. This lead to other problems as the standard way of loading the O/S is to download the desired O/S from Dallas Semiconductors and upload the firmware using the Javakit, which really didn't work. I downloaded the latest TINI O/S, specifically the TINI 1.17 version. Installation of this O/S creates a "Slush shell", which is like traditional Unix shells, but with no text editor and a very compressed and reduced command set. The O/S also contains a serial (TTY), FTP and Telnet server, to allow for easy uploading of the finished (compiled) programs and execute them in the slush shell/JAVA environment provided by the TINI O/S.

A quick Slush Primer

The initial connection to upload the firmware and O/S also proved to be problematic, as the recommended method of loading, namely using the Javakit, is no longer built or maintained for Windows operating systems. As someone trying to ramp up on Java, I found this a bewildering initial set of configurations that needed to be done to enable the Javakit, only to learn that it was no longer supported for Windows clients. My difficulties were compounded by the fact that the "serial cable" I was using was actually an old 9 pin null modem cable that I had lying around. I was able to find and download an older copy of this application, but setting up this software is very tricky, and I spent considerable time attempting to get this to work. Whether I was stymied by the inability of the Javakit to recognize the serial connection or because of some configuration error I never determined. I eventually tried an alternative to the Javakit in the form of the Microcontroller Tool Kit (MTK) which I downloaded from Dallas Semiconductors and was able to install and upgrade and install the Firmware and O/S with little trouble by following the Getting Started with TINI guide from Dallas Semi Conductor[3].

Once loaded the TINI O/S is fairly easy to use – there are three built in servers in the Operating System; FTP, telnet, and a Serial Server. The serial and telnet servers allow for access to the device, and the FTP server allows for uploading files. The slush environment itself is a very stripped down shell for the embedded environment. The default install leaves a blank TINI with a single /etc folder, containing a .startup file, entries in this file will execute automatically on startup, and a passwd file, like in a Unix environment. There is no built in editor in this environment, so text files needed to be uploaded, which was a fairly small problem, given the built in FTP and telnet servers. Reasonable help functionality provides guidance on the following commands;

Available Slush Commands:

addc	append	arp	cat
cd	chmod	chown	copy
cp	date	del	df
dir	downserver	echo	ftp
gc	genlog	help	history
hostname	ipconfig	java	kill
ls	md	mkdir	move
mv	netstat	nslookup	passwd
ping	ps	pwd	rd
reboot	rm	rmdir	sendmail
setenv	source	startserver	stats
stopserver	su	touch	useradd
userdel	wall	wd	who
whoami			

This is designed to be instantly familiar to anyone who has done any work in any sort of a Unix environment, and as such, is quite easily picked up.

Java Development in Embedded Environments – A cautionary tale

In retrospect, choosing to develop in Java for this project was not the greatest decision. I do not have a strong programming background, am not terribly familiar with object oriented programming, and have almost no experience with Java. I was enticed, however, by the hype surrounding Java, and the wealth of resources for doing development. Everything I read and people I spoke to indicated that embedded development is the forte of Java. Having traveled far down this road, I am not keen to go back and start over, but the early decision to ignore C as a development vehicle with the Kiel compiler may have been a mistake. In that I haven't done any work with the Kiel compiler I don't know if this is true or not.

The TINI platform is designed to run Java, comes with a built in Java Virtual Machine (JVM), according to my understanding, the JVM should allow any Java compiled application to be portable. It should run in any JVM, independent of the hardware. The DS80C400 is an 8 bit microcontroller, and does run Java compiled programs, but only after extensive modification. I found development difficult and time consuming, I could do the development of the application in an IDE (I used Netbeans initially) which was syntactically useful, but a program which compiles and runs in the IDE will not necessarily work on the TINI. So once changes were made, the project needed to be manually built, converted to a *.TINI file, uploaded and tested. Complicating this process was some sort of a short in the 1-Wire cabling on Pender island, I could still connect to and Telnet into the TINI, and upload my applications to run them from Calgary, but could not connect to any of the 1-Wire devices to test the application. I managed to duplicate the work environment by purchasing another DS80C400 (this one directly from Dallas Semiconductors) and a second weather station. I was able to build, test and run my applications, and should now be able to relatively easily modify the program as required to run in the real environment once the 1-Wire cabling problem is fixed, which should happen by April.

I eventually wound up using JCreator by Xinox software to work in Java – which was a free and simpler tool than Netbeans, I combined this with a batch file to automate and test the code. Using the batch file really helped me to get over an obstacle in the development process. I found that with Netbeans, once I finished the code and wanted to test it, I was endlessly looking up paths, build dependencies and 1-Wire container references along with syntax and typing in impossibly long strings at the command line. It took longer to compile and test the code than it had to write it. (as an example, I initially spent 6 hours scratching my head trying to figure out why my code couldn't be run on the TINI, assuming it was some terrible coding mistake, only to realize that the TINIconvert**o**r used in the creation of the executable *.tini file is not the same as the TINIconvert**e**r, which is what I had typed in). Automating this process simplified things immensely and allowed me to make modifications to code and then quickly compile and test them with a single command[4].

TINI with Java does support some excellent possibilities – one being the simple support of a web server – this allows for the possibility of combining 1-Wire device detection and considerable computing power directly with a Web Server. A programmer could then create a web site with forms that would allow for input from clients to modify and update commands or requests manually. Ideally, I see this as a good area for embedded development to go; information needs to be available to clients, and http/web is the simplest method (and widely used) for doing this. Any standard browser could connect to the server and get the data, if modifications needed to be made in the environment, a web form would be a able to configure the 1-Wire network or other devices, according to the commands of the user. One of the limitations of using the TINI as a web server is the limited memory, but through careful design, it should also be possible to use frames to maintain a small website on the TINI, while referencing larger more complex pictures and web pages for the client, appearing to seamlessly come from the TINI server.

Network topology

The physical network design impacts the design of the program on the embedded devices. The physical network is strung between several spread out buildings and resembles Figure 7.

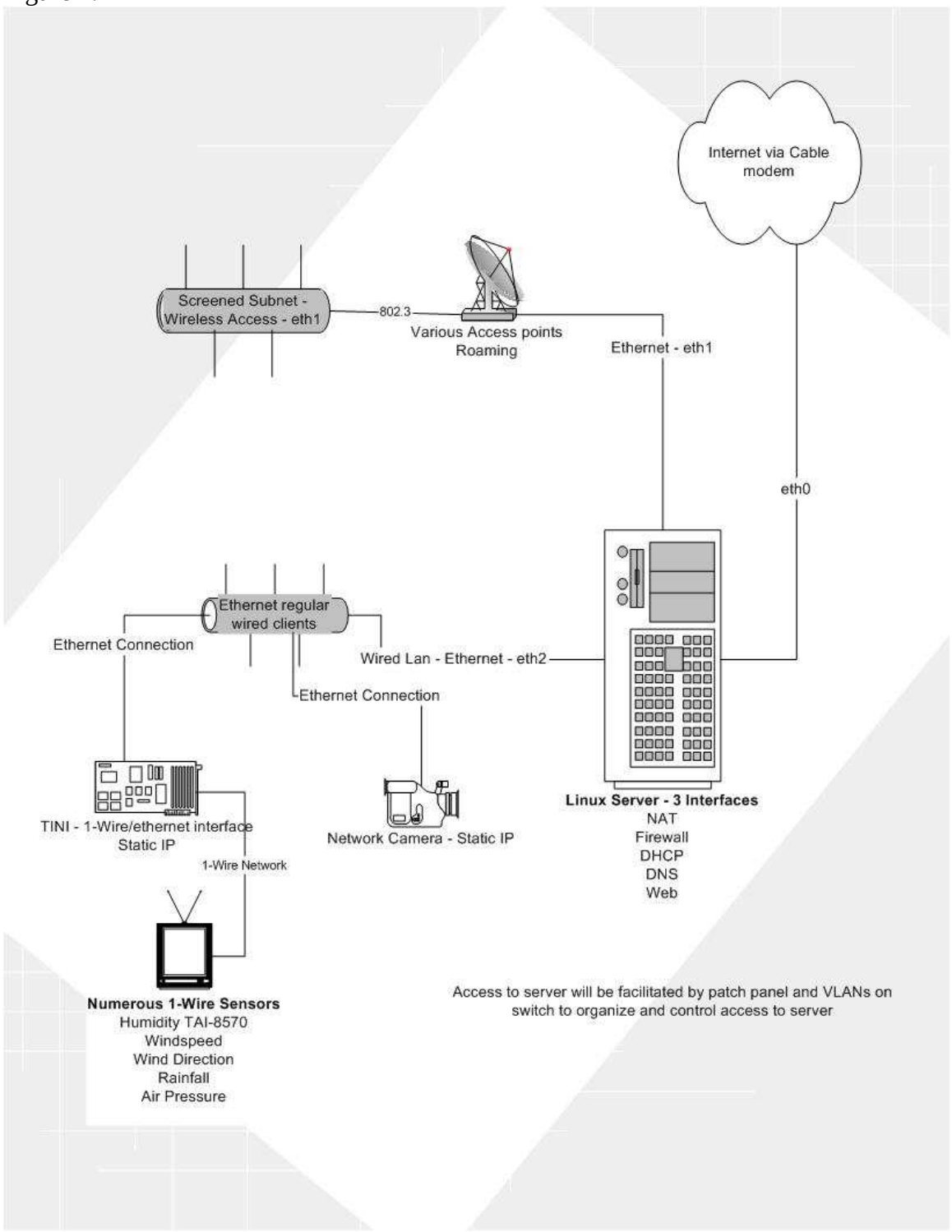

Figure 7 - Physical Network Layout

Linux server

The Linux server is running Debian (Etch) distribution. I chose this distribution because I am familiar with it, the operating system is free, supports all the services required and sports the very simple "apt" software configuration tools. Debian is now one of the higher quality and most popular Linux distributions.

One of the requirements of this server is that it is remotely configurable so it has SSH installed. The installation is a fairly simple and small one, with no X-Windows or extraneous software installed. Considerable care was taken to harden the server, using the guidelines supplied by Debian[5]. This server runs several roles, including firewall, NAT, DHCP, SSH, Http, DNS, and Apache.

Firewalling was accomplished through using some standard open source Linux packages, notably iptables, ipmasq, and dnsmasq. I needed to create some specific port forwarding rules to allow inbound traffic on specific ports to be forwarded to port 80 on three separate internal IP Camera's with different IP addresses (80 was taken by the www.portwash.ca website). Rules also needed to be created to allow SSH and web traffic to the Linux Server. Internally on the LAN I used telnet and FTP to connect to the TINI server, once I had connected through SSH to the Linux server.

Domain Name Registration

Having secured a dedicated Internet connection and static (sort of) IP address Pendercable, I registered the portwash.ca domain name online through a website called EasyDNS.com. I only mention this because of a problem that would never have occurred to me if it hadn't happened to me. ICANN requires certain information in order to register a Domain Name, it requires name, a Canadian address, and contact information, including an email address. As all Domain name registrations are public, within 48 hours of registering my email address in the ICANN database, I began, for the first time in my life, to be barraged by spam. I have since changed my registration to a dead end e-mail address, but some of the spam remains.

Uninterupted Power Supply

The power in the Gulf Islands is largely above ground, and as such, susceptible to outages, spikes, winter storms and the occasional drunk driver knocking over power poles. In that I need to manage this network remotely, I needed the capability to protect the server from power spikes, and have some way to reset the server in the event of a problem. I purchased a standard UPS with more than enough capacity for the power requirements to last for a considerable time. Power outages are common and often last for several hours. This results in losing remote access to the network. I toyed with the idea of a phone connection directly to the server or some sort of KVM control, but eventually bought and configured the UPS, which I needed for power protection anyway, to cycle the power to the server and the cable modem at 2 AM every third day, in the event of a loss of connection due to power outage, the system cycles and comes back online once every few days. The server, UPS and cable modem went into a small enclosed rack.

Figure 8 - Server, UPS enclosed in portable rack

Cabling

Cabling the network was also very problematic, the conditions were rough, requiring crawling through small, spider infested (see Figure 9) spaces in shorts. It is difficult to describe the size or virulence of these spiders without actually seeing one and being bitten by it.

Figure 9 - 3 Inch long Spider

The physical layout of the buildings meant for some very long cable runs, generally through conduit and PVC piping, which ran underground and was occasionally mouse infested, unlabelled and held no pull strings to begin with. A lot of ingenuity was required to get the standard cat5 cables through conduits, buildings and to various drop points.

Figure 10 - Cable run equipment shed

Figure 11 - Cable run, basement to Conduit

Switch

The switch was a fairly standard issue D-Link 24 port model, I didn't feel like there was enough need to justify an expensive Cisco switch. The D-Link switch sports a simple to use web interface which made setup and configuration simple.

