A Remote, Versatile Interface Between the Physical World and a Network Connected Controller

By

Michael T. DePlonty

B.S. Electrical Engineering
University of Michigan, 1998

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of

Master of Science in Electrical Engineering and Computer Science

at the
Massachusetts Institute Of Technology
February 2001

Signature of Author: .... [illegible]
Department of Electrical Engineering and Computer Science
January 19, 2001

Certified by: .... [illegible]
Chathan M. Cooke
Principle Research Engineer, Lecturer
Department of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by: .... [illegible]
Arthur C. Smith
Chairman, Committee on Graduate Students
Department of Electrical Engineering and Computer Science

A Remote, Versatile Interface Between the Physical World and a Network Connected Controller

By

Michael T. DePlonty

Submitted to the Department of Electrical Engineering and Computer Science
on January 19, 2001 in Partial Fulfillment of the
Requirements for the Degree of Master of Science in
Electrical Engineering and Computer Science

Abstract

Computerized devices that collect data and control processes are not new. Many such devices exist, often with the computer performing both the collecting and controlling functions. Examples occur in a wide range of applications and include adaptive controllers, hydraulic network controllers and ion accelerator controllers.

However, these devices often are application specific, require hands-on, local operation, and combine data collection with primary control. A generalized device, when properly designed, offers the opportunity to make implementation easier, simplify remote operation to extend the system's capabilities, and isolate historical data collection from primary control.

This thesis develops a process controlling / data collection device and calls it a concentrator. A concentrator is a general device that can be customized for specific applications, is intended to operate remotely, and separates data collection from primary control. In general, a concentrator interfaces the physical world with a network connected controller. To formalize this concept, this thesis defines a concentrator's properties, determines specific ideal features, and then tests a specific example.

The concentrator concept was tested with an oil dielectric strength test developed at the MIT High Voltage Research Laboratory. To physically implement a concentrator, custom software was developed, modeling it after the concentrator's ideal features. This software was then loaded onto a microprocessor, which in turn runs the oil dielectric strength test. Specifically, the microprocessor controls the oil tester, reads and processes the tester's results, then posts the data to a web server. Results indicate that the software adequately implements a concentrator, however long-term reliability and responding to a primary controller need further development.

Thesis Supervisor: Chathan M. Cooke
Title: Principle Research Engineer, Lecturer
Department of Electrical Engineering and Computer Science

Acknowledgments

First I would like to thank my advisor, Dr. Chathan Cooke, for giving me a chance and allowing me to achieve my degree. Thanks for giving me the freedom to succeed and fail; for being there to reign me in when needed; and for giving me direction and support.

Financial support for my thesis came from two main sources. First, Entergy Service, Inc. provided financial support for this research. Thank you for making my research possible and for supporting other forms of advanced work. Second, MIT provided a teacher's assistantship and a fellowship that allowed me to attend school and find this research project. Thank you for making it financially feasible for me to be here.

A special thanks goes to fellow lab members Tim Cargol and Will Johnson. As mentioned in the thesis, my device has limited value without a functional system. Tim and Will helped create this system and offered programming advice, computer help, and other random tips that aided my project.

Finally, many friends supported, prodded, and questioned me as I completed this degree. Mere words cannot express the gratitude I have and how much I feel I owe you for your friendship. So, thanks to:

- My parents John and Teri DePlonty
- Professor Alan Willsky
- Patrick Maurer
- Professor Paul Gray
- Francis Carr
- Andy Wang
- Carol Frederick

Contents

List of Figures

List of Tables

Chapter 1
Introduction

1.1 Motivation

Computerized devices that collect data and control processes are not new. Many such devices exist, often with a computer performing both the collecting and controlling functions. Wilkinson and Rees use this approach in their adaptive controller [**1**]. Jager, et al., also use this approach in their rule-based controller [**2**].

However, performing both data collection and process control on the same computer is not always best. Consider data intensive applications. Here, collecting and processing data may hog processor time, effectively eliminating all control capability. Or consider a complex process. Simply breaking up this complex process by having a separate computer monitor each part may provide the best solution. But this too creates a problem. A single computer would not have access to all the data, thus rendering it, or a human, incapable of making an informed decision.

Masmoudi and Vansteenkiste encountered this problem simulating a complex hydraulic network. To do this, they divided the process into smaller parts, but incorporated these divisions into a multi-level control process. This process uses "a higher level coordinator [that] has a global view of the system" and a local controller that interfaces controlled components with the high-level controller [**3**]. Their solution successfully separates data collection from primary process control by allowing the higher level coordinator to make informed decisions.

Bauman, et al., developed a similar system to control an electrostatic ion accelerator. They called the local controller a concentrator and they called the higher level coordinator a user interface application (UIA). The UIA is a graphical user interface and is completely human controlled. The concentrator can collect data, control a local process, respond to UIA commands, and share data with other concentrators [**4**].

Combine these devices and one has a remote, computerized device on a network that separates data collection from primary control. With this versatility, many more applications that once were unreachable by a computer can motivate further developments. For example, consider the application that drove this research.

Power companies need a way to remotely test oil dielectric strength. Oil is an important component in an electric power transformer's tap changer as it electrically and thermally insulates the tap changer's components. If the oil goes bad, the tap changer will fail and force an expensive maintenance and a disruption in power. So to prevent this, a power company routinely sends maintenance people out to test the oil.

This on site oil test involves retrieving an oil sample from the transformer. However remote and inaccessible transformers make this test very inconvenient so the oil does not get tested often [5]. To reduce this inconvenience, Cargol explored a test, originally developed by Cooke and Hagman, that does not require drawing an oil sample [5, 6]. But to make this truly useful, such a test must be conducted remotely and display the results without requiring a technician's presence at the transformer.

Of the three devices previously encountered, the concentrator seems to work best. It is can be remote, which is good for the remote transformers, and separates data collection from primary control, which is good for the reasons discussed above. The concentrator can control the external devices necessary for conducting Cargol's test and can be controlled by humans through the UIA. But though the concentrator possesses these qualities, it is still designed specifically for controlling an electrostatic ion accelerator. As a standard structure does not exist, this concentrator must be redesigned to fit the oil dielectric strength test.

Clearly there is a need for a well-defined device that is remote and separates data collection from primary control. An early attempt by McCabe [7] began the concept of an observer – concentrator pair, in which acquisition and some control are divided between two machines. This thesis develops the front-end device of such a pair and calls it a concentrator, borrowing Bauman et al.'s device name. It is a remote, versatile interface between the physical world and a network connected controller. It is also an important cog in the observer - concentrator system [7]. But before delving into the concentrator and the observer - concentrator system, consider other approaches to the problem of process monitoring and data acquisition.

1.2 Past Approaches to Process Monitoring and Data Acquisition

Many applications require a process controlling / data collection system. Figure 1.1 shows how a historical system takes physical world measurements. Such a system converts the measurements to a desirable format, manipulates the conversions, and acts on the results. This action could involve making control decisions, displaying the data, saving the data, or doing any combination of these three.

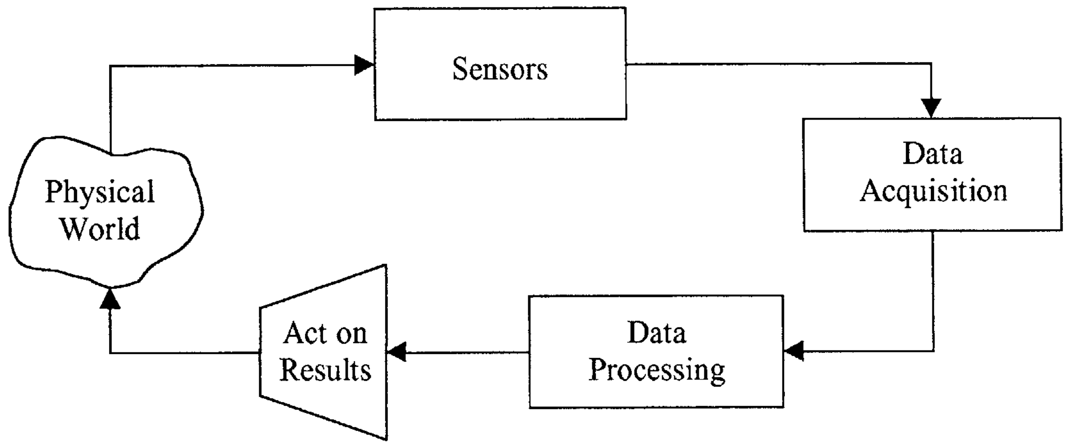


Figure 1.1
Historical Control/Collection System

For example, consider the classical feedback control loop shown in Figure 1.2. A little thought shows that this system is basically the same as Figure 1.1, only arranged and labeled differently. The control processor in Figure 1.2 performs both the data acquisition and data processing functions while the controllers act on the results.

Or consider the classical data collection system shown in Figure 1.3. Even though this system has no feedback to the physical world, Figure 1.1 still represents this system. The “data storage or display” box now performs the acquisition, processing (if any), and actions. Here, actions may include saving data or graphing data on a computer monitor.

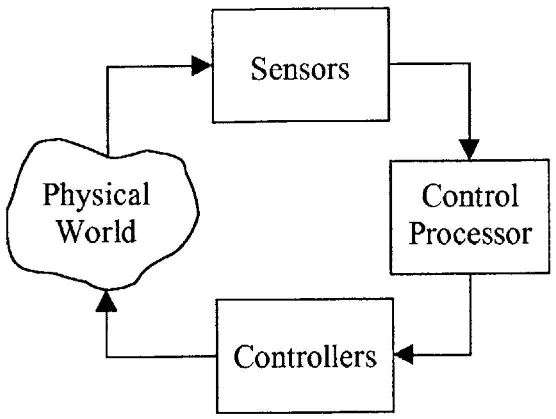


Figure 1.2
Classical Feedback Control

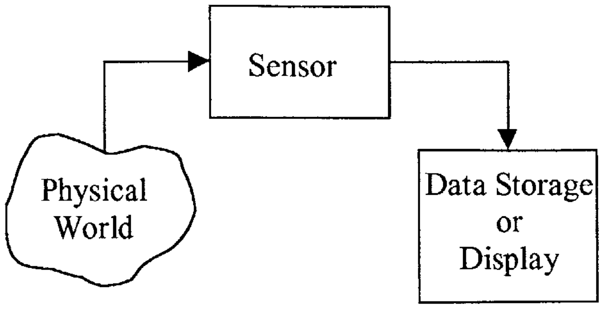


Figure 1.3
Classical Data Collection

Now before the development of computers, humans had to perform everything in Figure 1.1 except creating the measurements (actual devices such as voltmeters or pressure gauges performed this part). This human system was, understandably, error prone. Complex instruments, wrong inputs, incorrect recording, and improper data presentation were some error sources [8].

So with the development of computers came the development of computerized controlling / collection systems. Ideally these systems reduce human error by performing most, if not all of the repetitious tasks. So rather than a human implementing Figure 1.1, computer controlled hardware acquires data, the computer's software processes data, then software and/or a human acts on the results.

1.2.1 Implementation Schemes

There are basically two different computerized controlling / collection systems: plug-in cards and standalone systems. Plug-in cards, also referred to as data acquisition boards, place the A/D converter on a card connected directly to the computer's motherboard [9]. In such a system, the measuring devices' output must be connected directly to the computer and all interfacing between the analog and digital domains is implemented inside the computer. For example, various bus protocols developed to facilitate communication between the computer's microprocessor and the data acquisition board include STD (IEEE 961), STE (IEEE P1000), VME bus (IEEE1014), Futurebus (IEEE896.1) [**10**].

While the plug-in card system has the A/D converter inside the computer, the standalone system places the A/D converter outside the computer. Now the physical processes' output, after going through the A/D converter, can be connected to any computer. Thus the name "standalone system." This approach is particularly useful for processes that are remote or otherwise inaccessible, such as monitoring and controlling a missile's flight path [**11**].

Once an analog signal in a standalone system has been measured and digitized, it still must reach the computer. To do this, standalone systems transmit data over a communication channel using the channel's associated standards. Examples include radio and other wireless protocols; EIA serial bus standards such as RS-232-E, RS-423, RS-422, and RS-485; GPIB (General Purpose Interface Bus) standards such as IEEE-488.1, IEEE-488.2, and SCPI (Standard Commands for Programmable Instrumentation); VXI bus; optical Ethernet; and finally TCP/IP [**4, 10, 12**].

Once the data reaches the computer, it must be processed. There are three different data processing approaches. First, use software that comes with the specific devices such as a plug-in card. Second, develop custom software for a specific lab or company. Third, select software from a rich source of commercial programs such as LabView, LabWindows, or WorkBenchMac [**8, 9, 10**].

Finally, there are numerous ways this system can act on the data. Programs such as Labview or Microsoft Excel could graph it. Custom software could be programmed to make decisions and operate external controllers, which in turn control the physical process. Or a human could print the data, store the printout in a binder, then walk over and control the physical process manually. Taylor presents additional examples, ranging from reading the data to a file to fitting an equation to the measured data [**10**]. The possibilities are endless.

1.2.2 Local Control Limitations

Yet, a system modeled after Figure 1.1 and using these implementation schemes does have limitations. For example, suppose a remote device wishes to perform data acquisition and make control decisions without having to report to a master controller. The classical system discussed above does not allow this "local control."

Now before designing a new system, one should stop and think. Should the data acquisition stage even be allowed to control the physical process? Or as discussed above, should all of the control be placed in the primary controller. Well the answer, of course, depends on the application and available hardware. Consider three separate systems that illustrate the local control's range.

First, an example from personal experience and illustrates zero local control. The goal is to measure the engine temperature over an extended period of time. Thermocouples attached to the engine block output voltages that are measured by voltmeters, which are then connected to a computer via GPIB. A human then sits at the computer, tells the computer to retrieve the voltages, and if the results are undesirable, the human walks over and adjusts the engine. The voltmeters, which both measure and acquire data, understandably have no control over the engine's temperature.

The second system is the missile example mentioned above and illustrates a moderate level of local control. Here, the onboard data acquisition module controls the missile flight path through preprogrammed values. These values locally control the missile but are static, thus why this system illustrates a moderate level of control. Real-time control is reserved for the data processing module that is human controlled back at the control center [**11, 12**].

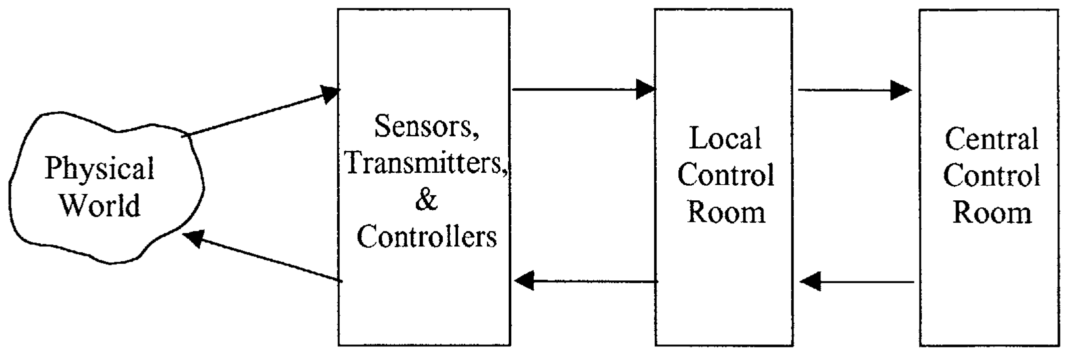


Figure 1.4
Borer's Three Level Distributed Control System

Third, Borer's distributed control system illustrates a high level of local control. This approach was discussed in section 1.1 and is shown in Figure 1.4. Layer one consists of devices that measure data and transmit it to the local control room. Layer two is the local control room and consists of the data gathering systems that gather, process, and transfer data to the central control room. Additionally, this layer regulates external processes in real time, thus the high level of local control. Layer three is the central control room, consisting of additional data processing systems and computers with a graphical user interface [13].

In summary, a process controlling / data collection system is straightforward. Humans have been implementing it for some time. Introducing computers into this system reduces some errors but creates other problems, such as designing and implementing a data acquisition module or deciding how to process data. One issue not addressed by classical approaches is local control. Borer presents one solution. Bauman et al., mentioned in section 1.1, presents another solution. The next section will introduce a third approach called an observer-concentrator system.

1.3 Observer - Concentrator System

The observer-concentrator system is a new method of combining data collection and physical process control into one system. Note, this section refrains from fully developing this system. Rather it introduces the observer-concentrator system so that the concentrator can later be discussed in the proper context.

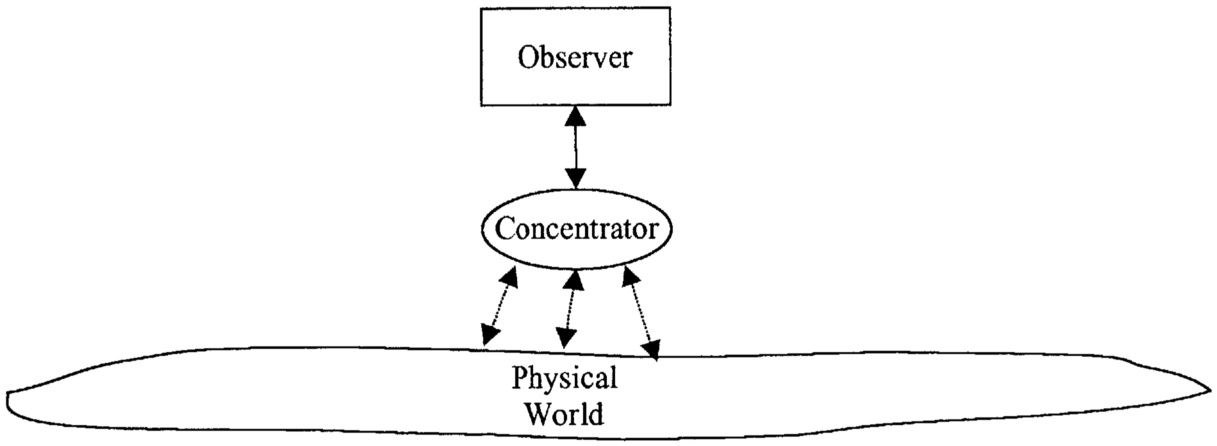


Figure 1.5
Basic Observer - Concentrator Relationship

Figure 1.5 shows the basic observer-concentrator relationship. To begin, an observer is a network connected controller that receives data from a concentrator and transmits commands to a specific concentrator. An observer can be an independent, intelligent machine making its own decisions or a standard user interface relying on human decisions. In Borer's approach, his central controller has some attributes similar to an observer.