All that was really required of the configuration was a method to VLAN the two separate internal subnets – wireless and LAN from each other to allow for separation without using a router to allow me to maintain the iptables firewalling rules.

Wireless

In the end, I wound up using Netgear "Rangemax" equipment. On testing I found that the wireless AP's that I originally purchased did not have the range required. The Rangemax access points use Multiple In Multiple Out (MIMO) technology, this is still not an 802.11n ratified devices, although they are hopefully advertised as "pre-n" access points as competing vendor technologies have not yet been agreed on.

This AP gave better range and coverage than the 802.11g Access points I had originally tried, and allowed me to set up a sizable wireless roaming network across the property, spanning 6 different access points. I originally setup the wireless network to be secure – the access points were capable of meeting a number of different standards for authorization and encryption. Given the known weaknesses of the WEP standard, I selected WPA2-PSK [AES] to encrypt all data –I quickly found that this configuration turned out to be completely unworkable from an administrative implementation standpoint. Every user who had problems connecting to the wireless network, which was almost all of them, soon contacted me and I had to try to troubleshoot the connection issues. Many older wireless cards were not capable of running the standard, and virtually all cards came with their own wireless connection utilities, which required learning different vendor utilities to troubleshoot users' connection issues. Much of this was done over the phone from Calgary to the West coast. It soon became clear to me that the WPA standard was unworkable, and reduced the security requirements to WEP, which proved little better in terms of its workability.

I eventually removed all security requirements and allowed unencrypted access; clearly vendors and industry have a lot of work to do in this area, probably 80% of all clients were unable to connect to the Wireless network at all once a security protocol was used for authentication and/or encryption, rendering the implementation completely useless. In a production environment where everyone had the same hardware and software it would be manageable, but in an environment where people were bringing their own equipment it was simply unworkable. I was unhappy with leaving the network open, but had little choice. I decided this was alright given the size of the property, and the difficulty I had getting enough range for even basic coverage inside the buildings. There was no coverage off the property and I did maintain strict iptable firewalling rules for traffic originating on the wireless network.

Network Camera's

An interesting sidebar to the project was the inclusion of IP Camera's as part of the project. The "scope creep" by this point was in full swing, and my straightforward implementation was entering my second trip to the coast and about three and a half weeks of dedicated time, and I had yet to even begin working with the TINI. I was dealing with a client who wanted every possible bell and whistle, and money was simply no object to getting them. Having agreed to a simple implementation, new possibilities were endlessly added as they occurred to the owner.

This part of the project stemmed from a casual discussion, with suitable beverages while watching the sun set, and I wound up installing three separate cameras. After searching for an appropriate "webcam", I realized that what was actually required was a "Network camera". Network (or IP) camera's, typically host, or at least stream, video display data to a web server. I first bought some cameras and tried them but the resolution was poor. I wound up purchasing some high end camera's from Axis. One "PTZ" – Pan Tilt Zoom camera (Axis214) and two fixed camera's (Axis211a). This introduced some new levels of complexity to the project. These cameras needed to be set up and integrated into the Ethernet network. The two fixed cameras were used as "public domain" cameras – one providing a view of the channel of the North End of Pender Island. Free public access was granted to users of Pender Cable as a promotion, and in return, Pender Cable provided us with a free Internet cable connection for our network. Since this is in the Gulf Islands, providers of even moderate bandwidth Internet connections are something of a rarity, so this was exactly what we needed. The second camera was fixed on a view of Port Washington – this camera is used to provide a view of the location where float planes regularly land in Grimmer's Bay – the exchange of a view of conditions in the Bay, along with the weather conditions – wind speed and direction proved to be useful in striking a deal with the local charter companies to provide cheap plane access to the island. The third camera provides a remote controllable view of the property.

All three of these cameras now needed to be integrated into the network, mounted, protected from the elements and sea water, and provided DC power. We erected an aluminum pole to mount the weather station and the cameras on the dock. One of the reasons I selected the fixed 211a cameras was the fact that they were standard Power over Ethernet (PoE) devices, which meant I also needed to find a PoE injector. I wound up pulling all cable to a centralized location in the basement of the main house, and attaching it to a patch panel. I placed the switch and the PoE at this location to centralize management. I had hoped to put all the equipment in the control shed where the Internet connection was brought in, but it turned out that one of the conduits running between two of the buildings had been crushed, so there was not enough room in the conduit to pull all the cables back to that location. I wound up leaving the server at in the control shed and centralizing the rest of the network in the basement of the house. (see Figure 12)

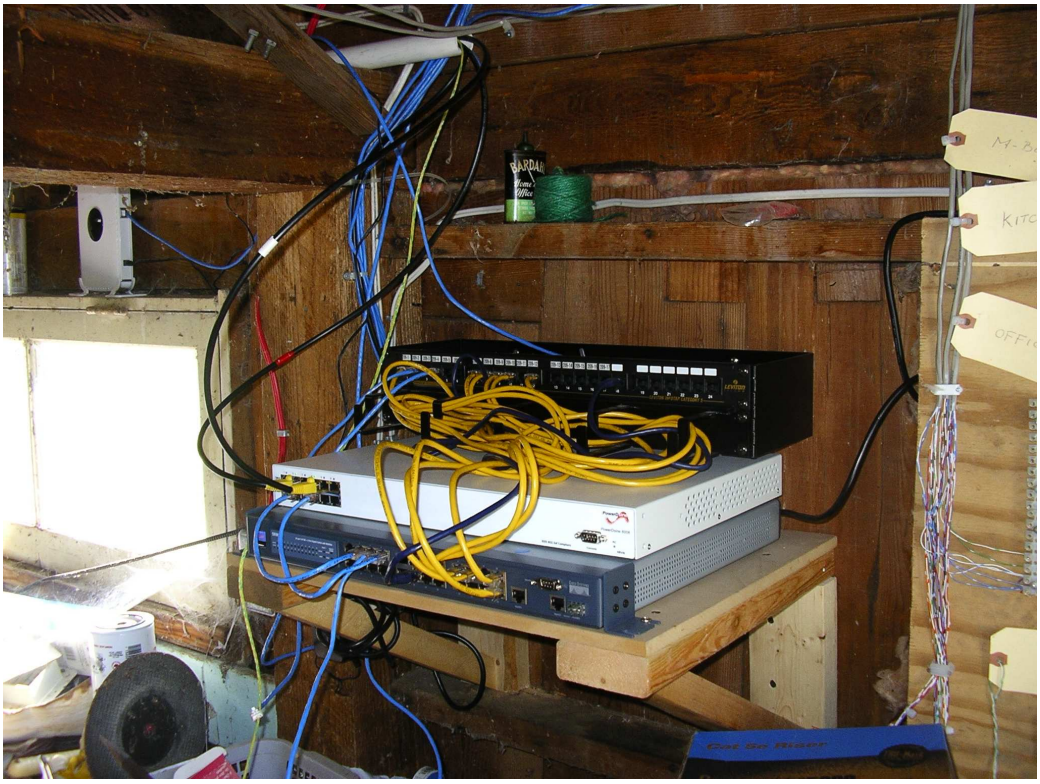

Figure 12 - patch panel and networking equipment in basement

All the cables pulled into the basement and the back of this patch panel. In this figure you can also see the switch and the PoE device (white box). The white PVC pipe at the top of the picture eventually (once the mouse nest was removed) runs to a point under the dock, some 70 meters away. Behind and to the left is one of the Wireless AP's.

The PTZ camera required DC power, for which we installed a transformer and strung both power and data cables through conduit to the cameras. To protect the camera's I purchased appropriate enclosures and mounted all three cameras and the weather station on an aluminum pole on the dock (as another aside, I actually got the opportunity to weld aluminum,- not only something I'd never done, but also something I didn't know was possible). This installation can be seen in Figure 13.

Figure 13 - Camera and Weather station installation

The cables were installed in the tubing to the left of the picture on the second rail. Pulling cable in exposed outdoor environments requires careful consideration of things you'd never have to do otherwise, like taking care in the slope of the conduit, it would never do to have water accumulating and running down the conduit into the "waterproof" enclosure. My experience this summer would teach me that waterproof devices are perfectly waterproof, provided you don't stand up and walk away from them. The ramifications of blowing a transformer because of a short when you are in a remote location and can not get access to new parts can easily add an extra five days to an installation.

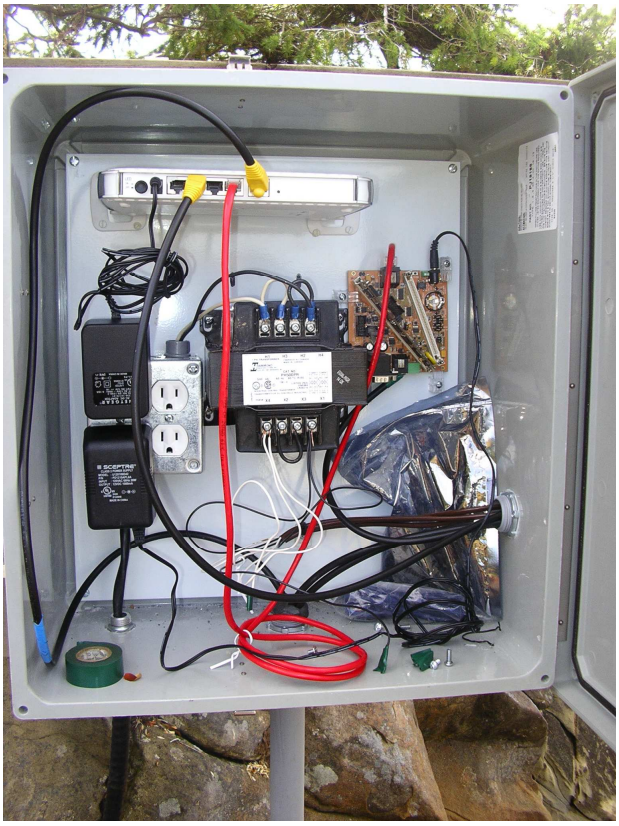

Figure 14 - Waterproof container holding Electrical Devices

Another issue that arose because of the cameras was that I now had four distinct (five depending on how I chose to implement the publishing of the TINI data) web servers using port 80 from our single internet connection. I set up port forwarding rules for the network Camera web servers to use non-standard ports.

1-Wire Functionality

The development of the 1-Wire standard is a joint effort of Dallas Semiconductor with various electronic companies. It follows an open source philosophy, with standards being openly shared; allowing various manufacturers to create devices, if a vendor device is compatible with the standard, then it is a 1-wire device. Although what primarily occurs is that companies will request 1-Wire devices from Dallas Semiconductor, and either have them placed into a circuit board by DS or in their own manufacturing facility. The following is a brief discussion of the technical aspects of the 1-Wire networking standards; more in depth information is included later in this report and is available on the web[6]. By using 1-wire devices, we can customize the type of sensing we wish to do, plug this device into a 1-Wire bus, and through communication with the "Master" on the bus, convert this information to a format which can be transmitted, from the sensor device, read by the master and forwarded to other nodes on different types of networks.

The miniaturization of all electronic components is well known. One need not think back too far to remember mammoth calculators, televisions, and stereos. In the computer industry the advantages of miniaturization are fairly obvious, allowing more circuitry to be packed into smaller, increasingly complex devices using less power. One disadvantage of increasingly small chip size coupled with increased complexity is that this has lead to smaller and smaller devices feeding larger and larger data buses. Finding space to put more pin connectors on small devices becomes more difficult. The 8 bit processors of 20 years ago were not significantly different in size than the 64 bit processors of today. This is an 8 fold increase in data carrying requirements, to say nothing of the other connectors to supply power and monitoring, etc. Chip manufacturers were under pressure to put larger data busses into smaller and smaller devices. Engineers at Dallas Semiconductors chose to take this to the extreme in the opposite direction – build a system with a single data bus, which would allow for scalability and expansion. The result was 1-Wire.

A 1-Wire network is somewhat misleadingly named; in this type of network, two wires are actually used, one is used to carry signal, while the other is used as ground. The reason we need a ground wire will become apparent with the discussion of the function of the network, although strictly speaking, the ground could be anywhere, it need not necessarily be a short to the ground wire. This type of network is diagramed in Figure 15.