Some important observer properties include:

1. Independence from other observers
2. No direct access to raw data
3. Active and passive interaction with concentrators
4. Only receives error messages or data from concentrators
5. Knowledge of each concentrator's function
6. Can control concentrator actions [7]

As mentioned and shown in Figure 1.5, a concentrator interfaces an observer with the physical world. Basically, a concentrator can control and/or collect data from multiple events in the physical world, put this data into desired units, then report the results to the observer. As a data collector, a concentrator "... is an independent entity that captures, stores, processes and forwards data to a host computer" [14]. In this system the host computer is the observer. As a controller, a concentrator can make its own decisions and control physical processes locally, wait for observer commands and serve as a remote controller, or do both knowing that any observer decisions overrule its own decisions.

Now for this system to work an observer must have at least one concentrator and each concentrator must have an associated observer. As a single observer - concentrator pair will not always be sufficient to solve a particular problem, an observer can control multiple concentrators. The transformer monitoring application mentioned in section 1.1 illustrates this problem. Remember, one concentrator just cannot physically be at multiple transformers.

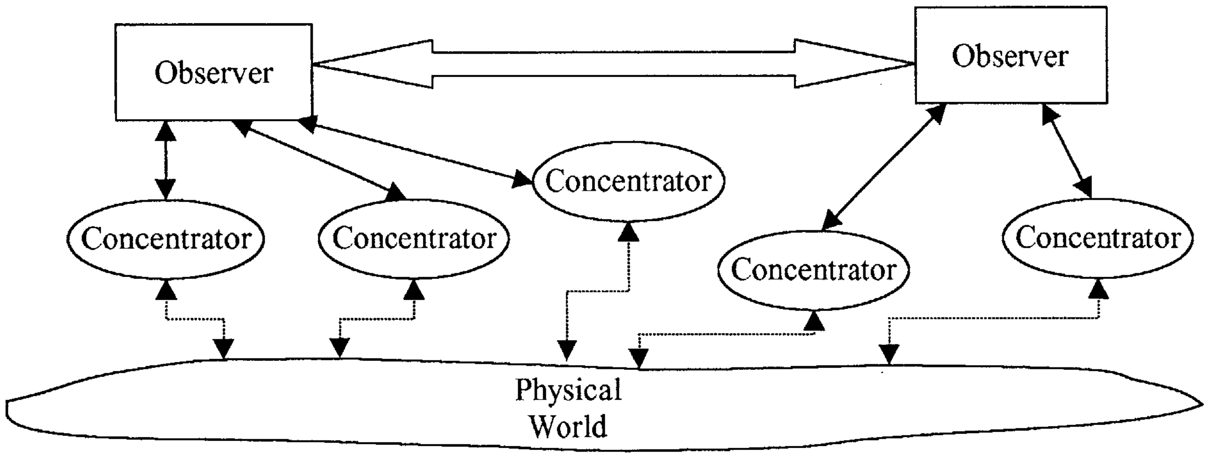


Figure 1.6
General Observer - Concentrator Network

Or suppose the physical world process is so complicated that multiple observers are necessary. And to properly control this process as discussed above, observers must share information. Therefore observers and concentrators must have networking capability. Figure 1.6 shows a general observer-concentrator network. Notice that observers can share data and communicate directly between other observers while concentrators cannot directly link to other concentrators. This is done for two reasons. One, it makes defining a concentrator clear. And more importantly two, it keeps the concentrators isolated and thus independent from each other. It is still possible for one concentrator's data to "influence" another, but this data must go through at least one observer first.

Also notice that while an observer can control multiple concentrators, multiple observers do not control one concentrator. This too makes defining a concentrator clear. Now a concentrator does not have to prioritize commands from separate observers. It also does not have to resolve conflicting commands or decide which observer should receive data. A concentrator simply knows one and only one observer exists.

This requirement also helps define an observer. If multiple observers monitor a complex application, each observer can easily monitor a specific subset of the application. Each observer would be specified for its subset, employing the necessary number of concentrators. Since an observer can control only the concentrators in its subset, the unique division is maintained. The observers can thus interact at a higher level and not be burdened by unrelated concentrator information.

In summary, an observer - concentrator system is a new approach to combining data collection and external process control. The observer acts as the master controller and, if needed, user-interface. The concentrator interfaces the observer with measurable events in the physical world. If needed, multiple observers and concentrators can be networked together to monitor and control complex processes or collect data in data intensive applications. Overall, this section introduced the observer-concentrator system only to reference further discussion about the concentrator. It omitted much of the details as exploring and formalizing the observer could constitute another thesis. On the other hand, the concentrator will be further defined in Chapter two.

1.4 Thesis Focus

This thesis develops a remote process controlling / data collection device and calls it a concentrator. The goal is to establish a versatile device that can be customized for specific applications. In this spirit, this thesis will not develop new error handling routines, require a certain network protocol for the device, or provide other specific implementation requirements. Rather, this thesis will establish a model concentrator by defining its properties, determining specific ideal features, and then testing an example.

1.5 Thesis Overview

Chapter one motivates the need for a concentrator and presents various approaches to designing a process controlling / data collection system. Chapter two establishes concentrator properties, laying out the requirements a device must fulfill to be considered a concentrator. Chapter three presents specific features for an ideal concentrator. Chapter four presents an implemented concentrator and discusses the test results. Finally, chapter five touches upon a few future improvements not addressed in this thesis.

Chapter 2

Desired Concentrator Properties

2.1 Introduction

As seen in Chapter one, there is a need for a link between the physical world and an observer that desires to quantify physical world features. A possible device to help perform this link is the concentrator. This concentrator obtains quantified information from the physical world and passes that information on to a network accessible observer. Figure 2.1 shows a typical concentrator located between the physical world and an external network. On the physical world side, a concentrator monitors and in some cases controls a physical process. A process could be a microprocessor's temperature variation or a refrigerator's changing contents. It may be complex or simple. In general, a process is an event, or series of events, producing data that a human or computer wishes to see.

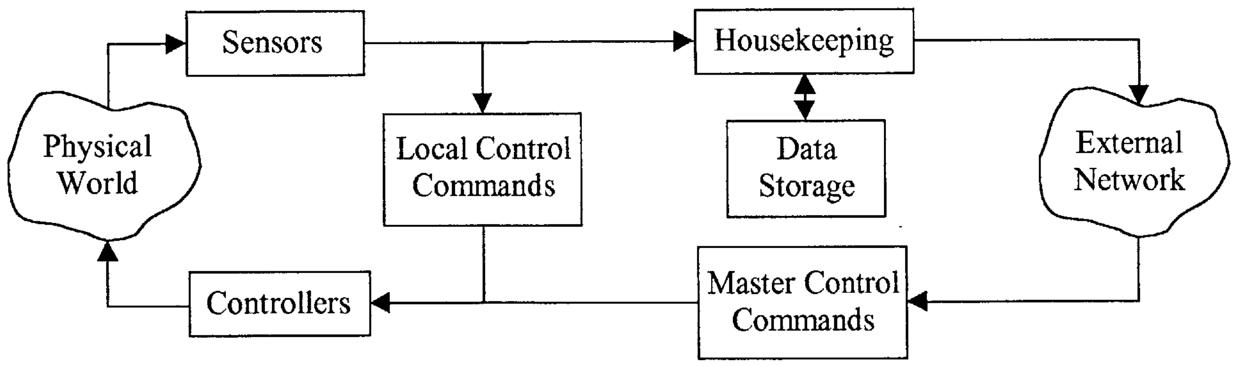


Figure 2.1
Typical Concentrator, Physical World and External Network

To effectively monitor a process, a concentrator must have data collection capability and control capability. Data is produced by the process and may take the form of discrete points or continuous signals. Similarly, control signals result from observer instructions or local decisions and may take the form of digital instructions or continuous signals. So to monitor a process, the concentrator collects data from the process, makes local decisions, and then issues control instructions. Or it collects data, processes and reports it to the observer, receives the observer's instructions, then issues control instructions. Or it collects data, processes and reports it to the observer, then collects more data without controlling the process. The combinations are too numerous to include.

On the network side, a concentrator sends data to the observer and receives data from the observer. Transmitted data are the processed points that were read from the physical world. Received data are instructions, software, and parameters. This way the observer receives the desired information and controls the concentrator.

Now this elucidates a concentrator's functions but only alludes to its properties. Specifically, a concentrator's properties can be broken up into three general areas: global properties, input / output requirements, and security constraints. Global covers all of the concentrator's important requirements and limitations, including its relationship with the observer. Input / output discusses just that, how a concentrator deals with input to and output from itself. Finally, security explores a fairly new issue due to the distributed nature of networks.

2.2 Global Properties

There are four important global properties. First, one observer via a general network connection controls a concentrator. As discussed in Section 1.3, only one observer controls a concentrator. If multiple observers could control a single concentrator, then the concentrator would need an established protocol that handles command priority, conflict resolution, command authorship, etc. Rather than address this complexity, and to make a concentrator more versatile and modular, this limitation is imposed.

To be controlled by an observer, a concentrator must have the following abilities:

1. Uniquely identify itself so that the observer can recognize it
2. Report information to the observer
3. Receive information from the observer

A little thought shows that requiring a unique identity makes sense. As also discussed in section 1.3, an observer may control multiple concentrators. Each concentrator monitors different processes or different parts of one process, thus each concentrator performs unique functions. Since an observer needs to keep track of each concentrator, this unique identity is necessary.

Reporting information and receiving information are best described as input / output requirements. They are mentioned here to complete the "controlled by observer" requirement but will be elaborated in section 2.2.

Second, a concentrator is remotely accessible and remotely configurable. This requirement follows directly from the problem stated in Chapter one. As discussed, a remote concentrator extends the observer – concentrator system and allows this system to cover a wider range of applications.

Accessing a remote concentrator is done through the observer via the general network connection. By design, this is the only way to access the concentrator short of physically going to it. This way only the observer sends instructions to the concentrator, ensuring the observer knows what the concentrator should be doing.

To be remotely configurable, a concentrator needs to easily receive and implement new parameters and software. Ideally the concentrator's software uses parameter files. These parameter files should be easy to change, thus allowing a user, or the observer, to easily tweak the concentrator's actions.

Additionally, a concentrator needs to accept and update new software so that it maintains its flexibility and ensures it is a remote device. As new software is developed, there must be an easy way to update the concentrator. Physically accessing the concentrator is undesirable as this may be time consuming. But more importantly, this also contradicts the "remote device" requirement. Consequently a concentrator must be able to remotely accept and update new software.

Third, a concentrator can simultaneously run separate programs. Each program can share data and is responsible for a specific physical process. This allows a single concentrator to collect data from multiple physical processes and form one data point. It also allows a concentrator to control a complex process by breaking it up into manageable chunks.

Finally fourth, a concentrator needs an internal, real-time clock. This allows time-date stamping of the raw data and enables code scheduling. Basically, the concentrator will know when to run a specific routine and when a specific data point was taken.

2.3 Input / Output Requirements

Since a concentrator can accept input and output from both the observer and the physical world, this section is divided into two main areas. First the input / output relationship between a concentrator and the physical world will be discussed. Then the input / output relationship between a concentrator and the observer will be discussed.

2.3.1 Physical World Input / Output

To handle input from the physical world, a concentrator must perform three actions. First, it needs to put a time-date stamp on the raw data. As discussed in section 2.1, this stamp uniquely specifies the data point's capture time. Furthermore, it, along with other stamps, uniquely identifies the data point at the observer.

Second, a concentrator needs to process the raw data by converting them to conventional values. These values are application dependent and should be specified by the observer. SI values are one example of conventional values; slugs would be another example.

Now processing the raw data is necessary. To begin, sensors are the only way a concentrator can receive data from the physical world. Since the overall monitoring system is electrical in nature, the sensors must convert their readings into an electrical signal before transmitting them to the concentrator. The electrical signal may be meaningful for the sensor and the concentrator, but it is usually meaningless to the observer. As the observer is a high-level device, it needs to receive data relevant to the process, not the electrical signals relevant to the sensor. It also must be isolated from the raw data per the requirements mentioned in section 1.3. Consequently, the concentrator must convert the raw data into observer defined units.

Third, a concentrator needs to check the physical world data for errors and know how to correct errors once they are discovered. This too adds to the concentrator's versatility and semi-independence. For one, error checking can help the concentrator make control decisions. It can also help the concentrator make diagnostic checks on the sensors and controllers. And finally it can help the concentrator make diagnostic checks on itself in case the data processing function became corrupted.

Now a concentrator's output to the physical world are control commands. When a concentrator actually issues these commands stems from previous discussions and definitions. As mentioned, the concentrator can operate as the observer's remote controller or operate as a local controller. Therefore the concentrator will output control commands at two specific instances.

First, a concentrator will output commands in response to observer commands. By design, the observer has no direct connection to a physical process. If the observer wants to control the process, like any device it will issue control commands. Only these commands are issued to the concentrator instead of the physical world controllers. So by design and definition, the concentrator must be able to accept and act upon observer commands.

Second, a concentrator will output commands in response to data and serve as a local controller. As shown in section 1.2, this maintains the concentrator's versatility. But this requirement also makes sense. Suppose the observer does not want to make control decisions all the time, or the observer's communication link is broken and a process needs immediate control. Without some amount of local control, a remote process would be stuck, potentially never recover, and may produce catastrophic results. Therefore, a concentrator should be able to make local decisions. It is important to note, however, that the observer's commands will always override concentrator decisions.

2.3.2 Observer Input / Output

As one side of a concentrator communicates with the physical world, the other side communicates with the observer. And just like the physical world side, the observer side handles input and output. Input from the observer is comprised of two types.

The first input type a concentrator receives from the observer is new parameters and software. As discussed in section 2.1, this ensures that the concentrator remains a versatile, remote device. But not only must the concentrator receive new parameters and software, it must safely and seamlessly update the old parameters and software. Otherwise the concentrator's monitoring actions might fail or be interrupted, neither of which is desirable.

The second input type is control commands from the observer. All the discussion prior to this dealt with commands destined for the physical world. However, these general control commands also include specific concentrator commands. For example, the observer can instruct the concentrator to take a data point, retrieve and transmit a stored data point, or transmit identification information. And as previously discussed, the observer can send specific commands for the physical world controllers. So due to this variety of commands, a concentrator must be able to discern each command and then properly act upon it.

A concentrator's output to the observer includes error messages, data, and status information. Error messages and data are self-explanatory. Status information includes concentrator identification, processor status, and other such information. Together, these three output forms provide information that allows the observer to control the concentrator.

2.4 Security Constraints

The concentrator's network nature requires three security constraints. One, a concentrator needs a nonvolatile data storage buffer. To begin, all data the concentrator processes must eventually arrive at the observer. Suppose the concentrator is too busy to transmit data or the network connection to the observer is unavailable. The buffer allows the concentrator to transmit data at a convenient time, when the network connection has been re-established, or when any other problem has been resolved.

But not only must there be a buffer, the buffer must be nonvolatile. Suppose the concentrator loses all temporary memory, as could happen if it loses power or is forced to reboot. Data stored in the nonvolatile buffer will still exist and will still be available to be transmitted to the observer. Consequently the processed data will still arrive at the concentrator as desired.

Now it is important to point out that the nonvolatile buffer is a buffer. The concentrator does not store data once it has reached the observer; this is not its responsibility. Rather it is the observer's responsibility to store data, or pass it along to the proper entity, as it is the master controller.

Now two, a concentrator needs to be robust with self-recovery mechanisms. This constraint arises from the remote device requirement. Since the concentrator can be remote, it is important for it to rarely fail and to be able to recover if it does fail. Otherwise a human would have to go service it, which reduces the concentrator's usefulness as a remote device.

Finally three, a concentrator must use encrypted data to communicate with the observer. This allows the concentrator to be used for secure purposes, for proprietary purposes, or when anybody just doesn't want other people to know what is being sent over the observer – concentrator connection.

2.5 Conclusion

This chapter defined the concentrator to be a secure, remote device that interfaces a network connected controller with the physical world. Overall, the concentrator's properties can be divided into three categories: global properties, input / output requirements, and security issues. The global properties specify general concentrator requirements, the input / output requirements specify how the concentrator communicates with the physical world and the network connected controller, and the security constraints ensure that the concentrator remains safe and reliable. Table 2.1 summarizes all of the properties discussed in this chapter.

Table 2.1
Concentrator Properties

Global Properties
- Controlled by a single observer
- Unique identification
- Report information
- Receive information
- Remotely configurable
- Run simultaneous programs
- Possess real-time clock

Security Constraints
- Nonvolatile data buffer
- Robust with self-recovery mechanisms
- Communicate with encrypted data

Input / Output Requirements
- Input from physical world
    - Put time-date stamp on data
    - Convert raw data to conventional values
    - Perform error checking
- Output to physical world
    - Control commands in response to observer commands
    - Control commands in response to processed data

Input from observer
- New parameters
- New software
- Control commands

- Output to observer
    - Error Messages
    - Data
    - Status Information

Chapter 3
Ideal Concentrator

3.1 Introduction

As elaborated in Chapter two, a concentrator interfaces a network controller with the physical world. On the network side, the concentrator can receive or send information. Received information may be new software, new parameters, or new commands while sent information may be data, error messages, or requests. On the physical world side, the concentrator also can receive or send information. Here, received information may be digital or analog raw data while sent information may be digital or analog control instructions.