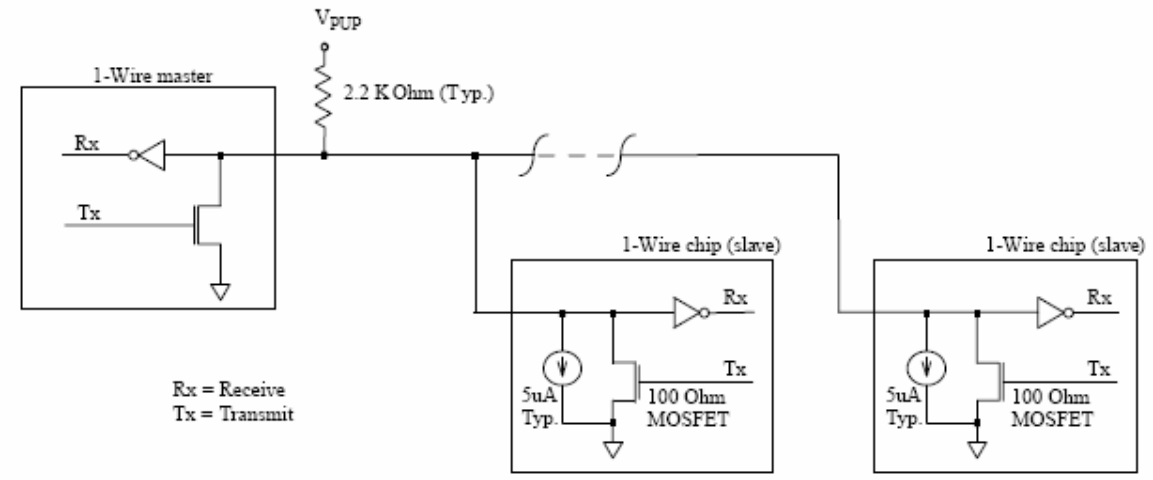

Figure 15 - A simple 1-Wire configuration[7]

This diagram shows a usual configuration structure of a 1-wire network – There is always a single "master" on a 1-Wire network, in our case the T-Stik/TILT board combination hosting the TINI O/S, and any number of "slave" nodes on the network. The nodes are effectively on a serial bus, and the master will control all access to the bus. The master is indicated to the left of the picture, the other points of interest are the MOSFET (metal-oxide-semiconductor field-effect transistor) gates, and the Pull Up transistor (V_{PUP}) which maintains a set level of electrical potential on the data bus, which can be shorted by 1-Wire slaves through the gates.

1-Wire Communication

Devices can communicate at either regular or overdrive speeds on the network – the regular speed is 16.3 kilobits/sec while the overdrive allows 144 kilobits/sec, but this capability can only be reliably used on small networks with few slaves. 1-wire devices are open drain driven and so they drive the bus low, and the bus is returned to a high state by the pull-up resistor, which is typically integrated into the master end of the bus.

The Master system can use four discrete signals to signal the other 1-wire devices;
1. reset sequence
2. write 0
3. write 1
4. read data

The reset sequence returns all devices on the bus to a known initial state. The reset sequence consists of a master generated reset pulse followed by a device generated presence pulse. The master will drive the bus low for a lengthy period and then release the bus to indicate it is ready to receive; the bus is then taken to a high state via the pull up resistor (V_{PUP} in above diagram) devices on the bus will then detect the rising edge on the bus and transmit the presence pulse, by simply driving the bus low for a predetermined (but short) amount of time. This interaction simply resets all devices into a listening state, and the presence pulse indicates the presence of devices other than the master on the bus.

Data is transmitted through the use of write 0 and write 1 signals. Data transmission is accomplished on the network by dividing time on the network into timeslots. The master drives the bus low for at least 1 μs and releases the bus. Voltage is returned to a high state by the pull up resistor and devices on the bus are synchronized to the master by monitoring the falling edge. In a write timeslot, a device can either drive the bus low (a 0) or leave the bus high (a 1). The master continuously samples the data line in the read time slots and can use this to determine if any of its transmissions had any corruption or contention issues.

1-Wire Transactions

Data on a 1-wire network is transmitted through a "transaction". A transaction consists of three phases;

Initializing
Addressing
Data exchange

a) Initializing phase
The master begins a transaction by initializing the network, which consists of the reset sequence described above; the master drives the bus low for an extended period (a minimum of 480 μs) and devices on the network respond by transmitting a presence pulse. This notifies the master that there are other 1-wire devices on the network.

b) Addressing phase
The master can then initiate the collection of data through the addressing phase. The addressing phase consists of the master transmitting the entire 64 bit address of the device which the master is interested in communicating with. All devices listen for and receive this broadcast. All devices except the targeted device then drop off the bus and master can then transmit a specific command to the targeted device to do something. Each 1-wire device has specific commands which it is capable of carrying out, depending on the function (or family) of the device. The bus can then be reset by the master to allow for another transaction. The following two figures demonstrate two transactions between the master and a DS18S20, a 1-wire device capable of sampling temperature.

Sample 1-Wire Transactions

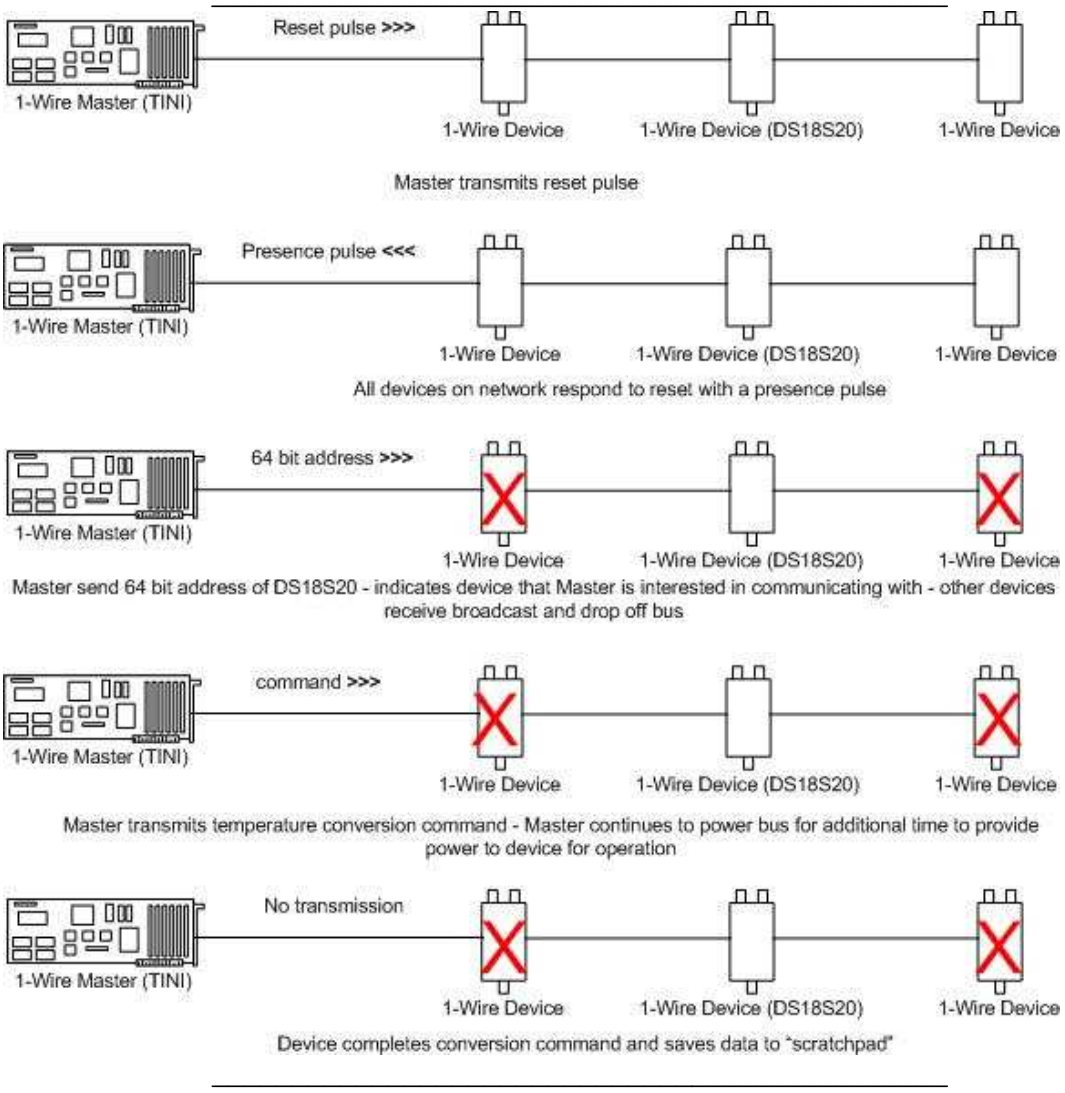

Figure 16 - An example transaction on a 1-wire network – getting a temperature conversion.

To begin the above transaction, the master will drive the bus low for at least 480 μsec, resetting the bus. This stops all incidental "noise" on the network, and notifies all devices (which are continuously monitoring the voltage of the bus) that a transaction is being commenced by the master. As the voltage is pulled up by the pull up transistor, the devices are all aware of the rising edge, and then signal their presence and awareness on the bus by driving the bus low in a presence pulse. The master can then transmit the unique address of the targeted device, causing all other devices to drop off the network and only leaving the master and the target device. The address of the device (as explained later) indicates the family of the target device and the commands which can e carried out by that device. The master then submits the command it wishes completed to the target device.

In this case there are a few interesting notes; the master initiates and controls all transmission on the network, dealing with contention issues. At the end of the transaction, all devices other than the master and the targeted device are idle – so to initiate communication in subsequent transactions, the master must always reset the bus at the start of every transaction. Also of interest is the ability of the master to power the bus high for a long enough period of time to allow the targeted device to do the work required, this means that 1-Wire devices do not require an independent power source. This is known as parasitic power, the device using the power on the line to complete its required commands. This has certain advantages and introduces concerns, sensing devices do not require any power, which is difficult in many locations where one wishes to do sensing. On a large 1-Wire network, many devices will drain the power from the line, causing devices to think that a reset sequence has been initiated. Likewise, the transmission of a large number of consecutive 0's may also inadvertently trigger a reset in devices, although this is extremely unlikely given the length of time of a reset sequence.

Of course at this point, the device has done the sampling and conversion, but the information it has stored has yet to be sent to the master, which will need to be taken care of in subsequent transactions.

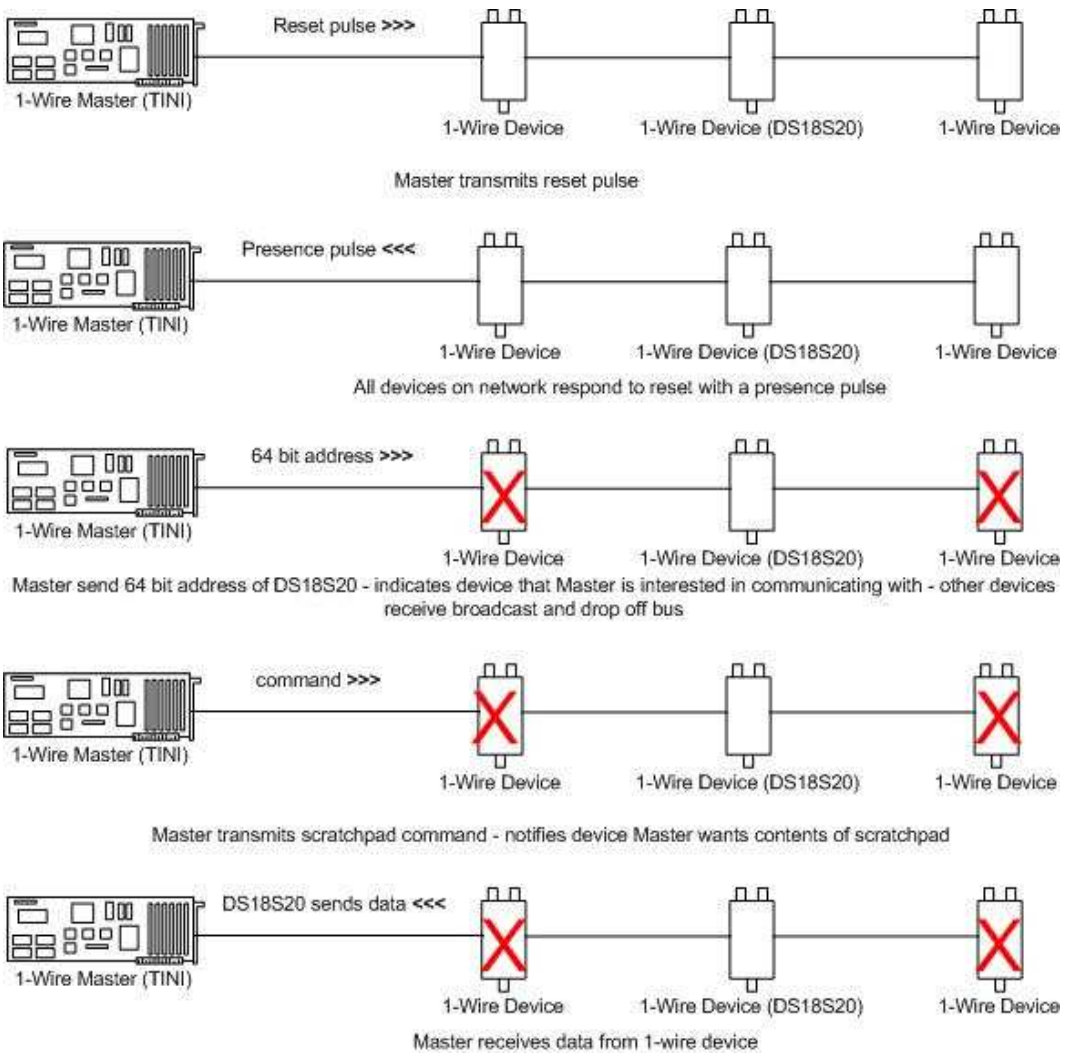

Figure 17 - An example transaction on a 1-wire network - receiving temperature conversion data

In this case the pattern is the same, with the master resetting the bus, targeting the specific device and then issuing the command, which in this case is a request for the contents of the "scratchpad", a 256 bit page in memory of the device which acts as a buffer for data. After the scratchpad command is issued, the device transmits the data in a series of write 0 and 1 commands by driving the bus low (or leaving it alone) in the appropriate time slots.

1-wire networks utilize this very simple communication protocol – all communications are initiated by the master, preventing collisions on the network, and providing power to remote devices at the same time. 1-Wire cabling has an outside physical length limitation of approximately 750 m^8, due to timing issues. The master also has the capability to drive the bus high to allow for longer distance transmission.

1-Wire Addressing

Obviously, given the steps of this communication protocol, every device on the network must be uniquely identified. To this end, all devices have a 64 bit address which is of the following format;

{[8 bit CRC] [48 bit device ID] [8 bit family ID]}

The family ID identifies which group of devices the appliance falls into, identifying it's capabilities to the master. The large Device ID ensures uniqueness, and the CRC portion allows confirmation of correct transmission of the identifier, for both the device ID and the family ID portions. There are literally hundreds of different 1-Wire devices, all with different capabilities. A complete list of 1-Wire devices and families can be found at Dallas semiconductor / maxim-com[9] Website. The simple nature of these devices allows a user to innovate and customize the sorts of sampling that is done by the 1-wire network (and how). I chose to use weather sampling as a typical sort of example, but really the possibilities are endless in terms of what is possible, allowing the user to customize what and how data is sampled, and provided in a format that is flexible enough to allow the user to again customize what is done with the data and how it is worked.

A list of the devices used in this project, family identifiers and capabilities is presented in Figure 18.

Device Name	Family	Description	Interfaces	MemoryBanks
DS1920 DS1820 DS18S20	10	Temperature and alarm trips	Temperature-Container	
DS2406 DS2407	12	1K EPROM memory, dual switch	SwitchContainer	MemoryBank PagedMemoryBankOTPMemoryBank
DS2423	1D	4K NVRAM memory with external counters		MemoryBank PagedMemoryBank
DS2450	20	quad A/D	ADContainer	MemoryBank PagedMemoryBank

Figure 18 - 1-Wire device list for devices used in this project[10]

Getting the Device ID

As mentioned in the above discussion, the Master needs to know the correct device ID to be able to communicate. This can be done in the course of the operation of the program on the master, but would normally be pre-determined by the programmer and placed into the program in some way. These ID's can be obtained in different ways; depending on the vendor, some devices come with the Device ID on the side of the appliance and Dallas Semiconductor provides snippets of Java code to do various tasks which can be implemented, including learning Device ID's. A programmer could embed the command to learn the ID's into their program, or determine the ID's ahead of time and hard code the device ID's, assuming they won't change, into your own code – which is what I initially did, or use a separate text file to list ID's and have the program query the file when required, which is what I did when I started using replacement equipment in Calgary, and knew that I would have to switch it back once the 1-Wire network on Pender was repaired.

Included with the download of the TINI OS (TINI SDK) is sample code for a variety of tasks that can be done using the TINI. The following is the output of the compiled code

running the ListOW application, which is designed to query the network to discover the addresses of all devices on the network[11].
The following is the redirection of the output of running the compiled ListOW.java program on the TINI;

TINI0247f4 /> java ListOW.TINI

Adapter: TINIExternalAdapter Port: serial1

A0000000008B6120
B200080014188810
75000000017A161D
F70000000106991D
9C0000000936D626

The above ListOW output allows us to not only determine the ID's of the device, but also identifies the family to which the device belongs, the capabilities inherent in that family, and which build dependencies will need to be included in the compilation of the application.

The above addresses can be examined in light of the discussion surrounding the 1-Wire addresses. Each address is discovered as 16 hex values, or 64 bits. As discussed, the last byte (2 hex values) indicate the family to which the device belongs. A complete list of families can be found on the web[12]. Examination of the last two hex values of each device in comparison with a list of 1-Wire families allows us to identify what families these devices belong to;

A0000000008B6120 - DS2450 – AtoD for Wind Direction FC=20
B200080014188810 - thermostat DS18S20 FC=10
75000000017A161D - DS2423 - Wind Speed FC=1D
F70000000106991D – Rain Gauge FC= 1D
9C0000000936D626 – Pressure Barometer FC=26

The first two hex values are the CRC values for the address (an in depth discussion of 1-Wire CRC values and algorithm can be found in Don Loomis's 1-wire development guide[13]). The first two values are for CRC calculation while the next 12 values ensure the uniqueness of the address, while the last two values indicate the family to which the device belongs.

Limitation in terms of number of devices on 1-Wire network

A 1-Wire network usually has what is known linear format, but can also take on one of two other topologies; star or stub.

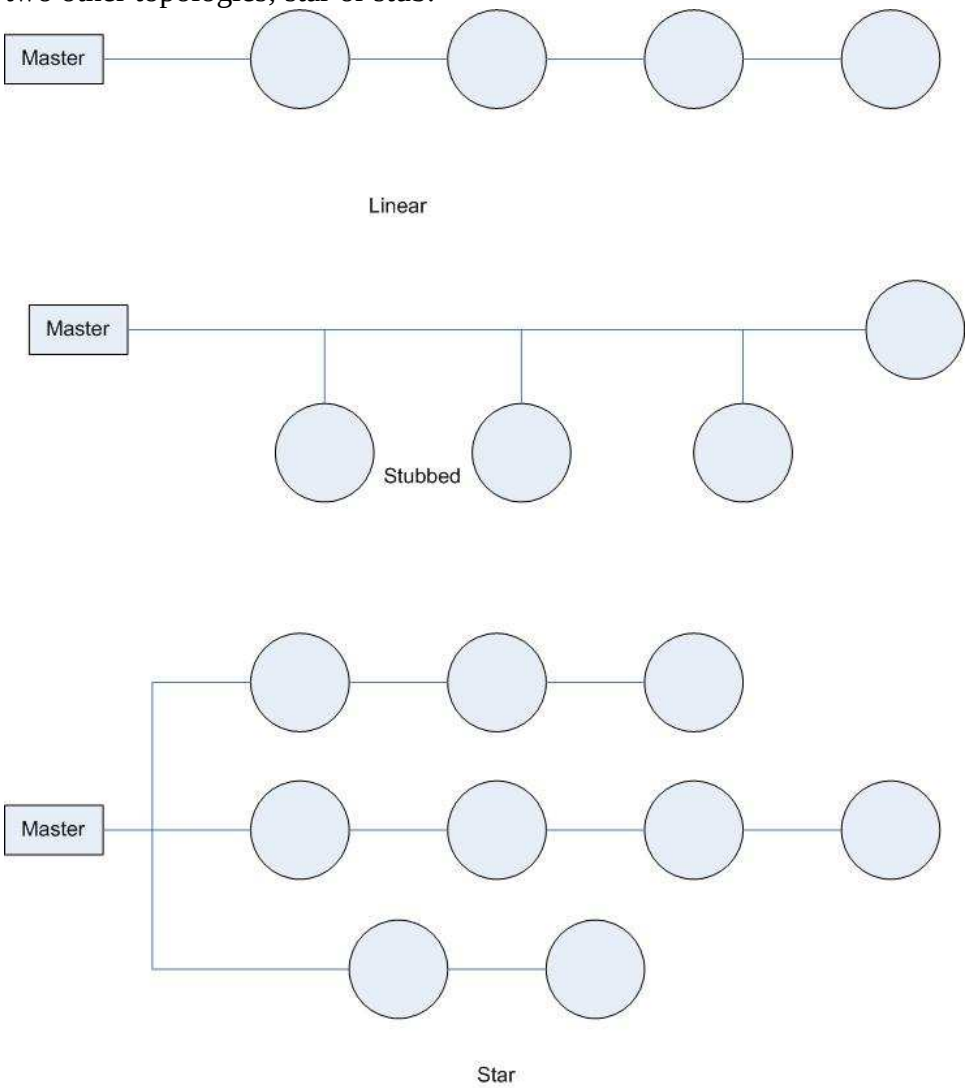

Figure 19 - 1-Wire Network topologies

There really is no difference between stub and Star topologies, a stub network with several devices on each stub is not considered a star, generally stub networks have arms less than 3m, while star topologies feature at least one arm over 3.

A 1-Wire network is often described as a Microlan, and can have different topologies and number of sensors. Two critical terms to the performance of a Microlan are the “weight” and the “radius”. The radius of the Microlan is described as the maximum distance of the

wire between the Master and the furthest 1-Wire device. The weight of the network is the distance, expressed in meters, of the total amount of wire used for the Microlan. As an example, you may have a star topology on your 1-Wire network, with 4 distinct arms of the following lengths; 5, 10, 15, and 20 meters. The radius would be 20 meters and the weight would be 50 meters. Since there is a single Pull Up resistor, larger weighted networks will impact the pull up times on the entire network. Once the pull up resistor is unable to overcome the weight of the network, sensing devices will think a reset signal has been initiated, or simply lose power entirely. The radius will impact the timing of the slowest (longest) signal reflections. A Microlan which features a star topology with two arms, one short and the other very long will have trouble regulating time slots for read and write operations. In this case the reflection of signals from the short arm effects the devices on the long arm rather like old unterminated 10base2 networks, and because of this, this design is not recommended. This problem does point out the need for inclusion of impedance in the network to dampen reflecting signals, and to terminate all cable runs with a 1-wire device.

Each 1-Wire or ibutton (an ibutton device is essentially a self contained 1-Wire device ‘in a can”) sensor adds to the capacitance of the network, and can be thought of in terms of the effect it has on the bus’ ability return to it’s normal resting potential, and in this way can be thought of as adding weight to the network equivalent to that of a small portion of wire. Each device has different operating requirements, depending on its Family Class, its functionality and what the master device requires of it. As a rule of thumb, an ibutton device is considered to add on average 1 meter of weight, while a regular 1-Wire device adds .5m. Any other connectors or circuit board traces will add weight to the network. Studies have shown that standard Microlans cannot exceed a radius of 750m, the timing delays between the master and furthest slaves becoming too great for the protocol. This is a smaller problem than weight, since weights exceeding 200m is beyond the capabilities of a simple pull up resistor. Intelligent pullup designs have increased the maximum weight to 500m through the use of higher voltage active pull-up resistor under some form of logic control. It is possible to get repeater devices and hubs to extend the ability of the resistor to carry more weight and manage complex topologies.

The limitations on a 1-Wire network caused by the weight of the Microlan is further compounded by the parasitic powering of 1-Wire devices, particularly on networks which are carrying a weight approaching the capacity of the pull-up resistor.

Other than the weight of the given network, there is a limitation to the number of devices a 1-wire Microlan can reasonably handle, depending on the frequency of the queries which need to be made.

1-Wire protocol/network limitations

If one considers the above discussion, it becomes apparent that the protocol used in 1-Wire communications comes with tremendous overhead. Even so, if we evaluate the transaction discussed in Figure 16 above, it is clear that the other problems associated

with weight are much larger than the issues of the number of devices on the network. Even with the relative speed of the bus, it takes a considerable period of time to do this transaction. This discussion assumes the standard 1-Wire transmission rate of 16.3 kilobits/second. 1-Wire is capable of an overdrive speed of 144 kilobits/second, but is only feasible over extremely short cable distances with few devices.