To implement this interface, a concentrator uses software modules. Now the module group, as a whole, must implement all of the properties discussed in Chapter two. When this is done, the concentrator becomes a computerized device and meets all the requirements motivated in Chapter one.

Ideally all the modules act simultaneously. This requires a separate processor for each module, but due to current technological and financial constraints, a concentrator is designed to be a single processor device. Therefore, one must use or develop a module coordination scheme that can be implemented on a single processor.

In addition to implementing the concentrator requirements, individual modules must validate received information. If the module encounters invalid information, it can either handle the problem itself or generate a fault. For example, suppose a module receives a command requiring unavailable data. Recognizing that this command is currently invalid, the module may choose to buffer the command until the data becomes available or it may generate a fault message, pass the message to the "Fault Handler" module, and let it take care of the problem.

Overall, the concentrator's modules are organized into the three groups shown in Figure 3.1. Physical world modules interface the concentrator with sensors and controllers, network modules interface the concentrator with the observer, and internal modules do everything else a concentrator should do. The remainder of this chapter explains each module group and describes each module's actions.

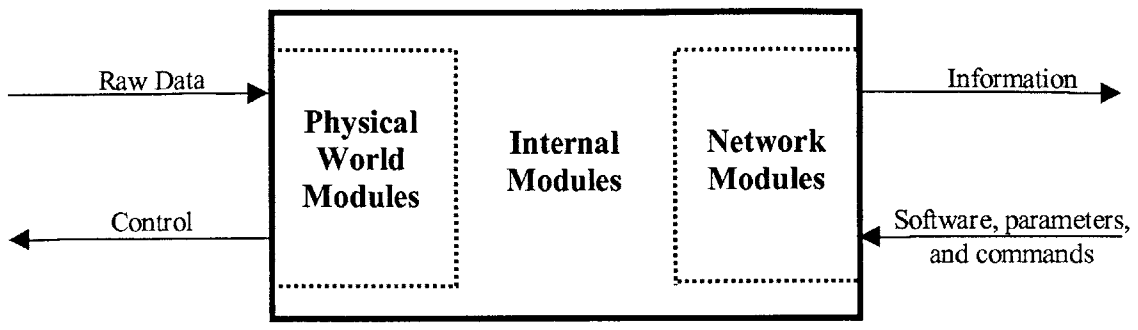


Figure 3.1
Ideal Concentrator

3.2 Physical World Modules

The physical world modules shown in Figure 3.2 interface sensors and controllers in the physical world with other concentrator modules. These modules handle physical world input, output, and error messages as well as the internal modules' input, output, and fault messages.

The internal modules' I/O is straightforward. Anything going to or coming from the internal modules is digital. Input to the internal modules consists of data and fault messages. Output from the internal modules consists of capture data commands and control commands. This will be elaborated further in the module sections below.

Physical world I/O is more complicated than the internal modules' I/O. First, input from the physical world is digital or analog raw sensor data. Since all data inside the concentrator is digital, analog input must be digitized. Consequently, this module group must contain an A/D converter and as discussed below, the "Capture Raw Data" module controls this digitization.

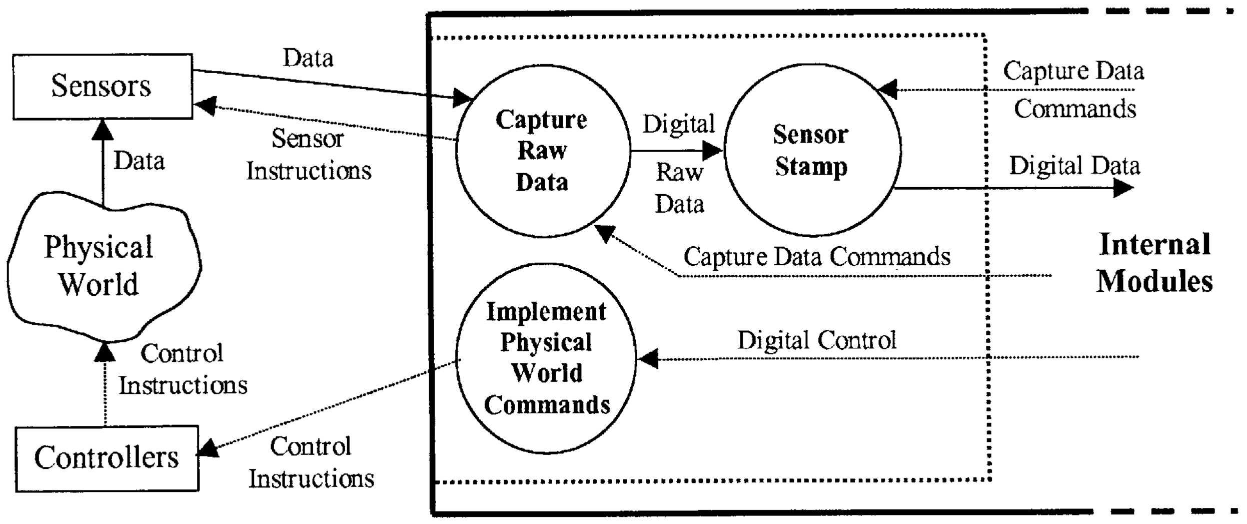


Figure 3.2
Physical World Modules

Output to the physical world is controller instructions or sensor instructions. While sensor instructions are digital, controller instructions may be analog or digital. And similar to analog input, desired analog instructions must be converted from digital commands. Therefore this module group may also contain a D/A converter which, as described below, is controlled by the "Implement Physical World Commands" module.

Error messages from the physical world fall into a special category. Technically these messages are input generated by the sensors and controllers, but because of three important characteristics, they are not classified as input. First, the concentrator does not actively seek error messages like it seeks raw data. Instead, sensors and controllers generate these messages and expect the concentrator to respond. Second, the concentrator only accepts digital error messages. Consequently error messages do not need to be digitized like some raw data points. And third, unlike raw data, all error messages do not go to the "Capture Raw Data" module. Instead, a particular error message returns to the instruction's origin. This means that sensors send error messages to the "Capture Raw Data" module and controllers send error messages to the "Implement Physical World Commands" module.

Figure 3.2 shows the three modules in this group. "Capture Raw Data" interacts with the sensors, "Implement Physical World Commands" interacts with the controllers, and "Sensor Stamp" labels data with sensor information.

3.2.1 Capture Raw Data

Overall, this module retrieves physical world data when instructed, digitizes it if necessary, and passes it to the "Sensor Stamp" module. This module's input may be sensor error messages, capture data commands, or raw data. Sensor error messages are digital messages generated by sensors and result from faulty commands, missing data, or general device failure. "Capture Raw Data" should detect and handle these messages.

Capture data commands arrive from the "Error Checker," the "Observer Message Processor," or the "Local Controller," all modules located in the internal modules group. Since data can come from many sensors, the capture command must include which sensor this module should read from. With this knowledge, "Capture Raw Data" converts the request into a sensor specific instruction and then sends this instruction to the desired sensor.

Finally, raw data arrive from the sensors. If the data is analog, it is immediately digitized and then, like digital raw data, sent to the "Sensor Stamp" module. Since this module knows which sensors it retrieves data from, it also knows whether or not the incoming raw data will be analog. This makes controlling the A/D converter easier. If it expects analog raw data, this module does not need to be told there is data waiting to be converted. Instead it only needs to wait for the data to arrive at the A/D converter's input.

In addition to accepting input, this module also outputs fault messages, sensor instructions, and digital raw data. Fault messages are generated when the module creates an error that it cannot recover from. Instead, it passes the fault message to the "Fault Handler."

Sensor instructions are digital messages in the sensor's required format. As a concentrator may use multiple sensors, these instructions allow data retrieval from specific sensors. If there are sensors that do not receive digital instructions, these "instructions" may be a multiplexer's channel select. There are numerous other examples, but the main point is that these instructions allow "Capture Raw Data" to control the sensor. And as for actually retrieving raw data, data is present at the input only when the concentrator issues a request.

Finally, digital raw data is the data gathered from the sensors. And as discussed above, it is immediately sent to the "Sensor Stamp" module.

3.2.2 Implement Physical World Commands

This module converts digital commands into controller specific instructions and then sends these instructions to the appropriate controller. Input consists of controller error messages and control commands. Controller error messages are similar to the sensor error messages as controllers generate these messages and expect this module to receive and act upon them.

Control commands are digital commands generated by the internal modules "Local Control" and "Observer Command Processor." Each command contains two fields: the desired response and the command destination. The desired response is the desired controller response, not the physical world response. Other modules, namely the "Local Control" and "Observer Command Processor" modules discussed in section 3.3, are responsible for converting physical world responses into controller responses. For example, suppose a concentrator controls an engine's throttle, which in turn controls the engine's rpm's. The observer issues a command "speed up," "Observer Command Processor" receives this command, converts it to a new command "open throttle." It then sends this new command to "Implement Physical World Commands" which converts this command into the proper format required by the throttle.

Now to properly convert and route these commands, the module must know the command's destination. Since a concentrator can utilize multiple controllers, the command destination informs this module which instruction format to use as well as which controller is to receive the instruction. Basically, the command destination ensures that the proper controller receives the proper instruction in the proper format.

The module's output is fault messages and controller instructions. Fault messages were covered in the chapter's introduction and are mentioned here for completeness. Controller instructions are controller specific commands that cause the controller to act. Instructions may be analog or digital, so those that must be analog must first pass through a D/A converter. Assuming the D/A converter does not have a buffer, this module should control the D/A converter to ensure that instructions are not lost or garbled while waiting for the conversion to finish.

3.2.3 Sensor Stamp

This module performs two tasks, stamp digital raw data with a sensor's ID and send the result to the proper internal module. Before data reaches any other module, it must be stamped with sensor specific information. This provides a unique match between a data point and a sensor. For example, suppose a concentrator receives data from two different temperature sensors. Both measure in degrees Kelvin and output voltage, however one result is erroneous. Or suppose one sensor measures in Centigrade while the other measures in Kelvin. Again both output voltage, but no physical world module converts the measurements into "standard units." The sensor stamp allows the concentrator to determine the error source in the first example, and in the second example, convert data into proper units.

To stamp data with the proper sensor ID, this module must receive the same capture data command that the "Capture Raw Data" module receives. This command, which originates from internal modules, contains the data's sensor origin. And since data is not buffered, received data will exactly correspond to the command's sensor information. Consequently, data points will be stamped with the correct sensor ID.

After stamping data, this module sends it to one of the following internal modules: "Local Control" or "Time-Date Stamp." Like embedded sensor information, the capture data command must contain the data's destination. And by construction, there is no lag between received commands and received data. Thus there is a one to one correspondence between data and commands, ensuring the data gets sent to the proper location.

3.3 Network Modules

Figure 3.3 shows the concentrator's four network modules: "Concentrator ID," "Listener," "Send Information," and "Fault Handler." Three modules facilitate communication with the observer while the fourth provides a recovery mechanism. Some transmit the internal modules' information to the observer while others transmit the observer's information to the internal modules. Each module is described below.

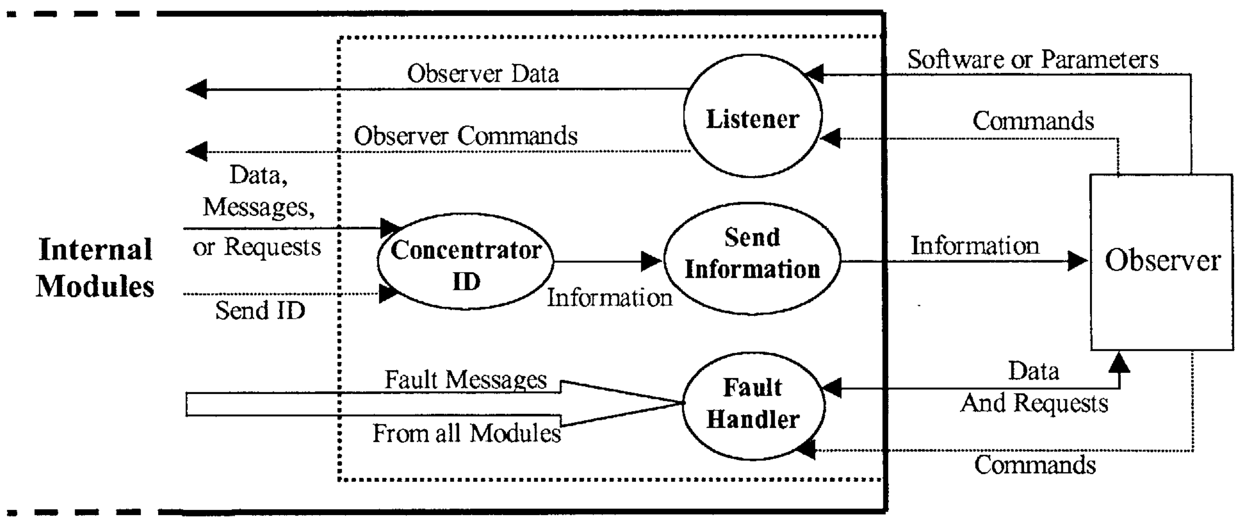


Figure 3.3
Network Modules

3.3.1 Concentrator ID

Overall, this module has two responsibilities, stamp outgoing information and describe the concentrator's responsibilities. To do this, "Concentrator ID" should handle input and produce output.

First, this module handles two types of input: information and commands. Information is anything the concentrator is sending to the observer: data, error messages, etc. Commands instruct this module to send the description mentioned above. Now input may arrive from multiple sources, at non-sequential times, with varying priorities. As mentioned in the beginning of the chapter, this scheduling and conflict resolution is ideally handled in the module. However, these duties may be relegated to a "controlling" module for ease of implementation.

Second, this module produces two types of output: stamped information and detailed descriptions. All information the concentrator sends to the observer must be stamped with the concentrator's ID. Since multiple concentrators may report to one observer, this stamp ensures a unique match between information and a concentrator.

Additionally, this module provides a detailed description of the concentrator's responsibilities. Such a description may mention the processes the concentrator controls or the data it reads, including a listing of sensors and controllers the concentrator uses. As a guide, the description should be detailed enough such that the observer can be self-sufficient. This way, a concentrator can be added to an existing system with minimal amount of human intervention.

3.3.2 Fault Handler

This module provides a recovery mechanism from general system faults. A fault is basically a flaw in a hardware or software component, and if left undetected, will eventually lead to a system failure. The fault may be a permanent, transient, or intermittent event that affects either a specific component or multiple components. Further, a fault may or may not be time invariant [15].

Now to satisfy the concentrator's robustness property, the software should be able to detect and reconfigure faulty components. In this context, reconfiguration means the software will eliminate the faulty entity from the system and restore the system to an operational condition [15]. An operational condition may imply the system is fully functional, especially if the fault is simple and can be easily corrected. But in general, this means the concentrator simply continues working with limited capabilities and will need further help, either human or observer, to restore full functionality.

To properly perform this reconfiguration, the software must perform four tasks [15].

1.	Fault Detection:	Recognize a fault occurred.
2.	Fault Location:	Determine where a fault occurred.
3.	Fault Containment:	Isolate the fault and prevent its effects from propagating through the system.
4.	Fault Recovery:	Restore system to an operational condition.

By design, each module performs the first three tasks. When a module detects a fault, it ceases operation (thus containing the fault) and passes a message to "Fault Handler." And since each module performs a unique task, "Fault Handler" knows exactly where the fault occurred. So to complete the reconfiguration, "Fault Handler" must perform task four.

For "Fault Handler" to perform fault recovery, it must receive fault messages from all the modules and communicate with the observer. Fault messages give this module necessary information that allow it to determine the correct action. Therefore, the fault message must specify whether the fault occurred in a hardware component or in a software module.

Hardware faults are relatively easy to detect as only these nine modules interact with hardware.

1. Capture Raw Data
2. Delete Data
3. Fault Handler
4. Implement Physical World Commands
5. Listener
6. Retrieve Data
7. Send Information
8. Store Data
9. Update

With "Capture Raw Data" and "Implement Physical World Commands," actual devices may produce an error message or simply not respond. With "Fault Handler," "Listener," "Send Information," and "Update," network connections may be noisy or unavailable. And with "Delete Data," "Retrieve Data," and "Store Data," memory may be inaccessible, illegally accessed, or full. However these modules are designed to detect these faults, thus making this task that much simpler. Now if the microprocessor fails, well, nothing is perfect.

Software faults, unlike hardware faults, are relatively difficult to detect and handle. Logic, design, and implementation errors can be subtle, escaping even the best human's detection. So it is tempting to assume the software is designed and implemented properly. Tempting, but this doesn't fully specify a robust system. Now according to Pradhan, there are three methods of handling faulty software.

1.	Robustness:	Software handles invalid inputs.
2.	Temporal Redundancy:	Software re-executes a program after encountering an error.
3.	Software Diversity:	Software detects all fault conditions and provides backup routines for critical routines [**15**].

The modules are designed to be robust, and any reasonable implementation incorporates diversity concepts. Therefore, "Fault Handler" simply performs temporal redundancy.

Now if "Fault Handler" cannot restore full functionality, it must be able to communicate with the observer. Assuming the network is available, "Fault Handler" will transmit the known faults and rely on an outside entity, be it observer or human, to fix the problem. It will then await instructions and data from the observer while "removing" the faulty module from the system. In this fashion, the concentrator will still maintain some functionality and hopefully regain full functionality once the observer, or human, responds.

For example, suppose the concentrator reads from only one sensor and has collected and stored twenty data points without sending them to the observer. But when it attempts to read the twenty-first data point, the sensor does not respond. "Fault Handler" cannot repair a defective sensor, so it informs the observer and prohibits the concentrator from using the "Capture Data" module. Since the concentrator cannot take any more readings, it transmits the twenty data points it did capture and then waits for further instructions. Now the observer cannot fix a defective sensor, so it notifies a technician. After repairing the sensor, the technician instructs the observer, which then instructs the concentrator, to resume capturing data. "Fault Handler" receives this instruction and enables the "Capture Data" module. The concentrator then regains full functionality and the problem has been solved.

In summary, "Fault Handler" allows the concentrator to recover from hardware and software faults. All of the concentrator modules can pass detailed fault messages to "Fault Handler," which then uses them to decide on an action. In most cases "Fault Handler" simply removes the faulty module, notifies the observer, and then waits for observer instructions. In some cases "Fault Handler" may simply re-execute the faulty module in case the fault was transient. If the fault persists, "Fault Handler" removes the faulty module, notifies the observer, and then waits for observer instructions.

3.3.3 Listener

This module handles observer generated communication requests. The only time the observer initiates communication with the concentrator is when it wants to send physical world commands, new software, new parameters, or concentrator commands. And since these have to be routed to the proper module, this module simply receives the observer's information, passes it to the "Observer Message Processor" for routing, and then closes the connection.

3.3.4 Send information

This module receives information from "Concentrator ID" and sends it to the observer. Since the observer may require a different format for each information type, this module must know what is to be sent and how to properly format it. For example, suppose the observer expects encoded data points but does not expect encoded requests. Then this module must know that if it receives a data point, it must be encoded before transmission but if the module receives a request, it is to be sent directly.

In addition to properly formatting information, this module must handle the actual communication protocols and transmission results. The communication protocols depend on the network and only affect this module. On the other hand, the transmission results affect both this module and the "Delete Data" module. As described in the "Delete Data" section, the concentrator clears the buffer once the observer receives the data. As the transmission result signifies whether or not the concentrator received the data, the "Send Information" module should make decisions based on this result. If the transmission succeeded, this module issues a delete data command. If the transmission failed, this module generates a fault and lets the "Fault Handler" determine the proper course of action.

3.4 Internal Modules

The internal modules perform a majority of the concentrator's functions. They manage the buffer, interpret observer commands, and perform local control and error checking routines. In general, they satisfy and implement all the I/O requirements listed in Table 2.1. These modules are illustrated in Figure 3.4 (notice the similarity to Figure 2.1) and explained below.

3.4.1 Delete Data

This module deletes data from the temporary data buffer and should be capable of deleting multiple data points or single data points. The only time data should be deleted from the concentrator's memory is when the observer confirms it received the data. Therefore the "Fault Handler" is the only module to issue a delete data command. And to ensure deletion of the correct data points, this command should include both the number of points and the specific points to delete.

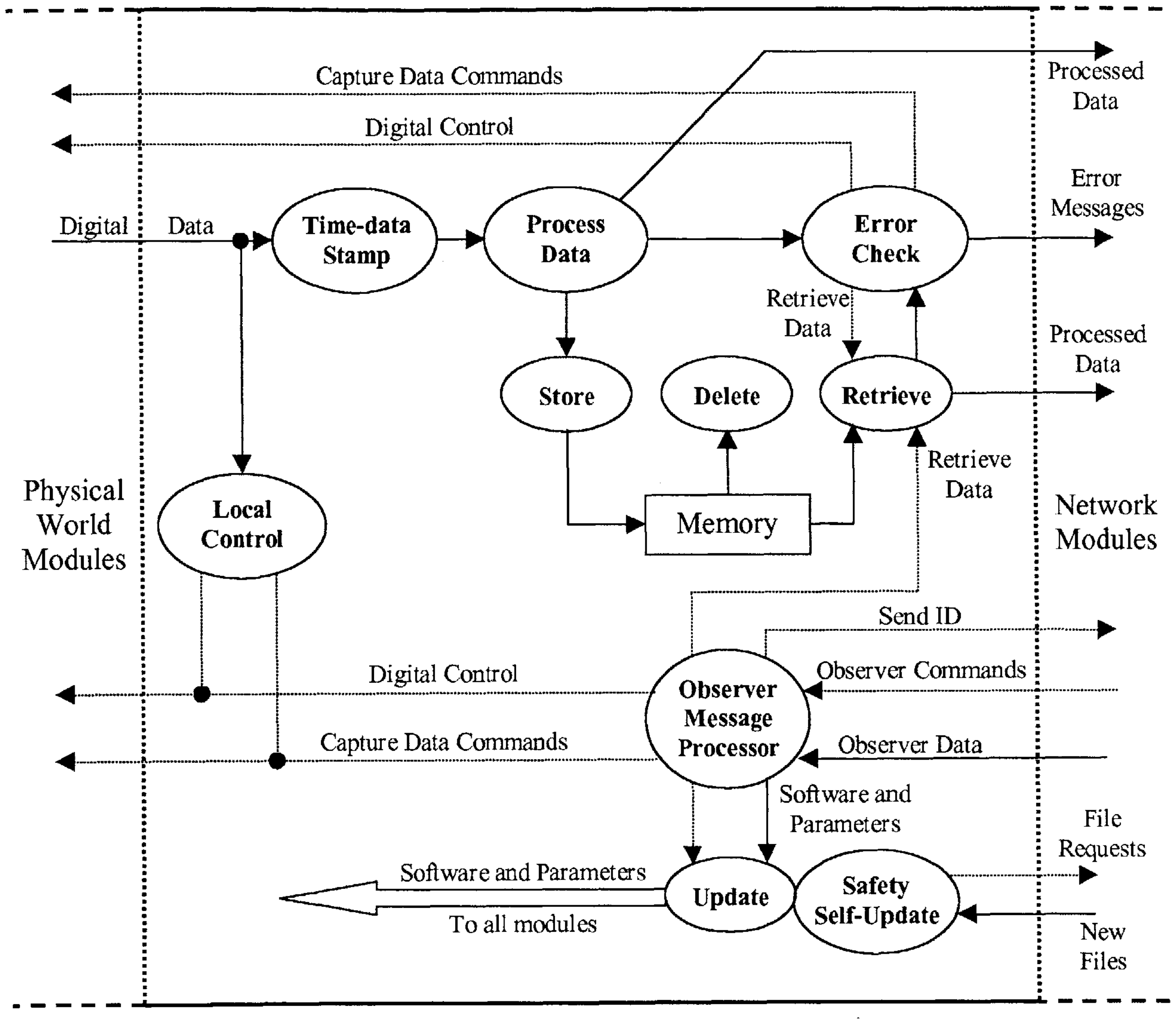


Figure 3.4
Internal Modules

3.4.2 Error Check

This module checks for data errors and data consistency. Now once a new data point arrives from the "Process Data" module, it is checked for consistency. To check for data consistency, this module uses a priori knowledge, which is gained from the sensor stamp, to compare the data point to its known limits. It also can use the sensor stamp as a redundancy check, making sure the raw data was converted into the proper units.

Checking statistical errors requires more than a single data point. Therefore, this module can issue a retrieve data command and receive data from the "Retrieve Data" module. With multiple data points, the desired error checking routine can be properly implemented.

If an error is discovered, this module may do three things. One, it may send a capture data command to the "Capture Raw Data" module, then use this new data point for further error checking. Two, this module may send a control command to the "Implement Physical World Commands" module, produce an expected response, then check if the result equals the expected response. And three, this module may send an error message to "Concentrator ID" and let the observer handle the problem.

3.4.3 Local Control

Overall, this module accepts raw data, performs control routines, then outputs digital control commands. Before performing any control routines, this module must receive data from the "Sensor Stamp" module. Now data may originate from multiple sensors, which in turn produce readings from multiple processes that require separate control routines. The sensor stamp allows this module to match sensor data with the appropriate control routine.

Next, this module performs the proper routine. The actual control routine depends on the application and is performed independent of the observer. As the concentrator may monitor multiple processes, so there are multiple control routines. Each routine, when completed, creates a control command for the associated controller (or a series of control commands for a group of associated controllers).

Finally, this module sends the derived control command to the "Implement Physical World Commands" module. "Local Control" does not need to know the command format the controllers expect. Rather, it must produce a command that exists in the "Implement Physical World Commands" command library. The generated command must also include the command's destination. As discussed in section 3.1.3, the command destination aids the "Implement Physical World Commands" module and ensures that the command reaches the proper controller.

3.4.4 Observer Message Processor

This module receives observer messages and routes them to the appropriate module. Input from the observer consists of data and commands. Received data is simply new software or new parameters and is destined for the "Update" module. Table 3.1 lists the allowed commands and their destined modules.

Table 3.1
Allowed Commands from Observer

Command	Module Destination
Capture data	"Capture Raw Data"
Physical world control	"Implement Physical World Commands"
Retrieve data	"Retrieve Data"
Send ID and/or detailed description	"Concentrator ID"
Update	"Update"

As this module simply routes the data or commands to the appropriate module, the output is exactly the same as the input. But if this module cannot route the input because it lacks information, it generates a fault explaining what it needs and then waits for the "Fault Handler" to provide the missing information.

3.4.5 Process Data

This module receives raw data, converts it to user defined "standard" units, then outputs the processed data. Raw data first arrives from the "Time-Date Stamp" module containing the following information: the sensor stamp, time-date stamp, the destination stamp, and the data value. Next, raw data is processed, using the sensor stamp to match data with the correct units. If a data point arrives with an unrecognizable sensor stamp, this module should generate a fault and pass it to the "Fault Handler." Then after completing the conversion, this module reads the destination stamp and sends the processed data to the "Error Check" module, the "Concentrator ID" module, or the "Store Data" module.

3.4.6 Retrieve Data

This module, when commanded, retrieves data from the data buffer and sends it to an appropriate module. A command may originate from "Error Check," "Observer Message Processor," or "Fault Handler" and should include three fields. One, the command should include the number of points to retrieve. Two, the command should include which point, or points, the module should receive. And three, the command should include the data's destination.

After retrieving data, this module utilizes the command's destination field to determine which module should receive data. Allowable destinations are the "Error Check" module and the "Concentrator ID" module.

3.4.7 Store Data

This module receives processed data from the "Process Data" module and immediately stores it in the nonvolatile data buffer. Since "Process Data" may be implemented with a temporary buffer and thus pass multiple data points, this module should be capable of saving both single and multiple data points.

3.4.8 Time-Date Stamp

This module receives data from the "Sensor Stamp" module, uses the real-time clock to stamp data with the current time and date, then sends the data to the "Process Data" module. Ideally, this stamping occurs simultaneously with capturing data. However this timing may be difficult, forcing the user to measure and account for this lag.

3.4.9 Update

This module updates a module's software and parameter files. Update commands arrive from "Observer Message Processor" and contain which module to update as well as what is being updated. This basically informs "Update" which file to rewrite. So an example command may be "Update Process Data's parameter file."

The new software and the new parameter files also originate from "Observer Message Processor" and are sent at the same time as the update command. If "Update" is replacing software, it communicates with the desired module and stops it at a convenient time. Once stopped, "Update" rewrites its software. If "Update" is replacing parameter files, it simply rewrites the file, relying on the module to check for the new parameters.

"Update" also contains two safety features. The first feature is the "Safety Self-Update" (SSU) submodule. SSU updates the "Update" module but exists in the concentrator's ROM so that it can never be remotely updated. This way, the concentrator cannot corrupt the "Update" module and prevent future remote updates.

The second safety feature is observer communication. In case the network modules are corrupted with an update, this module can notify the observer and receive new software. This too maintains a concentrator's remote configuration ability.

3.5 Conclusion

This chapter defined an ideal concentrator to consist of physical world modules, network modules, and internal modules. These groups implement the physical world – network interface required from Chapter two and satisfy all of the concentrator properties listed in Table 2.1. Table 3.2 summarizes all commands, data, and required formats encountered in this chapter.

Table 3.2
Internal Commands, Data, and their Required Format

Commands	**Format**
Digital Control Commands:	Desired controller, desired controller response
Capture Data Commands:	Desired sensor, data's module destination
Retrieve Data:	Number of points, which points, data's module destination
Send ID:	ID or description flag
Update:	Module to update, software or parameters flag
Data	**Format**
Digital Raw Data:	Value, sensor ID, data's module destination

Chapter 4
Concentrator Demonstration

4.1 Introduction

A technique to remotely test oil dielectric strength in a power transformer's tap changer has been a research topic at the MIT High Voltage Research Laboratory and has been sponsored by Entergy Service, Inc. The MIT method for this test can be separated into three components: a DSI tester, an observer, and a concentrator. The DSI tester tests the oil's dielectric strength and produces a voltage that indicates the amount of oil degradation [5]. The observer receives results from the concentrator and displays them in a user-friendly format. And as defined, the concentrator joins the two together.

Entergy's problem provided an excellent test for the concentrator. Chapter four introduces this test and presents the test results. Section 4.1 specifies the test requirements imposed on the concentrator. Section 4.2 describes the set-up. Section 4.3 presents the test results. Finally, section 4.4 discusses these results.

4.2 Test Requirements

In addition to meeting the theoretical requirements presented in Chapter three, this test imposed four additional requirements. One, the concentrator must read three analog voltages produced by the DSI tester. Two, the concentrator must use three switches to control the DSI tester. Three, the concentrator must convert raw data into units of voltage. And four, the concentrator must be robust, capable of recovering from either self-generated or non self-generated faults. Table 4.1 summarizes these requirements.

Table 4.1
Test Requirements

1. Read three analog voltages.
2. Control three switches.
3. Convert units to voltage.
4. Recover from faults.

4.2.1 Proposed Test

Before proposing the test, a couple definitions are necessary.

Taking a shot:	Firing the DSI tester, reading the voltages, processing these voltages, and storing the results in a buffer.
Group:	Result of taking one shot.
Sequence:	Collection of n groups.

Now the proposed test contains four steps. The concentrator should

1. Take a twenty shot sequence with specified time intervals between each shot.
2. Post the twenty shots to a web server after the sequence is complete.
3. Wait a specified time.
4. Repeat steps one through three a specified number of times.

4.3 Hardware Set-up

Table 4.2 lists the equipment used to conduct the test.

Items one through seven are the physical equipment used to conduct the test. Figure 4.1 illustrates the equipment layout. In this test, TINI, together with the TINI Socket Plus development board, serve as the concentrator. TINI is a mini computer programmed with the concentrator's functions and is basically responsible for firing the DSI tester, reading its voltages, and posting the processed data to a web page.

Table 4.2
Equipment List

1. Dallas Semiconductor's TINI, Revision C
2. TINI Socket Plus development board, Revision A
3. DSI test apparatus
4. Dallas Semiconductor One-Wire Sensors
   - One DS2450 – Four channel A/D Converter
   - Three DS2405's – Addressable Switch
5. Web server that accepts TINI's data
6. Existing Ethernet
7. Standard PC
8. TINI 1.01 firmware for TINI's flash memory
9. JavaKit
10. Sun's Java Communications API, **javax.comm**
11. Sun's JDK 1.3 API
12. Dallas Semiconductor's TINI API

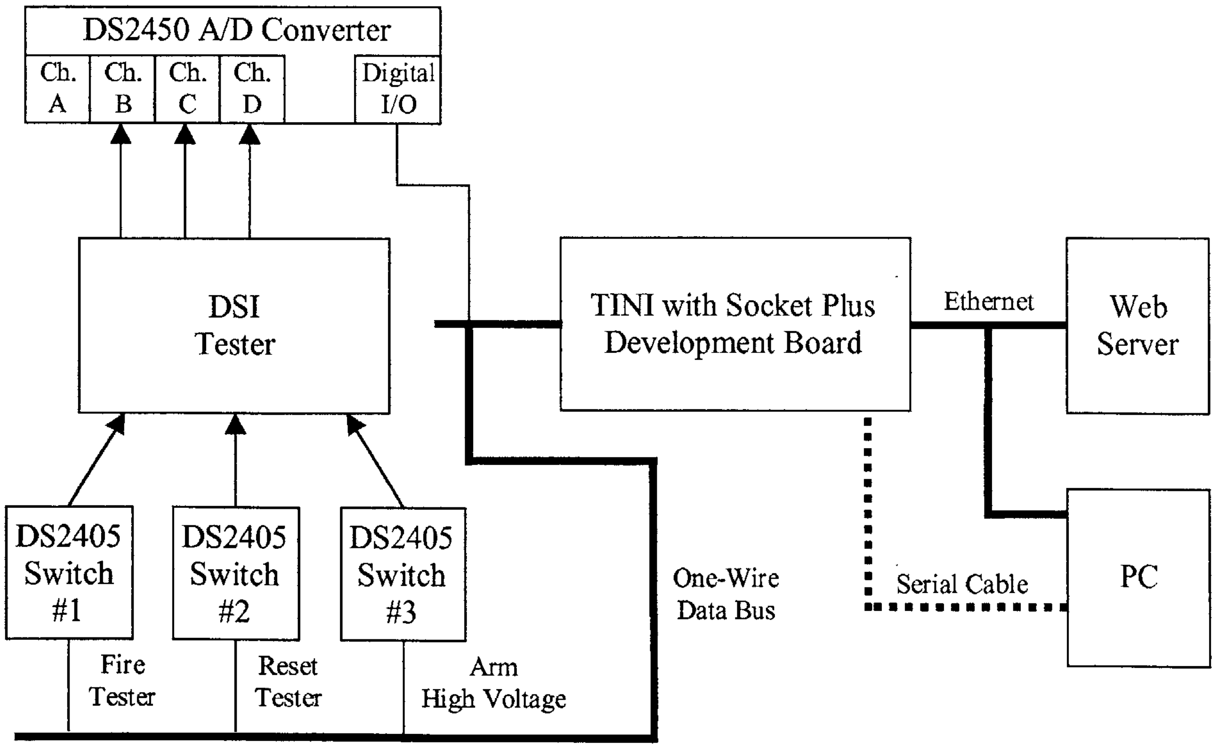


Figure 4.1
Hardware Set-Up

As mentioned in the introduction, the DSI Tester measures the oil's dielectric strength. To make this measurement, the tester must perform three actions.