To initiate the above sequence, the reset pulse lasts for at least 480 µsec. A Microlan with a larger radius and weight would require even more time than the minimum prescribed in the protocol. The presence pulse can then be sent back to the master. Following the reset pulse, the devices detect the rising edge on the bus and wait up to 60 µsec and then drive the bus low for up to a further 240 µsec. Following the presence pulse, devices communicate in master controlled timeslots, which are 60 µsec each. The master then transmits the match address command byte (0x55), ordering all devices to listen for the target address (480 µsec). The master transmits the 64 bit address of the target device (3840 µsec) and then transmits the requested function (depends on device) – usually a single byte specific to the functions available on the device (480 µsec). If required (as in the example in Figure 16), the bus is then maintained in a high state for a prolonged period to provide power to the device – in this case approximately 750 µsec. So in this case a single transaction from the bus initiation to conclusion can take up to 6330 µsec. To then acquire the data generated in the previous step, much of the same process is repeated and takes a further 20940 µsec to return the data to the master, in all for the Master to initiate the communication and receive the data back from the device will take up to approximately 27270 µsec or almost 30 milliseconds. Assuming no contention or data corruption issues requiring retransmission, then every device on the network would require 30 milliseconds to be polled; 10 devices would require .03 seconds, allowing us to sample up to 2000 devices a minute. On a real 1-Wire network there would likely be more issues, but the primary area for failure is shortages, insufficient power to restore the bus to its potential, usually determined by the overall weight of the network.

TINI limitations

The other limitation in the DS80C400 is the ability of the processor, and the amount of memory available to it. The 1-Wire bus might comfortably hold 150 different devices, but the TINI must query these devices, save information, run any other services required of it. The TINI is capable of hosting a small website, but the memory available and clock speed of the processor result in web page's being small, simple and low in physical memory. If we were to try to sample 1-Wire devices with any regularity and log the information, we would run out of room on the device within a short period (in the range of a few days to a few weeks depending on the quantity and type of data). If the TINI is working to continuously sample data, crunch the data it receives, publish a website and log, the DS80C400 will be sorely taxed and response times will increase greatly.

Software Development

Compiling programs to run on the TINI proved to be challenging. The DS80C400 is a microcontroller designed to hold the TINI OS – ideally it is designed to hold Java programs, but it can also run C compiled programs. The java interface allows for the use of the built in FTP, Telnet, and Serial Server. Java allows programs to be written in an object oriented format, and can also run a simple http server. There are some great possibilities for using the TINI as a standalone web server. For these reasons I chose to go with Java rather than trying to develop my programs in C, which is possible with a Kiel compiler. However, not being familiar with Java, I found compiling, even simple Java programs, into a format that is could be run on the TINI extremely difficult. The process of compiling these java programs is as follows;

A current version of java needs to be downloaded, and can be obtained from the sun website[14]. I downloaded java 1.4.2 and installed it, although a newer version of Java has been released since I began this project. The programs can be written in a regular java format, but then need to be compiled and converted into a format that will run on an 8-bit Microprocessor. This can be done by downloading the TINI Software Development Kit (SDK) System from Dallas Semiconductors[15]. There were 2 primary downloads from this site, an older version (1.02), and a newer (1.17) which has substantial changes over the earlier version. I downloaded and ran all development in version 1.17. The 1.17 firmware comes in a tar file and extraction of the file resulted in the firmware, source code, files a number of required "jar" files including the TINI.jar, TINIclasses.jar, TINI.db, and the TINI_400.tbin.

The TINI.jar file is a Java ARchive which contains three utilities, the Javakit (responsible for firmware loading) the TINIconvertor modifies Java class files and creates binary versions suitable for execution in a TINI environment and the TINI.db is a database used by the TINIconvertor to modify the Java code to TINI compatible executables.

TINIclasses.jar contains all the classes in TINI's API.

TINI_400.tbin is a TINI binary and is the firmware that needs to be loaded to the Flash ROM on the TINI server. The tbin file holds a combination of the native operating system and the Java API – the TINI Java runtime environment and slush shell application.

As previously mentioned, uploading the tbin file was problematic, both because of the obsolescence of the javakit and the lack of finding an "old school" serial cable to upload the TINI_400.tbin. This was eventually accomplished, and by utilizing the built in functionality of the TINI OS, I was able to log on to the TINI server over the network via telnet. The Operating system is small and simple, and working in the Slush environment intuitive to anyone who has ever worked in a UNIX shell like bash. The file system is very simple and the directory structure holds an etc directory with a configuration file and

a passwd file. The commands accepted by the TINI OS are typically Unix shell based, but stripped down and can be viewed with a "help" command.

To create a program which can run on the TINI device, we first need some Java code. This takes the form of one (or more likely several) *.java files, these will represent different class files, and the classes are declared in java files of the same name. So given a number of java files, they can be compiled using the java compiler;

Using the java compiler requires a few things, the bin directory for the Java development kit (JDK) needs to be in the current path, and later versions of Java requires the input of the –target 1.1 option.

C:\javac –target 1.1 myfiles.java

This will result in the creation of a "myfiles.class" file… sometimes.

Some commands have changed between different releases of java, so we can force the compatibility for different the source using a –source switch. If the java file contains commands which are not part of the regular Java API, the compiling needs to be done as follows;

C:\javac –target 1.1 – bootclasspath –source 1.2 c:\pathtotinifiles\tiniclasses.jar myfiles.java

This might be a little difficult to see in a written document, but these commands should all be inserted on a single line. If there are multiple java files, the compiler can be helped out with the use of a wild card character – by running the javac command from the folder above the source code, it should pick up and compile all *.java files into class files. The above command should create a file named myfiles.class, which then needs to be converted into a form which is executable on the TINI.

To do this we can use the tiniconvertor;

C:\java –classpath c:\pathtotinifiles\tini.jar TINIConvertor –f myfiles.class –o myfiles.tini –d c:\pathtotinifiles\tini.db

this command will take the class file(s) and output a single file called "myfiles.tini" (all existing "class" are left in the folder and need to be deleted manually). *.tini files can be uploaded to a TINI via the FTP utility and run using the java command in the slush environment;

tini>java myfiles.tini

A couple of other things to mention in this process, the first is that copying the tini "jar" files from their location on the hard drive to the \bin folder of the JDK (which is one of

the things a well meaning colleague advised me to do make compiling easier) results in programs which do not compile and run correctly in the tini, they must be specified by the appropriate –classpath or -bootclasspath option. Another consideration is that if we are going to be querying 1-Wire devices with this code, we will need to make further modifications to this process. The one wire API (OWAPI needs to be compiled into the class file, along with containers that will be referenced in the code. For example, if the above example was being used to communicate on a oneewire network, and was going to be querying a 1-Wire device which was a temperature sensor, meaning we would need a temperature container from the 1-Wire family "10". The device ID in this case would certainly end in the number 10 for the last two digits. In this case we would need to modify the compiling and conversion process to include all the build dependencies as follows;

Compile:

C:\javac –target 1.1 -source 1.2 – bootclasspath c:\pathtotinifiles\tiniclasses.jar -classpath c:\pathtotinifiles\owapi_dependencies_TINI.jar myfiles.java

Then build:

C:\java –classpath c:\pathtotinifiles\tini.jar BuildDependency –x c:\pathtotinifiles\owapi_dep.txt -p C:\pathtotinifiles\owapi_dependencies_TINI.jar TINIConvertor –f myfiles.class –o myfiles.tini -d c:\pathtotinifiles \tini.db -add OneWireContainer10

Following this the resultant tini file (myfiles.tini) can be uploaded and run on the TINI as in the above example.

To develop the software to; a) run in a TINI environment and b) query 1-Wire devices can be developed in a couple of languages. As previously discussed, I chose to do the development in Java, for the flexibility, power and ease of the development. I had never done any programming whatsoever in Java, my experience being limited to some very old languages like BASIC, Pascal and more recently C, none of which are object oriented programming languages. I would like to point out that the "ease" part of my expectation in terms of development was sorely misplaced; my expectation was that compiling in Java would be like compiling in C. It wasn't. My experience with gcc is that it always works as expected, and once compiled that program can be consistently run. In Java, this isn't always the case. Installing java is a large download, resulting in many directories and files being installed. Considerable configuration needs to be done to properly set variables like the PATH and CLASSPATH before Java will compile a single program. For a Java neophyte it was extremely difficult to figure out, although there is considerable help available online and in texts, much of this information is spread out, and needs to be pulled together. The getting started with Tini guide, for example, clearly explains the building and compiling process, to run applications on TINI but covers none

the build dependencies or the necessity of including the owapi dependies or the container files from the tini.db. Errors generated during attempts at compiling did not clearly indicate a dependency problem, and for a new Java programmer, it was easy to assume that the problem had to do with something in the code itself.

In addition to this was a physical wiring problem on the Pender, which made it impossible for me to test my code in a 1-Wire environment. I was able to SSH to the server, and from inside the network telnet to the TINI device, but some sort of a short in the wire did not allow any communication with the 1-Wire devices, making it impossible to test the 1-Wire network portions of the code. This worked when I left Pender, but ceased working sometime shortly afterwards. To rectify this I eventually got a second TINI device and some 1-wire gear to test my code locally, and am still waiting to return to Pender to troubleshoot the wiring problem.

It took some time to get used to the Java concept of "classes" and class files, I tried to think of a class as a function, and that helped. When compiling Java, the Java compiler searches for and compiles all class files required in the code. The code I developed generally placed whatever functionality I was attempting to add into the program into a separate class file. All the separate class files were referenced by the "mintwx" class file, containing the main() function.

In the end I wound up with 8 classes making up my program, the MintWX class being the center of the program. Due to the length of these programs, they can't reasonably be included here, I will discuss two of these classes – MintWX and Onewire, and provide some sample output captured via file redirection on the TINI.

Sample TINI output
MintWX.tini running example

The following text in italics is the output from a sample run on the TINI – I have included debugging output to demonstrate. The program starts and parses a file called prefs.ini on the TINI device. 1-Wire ID's are included in the preferences file. I changed the frequency of calculating wind direction, and sending data via a socket for a shorter demonstration. The wind direction calculation is taken from Tim Bitson's Extreme Tech Weather server (with permission) and is quite crude, but this turns out to be very difficult to do inside the TINI runtime due to the math libraries available. This analysis gets current values for time and temperature, and simply takes the current value for wind direction at sampling time. It is actually set to take an average over the sample period (5 minutes), but I have this shortened to a single minute in this example.

Following the collection and storage of these values, MintWx connects to a socket server program running on the server and dumps the data in a format which is mySQL friendly, from where it can be pulled and published in the much more powerful Apache server.

```
Starting MintWx v.4
debug on
Setting Tini's Timezone...
TINIOS Timezone = MST
MintWx Local Time = Tue Jan 09 01:17:04 MST 2007
MintWx v.4 Started
Found Adapter: TINIExternalAdapter
Invalid or No Rain Counter Device in prefs.ini
Invalid or No Pressure Device in prefs.ini
Device Error Counter Reset
Resetting 1-wire bus
Average reset
Getting weather at 1:17
Temperature: DS1920 F20008001B46B810
Temperature = 64.287498 degsF

Resetting 1-wire bus
Resetting 1-wire bus
Wind Speed: DS2423 2000000001508E1D
Count = 0 during 1168330654394ms calcs to 0.0MPH
Resetting 1-wire bus
Wind Direction: DS2450 4800000000E65A20
Wind Dir AtoD Ch A = 0.055782102
Wind Dir AtoD Ch B = 4.4328799
Wind Dir AtoD Ch C = 4.4292864
Wind Dir AtoD Ch D = 4.4386615
Wind Direction     = 12


Resetting 1-wire bus
Resetting 1-wire bus
Updating Weather Page
24 Hour Index = 7
GetWindDirectionString input = 12 and cal = 0
Wind Direction Decoded = 12 =  W
Bin 0 = 0
Bin 1 = 0
Bin 2 = 0
Bin 3 = 0
Bin 4 = 0
Bin 5 = 0
Bin 6 = 0
Bin 7 = 0
Bin 8 = 0
Bin 9 = 0
Bin 10 = 0
Bin 11 = 0
Bin 12 = 2
Bin 13 = 0
Bin 14 = 0
Bin 15 = 0
Wind Dir: Found max of 2.0 at index 8 Offset of 4
GetWindDirectionString input = 12 and cal = 0
Wind Direction Decoded = 12 =  W
Wind Dir = 12 =  W
```

```
GetWindDirectionString input = 12 and cal = 0
Wind Direction Decoded = 12 =  W
Device Error Counter Reset
Connecting to 192.168.1.153
message =&dateutc=2007-01-
09+08%3A18%3A00&tempf=64.3&windspeedmph=0.0&windgustmph=0.0&winddir=270
sending message
waiting for response…
waiting for response
response=Data received at server, thank you!
Average reset
```

The mintwx.class

The mintwx.class file follows a fairly straightforward layout with the usual io, java and onewire imports – these imports are part of the TINI O/S download and the TINI sdk (software development kit). The entire code for this class can be found in the appendix. The program follows with the declaration of the Mintwx.class, which holds the "main" function. This class begins by pulling together the standard constants, specific 64 bit 1-Wire addresses and initiates a loop which sleeps for a minute and then queries the 1-Wire devices. Specifics of querying other 1-wire devices, data logging, socket creation and sending data to the web server are handled in other classes, which are called as required.