1. Arm High Voltage:	Turn on the high voltage supply.
2. Fire Tester:	Discharge a high voltage pulse.
3. Reset Tester:	Reset the fire detector so that subsequent firings can be detected.

Now the tester was designed such that a switch can perform each action. TINI uses three DS2405 switches to control these actions, and as depicted in Figure 4.1, each switch performs the following:

- Switch 1 – Fire Tester
- Switch 2 – Reset Tester
- Switch 3 – Arm High Voltage

While measuring the oil's dielectric strength, the DSI Tester produces three analog voltages.

1. Charge voltage:	Pulse voltage the DSI Tester uses to conduct the oil dielectric strength test.
2. Check-fire voltage:	Voltage that determines if the pulse fired.
3. Result voltage:	Result of the oil dielectric strength test.

To measure and digitize these voltages, the concentrator uses the DS2450 analog to digital converter. The four channels shown in Figure 4.1 measure the following:

- Channel A – not used
- Channel B – charge voltage
- Channel C – check-fire voltage
- Channel D – result voltage

TINI communicates with the A/D converter and the switches by using Dallas Semiconductor's one-wire bus protocol. Each one-wire device has specification sheets detailing the necessary steps that facilitate one-wire bus communication. For further information, please see the appropriate specification sheet.

Once TINI has processed the data, it must post the results to a web server. The web server functions as a basic observer in Chapter two's concentrator - observer concept. From the TINI's point of view, it simply receives data and returns a confirmation.

Finally, a standard PC is used to interact with TINI. First, the PC initializes TINI, loading the TINI 1.01 firmware over a serial connection. JavaKit is a Java application that facilitates PC serial communication with TINI and requires **javax.comm**. **javax.comm** "allows Java applications to send and receive data to and from the serial and parallel ports of the host computer" [16]. Please refer to Appendix A for further information on setting up and configuring TINI.

Second, a PC is used to create TINI's software. Sun's JDK 1.3 API, a Java development platform, and Dallas Semiconductor's TINI API provide Java class files used to compile TINI's code. Please refer to Appendix B for a more detailed description of building TINI's applications.

4.4 Software Structure

As previously mentioned, TINI's software performs the concentrator tasks outlined in Chapter three. And since TINI should also meet the concentrator properties discussed in Chapter two, the software uses two configuration files. It also produces two output files, one for diagnostic purposes, one that serves as the data buffer.

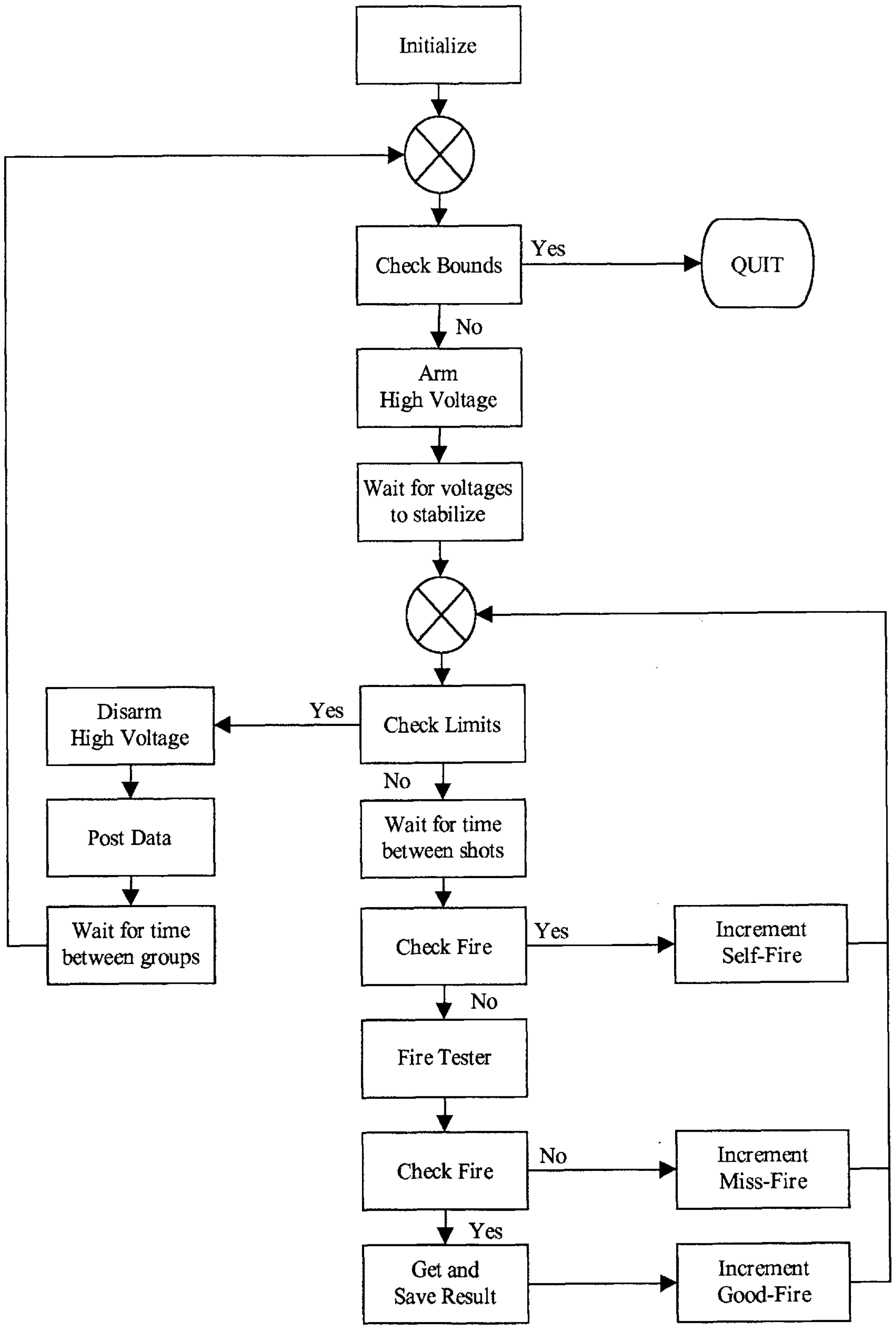


Figure 4.2
TINI Software Flow Chart

Table 4.3
Subroutines for TINI Software Flow Chart

Check Bounds

1) If number of groups ≥ limit, then return "Yes"
2) Else return "No"

Check Fire

1) Read channel C
2) Set "Check Fire" switch to high, then low, then high
3) If result of read is logic high, then return "Yes"
4) Else return "No"

Arm High Voltage

1) Set "Fire" switch to high
2) Set "Arm High Voltage" switch to low
3) Set "Check Fire" switch to high, then low, then high

Fire Tester

1) Set "Fire" switch to low, then high

Check Limits

1) If number of Good-Fires ≥ 20, then return "Yes"
2) If number of Self-Fires ≥ 20, then return "Yes"
3) If number of Miss-Fires ≥ 20, then return "Yes"
4) Else return "No"

Get and Save Result

1) Read channel D
2) Stamp and save data

4.4.1 Concentrator Software

The concentrator software consists of one program that controls all the modules' scheduling and routing issues. Basically, it performs the four steps mentioned in section 4.1. It first fires the DSI Tester, gets the raw data and creates a data point. A data point consists of the following fields:

1. Date
2. Time
3. Result
4. Units
5. Sensor ID
6. A/D Channel Number
7. TINI ID (the concentrator ID)
8. Software Version
9. Test Type ID
10. Sequence Number
11. Group Number

After creating twenty data points, each point is posted to the web server. Then the software waits for a specified duration and repeats this process a specified number of times. Figure 4.2 presents a more detailed flow chart of this program and Table 4.3 specifies subroutines represented in Figure 4.2.

Now for this program to correctly work, it must implement the modules presented in Chapter three. In this program, the main routine uses separate class files that represent most of these modules. Below is a list of the modules that fall into this category.

- Capture Raw Data
- Concentrator ID
- Delete Data
- Fault Handler
- Implement Physical World Commands
- Process Data
- Retrieve Data
- Send Information
- Sensor Stamp
- Store Data
- Time-Date Stamp

The following modules do not exist as separate class files. Rather the main routine incorporates their functionality.

- Error Check
- Listener
- Local Control
- Observer Message Processor

Finally, the main routine uses the following class files for implementation purposes. A brief summary of their functionality is included.

• Configure Test:	Uses "Load Configuration File" to read the configuration file and store results in class variables.
• DS2450 Library:	Library encapsulating useful actions and properties associated with the DS2450 A/D converter.
• Find iButtons Console:	Dallas Semiconductor class file that lists all one-wire devices (iButtons) on the one-wire bus. Used in "Concentrator ID" to list TINI's controllers and sensors.
• iButtonContainer05:	The DS2405 device driver.
• iButtonContainer20:	The DS2450 device driver.
• Load Configuration File:	Driver set up to read the concentrator's configuration file and write to specified locations.
• Post Data Library:	Driver that first formats data for the web server and then posts it via the Ethernet port.
• TINI Data:	Creates and manipulates a vector of TINI Data Points.
• TINI Data Point:	Creates and manipulates a vector that contains the eleven data fields.

4.4.2 Fault Handler Configuration File

faultnum.cfg configures the "Fault Handler" module and only contains the number of faults the module has handled (fault number). Each time a fault is passed in, "Fault Handler" reads this file. If the fault number is within an acceptable range, "Fault Handler" writes the fault to an error log, increments the fault number, then writes the new fault number to **faultnum.cfg**. If the fault number is too big, "Fault Handler" first zeros the fault number, erases the error log, and then proceeds as if the fault number were acceptable.

4.4.3 Concentrator Configuration File

concentrator.cfg configures the entire concentrator and allows users to easily change operating parameters. Below is a list of chosen operating parameters that are customizable.

1. Arm High Voltage Switch address
2. Reset Switch address
3. Fire Tester Switch address
4. A/D Converter address
5. Test Type ID
6. Number of shots to take
7. A/D converter's channel test result appears on
8. Time between shots
9. Time between sequences
10. Number of sequences to take
11. Sequence number

Please see Appendix E for more details.

4.4.4 Error Log

ErrorLog.txt logs a maximum of twenty faults passed to the "Fault Handler." As mentioned above, this file is erased when the twenty-first fault occurs. This fault then begins a new log file.

4.4.5 Software Data Buffer

buffer.txt serves as the nonvolatile data buffer required in Chapter two. Each line in the file is one data point and consists of the following tab separated values:

1. Date stamp
2. Time stamp
3. Result voltage
4. Result's units
5. Sensor ID
6. A/D channel result came from
7. TINI (Concentrator) ID
8. Software version
9. Test Type ID
10. Sequence number
11. Group number

An example data buffer line follows (formatted for readability):

Date	**Time**	**Result**	**Units**	**Sensor ID**	**A/D Channel**
11/27/2000	5:6:24	4.26	Volts	96000000011F7C20	D

TINI ID	**Software Version**	**Test Type ID**	**Sequence Number**
0:60:35:0:64:ac	1.1	108	14975301491400

Group Number
1

4.5 Results

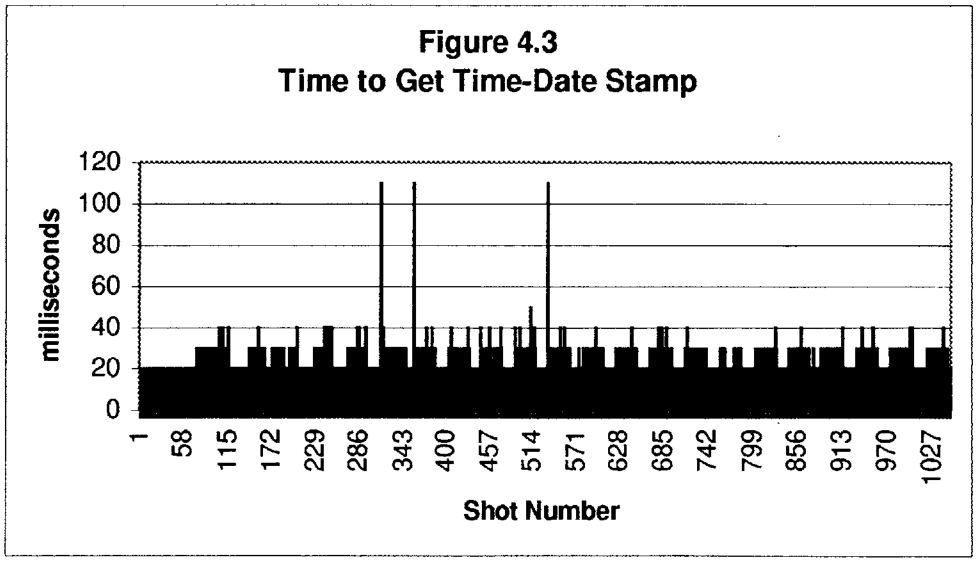
TINI successfully controlled the DSI Tester, took readings, processed the raw data, and posted results to the web server. This section presents additional results to help quantify how well TINI actually performed as a concentrator.

4.5.1 Time Needed to Get Time-Date Stamp

To measure the time required to get the time-date stamp data into memory, the following experiment was conducted:

1. Stamp dummy data and get time in hundredths of seconds
2. Stamp dummy data and get new time in hundredths of seconds
3. Write results of steps one and two as TAB separated values on one line in a text file.

Figure 4.3 plots this time as a function of the shot number. Following this figure is the maximum, minimum, and average time of one thousand fifty points, as calculated in Microsoft Excel.



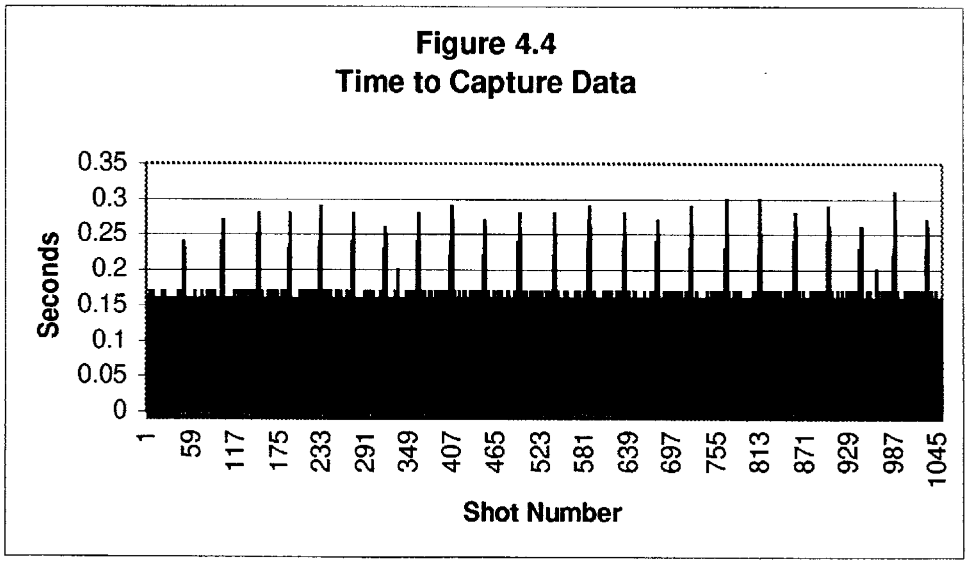
Max Time:	110	milliseconds
Min Time:	10	milliseconds
Ave Time:	24.4	milliseconds

4.5.2 Time Needed to Capture Raw Data

To measure the time TINI needs to capture raw data, the following experiment was conducted:

1. Stamp dummy data and get time in hundredths of seconds
2. Issue controller instructions
3. Perform A/D conversion and read result
4. Stamp dummy data and get new time in hundredths of seconds
5. Write results of steps one and four as TAB separated values on one line in a text file.

This measures the time elapsed between TINI issuing a controller instruction and TINI receiving the raw data. Figure 4.4 plots this time as a function of the shot number. Following this figure is the maximum, minimum, and average time of one thousand fifty points, as calculated in Microsoft Excel.



Max Time:	310	ms.
Min Time:	140	ms.
Ave Time:	170	ms.

4.5.3 Time Needed to Perform A/D Conversion

Normal one-wire communication speed is 16.3k bits per second. Now to perform A/D conversions using the DS2450, one must first write to its memory, tell it to convert, then read from the memory buffer. Reading from memory is a crucial step as the DS2450's buffer is replaced with new values upon each conversion. Accounting for the number of bytes that must be sent to and received from the DS2450, the times for each of the three steps are calculated below.