The central program in this project is the MintWx.java – this programs contains the main class, and the main method, it sets up the operating environment – in general sets up the main () method to loop once a minute, querying the 1-Wire devices on the network and saving the information. Once every 5 minutes it calculates the average values and creates a socket connection to the web server and dumps the data to a server program that I created in C which listens on a specific port (5446) and then places the data into a SQL database for long term storage and reference, and publishing to the web in Apache. I have borrowed (with permission) from Tim Bitson's extreme tech weather server, the weather calculating routines in this program, the logging routine and the techniques for setting and using preferences. The largest difference between the programs is that the ETWS generates and serves a web page directly from the TINI, rather than forwarding the data to a centralized server. I see my design as having two major advantages; scalability and performance. Any number of TINI devices could then be plugged into an Ethernet LAN forwarding data to a single reference point. This design also avoids some of the limitations of the TINI itself, with memory limitations, the ETWS will freeze after a few days if full logging is implemented as the available memory is used up. The TINI performance is also slower than my server's P4 processor with a 2G of RAM, and has a hard limitation of 24 concurrent socket connections.

I originally built the Mintwx.java program with the 1 minute query and 5 minute loop and then began feathering in more functionality through the addition of new classes. For example, when the OWSocket.java program (class) is called and it generates a socket

connection to the server, and queries the other classes to get the weather data, then dumps that information into the socket.

A quick run through of the code (refer to the appendix) reveals a reasonably small program, and follows the convention of putting the main() function at the end. The file begins with includes and creation of variables. Time keeping and creation of all necessary variables for the later loops follows.

Inside the loop, the key line in terms of getting the weather is;

weather.getWeather();

this line basically calls the getWeather function in the weather class. A truncated version of the getweather function is;

```
public void getWeather()
 {
  String device = "";       // device name for error message
   try
  {
    // get data from each sensor…
    device = "Temperature";
    temperature = ws.GetTemperature();
    ws.resetBus();

    device = "Pressure";
    pressure = ws.GetPressure();
    ws.resetBus();

    device = "";

    sumTemp += temperature;
    sumWindSpeed += windSpeed;
    sumPress += pressure;

    dir[windDirection]++;

    numSamples++;

    if (windSpeed > windPeak)
      windPeak = windSpeed;

  }
```

Inside this function, each sensor is queried in turn by the lines

```
temperature = ws.GetTemperature();
ws.resetBus();
```

ws is actually a variable which represents the onewire class, so this command is actually filling the “temperature” variable with the contents of the Gettemperature() function found in the onewire class.

The onewire class is responsible for querying the devices on the 1-wire network, this class begins with the declaration of the tempDevice using the temperature container used during compiling.

```
private OneWireContainer10  tempDevice = null;
```

which will allow us to use tempDevice in the GetTemperature function

```
byte[] state = tempDevice.readDevice();
 tempDevice.doTemperatureConvert(state);

 state = tempDevice.readDevice();
 temperature = (float)tempDevice.getTemperature(state);
```

This seems fairly explanatory – the container has been identified, and then uses this value to execute the readDevice() function, eventually assigning the the sensed value to the temperature variable.

Areas of Future Research

It seems that a move towards greater convergence in network technologies is long overdue. We are finally beginning to see things like phone connectivity being converged with the data network. Investigation into the standards and protocols required for network integration by all sorts of vendors, like companies such as Leviton, Honeywell and General Electric should begin to allow for the development of a universal standards based development of controls for viewing, managing and integrating all sorts of these devices with a network.

1-Wire's suitability for this process should be examined. Further investigation into integration between with types of network technologies would also be an interesting exercise, combination with wireless, Bluetooth sensors (available on many common household devices like phones) would allow for an interesting tie in for smart home technologies. Possibly trying to find other media to apply this protocol too might allow for expansion to other types of media, like wireless. The necessity of using electrical signal with the pull transistor might make that difficult.

There is more than one way to connect to a 1-Wire device, it strikes me as feasible to create simple embedded DS80C400 microcontrollers onto a standard electronic interface, like a PCI card, that should allow simple, direct connection with between a PC and the 1-Wire network and the Ethernet. This might allow for the development of a "device sensing server" that interfaces directly with Ethernet.

Some work in the open source community has developed some work towards directly connecting to 1-wire weather sensors in a Linux environment, searching for the OWW project on sourceforge, will yield results.

An interesting project would be to further stress test the limits of reliability of 1-wire network design, closing examining the impact of weight and radius on the performance of the network. It would also be interested in examination of ways to increase the maximum supportable weight of network.

Another area of research might be to examine possible protocol enhancements, application of protocol outside 1-wire technologies.

Accommodations for using one wire to modify settings on devices depending on the data being collected, research into possible X-10 tie in to allow power on/off of regular electrical devices based on the data acquired from the 1-Wire network.

Conclusions

This project was an investigation into embedded networking and sensing using 1-Wire network technologies. In the course of this project I was able to ascertain a number of things;

The DS80C400 is a very stable, functional platform. It is inexpensive, and provides an easy to operate interface between serial, Ethernet and 1-wire networks. The TINI O/S is an excellent operating system that is easy to use, and flexible, with support for java built applications, and built in FTP, telnet, and serial servers. Incorporation of this embedded platform into custom built microcomputers should be easy, affordable and flexible.

1-Wire networks utilize a very interesting protocol for remote data sensing using 1-Wire and ibutton devices. This protocol is stable, robust, and the use of parasitic power in 1-wire devices makes it an ideal candidate for doing remote sensing. To try to pull AC and or DC power to every device on a large sensor network is not feasible, but 1-Wire makes this simple, cheap an effective.

Embedded development in java proved very difficult for me in the beginning, but I have to confess that as I come to slowly understand it better, I am liking the use of it more all the time, and now that I have gotten over the initial challenges of how to compile, build and run the code, I would not have too much hesitation to embark on using the java platform in this environment again. That being said, further investigations into using C as a development language for embedded platforms would be a useful undertaking.

My goal in this project was to get a solid understanding of 1-Wire technologies and embedded controllers, and querying 1-Wire devices to gather information in a practical environment.

References

Dallas Semiconductor Maxim. "TINI_GUIDE," Rev 0.7/04 Retrieved December 2005 from: http://www.maxim-ic.com/products/TINI/pdfs/TINI_GUIDE.pdf

Stevens, Richard W., Bill Fenner, Andrew Rudoff, (2004) Unix Network Programming, vol 1, 3rd Edition, Addison Wesley..

Waite, Mitchell, Stephen Prata, (1994) New C Primer Plus, Sams Publishing,.

Loomis, Don, (2001) TINI Specification and Developers Guide, Addison Wesley, Toronto.

Axelson, Jan, (2003) Embedded Ethernet and Internet Complete, Lakeview Research,.

Sobell, Mark, (2005) A Practical Guide to Redhat Linux, 2nd Ed, Prentice Hall,.

Bitson, Tim, (2006) Weather Toys, Wiley Publishing Inc.,.

Arnold, Ken and james Gosling, (1998) The Java Programming Language Second Edition, Addison Wesley,.

Reilly, Michael and David Reilly, (2002) Java Network Programming and Distributed Computing, Addison-Wesley,.

Horstmann, Cay and Gary Cornell, (2001) Core Java vol1- Fundamentals, Sun Microsystems Press,.

Liang, Daniel, (2005) Introduction to Java Programming, Pearson Prentice Hall.

Maxim/Dallas Semiconductors (2005)Guidelines for Reliable 1-Wire Networks.

Appendix A – ListOW.java for 1-Wire addressing discovery

```
/*---------------------------------------------------------------------------
 * Copyright (C) 1999,2000 Dallas Semiconductor Corporation, All Rights Reserved.
 *
 * Permission is hereby granted, free of charge, to any person obtaining a
 * copy of this software and associated documentation files (the "Software"),
 * to deal in the Software without restriction, including without limitation
 * the rights to use, copy, modify, merge, publish, distribute, sublicense,
 * and/or sell copies of the Software, and to permit persons to whom the
 * Software is furnished to do so, subject to the following conditions:
 *
 * The above copyright notice and this permission notice shall be included
 * in all copies or substantial portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS
 * OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
 * MERCHANTABILITY,  FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
 * IN NO EVENT SHALL DALLAS SEMICONDUCTOR BE LIABLE FOR ANY CLAIM, DAMAGES
 * OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE,
 * ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR
 * OTHER DEALINGS IN THE SOFTWARE.
 *
 * Except as contained in this notice, the name of Dallas Semiconductor
 * shall not be used except as stated in the Dallas Semiconductor
 * Branding Policy.
 *---------------------------------------------------------------------------
 */

import com.dalsemi.onewire.OneWireAccessProvider;
import com.dalsemi.onewire.adapter.DSPortAdapter;
import com.dalsemi.onewire.container.OneWireContainer;
import java.util.Enumeration;


/**
 * Minimal demo to list device found on default 1-Wire port
 * @version    0.00, 28 August 2000
 * @author     DS
 */
public class ListOW
{

   /**
    * Method main
    *
    *
    * @param args
    *
    */
   public static void main (String args [])
   {
      OneWireContainer owd;
```

```
      try
      {

         // get the default adapter
         DSPortAdapter adapter = OneWireAccessProvider.getDefaultAdapter();

         System.out.println();
         System.out.println("Adapter: " + adapter.getAdapterName()
                            + " Port: " + adapter.getPortName());
         System.out.println();

         // get exclusive use of adapter
         adapter.beginExclusive(true);

         // clear any previous search restrictions
         adapter.setSearchAllDevices();
         adapter.targetAllFamilies();
         adapter.setSpeed(adapter.SPEED_REGULAR);

         // enumerate through all the 1-Wire devices found
         for (Enumeration owd_enum = adapter.getAllDeviceContainers();
                 owd_enum.hasMoreElements(); )
         {
            owd = ( OneWireContainer ) owd_enum.nextElement();

            System.out.println(owd.getAddressAsString());
         }

         // end exclusive use of adapter
         adapter.endExclusive();

         // free port used by adapter
         adapter.freePort();
      }
      catch (Exception e)
      {
         System.out.println(e);
      }

      return;
   }
}
```

Appendix B - Build.bat - Batch file compiling and uploading to TINI

```
@echo off
rem usage; from command line; "build 1.2.3.4"
rem where 1.2.3.4 represents the IP address of the TINI - or FTP server
rem this script will copy the current *.java and finished *.TINI files to a separate
rem location for backup, compile the current java files into associated class
rem files, and then use the TINIconvertor to create  a single runnable
rem file (MintWX.TINI). This script will then redirect a series of commands into a
rem single file (TINI.cmd) which automates the connection to the TINI FTP server and
rem uploading of the *.TINI file.

rem ***************************************************************
rem I am heavily indebted to "chenot" - a moderator in the TINI forum located at;
rem http://discuss.dalsemi.com/index.php?showforum=5
rem for suggesting this batch filing approach, it saved years of time and frustration
rem in building, compiling and testing the code. I had never seen the technique for
rem redirection to automate FTP connection before. Generally I used this forum when I
rem ran into trouble and it was often very helpful.
rem***************************************************************

echo TINI Build Script
echo FTP Address =  %1

echo "backing up source files"
del c:\aab\*.* /Q
copy C:\amint\Source\src\*.* c:\aab\*.*
copy C:\amint\Source\bin\*.* c:\aab\*.*

del bin\*.* /Q

echo Compiling...
javac -target 1.1 -source 1.2 -bootclasspath ..\..\TINI1.17\bin\TINIclasses.jar -
classpath ..\..\TINI1.17\bin\owapi_dependencies_TINI.jar -d bin src\*.java

echo Building....
java -classpath ..\..\TINI1.17\bin\TINI.jar BuildDependency -x
..\..\TINI1.17\bin\owapi_dep.txt -p ..\..\TINI1.17\bin\owapi_dependencies_TINI.jar -f
bin -o bin\MintWX.TINI -d ..\..\TINI1.17\bin\TINI.db -add
OneWireContainer10,OneWireContainer12,OneWireContainer1D,OneWireContainer20,OneWireCo
ntainer26

echo Attempting FTP connection with "%1"
echo root>> TINI.cmd
echo TINI>> TINI.cmd
echo cd bin>> TINI.cmd
echo put .\bin\MintWX.TINI>> TINI.cmd
echo bye>> TINI.cmd
ftp -s:TINI.cmd "%1"

del TINI.cmd
del bin\*.class

echo done
```