Write Memory

Send first 12 bytes:	(8 bits/byte) * (12 bytes) * $(16.3 * 10^{\wedge 3}$ bits/sec$)^{\wedge -1}$	=	5.89 ms
Receive confirmation:	(8 bits/byte) * (3 bytes) * $(16.3 * 10^{\wedge 3}$ bits/sec$)^{\wedge -1}$	=	1.47 ms
Send last byte:	(8 bits/byte) * (1 bytes) * $(16.3 * 10^{\wedge 3}$ bits/sec$)^{\wedge -1}$	=	0.49 ms
Receive confirmation:	(8 bits/byte) * (3 bytes) * $(16.3 * 10^{\wedge 3}$ bits/sec$)^{\wedge -1}$	=	1.47 ms
Total:			9.32 ms

Perform Conversion

Send 12 bytes:	(8 bits/byte) * (12 bytes) * $(16.3 * 10^{\wedge 3}$ bits/sec$)^{\wedge -1}$	=	5.89 ms
Receive confirmation:	(8 bits/byte) * (3 bytes) * $(16.3 * 10^{\wedge 3}$ bits/sec$)^{\wedge -1}$	=	1.47 ms
Hold Power:			1.00 ms
Total:			8.36 ms

Read Memory

Send 12 bytes:	(8 bits/byte) * (12 bytes) * $(16.3 * 10^{\wedge 3}$ bits/sec$)^{\wedge -1}$	=	5.89 ms
Receive confirmation:	(8 bits/byte) * (3 bytes) * $(16.3 * 10^{\wedge 3}$ bits/sec$)^{\wedge -1}$	=	1.47 ms
Total:			7.36 ms

Total time for one conversion: 25.04 ms

The DS2450 does support an "overdrive" mode where the one-wire communication speed is boosted to 142k bits per second. Performing the calculations above with this new speed produces the following results:

Total time to write memory:	1.07 ms
Total time to convert:	1.84 ms
<u>Total time to read memory</u>:	<u>0.84 ms</u>
Total time for one conversion:	3.75 ms

4.5.4 Time Needed to Toggle DS2405

The DS2405 uses normal one-wire communication speed (16.3k bits per second) and does not support overdrive mode.

<u>Toggle Switch</u>

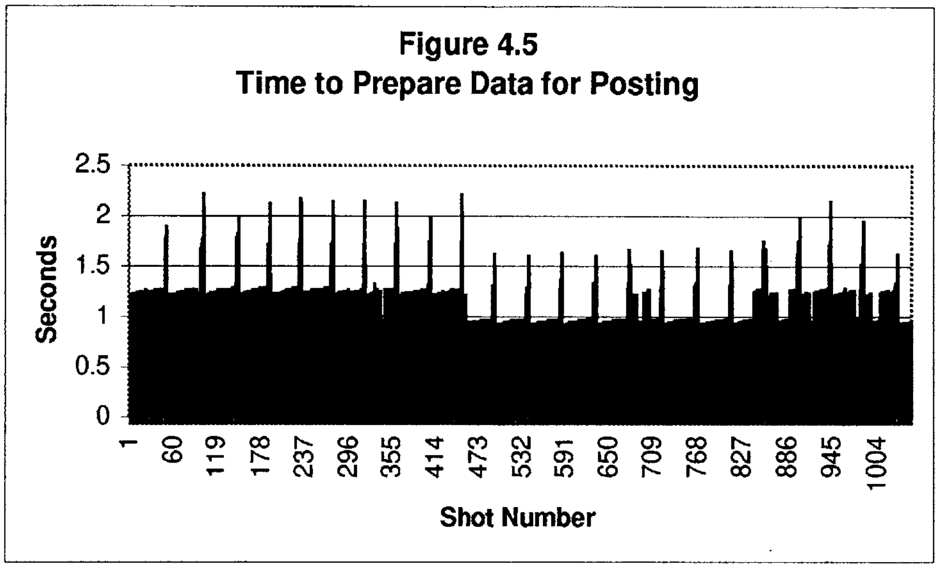
Send 10 bytes: (8 bits/byte) * (10 bytes) * (16.3 * 10^3 bits/sec)^-1 = 4.91 ms

4.5.5 Time Needed to Prepare Data for Posting

To measure the time TINI needs to prepare data for posting, the following experiment was conducted:

1. Stamp dummy data and get time in hundredths of seconds
2. Stamp with time and date
3. Process Data
4. Stamp with units
5. Stamp with sensor ID
6. Stamp with A/D channel information
7. Stamp with TINI's ID
8. Stamp with software version
9. Stamp with test type ID
10. Stamp with sequence number
11. Stamp with group number
12. Stamp dummy data and get new time in hundredths of seconds
13. Write results of steps one and four as TAB separated values on one line in a text file.

Figure 4.5 plots this time as a function of the shot number. Following this figure is the maximum, minimum, and average time of one thousand fifty points, as calculated in Microsoft Excel.



Max Time:	2.22	seconds
Min Time:	0.89	seconds
Ave Time:	1.13	seconds

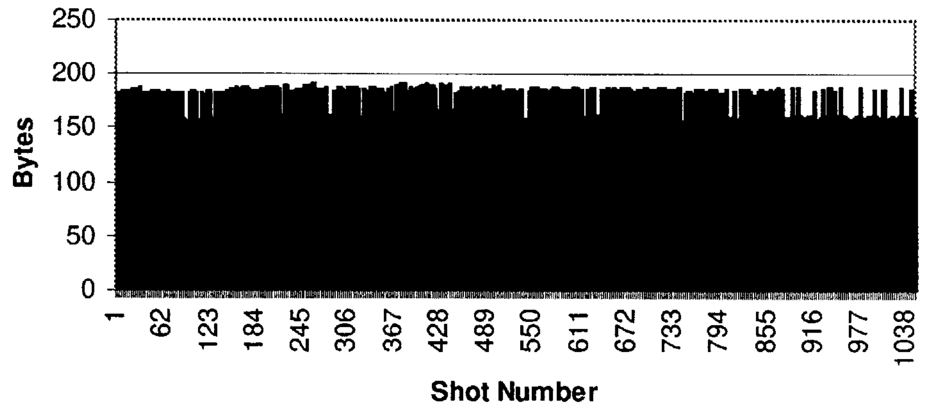
4.5.6 Size of Posted Data

To measure the size of the data TINI posts to the web server, the following experiment was conducted:

1. Capture raw data
2. Process raw data and stamp it with all the data fields
3. Measure size of data and data fields
4. Write results to text file

Figure 4.6 plots this size as a function of the shot number. Following this figure is the maximum, minimum, and average size of one thousand fifty points, as calculated in Microsoft Excel.

Figure 4.6
Size of Posted Data



Max Size:	192	Bytes
Min Size:	152	Bytes
Ave Size:	169	Bytes

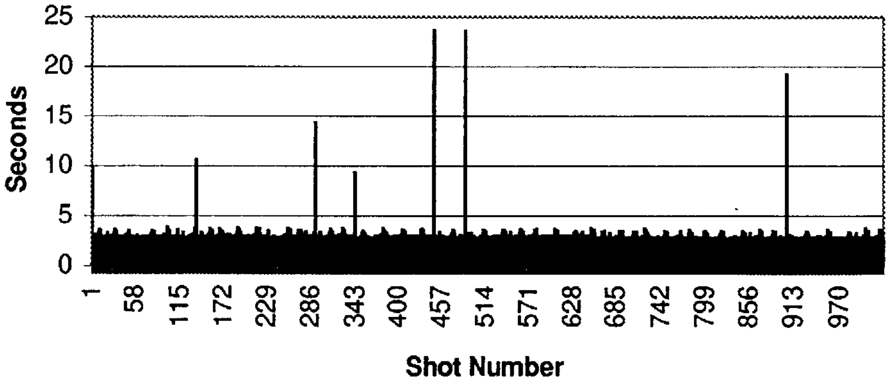
4.5.7 Time Needed to Post Data and Receive Confirmation

To measure the time TINI needs to post data and receive a confirmation, the following experiment was conducted:

1. Stamp dummy data and get time in hundredths of seconds
2. Get point to post
3. Post data point
4. Receive confirmation from web page
5. Stamp dummy data and get new time in hundredths of seconds
6. Write results of steps one and four as TAB separated values on one line in a text file.

Figure 4.7 plots this time as a function of the shot number. Following this figure is the maximum, minimum, and average time of one thousand twenty-six points, as calculated in Microsoft Excel.

Figure 4.7
Time to Post and Receive Confirmation



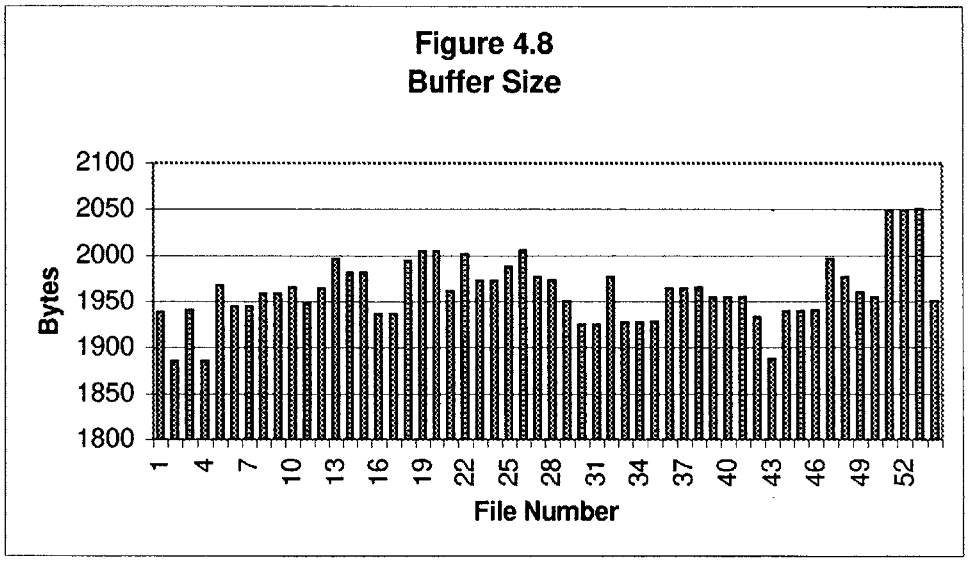
Max Time:	24	seconds
Min Time:	2	seconds
Ave Time:	3	seconds

4.5.8 Data Buffer Size

To measure the data buffer's file size, the following experiment was conducted:

1. Take twenty shots
2. Get file size
3. Write result to a file

Figure 4.8 plots buffer size as a function of the file number. Following this figure is the maximum, minimum, and average size of fifty-four files, as calculated in Microsoft Excel.



Max Size:	2050	Bytes
Min Size:	1885	Bytes
Ave Size:	1961	Bytes

4.5.9 Long Term Operation

Long term operation seems limited at this time. Sometimes, an unknown error disrupts TINI's ability to communicate over the network. When TINI attempts the next data post, the operating system generates an error message, the software catches this error, then forces TINI to reboot. This usually solves this problem.

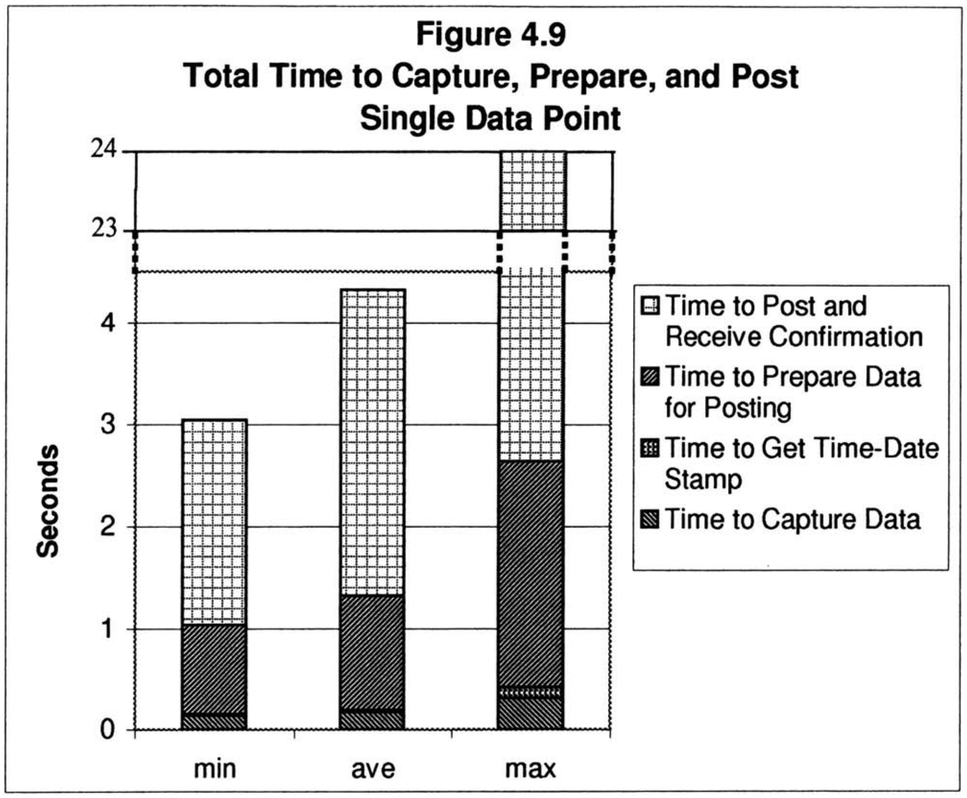
However, another unknown error, potentially related to the first, sometimes prevents TINI from executing the concentrator software after a reboot. When the software catches an error, it executes the Fault Handler routine, which in turn reboots TINI. Ideally, TINI executes a startup routine that will restart the concentrator software. Yet during this process, sometimes an error occurs and TINI will reboot and just not execute the concentrator software. As this error occurs when the software has been terminated, it is not detected. So with these two known errors, the software has not run longer than two consecutive days.

4.5.10 Total Time Needed to Capture, Prepare, and Post Data

Figure 4.9 displays the total time a concentrator requires to capture raw data, prepare it for posting by processing it and stamping it with the appropriate labels, post an average of 170 bytes of data to the observer, and then receive a confirmation. In summary, this time measures the following steps:

1. Capture Data:	Send fire command, digitize, then read into TINI's RAM
2. Get Time-Date Stamp:	Create time-date data in TINI's RAM
3. Prepare Data for Posting:	Process data point, stamp data point with time-date data created in step 2, then create and associate all remaining stamps discussed in section 4.5.5
4. Post and Receive Confirmation:	Post data point to observer then receive confirmation from observer

The charted data are results calculated and presented in the previous sections.



Max Time:	26.64	seconds
Min Time:	3.04	seconds
Ave Time:	4.32	seconds

4.6 Discussion

Except for long-term stability, TINI adequately served as a concentrator, interfacing the physical world and the network controller with minimal problems.

4.6.1 Physical World Connection

As already shown, TINI connects to the physical world by using one-wire devices. One-wire devices, in this thesis, are Dallas Semiconductor iButtons that communicate over a one-wire bus. Now many one-wire devices either have available software drivers or are simple enough to create custom drivers. Additionally, the TINI API provides well-developed classes that are devoted to one-wire communication and can be used to create these drivers. Together, this cuts down on necessary programming. Throw in a relatively small cost, small size, and general availability, these devices are attractive options.

However, the one-wire devices used in this test exhibit two drawbacks. One, they are not suited for applications requiring times faster than tens of milliseconds. The DS2450, at a normal bus rate, requires about 25 milliseconds to convert data, 4 milliseconds using the overdrive rate. The DS2405 requires about 5 milliseconds to toggle the switch. Note that these rates do not include the processor time TINI uses to implement the appropriate commands. Include this time and one learns that TINI takes an average of 146 milliseconds to fire the DSI Tester and read the result (calculated by subtracting the average time to time stamp data from the average time to capture raw data).

Even though these times appear slow, it did not hamper this test. First, the DSI Tester, by design, holds the desired voltages long enough for them to be converted. Second, the DSI Tester has built in safety mechanisms that prevent damaging racing conditions. Consequently the tester, and thus this test, did not rely on one-wire devices to detect and react to rapidly changing signals.

However, the second drawback did prove problematic. The one-wire devices, in particular the DS2405 addressable switches, are susceptible to noise. Firing the high voltage pulse generates transient noise and often causes both the A/D converter and the switches to enter unknown states. Basically, this means the A/D converter returns unreasonable values and the switches do not return their switch state when asked by TINI.

This proved annoying, but not insurmountable. To begin, both the one-wire devices and the one-wire bus were shielded from the noise. With this shielding, the A/D converter provides reliable data and TINI can poll the switches for their current state. Yet two problems still remain. One, TINI still has to wait for the noise to dissipate before it can ask for the switch's current state. And two, the noise still causes the switches to change state.

So to accommodate these problems, the software uses two methods. One, it accepts that it can not determine a switch's state immediately after a pulse fire. So, it simply waits. And two, it assumes the switches change state after each shot. So before the next shot, the program re-initializes each switch to the proper state.

Overall, TINI has little trouble connecting to the physical world. The one-wire devices seem slow, but the DSI Tester compensates for this. Noise disturbs the one-wire devices, but shielding and software compensate for this. Therefore, TINI can reliably communicate with the physical world and thus meets the concentrator's physical world requirements.

4.6.2 Network Connection

Unlike the physical world connection, TINI has no problems connecting to a network. Using TCP/IP protocol, TINI takes three seconds, on average, to post an average of 169 bytes to the web server (the spikes in Figure 4.7 can be attributed to the randomness in the network connection). This may seem slow, but from the concentrator's point of view, posting is not time-critical. If another operation must be performed before all the data points are posted, posting can cease as long as the remaining data points are stored in the buffer. Then when there is available time, the concentrator can read from the buffer and pick up where it left off.

Now if the server is down or there is a problem transmitting and receiving information, the "Send Information" module generates a fault that is handled by the "Fault Handler." Once this problem is resolved, posting can continue since, like above, the data is stored in the buffer. Therefore TINI successfully connects to the network.

4.6.3 Concentrator Software

TINI's software does not exactly match Chapter three's suggestions. Most of the modules were implemented as suggested and work quite nicely. These modules are listed in section 4.3.1. However, certain circumstances arose that forced a re-adaptation of some modules. These four adaptations are described below.