Appendix C - List of 1-Wire devices and capabilities

Device Name	Family	Description	Interfaces	MemoryBanks
DS1990A DS2401	01	1-Wire Address only		
DS1991 DS1425	02	Secure memory device		
DS1994 DS2404	04	4K NVRAM memory and clock, timer, alarms	ClockContainer	MemoryBank PagedMemoryBank
DS2405	05	Single addressable switch		
DS1993	06	4K NVRAM memory		MemoryBank PagedMemoryBank
DS1992	08	1K NVRAM memory		MemoryBank PagedMemoryBank
DS1982 DS2502	09	1K EPROM memory		MemoryBank PagedMemoryBankOTPMemoryBank
DS1995	0A	16K NVRAM memory		MemoryBank PagedMemoryBank
DS1985 DS2505	0B	16K EPROM memory		MemoryBank PagedMemoryBank OTPMemoryBank
DS1996	0C	64K NVRAM memory		MemoryBank PagedMemoryBank
DS1986 DS2506	0F	64K EPROM memory		MemoryBank PagedMemoryBankOTPMMemoryBank
DS1920 DS1820 DS18S20	10	Temperature and alarm trips	Temperature-Container	
DS2406 DS2407	12	1K EPROM memory, dual switch	SwitchContainer	MemoryBank PagedMemoryBankOTPMemoryBank
DS1983 DS2503	13	4K EPROM memory		MemoryBank PagedMemoryBank OTPMMemoryBank
DS1971	14	256bit EEPROM memory and OTP register		MemoryBank PagedMemoryBankOTPMMemoryBank

DS1954	16	Java Powered Cryptographic iButton		
DS1963S	18	4K NVRAM memory and SHA-1 engine		MemoryBank PagedMemoryBank
DS1963L	1A	4K NVRAM memory with write cycle counters		MemoryBank PagedMemoryBank
DS2423	1D	4K NVRAM memory with external counters		MemoryBank PagedMemoryBank
DS2409	1F	dual switch, coupler	SwitchContainer	
DS2450	20	quad A/D	ADContainer	MemoryBank PagedMemoryBank
DS1921	21	Thermochron temperature logger	Temperature-Container ClockContainer	MemoryBank PagedMemoryBank
DS1822	22	Econo temperature	Temperature-Container	
DS1973	23	4K EEPROM memory		MemoryBank PagedMemoryBank
DS1904 DS2415	24	Real-time-clock	ClockContainer	
DS2438	26	Temperature, A/D	ClockContainer ADContainer Temperature-Container Humidity-Container	
DS2417	27	Real-time-clock with interrupt	ClockContainer	
DS18B20	28	Adjustable resolution temperature	Temperature-Container	
DS2890	2C	single channel digital pot	Potentiometer-Container	
DS2760	30	Temperature, current, A/D	ADContainer Temperature-Container	
DS1961S DS2432	33	1K EEPROM memory with SHA-1 engine		MemoryBank PagedMemoryBank

Appendix D – MintWx.java

```
/*****************************************************************************

  File name:    MintWx.java

 MintWx is a program that serves weather data collected using an
 AAGelectronica 1-wire weather station. Weather data is collected once
 per minute. After every 5 collections (5 minutes) the data is averaged and
 forwarded via a stream socket connection to a web server, weather the data is
 put together and published.

 This file is the contains the main() method. And queries all other class files
 as required.

 *****************************************************************************/

//includes
import java.net.*;
import java.io.*;
import java.util.*;
import com.dalsemi.system.*;

//class must be declared in a java file of the same name
public class MintWx
{
          // class variables and constants
  private static Prefs prefs;
  private static final String versionStr = "MintWx v.4";
  protected weather weather;
  protected Logs todaysLog, yesterdaysLog = null;
  protected static boolean debugFlag = false;
  protected static Object  lock;
  private static boolean firstTime = true;
  private static long startTime;


  public MintWx() throws IOException
  {
    // get the start time of this session
    startTime = d.getTime();
    Date d = new Date();
   }

// need to declare a few more things for the main program loop: Start the server,
// get the weather and send data to webserver via socket created in OWSocket.java

  public void mainLoop()
  {
    boolean resetFlag = true;
    Date    date;
    int     minute, hour;
    OWSocket wg = new OWSocket();

    try
    {

      // create a new time setter class,
```

```
    // then loops forever, getting weather data
    // will later compare Minute=lastMinute – will activate when condition not met
    TiniTime ts = new TiniTime(prefs.timeServerUrl, prefs.timeOffset, debugFlag);
    int lastMinute = -99;
    int lastHour = -99;
    date = new Date();


    // start of primary loop – loop forever
    while(true)
    {
      // sleep for 1 second - specified milliseconds
      Thread.sleep(1000);


      // check current time – update minute and hour variables
      date.setTime(System.currentTimeMillis());
      minute = date.getMinutes();
      hour = date.getHours();


      // only loop once a minute – at end of loop lastMinute is updated
      if (minute != lastMinute)
      {
        // gc function is the TINI "garbage collector" – will free up memory
        java.lang.System.gc();


        // *** here's the call that actually gets the weather data ***
        weather.getWeather();

        //weather collected each minute but
        // if time = 5 minute, gather weather data calculate, and forward it
        // to server via socket (5446)
        if ((minute % 5) == 0)
        {
          // calculate weather data from weather class
          weather.crunchWeather(index, resetFlag);

          // send data via OWSocket class to server
          if (prefs.wuEnabled)
            wg.send(date, weather);

          // now clear avgs and reset flag for new set
          weather.resetAvg();
          resetFlag = false;
        }

       // update the time to complete loop
        lastMinute = minute;
        lastHour = hour;

        // clean up the any mess we left behind
        java.lang.System.gc();
      }
    }
  }
```

```
    catch(Throwable t)
    {
      ErrorLogs.log("drive(): " + t);
    }
  }
  //Need to carefully keep track of time
  public static boolean setTimeZone()
  {
    try
    {
      // get the OS timezone
      String tz = com.dalsemi.system.TINIOS.getTimeZone().trim();

      // get if time zone is default
      if (!tz.equals("GMT"))
      {
        try
          {
            // now try to set the java timezone
            TimeZone zone = TimeZone.getTimeZone(tz);
            if (zone == null)
            {
              try
              {
                if (tz.charAt(0) != '-')
                  tz = "+" + tz;

                com.dalsemi.system.TINIOS.setTimeZone(tz);
                zone = TimeZone.getDefault();
              }
              catch(NumberFormatException nfe)
              {  }
            }

            if (zone!=null)
              TimeZone.setDefault(zone);
          }
       }
    }
    return true;
  }
//get preferences from prefs class and prefs.ini file
  public static Prefs getPrefs()
  {
    return prefs;
  }
  //is debugging turned on?
public static boolean getDebugMode()
  {
    return debugFlag;
  }
//****************MAIN() function*********************
//in java the convention is to have main() at end of file
  public static void main(String[] args)
  {
    System.out.println("Starting " + versionStr);
```

```
    if (args.length != 0)
    {
      // check if –d option was used on starting of command, if so
      // print all debug flags
      if (args[0].equals("-d"))
      {
        System.out.println("debug on");
        debugFlag = true;
      }
    }

        // set Tini's Timezone
    setTimeZone();
    System.out.println("MintWx Local Time = " + new Date());

    try
    {
      prefs = new Prefs();
      prefs.read(debugFlag);
      ErrorLogs.log(versionStr + " Started");

      MintWx weatherServer = new MintWx();
      weatherServer.weather   = new weather(debugFlag, prefs);
      weatherServer.mainLoop();
    }
    catch(Throwable t)
    {
      ErrorLogs.log("Exception: Main() " + t);
    }
    finally
    {
      ErrorLogs.log("MintWx Stopped");
      System.exit(1);
    }
  }
}
```