First, to make implementation easier, the concentrator software uses a central controlling program. This program schedules modules and controls data flow. It also operates with a defined command set. Therefore, a module technically neither passes data to other modules nor receives commands from other modules. A module also does not check for command validity since the control program uses only pre-defined commands.

Second, the "Listener," "Observer Message Processor," and "Update" modules were not fully developed due to observer limitations. Now the input from the observer is limited as it only sends a confirmation that the data arrived. Since only a confirmation arrives, the "Listener" module was best implemented in the "Post Data Library" class. Here, the class simply posts data, awaits a reply, and passes the result to the control program. The control program then determines how to best proceed, thus eliminating the need for the "Observer Message Processor" module.

Since the "Observer Message Processor" module does not exist, other concentrator functions are reduced. For one, TINI does not issue capture data or digital control commands based on observer messages. Consequently this limits the functionality of "Implement Physical World Commands" and "Capture Data." TINI also does not retrieve data or send its detailed description upon command, thus limiting the "Retrieve Data" and "Concentrator ID" functionality.

The final observer imposed limitation occurs in the "Update" module. TINI does not receive new software, new parameters, or update commands, so the "Update" module is not needed. But unlike the previous limitations, the remote update functionality still exists. Currently, one must use ftp to place new software and parameter files on TINI. If only new parameter files are loaded, the software will automatically use these files when it reconfigures itself. But if new software is loaded, then the user must open a telnet session and start the new software. Since this interrupts the test and is less than ideal, this should be fixed in future revisions.

Third, in addition to the "Listener," "Observer Message Processor," and "Update" modules, the "Error Check" and "Local Control" modules were not implemented as separate entities. Where the observer imposed limitations on the first three modules, it did not limit these two modules. Rather specific features of the DSI test rendered them unnecessary. "Error Check" was not implemented as a separate module because the DS2450 provides a basic error checking capability called Cyclic Redundancy Checks (CRCs). The DS2450 generates a CRC number with each memory read and write. This result is then read by the DS2450 device driver and used to check data validity. Although the current software version does not utilize this feature, simple error checking is available.

"Local Control" was not implemented as a separate module because the DSI Test was simple. The test used one sensor, three simple controllers, and a very simple control routine. This routine does not have to differentiate between multiple sensors, which makes implementation easier, and only uses simple comparison checks at appropriate times. So instead of using a separate module, the software simply incorporates the control routine into the master program.

Finally fourth, the "Fault Handler" exists as a separate module but is not fully developed. Currently, any generated fault is passed to the "Fault Handler," just as it should. However, this module simply ensures the log file's size stays finite, logs the fault, and then reboots TINI. Now when TINI reboots, it automatically restarts the concentrator software, so when a fault occurs, TINI is able to recover and continue.

But this recovery is awkward. "Fault Handler" simply halts the software instead of gracefully terminating it. "Fault Handler" also does not determine what caused the fault, so upon rebooting, the fault source may still exist. If TINI reboots every time and the faults are not fixed with the reboot, the software could enter an infinite loop. Clearly this is not ideal and should also be fixed in future revisions.

4.6.4 Data Buffer File

TINI implements the temporary buffer by writing data to a file. This file is located in TINI's nonvolatile memory, thus meeting one of the concentrator's desired properties. Originally, the buffer was to hold a week's worth of data. In this scenario, TINI would take twenty shots per hour, twenty-four hours a day, seven days a week. Now when the buffer file contains twenty shots, the average file size is about 2 kBytes. After performing the math, one finds that the buffer file would require, on average, 336 kBytes. TINI currently has 512 kBytes of nonvolatile RAM, expandable to 1 MByte. Without expanding TINI's memory, storing this much data leaves 176 kBytes for the concentrator software and configuration files. The current software version requires about 22 kBytes while the configuration files require about 1 kByte. Given this, TINI can safely store a week's worth of data.

However, this feature was not implemented primarily for debugging purposes. While debugging the system, the number of shots and the number of sequences per hour varied. So instead of only taking twenty shots once an hour, one could take one shot per minute for the entire day. The next day, one could take one shot every thirty seconds during one hour then return to taking one shot per minute for the rest of the day. Consequently, one could not predict the buffer's size after operating TINI for a week.

So currently the buffer holds the defined number of shots. If the shots are successfully posted to the web page, the file is deleted. But if the shots are not successfully posted to the web page, TINI reboots and keeps the file. The next time TINI tries to post data, it should post both the newly gathered data and the old data in the buffer. However, the current software version simply posts the new data, and if successful, deletes the old. But if the posting is not successful, the new data is added to the buffer and TINI reboots.

Now this elucidates a previously unknown bug. Suppose TINI collects a sequence but fails to post them to the web server. TINI reboots, collects another sequence, but again fails to post them. If TINI continually fails to post data, this loop will continue ad nauseum. Since TINI neither checks the buffer's size nor checks for available memory, the buffer will continue to grow. Once the buffer exceeds the available memory, TINI chokes and ungracefully halts the program. Although the chances of this happening are rare, it should be fixed to ensure long term reliability.

4.6.5 TINI as a Concentrator

Although the software does not implement all the modules exactly as outlined in Chapter three, TINI successfully implements a concentrator. It can retrieve data from the physical world and control processes without observer input. TINI can properly stamp and process raw data, storing it in the temporary buffer if necessary. And TINI can communicate with the observer by posting data and receiving confirmations. Table 4.4 restates the concentrator properties in Table 2.1 and includes how TINI satisfies each property.

Table 4.4
Satisfied Concentrator Properties

1. **Global Properties**

• Controlled by a single observer:	Satisfied by definition. Posts to and receives information from one web server.
• Unique identification:	Satisfied by TINI's unique Ethernet address.
• Report information:	Satisfied by software module "Post Information."
• Receive information:	Partially satisfied by communication contract with web server. After each post, web server sends confirmation.
• Remotely configurable:	Partially satisfied. User must place new configuration files on TINI using ftp. Currently does not accept new files from web server.
• Run simultaneous programs:	Satisfied by TINI's operating system.
• Possess real-time clock:	Satisfied by TINI's peripheral chips.

2. **Security Constraints**

• Nonvolatile data buffer:	Satisfied by TINI's ROM that stores the buffer file.
• Robust with self-recovery mechanisms:	Partially satisfied by "Fault Handler."
• Communicate with encrypted data:	Not satisfied by current software.

3. **Input / Output Requirements**

• Input from physical world:	Partially satisfied by software. Current version puts time-date stamp on data, converts raw data to conventional values.

Table 4.4 (continued)
Satisfied Concentrator Properties

• Output to physical world:	Partially satisfied by software. Current version sends control commands in response to processed data but not in response to observer commands.
• Input from observer:	Partially satisfied by software. Current version accepts posting confirmation that is used as a concentrator control command, but does not accept new software or new parameters.
• Output to observer:	Partially satisfied by software. Current version sends data automatically and is capable of sending status information but does not send data when commanded.

Chapter 5
Conclusions and Future Work

5.1 Concentrator Theory

A concentrator successfully implements a well-defined, remote device that helps separate primary control from data collection. As a well-defined device, the concentrator possesses clearly defined properties and attributes that are logically grouped into global properties, input / output requirements, and security constraints. These attributes led to well-defined software modules with clearly defined input / output requirements, acceptable failure modes, and failure contingency plans. Together these modules implement a concentrator.

The concentrator also operates as a remote device due to carefully specified properties. To ensure the concentrator succeeds as a remote device, it must be robust with self-recovery mechanisms so that a human would not have to service it. The concentrator also has a nonvolatile data buffer so that data would not be lost. It has a unique identification so that the primary controller can identify it in a network. And finally the concentrator can accept new software, parameters, and commands that ensure it is remotely configurable.

Finally, the concentrator helps separate primary control from data collection because it is a member of the observer – concentrator system. The observer is the primary controller. It transmits commands, new software, and new parameters to the concentrator while expecting the concentrator to transmit data, status information, and error messages. The concentrator performs the data collection. It retrieves data from sensors, processes it, then transmits the results to the observer.

However, the concentrator is designed to be much more than a simple data collector. It can also operate as a controller. The concentrator can issue instructions for the physical world controllers either when instructed by the observer or after implementing its control routine. In this fashion, the concentrator serves as the observer's remote controller and as a local controller.

As mentioned many times, the concentrator interfaces the physical world with a network connected controller. It operates as a data collector, local controller, and remote controller and helps separate data collection from primary control. Specific properties define a concentrator while software modules implement it. Overall, the concentrator expands computerized control and data collection.

5.2 Concentrator Demonstration

As shown in Chapter four, the physical demonstration concentrator satisfies some of the defined requirements and adequately works as a data collector and local controller. It can retrieve data and control processes without observer input. It can properly stamp and process raw data then safely store the results in a nonvolatile buffer. And it can send data to the observer.

However, certain requirements were not addressed. For one, the physical concentrator is not very robust. After the concentrator's software runs for a few hours, it loses Ethernet capabilities with rebooting the device the only way to exit this state. As this interferes with the concentrator's abilities to communicate with the observer, this should be further explored.

In addition to not being very robust, the developed software uses a very basic self-recovery method. Once the "impaired Ethernet state" occurs, the device simply detects this state, forces a reboot, then restarts the concentrator software. An improvement would be gracefully terminating the program then restarting the software at the point in the routine where the error occurred.

The physical concentrator also has limited remote configuration capability. Currently the software uses parameter files that can be updated remotely and seamlessly. Unfortunately, the software itself can be updated remotely but not seamlessly. Currently all the modules' software is compiled into one program, so to update one part of the software, the concentrator must replace the whole program. This requires that the old software first quit running. Obviously this is far from seamless.

Finally, the physical concentrator does not respond to observer commands. This feature was not explored in this demonstration and should be developed in future versions.

5.3 Future Work

Forcing concentrators to remain isolated from other concentrators may be too restrictive. True, this requirement helped define a concentrator and make it more versatile, but to be a true local controller, concentrators should be able to share information. Again revisit the complex process used throughout this thesis. Suppose an observer uses multiple concentrators to monitor this process and wants them to mainly operate as local controllers. If one concentrator must make a local decision but first must get data from another concentrator, it has to wait for the data to leave the second concentrator, arrive at the observer, and then arrive at the appropriate module. With a small number of concentrators, this time may be comparable to the amount of time used in concentrator to concentrator communication. But with hundreds or thousands of concentrators, the observer can quickly become overwhelmed with requests. The time to correctly implement a local decision quickly rises, and limits the system's usefulness.

Also, the observer should be carefully defined. The concentrator is only useful if an observer – concentrator network exists and without a well-defined observer, implementing and using a concentrator becomes difficult.

Finally, the demonstration's software improvements mentioned above, as well as the shortcomings mentioned in Chapter four, should be addressed. The existing demonstration is a basic proof of concept but is far from complete. A few bugs exist, such as the potential for the data buffer to run out of memory and the potential for the software to enter an infinite loop and never terminate. Also some requirements were not met, such as long term reliability, responding to observer commands, and configuring remotely. Overall, there are many remaining tests that will help explore the physical concentrator.

Earlier attempts at the concentrator concept [7] used a full size desktop computer, but this is the first attempt to reduce the concentrator's hardware requirement to a small, diskless system. This work successfully demonstrates the capability to achieve an efficient, easy to use, low-cost concentrator structure.

Appendix A

Initialization and Set-Up of TINI

Appendix A highlights steps that set up and configure Dallas Semiconductor's TINI. This assumes the reader has:

- A TINI board
- A board that at least connects TINI to a power supply, an RS-232 connection, and an Ethernet connection. Dallas Semiconductor currently provides a TINI Socket Eurocard 72-pin Revision C board for development work.
- A PC with RS-232 serial output

The TINI web page http://www.ibutton.com/TINI/index.html, Dallas Semiconductor's home page http://www.dalsemi.com/, and Sun's Java home page http://www.javasoft.com/ are valuable sources to find additional information and locate the files mentioned below. Also, please consult the README.txt file (see step 3) for a more detailed description of the installation procedure.

A.1 PC Installations and Configurations

1. Install Sun's Java Development Kit (JDK). This thesis installed JDK 1.3 into the **c:\jdk1.3** directory (<jdk>) and the **c:\Program Files\JavaSoft** (<jdk_prog>) directories
2. Install Sun's Java Communications API, **javax.comm**. This thesis installed **javacomm20-win32.zip** into the **c:\commapi** directory (<comm>)
3. Install the TINI software package. This thesis installed **tini1_01.tgz** into the **c:\tini1.01** directory (<tini1.01>). <tini1.01> contains the **README.txt** file mentioned above
4. Place **<comm>\win32com.dll** in **<jdk>\jre\bin** and **<jdk_prog>\JRE\1.3\bin**
5. Place **<comm>\comm.jar** in **<jdk>\jre\lib\ext** and **<jdk_prog>\JRE\1.3\lib\ext**
6. Place **<comm>\javax.comm.properties** in **<jdk>\jre\lib** and **<jdk_prog>\JRE\1.3\lib**
7. Edit the AUTOEXEC.BAT file so that the **tini.jar** and **comm.jar** files are in the classpath. This thesis entered the following paths into the classpath:
   - <tini1.01>\bin\tini.jar;
   - <jdk>\jre –cp <jdk>\jre\lib\ext\comm.jar;

8. Create the JavaKit.BAT file (or equivalent) found in Appendix C (optional)

A.2 Loading TINI Firmware

Before proceeding, make sure TINI is powered and connected to the PC with a reliable serial cable.

1. Start JavaKit
2. Make sure baud rate is set to 115200
3. Select desired PC serial port
4. Press the "Open Port" button
5. Press the "Reset" button
6. Make sure TINI has the latest bootstrap loader (see the README.txt file referenced above)
7. Load tini.tbin
   - Select "Load File" from the File menu
   - Choose **<tini1.01>\bin\tini.tbin**
8. Clear the heap
   - Type b18 (changes to bank 18)
   - Type f0 (fills bank 18 with 0's)
9. Load slush.tbin
   - Select "Load File" from the File menu
   - Choose **<tini.101>\bin\slush.tbin**

Congratulations. TINI's firmware has been loaded. However, any previous files on TINI, as well as TINI's past configurations, have been lost. This includes network configurations. So until the network configurations are reset, TINI's ftp and telnet services will not work.

A.3 Setting Network Configurations

1. Start JavaKit
2. Select desired PC serial port
3. Press the "Open Port" button
4. Press the "Reset" button
5. Type E and press Return
6. Login to TINI

This enters slush, TINI's operating system. Now use the "ipconfig" command to set the network configurations. For further information, type "help ipconfig" at the slush prompt.

Appendix B
Compiling Programs for TINI

This assumes the reader has some familiarity with Java and Java programs. Also, the reader must install the TINI software as outlined in Appendix A. In particular, the **tini.jar** file must be included in the classpath or placed in the **c:\jdk1.3\JRE\1.3\lib\ext** directory. For this thesis, the **tini.jar** file was included in the classpath. Please read **Building_Applications.txt** located in the TINI install directory under the docs directory for further information.

To begin, TINI applications must be compiled and converted into a **.tini** format. Since TINI applications are written in Java, the Java compiler will successfully convert them into a **.class** format. However, TINI's operating system, which is Java based, does not use Java's entire standard API, so all TINI applications must be compiled with TINI's API.

There are multiple ways to accomplish this. For this thesis,.applications were compiled by specifying Java's bootclass path. Assuming the TINI software was installed in a directory referred to as <TINI Install Dir>, typing the following command at a DOS prompt would convert **Concentrator.java** into a **.class** format using TINI's API.

```
javac -bootclasspath <TINI Install Dir>\tiniclasses.jar Concentrator.java
```

Now the class file must be converted into TINI's format. This is done using the TINIConverter tool included in the **tini.jar** file. To convert **Concentrator.class** to **Concentrator.tini**, type the following at a DOS prompt:

```
java TINIConvertor -f Concentrator.class -o Concentrator.tini -d <TINI Install
Dir>\bin\tini.db
```

If multiple class files must be converted, all the files must first be placed into a directory before the whole directory can be converted. So if compiling

Concentrator.java, as above, creates more than one class file, each file must be placed into one directory. This directory can then be converted into a single **.tini** program. For example, to convert the directory called mydir into **Concentrator.tini** type the following:

```
java TINIConvertor -f mydir -o Concentrator.tini -d <TINI Install Dir>\bin\tini.db
```

Repeatedly typing these lines quickly becomes cumbersome and time consuming. So to make this task easier, the following batch files were created. Each batch file uses the DOS program GETKEY, explained in Appendix D, to get keyboard input. These batch files can either compile and convert a single **.java** file that produces one **.class** file or compile and convert a single **.java** file that produces multiple **.class** files. If the application produces multiple **.class** files, the user must manually place all the files into the desired directory.