Appendix E – Onewire.java

```
/*****************************************************************************


 This file provides the primary interface to the 1-wire devices in the
 weather station. It utilizes a pref class to provide weather station
 specific data such as device serial number (ID) and calibration values.

 *****************************************************************************/
//imports
import com.dalsemi.onewire.*;
import com.dalsemi.onewire.adapter.*;
import com.dalsemi.onewire.container.*;

public class onewire
{

  //variable declaration
  private OneWireContainer10  tempDevice = null;
  private OneWireContainer1D  windSpdDevice = null;
  private OneWireContainer20  windDirDevice = null;
  private OneWireContainer1D  rainDevice = null;
  private OneWireContainer26  pressureDeviceAtoD = null;
  private TAI8570        pressureDeviceTAI = null;
  protected float        temperature;
  protected float        windSpeed;
  protected int          windDirection;
  protected float        pressure;
  protected float        rain;
  private long          lastCount = 0;
  private long          lastTicks = 0;
  private long          lastLCount = 0;
  private long          lastLTicks = 0;
  private int  tempErrors;
  private int  windSpdErrors;
  private int  windDirErrors;
  private int  pressureErrors;
  private int  rainErrors;
  private DSPortAdapter    adapter;
  private static boolean    debugFlag;
  private static Prefs      prefs;
  public static final long  TICKS_PER_SECOND = 1000L;
  public onewire(boolean debugFlag, Prefs prefs) throws OneWireException
  {

    // debug flag for print statements
    this.debugFlag = debugFlag;
    this.prefs = prefs;

    // get an instance of the TINI external adapter – must have this to communicate
    // on 1-wire network
    adapter = OneWireAccessProvider.getDefaultAdapter();
    if (adapter != null)
    {
      if (debugFlag)
        System.out.println("Found Adapter: " + adapter.getAdapterName());
```

```
}
else
{
  ErrorLogs.log("Error: Unable to find adapter!");
  throw new OneWireException("1-Wire Adapter Not Found");
}
// using the prefs data, we can now get instances of each weather sensor
//we need to determine devices before we can query them

// ***temperature device***
// routine to check for device sensitivity from Dallas Semiconductor
if(prefs.tempDeviceAvailable)
{
  //check to make sure we have ID and can communicate
  tempDevice = new OneWireContainer10(adapter, prefs.tempDeviceID);
  if (tempDevice == null)
    ErrorLogs.log("No DS1820 Temperature Sensor found - Disabling Device");

  else
  {
    // does this temp sensor have greater than .5 deg resolution?
    try
    {
      if (tempDevice.hasSelectableTemperatureResolution())
      {
        // set resolution to max
        byte[] state = tempDevice.readDevice();
        tempDevice.setTemperatureResolution(tempDevice.RESOLUTION_MAXIMUM,
        state);
        tempDevice.writeDevice(state);
        ErrorLogs.log("Temp Device Supports High Resolution");
      }
    }
    catch (OneWireException e)
    {
      ErrorLogs.log("Error Setting Resolution: " + e);
    }
  }
}
else
  System.out.println("Invalid or No Temp Device in prefs.ini");


// ***check for wind speed device
if (prefs.windSpdDeviceAvailable)
{
  windSpdDevice = new OneWireContainer1D(adapter, prefs.windSpdDeviceID);
  if (windSpdDevice == null)
    ErrorLogs.log("No DS2423 Wind Counter found - Disabling Device");
}
else
  System.out.println("Invalid or No Wind Speed Device in prefs.ini");

// check for wind direction device
if (prefs.windDirDeviceAvailable)
{
  windDirDevice = new OneWireContainer20(adapter, prefs.windDirDeviceID);
```

```
      if (windDirDevice == null)
        ErrorLogs.log("No DS2450 Wind A to D Device found - Disabling Device");
    }
    else
      System.out.println("Invalid or No Wind Direction Device in prefs.ini");

    // check for rain device
    if (prefs.rainDeviceAvailable)
    {
      rainDevice = new OneWireContainer1D(adapter, prefs.rainDeviceID);
      if (rainDevice == null)
        ErrorLogs.log("No DS2423 Rain Counter found - Disabling Device");
    }
    else
      System.out.println("Invalid or No Rain Counter Device in prefs.ini");

    // check for pressure device
    if (prefs.pressureDeviceIsTAI8570)
    {
      // make a new TAI8570
      pressureDeviceTAI = new TAI8570(adapter, prefs.pressureDeviceID1,
prefs.pressureDeviceID2, debugFlag);
      // load the calibration values
      if (!pressureDeviceTAI.readCalibration())
      {
        ErrorLogs.log("Unable to Read Cal Values from TAI8570 - Disabling Device");
        pressureDeviceTAI = null;
      }
    }
    else
      System.out.println("Invalid or No Pressure Device in prefs.ini");


    // reset the 1-Wire bus
    resetBus();
  }

      try
      {
        // ****read temperature*****
        if (debugFlag)
          System.out.println("Temperature: " + tempDevice.getName() + " " +
tempDevice.getAddressAsString());

            //this is a straightforward call – we have determined in device ID
            //is correct, now we can read device by readDevice() call
        byte[] state = tempDevice.readDevice();
        tempDevice.doTemperatureConvert(state);
        state = tempDevice.readDevice();
        temperature = (float)tempDevice.getTemperature(state);

        // convert to degs F
        if (prefs.tempF == true)
          temperature = temperature * 9.0f/5.0f + 32f;

        if (debugFlag)
```

```
          System.out.println("Temperature = " + temperature + " degs" +
prefs.tempUnits + "\n");
      }
    }
    return temperature;
  }


  public float GetWindSpeed() throws OneWireException
  {
    if (windSpdDevice != null && windSpdErrors < prefs.deviceErrorsBeforeDisable)
    {
      // is device there?
      if (!windSpdDevice.isPresent())
      {
        ErrorLogs.log("Wind Speed Counter Not Present");
        windSpdErrors++;

        // have we exceeded the number of allowed errors?
        if (windSpdErrors >=  prefs.deviceErrorsBeforeDisable)
        {
          ErrorLogs.log("Windspeed Disabled");
          windSpeed = 0.0f;
        }
      }
      else
      {
        // read current wind count & time and compare to last count & time
        if (debugFlag)
          System.out.println("Wind Speed: " + windSpdDevice.getName() + " " +
windSpdDevice.getAddressAsString());

        long currentCount = windSpdDevice.readCounter(15);
        long currentTicks = System.currentTimeMillis();

        if (lastTicks != 0)
        {
          // calculate the wind speed based on the revolutions per second
          if (prefs.speedMph)
            windSpeed = ((currentCount-lastCount)/((currentTicks-lastTicks)/1000f)) /
2.0f * 2.453f;  // MPH
          else
            windSpeed = ((currentCount-lastCount)/((currentTicks-lastTicks)/1000f)) /
2.0f * 3.862f;  // KPH
        }

        if (debugFlag)
          System.out.println("Count = " + (currentCount-lastCount) + " during " +
(currentTicks-lastTicks) + "ms calcs to " + windSpeed + prefs.speedUnits);

        lastCount = currentCount;
        lastTicks = currentTicks;

      }
    }

    // on startup, wind might be negative, so cap it at 0
```

```
    if (windSpeed < 0)
      windSpeed = 0.0f;

    return windSpeed;
  }

  public int GetWindDirection() throws OneWireException
  {
    if (windDirDevice != null && windDirErrors < prefs.deviceErrorsBeforeDisable)
    {
      // is device there?
      if (!windDirDevice.isPresent())
      {
        ErrorLogs.log("Wind Direction A to D Not Present");
        windDirErrors++;

        // have we exceeded the number of allowed errors?
        if (windDirErrors >= prefs.deviceErrorsBeforeDisable)
        {
          ErrorLogs.log("Wind Direction Disabled");
          windDirection = 16;
        }

      }
      else
      {
        // setup the wind AtoD to read all 4 channels 8 in bit mode with 5.12 volts
full scale
        if (debugFlag)
          System.out.println("Wind Direction: " + windDirDevice.getName() + " " +
windDirDevice.getAddressAsString());

        byte[] state = windDirDevice.readDevice();

        windDirDevice.setADResolution(OneWireContainer20.CHANNELA, 8, state);
        windDirDevice.setADResolution(OneWireContainer20.CHANNELB, 8, state);
        windDirDevice.setADResolution(OneWireContainer20.CHANNELC, 8, state);
        windDirDevice.setADResolution(OneWireContainer20.CHANNELD, 8, state);

        windDirDevice.setADRange(OneWireContainer20.CHANNELA, 5.12, state);
        windDirDevice.setADRange(OneWireContainer20.CHANNELB, 5.12, state);
        windDirDevice.setADRange(OneWireContainer20.CHANNELC, 5.12, state);
        windDirDevice.setADRange(OneWireContainer20.CHANNELD, 5.12, state);
        windDirDevice.writeDevice(state);

        windDirDevice.doADConvert(OneWireContainer20.CHANNELA, state);
        windDirDevice.doADConvert(OneWireContainer20.CHANNELB, state);
        windDirDevice.doADConvert(OneWireContainer20.CHANNELC, state);
        windDirDevice.doADConvert(OneWireContainer20.CHANNELD, state);

        float chAVoltage =
(float)windDirDevice.getADVoltage(OneWireContainer20.CHANNELA, state);
        float chBVoltage =
(float)windDirDevice.getADVoltage(OneWireContainer20.CHANNELB, state);
        float chCVoltage =
(float)windDirDevice.getADVoltage(OneWireContainer20.CHANNELC, state);
```

```
        float chDVoltage =
(float)windDirDevice.getADVoltage(OneWireContainer20.CHANNELD, state);

        windDirection = GetWindDir(chAVoltage, chBVoltage, chCVoltage, chDVoltage);

        if (debugFlag)
        {
          System.out.println("Wind Dir AtoD Ch A = " + chAVoltage);
          System.out.println("Wind Dir AtoD Ch B = " + chBVoltage);
          System.out.println("Wind Dir AtoD Ch C = " + chCVoltage);
          System.out.println("Wind Dir AtoD Ch D = " + chDVoltage);
          System.out.println("Wind Direction     = " + windDirection + "\n");
        }

        if (windDirection == 16)
        {
          ErrorLogs.log("Wind Direction Error: " +
                  "Wind A2D Ch A = " + chAVoltage +
                  "Wind A2D Ch B = " + chBVoltage +
                  "Wind A2D Ch C = " + chCVoltage +
                  "Wind A2D Ch D = " + chDVoltage);
        }
      }
    }

    return windDirection;
  }

  public float GetRain() throws OneWireException
  {

    if (rainDevice != null && rainErrors < prefs.deviceErrorsBeforeDisable)
    {
      // is device there?
      if (!rainDevice.isPresent())
      {
        ErrorLogs.log("Rain Counter Not Present");
        rainErrors++;

        // have we exceeded the number of allowed errors?
        if (rainErrors >= prefs.deviceErrorsBeforeDisable)
        {
          ErrorLogs.log("Rain Counter Disabled");
          rain = 0.0f;
        }
      }
      else
      {
        // read rain count from counter 15
        if (debugFlag)
          System.out.println(rainDevice.getName() + " " +
rainDevice.getAddressAsString());

        rain = (rainDevice.readCounter(15)/100F);

        if (debugFlag)
          System.out.println("Rain Counter: " + rain + " inches");
```

```
        // convert to centimeters if required
        if (!prefs.rainInches)
          rain *= 2.54f;

        // subtract rain offset value from prefs
        rain = rain - prefs.rainZero;

        if (debugFlag)
        {
          System.out.println("Rain Offset : " + prefs.rainZero);
          System.out.println("Rain YTD    : " + rain + " \n");
        }
      }
    }

    return rain;
  }

  public float GetPressure() throws OneWireException
  {
     if (prefs.pressureDeviceIsTAI8570)
    {
      if (pressureDeviceTAI != null && pressureErrors <
prefs.deviceErrorsBeforeDisable)
      {
        // is device there?
        if (!pressureDeviceTAI.isPresent())
        {
          ErrorLogs.log("TAI Pressure Sensor Not Present");
          pressureErrors++;

          // have we exceeded the number of allowed errors?
          if (pressureErrors >= prefs.deviceErrorsBeforeDisable)
          {
            ErrorLogs.log("Pressure Sensor Disabled");
            pressure = 0.0f;
          }
        }
        else
        {
          // read the pressure and scale results
          if (pressureDeviceTAI.readPressure())
          {
            // get the pressure
            if (prefs.baroInHg)
              pressure = pressureDeviceTAI.getPressureInHg();
            else
              pressure = pressureDeviceTAI.getPressureMb();

            if (debugFlag)
            {
              System.out.println("TAI9870 Temp       = " +
pressureDeviceTAI.getTempF());
              System.out.println("Station Pressure   = " + pressure +
prefs.baroUnits);
              System.out.println("Gain               = " + prefs.baroSlope);
```

```
            System.out.println("Offset             = " + prefs.baroIntercept);
          }

          // apply calibration
          pressure = pressure * prefs.baroSlope + prefs.baroIntercept;

          if (debugFlag)
            System.out.println("Corrected Pressure = " + pressure + "\n");
        }
        else
          pressureErrors++;
      }
    }
  }

  return pressure;
}

 static final float lookupTable[][] = { {4.5F, 4.5F, 2.5F, 4.5F}, // N    0
{4.5F, 2.5F, 2.5F, 4.5F}, // NNE   1
{4.5F, 2.5F, 4.5F, 4.5F}, // NE    2
{2.5F, 2.5F, 4.5F, 4.5F}, // ENE   3
{2.5F, 4.5F, 4.5F, 4.5F}, // E     4
{2.5F, 4.5F, 4.5F, 0.0F}, // ESE   5
{4.5F, 4.5F, 4.5F, 0.0F}, // SE    6
{4.5F, 4.5F, 0.0F, 0.0F}, // SSE   7
{4.5F, 4.5F, 0.0F, 4.5F}, // S     8
{4.5F, 0.0F, 0.0F, 4.5F}, // SSW   9
{4.5F, 0.0F, 4.5F, 4.5F}, // SW   10
{0.0F, 0.0F, 4.5F, 4.5F}, // WSW  11
{0.0F, 4.5F, 4.5F, 4.5F}, // W    12
{0.0F, 4.5F, 4.5F, 2.5F}, // WNW  13
{4.5F, 4.5F, 4.5F, 2.5F}, // NW   14
{4.5F, 4.5F, 2.5F, 2.5F}, // NNW  15
};


 /* convert wind direction A to D results to direction */
private int GetWindDir(float a, float b, float c, float d)
{
  int i;
  int direction = 16;


  for(i=0; i<16; i++)
  {
    if(((a <= lookupTable[i][0] +1.0) && (a >= lookupTable[i][0] -1.0)) &&
            ((b <= lookupTable[i][1] +1.0) && (b >= lookupTable[i][1] -1.0)) &&
            ((c <= lookupTable[i][2] +1.0) && (c >= lookupTable[i][2] -1.0)) &&
            ((d <= lookupTable[i][3] +1.0) && (d >= lookupTable[i][3] -1.0)) )
    {
      direction = i;
      break;
    }
  }
  return direction;
}
```

```
  /* convert direction integer into compass direction string */
  public String GetWindDirectionString(int input)
  {
    String[]  direction = {" N ", "NNE", "NE ", "ENE",
    " E ", "ESE", "SE ", "SSE",
    " S ", "SSW", "SW ", "WSW",
    " W ", "WNW", "NW ", "NNW",
    " ERR"};
    // valid inputs 0 thru 16
    if (input < 0 || input >= 16)
      input = 16;
    else
      input = (input + prefs.northAdjust) % 16;

    if (debugFlag)
      System.out.println("Wind Direction Decoded = " + input + " = " +
direction[input]);

    return direction[input];
  }

  // reset error count for devices
  public void ResetDeviceErrors()
  {
    tempErrors       = 0;
    windSpdErrors    = 0;
    windDirErrors    = 0;
    pressureErrors    = 0;
    rainErrors      = 0;

    if (debugFlag)
      System.out.println("Device Error Counter Reset");
  }

  // reset 1-wire bus
  public void resetBus()
  {

    if (debugFlag)
      System.out.println("Resetting 1-wire bus");

    try
    {
      int result = adapter.reset();

      if (result == 0)
        ErrorLogs.log("Warning: Reset indicates no Device Present");
      if (result == 3)
        ErrorLogs.log("Warning: Reset indicates 1-Wire bus is shorted");
    }
    catch (OneWireException e)
    {
      ErrorLogs.log("Exception Reseting the bus: " + e);
    }
  }
}
```

FOOTNOTES

[1] http://www.aagelectronica.com/aag/index.html
[2] http://www.maxim-ic.com/products/microcontrollers/TINI/
[3] Getting Started With TINI, Dallas Semiconductor/MAXIM Wireless, 2004.
[4] See “build.bat” in the Appendix for further particulars regarding the batch file.
[5] Securing Debian Manual, Debian, 2006, http://www.debian.org/doc/manuals/securing-debian-howto/
[6] The TINI Specification and Developers Guide, Don Loomis, Dallas Semiconductor Corporation, 2001.
[7] The TINI Specification and Developers Guide, Don Loomis, Dallas Semiconductor Corporation, 2001
[8] http://www.maxim-ic.com/appnotes.cfm/appnote_number/148 - Guidelines for reliable 1-Wire Networks
[9] http://pdfserv.maxim-ic.com/en/an/AN155.pdf
[10] See list in Appendix for more comprehensive list
[11] ListOW.java from TINI 1.17 examples folder – code available in appendix
[12] http://owfs.sourceforge.net/simple_commands.html
[13] The TINI Specification and Developers Guide, Don Loomis, Dallas Semiconductor Corporation, 2001
[14] http://java.sun.com/j2se/1.4.2/download.html
[15] http://files.dalsemi.com/TINI/index.html