B.1 Steps to Compile and Convert Single Class File

1) Type "BuildTini myfile" at DOS prompt (without quotation marks and substituting desired file name for myfile)
2) Type "Y" (It must be a capital Y. Anything else will cause the program to quit.)
3) Type "1"

B.2 Steps to Compile and Convert Multiple Class Files

1) Type "BuildTini myfile mydir" at DOS prompt (without quotation marks and substituting desired file name for myfile and desired directory for mydir)
2) Type "Y" (It must be a capital Y. Anything else will cause the program to quit.)
3) Place all **.class** files that were just created into the mydir directory

4) Type "2"

B.3 BuildTini.BAT

```
ECHO OFF

REM ****************************************************************
REM *                                                              *
REM *  This takes two parameters specified at the command line:    *
REM *        1) File name to compile (no extenstions)              *
REM *        2) Directory to build (if you are building            *
REM *                              a directory)                    *
REM *                                                              *
REM *  Example:  You want to compile foo.java and then build the   *
REM *            entire directory bar (which is located in the     *
REM *                              same directory as foo.java)     *
REM *                                                              *
REM *        So you would type (at the DOS prompt):                *
REM *               buildTini foo bar                              *
REM *                                                              *
REM ****************************************************************

CLS

ECHO %1

javac -bootclasspath <TINI install dir>\bin\tiniclasses.jar %1.java

ECHO Do you want to build a TINI executable file (Y/N)?
GETKEY
IF ERRORLEVEL 78 IF NOT ERRORLEVEL 79 ECHO  N WAS PRESSED

IF ERRORLEVEL 89 IF NOT ERRORLEVEL 90 buildTini1 %1 %2
    REM ECHO  Y WAS PRESSED
```

B.4 BuildTini1.BAT

```
ECHO OFF
CLS

ECHO %1

ECHO Enter:
ECHO    (1) to build a single class
ECHO    (2) to build a directory

GETKEY

 IF ERRORLEVEL 49 IF NOT ERRORLEVEL 50 java TINIConvertor -f %1.class -o
        %1.tini -d <TINI install dir>\bin\tini.db
 IF ERRORLEVEL 50 IF NOT ERRORLEVEL 51 java TINIConvertor -f %2 -o %1.tini
        -d <TINI install dir>\bin\tini.db
IF ERRORLEVEL 50 IF NOT ERRORLEVEL 51 ECHO %2 %1
```

Appendix C
JavaKit Batch File

This appendix assumes the reader has successfully followed the installation steps in Appendix A and placed the files in the directories used for this thesis. If using different directories, please make the appropriate changes in the following code.

TINI's serial connection provides a "fail-safe" connection, is used to load TINI's operating system, and offers another development platform. In development work, the serial connection can potentially be used regularly. So to facilitate easy communication between the PC's serial port and TINI's serial port, Dallas Semiconductor provides a program called JavaKit.

JavaKit is a Java program that serves as a user interface with TINI's serial port. But in order for it to run on a PC, the Java program executor must have its classpath point to two files: tini.jar and tiniclasses.jar. Typing the following line at a DOS prompt can do this:

```
        C:\JDK1.3\bin\java -classpath C:\TINI1.01\BIN\tini.jar;
C:\TINI1.01\BIN\tiniclasses.jar JavaKit
```

However, this is too cumbersome to type frequently. So to provide a quick way to start JavaKit, the following DOS batch file was created.

C.1 JavaKit.BAT

```
        @ECHO OFF

        C:\JDK1.3\bin\java -classpath C:\TINI1.01\BIN\tini.jar;
C:\TINI1.01\BIN\tiniclasses.jar JavaKit
```

Appendix D
DOS Program that Allows Keyboard Input During Batch File Execution

Typical DOS configurations do not allow keyboard input, which limits most batch files. GETKEY.COM lets these files use keyboard input [17]. Follow these steps to create GETKEY.COM.

1. Bring up a DOS prompt
2. Change to the directory GETKEY.COM will be located in (<desired directory>)
3. Enter DEBUG <desired directory>\GETKEY.COM
4. Type:
   - A100
   - MOV AH,1
   - INT 21
   - MOV AH,4C
   - INT 21
   - (press RETURN)
   - RCX
   - 8
   - W
   - Q

Figure D.1 shows a screen shot where the user created GETKEY.COM in the directory C:\tini.

```
C:\tini>DEBUG C:\TINI\GETKEY.COM
File not found

-A100
0ADC:0100 MOV AH,1
0ADC:0102 INT 21
0ADC:0104 MOV AH,4C
0ADC:0106 INT 21
0ADC:0108
-RCX
CX 0000
:8
-W
Writing 00008 bytes
-Q

C:\tini>
```

Figure D.1
Screen Shot of Creating GETKEY.COM

Appendix E

Concentrator Configuration File

concentrator.cfg configures the concentrator to conduct the DSI experiment. Below is the configuration file used in this thesis; the line numbers are added for discussion purposes. Note, any line that begins with the pound sign, #, is treated as a commented line by the software and thus has no effect on the concentrator.

```
1.  #number of uncommented lines in configuration file
2.  #this determines the length of the array which stores configuration data
3.  14
4.  #number of addresses present
5.  4
6.  #Address for Arm High Voltage Switch
7.  F50000000C252605
8.  #Address for Reset Switch
9.  C70000000C42D605
10. #Address for Fire Tester Switch
11. 820000000C77D805
12. #DS2450 (A/D converter) address
13. 12000000012A0120
14. #Test Type ID Number
15. 108
16. #number of groups to take
17. 20
18. #Channel number on the DS2450 (A/D converter) that result appears on
19. 3
20. #Time between groups (in seconds)
21. 60
22. #Time between sequences (in minutes)
23. 1
24. #number of sequences to take (enter 0 if desire infinite number)
25. 0
26. #sequence number (this must be the last item in the configuration file)
27. 1
```

Most of the lines are self explanatory with the following definitions.

- Group: One data point, including all the proper identification tags
- Sequence: A collection of m groups.
- The three channels used on the DS2450 (A/D converter) in this implementation are specified in the compiled code and are not configurable by **concentrator.cfg**

A few lines, though apparently self-explanatory, need some elaboration. As the comment suggests, line three initializes an array in the software. This array contains the values on all the uncommented lines, except for this number, and is used to initialize specific variables in the software.

Line five tells the software how many one-wire device addresses follow. It is important that all the one-wire device addresses follow this line else the software will crash.

Originally, lines three and five were intended to make adding parameters and devices easier. These lines do achieve this goal under certain qualifications. The user must still change the software by adding the appropriate variable and matching it with the appropriate array position. Also, new one-wire devices can be added as long as the variables, with the appropriate drivers, exist in the code. So the user can add devices and parameters to the code and not worry about the initialization array's length, as long as they update the code and change the appropriate numbers in the parameter file. This makes software changes a little easier and less susceptible to human forgetfulness.

Finally, line twenty-seven must be the last line in the parameter file. The software updates the sequence number internally, then rewrites it to the parameter file. Rather than rewriting the entire parameter file, the software simply jumps to the appropriate location in the file and rewrites the number. This location is at the end of the file for one reason. The sequence number can continually grow, and storing a variable that uses more and more bytes in a file forces the software to rewrite everything after the sequence number, otherwise other data will be overwritten. So the only place in the file that does not have data after the sequence number is the end of the file.

Appendix F

Concentrator Software

F.1 Concentrator.java

F.1.1 Code Description

This program implements the flow diagram depicted in Figure 4.2. It configures variables using the **concentrator.cfg** file and is implemented with two loops: the "Big Loop" and the "Inner Loop." The "Big Loop" takes the specified number of groups, posts the results, reconfigures the variables, and then checks to see if it should loop again. The "Inner Loop" operates the DSI tester, captures processes data, then stamps and saves the results. Each loop number is specified in the configuration file.

F.1.2 Actual Code

```
/*******************************************************************
 *
 *        Concentrator:
 *
 *
 *        Author:  Michael T. DePlonty
 *
 *        Date:    November 30, 2000
 *
 *******************************************************************/
import com.ibutton.iButtonException;
import com.ibutton.adapter.*;

class Concentrator
{
  private static final String BUFFER_FILE_NAME      = "buffer.txt";
  private static final String DEFAULT_URL =
"http://192.168.0.2/entergy/listen4data_full.asp";

  private static final double      FIRE_LIMIT        = (double) 2.5;
private static final int          CHRG_VLT_CHNL = 1;  //channel B on A/D
                                                      //converter will read
                                                      //the charge voltage
private static final int          CHK_FIRE_CHNL = 2;  //channel C on A/D
                                                      //converter will read
                                                      //the check fire voltage
private static final int          RSLT_VLT_CHNL = 3;  //channel D on A/D
                                                      //converter will read
                                                      //the result voltage

private static ImplementPWorldCmds  armingSwitch;
private static CaptureRawData       captureData;
private static ConcentratorID       concentratorID;
private static ConfigureTest        configureTest;
private static DeleteData           deleteData;
private static ImplementPWorldCmds  fireSwitch;
private static TiniDataPoint        onePoint;
private static ProcessData          processData;
private static ImplementPWorldCmds  resetSwitch;
private static RetrieveData         retrieveData;
private static SendInformation      sendInformation;
private static SensorStamp          sensorStamp;
private static StoreData            storeData;
private static TimeDateStamp        time_dateStamp;
private static TiniData             vectorDataPoints;

private static String[]             command;

private static String               a2dConverter_address;
private static String               arm_HV_address;
private static String               fire_DSI_address;
private static String               lapsedMillisecondsString;
private static String               reset_switch_address;
private static String               resultOfPosting;
private static String               sequenceNumberString;

private static long                 groupNumber;
private static long                 sequenceNumber;

private static double               charge_voltage;
private static double               check_fire;
private static double               result_voltage;

private static int                  num_miss_fires;
private static int                  num_self_fires;
```

```
  private static int                      num_seq_limit;
  private static int                      num_seq_limit_new;
  private static int                      num_shots_limit;
  private static int                      numAddresses;
  private static int                      resultChannel;
  private static int                      shotRestTime;
  private static int                      tempBuffer;
  private static int                      testTypeID;
  private static int                      time_between_sequences;

  private static byte                     posting_flag;

  public static void main(String args[])
  {
    try
    {
      System.out.println("Welcome to the concentrator.");
      configure();
      init();
      System.out.println("Done initializing.");
      num_seq_limit = num_seq_limit_new;
      int tempCounter = 0;

/*******************************************************************
 *
 *    Run the test
 *
 *******************************************************************/

      do
      {
/*******************************************************************
 *
 *    Initialize these at the start of every sequence, which is the begining of this
 *  do loop (one iteration of the do loop corresponds to one sequence).
 *
 *******************************************************************/
        num_self_fires = 0;
        num_miss_fires = 0;
        posting_flag   = (byte) 0;  //init. flag such that it is not set
        vectorDataPoints = new TiniData();

        lapsedMillisecondsString =
                              Long.toString(time_dateStamp.getLapsedMilliseconds());
    //gets time in milliseconds elapsed since Jan. 1., 1970.  Used to create unique
sequence number
```

```
        sequenceNumberString = Long.toString(sequenceNumber);
        sequenceNumberString =
                              sequenceNumberString.concat(lapsedMillisecondsString);

        System.out.println("Starting Big Loop number " + tempCounter);
        System.out.println(" ");

System.out.println("*****************************************************");

        groupNumber    = (long) 1;  //reset group numbers for a new sequence


        System.out.println("Parameters the test is operation on:");
        System.out.println("test type id = " + testTypeID);
        System.out.println("num groups limit = " + num_shots_limit);
        System.out.println("resultChannel = " + resultChannel);
        System.out.println("Shot Rest time = " + shotRestTime);
        System.out.println("Time Between Sequences = " + time_between_sequences);
        System.out.println("number seqences limit = " + num_seq_limit);
        System.out.println("Sequence Number = " + sequenceNumber);

/*****************************************************************
 *
 *    Arm HV
 *
 *****************************************************************/
        fireSwitch.initializeSwitch("open");
        armingSwitch.initializeSwitch("closed");
        resetSwitch.initializeSwitch("open");
        resetSwitch.initializeSwitch("closed");
        resetSwitch.initializeSwitch("open");
        System.out.println("Armed the high voltage");

/*****************************************************************
 *
 *    Wait for voltages to stabalize
 *
 *****************************************************************/
        try
        {
          System.out.println("Starting to sleep for 30 secs.");
          Thread.sleep(1000*30);   //wait 30 seconds between runs
        }//try
        catch (InterruptedException e)
        {
          System.out.println("Interrupted Exception");
        }//catch

/*****************************************************************
 *
 *     Take a specified number of shots
 *
 *****************************************************************/
        System.out.println("Number of shots = " + num_shots_limit);

        for(int num_shots=0; num_shots < num_shots_limit; num_shots++)
        {
          System.out.println("Top of inner loop");
          System.out.println(" ");
          System.out.println("Loop Number " + (num_shots+1));
          System.out.println(" ");

          initSwitches();  //put switches into known state

          try
          {
            System.out.println("Sleeping for " + shotRestTime + " sec.");
            Thread.sleep(1000*shotRestTime);          //wait between runs
          }//try
          catch (InterruptedException e)
          {
            System.out.println("Interrupted Exception");
          }//catch

          charge_voltage =
processData.convertToStandardUnits(captureData.retrieveData(CHRG_VLT_CHNL));
          System.out.println("charge voltage = " + charge_voltage);
          check_fire     =
processData.convertToStandardUnits(captureData.retrieveData(CHK_FIRE_CHNL));
          System.out.println("check fire = " + check_fire);

/*****************************************************************
 *
 *     Device did self fire
 *
 *****************************************************************/
```

```
          if (FIRE_LIMIT < check_fire)  //device did a self-fire as
                           //result was above the threshold
          {
            System.out.println("Device did a self-fire");
            ++num_self_fires;  //increment number of self fires
            if (num_self_fires > num_shots_limit/2)
            {
              //signify test failed

              for (int index=0; index < vectorDataPoints.size(); index++)
              {
                System.out.println("Changing Test Type ID.");
                onePoint = vectorDataPoints.getDataPoint(index);
                onePoint.setTestTypeID(11010);
                vectorDataPoints.setElementAt(onePoint, index);
              }//for

              onePoint = null;  //free up this object for future use
              System.out.println("test failed because of self-fire");
              break;
            }//if (num_self_fires > num_shots_limit/2)

            else  //reset fire flip-flop and continue
            {
 //to reset, resetSwitch must go Low, then High.  resetSwitch
 //is already High, so we only need to toggle twice.
              System.out.println("Resetting switch cause number of self fires is not too
big");
              resetSwitch.initializeSwitch("open");
              resetSwitch.initializeSwitch("closed");
              resetSwitch.initializeSwitch("open");
            }//else
          }//if(FIRE_LIMIT < check_fire)

       //to fire, fireSwitch must go Low, then High.  fireSwitch
       //is already High, so we only need to toggle twice.

          fireSwitch.initializeSwitch("closed");
          fireSwitch.initializeSwitch("open");
          check_fire    =
processData.convertToStandardUnits(captureData.retrieveData(CHK_FIRE_CHNL));
          System.out.println("check fire = " + check_fire);

          sensorStamp.acceptCommand(command);
          result_voltage =
processData.convertToStandardUnits(captureData.retrieveData(RSLT_VLT_CHNL));
```

```
        time_dateStamp.stamp();

/*****************************************************************
 *
 *    Test fired as desired
 *
 *****************************************************************/

        if (FIRE_LIMIT < check_fire)  //test fired as desired
        {
          System.out.println("Test fired as desired");
          System.out.println("result = " + result_voltage);
          resetSwitch.initializeSwitch("open");
          resetSwitch.initializeSwitch("closed");
          resetSwitch.initializeSwitch("open");   //reset the fire flip-flop
          createTiniDataPoint();

/*****************************************************************
 *
 *    Saving Data
 *
 *****************************************************************/
          System.out.println("Saving data point");
          storeData.writeData(onePoint, true);
          System.out.println("Saved data point");
          onePoint = null; //free up object for future use

          ++groupNumber;
        }//if (FIRE_LIMIT < check_fire)

/*****************************************************************
 *
 *    Test didn't fire (misfire)
 *
 *****************************************************************/

        else if (check_fire <= FIRE_LIMIT)   //test didn't fire
        {
          System.out.println("Test didn't fire.");
          ++num_miss_fires;  //increment the number of miss fires
          if (num_miss_fires > num_shots_limit/2)
          {
          //signify test failed - voltage too low
```

```
              for (int index=0; index < vectorDataPoints.size(); index++)
              {
                System.out.println("Changing Test Type ID.");
                onePoint = vectorDataPoints.getDataPoint(index);
                onePoint.setTestTypeID(11001);
                vectorDataPoints.setElementAt(onePoint, index);
              }//for
              onePoint = null;  //free up this object for future use
              System.out.println("test failed - voltage too low");
              break;
            }//if (num_miss_fires > num_shots_limit/2)
            num_shots--;  //make sure we take another point and not count this loop
iteration
            System.out.println("Resetting switch cause number of misfire");
            resetSwitch.initializeSwitch("open");
            resetSwitch.initializeSwitch("closed");
            resetSwitch.initializeSwitch("open");

          }//else if (0 <= check_fire < FIRE_LIMIT)
        }//for

/*****************************************************************
 *
 *    Dis-arm HV
 *
 *****************************************************************/
        armingSwitch.initializeSwitch("open");
        System.out.println(" ");

System.out.println("****************************************************");
        System.out.println(" ");
        System.out.println("Turned off HV");
        System.out.println(" ");

System.out.println("****************************************************");
        System.out.println(" ");

        sequenceNumber = sequenceNumber + (long) 1;


/*****************************************************************
 *
 *    Post data
 *
 *****************************************************************/
```

```
        for (int index=0; index < vectorDataPoints.size(); index++)
        {
          System.out.println("Trying to post the data");
          onePoint = vectorDataPoints.getDataPoint(index);
          try
          {
            sendInformation = new SendInformation(DEFAULT_URL);

            System.out.println(" ");

System.out.println("******************************************************");
            System.out.println("Posting this data point: " + index);
            System.out.println(" ");
            System.out.println(onePoint.getDate());
            System.out.println(onePoint.getTime());
            System.out.println(onePoint.getValueAsString());
            System.out.println(onePoint.getUnits());
            System.out.println(onePoint.getSourceID());
            System.out.println(onePoint.getChannel());
            System.out.println(onePoint.getTiniID());
            System.out.println(onePoint.getSoftwareVersionAsString());
            System.out.println(onePoint.getTestTypeIDAsString());
            System.out.println(onePoint.getSequenceNumberAsString());
            System.out.println(onePoint.getGroupNumberAsString());
            System.out.println(" ");
System.out.println("******************************************************");
            System.out.println(" ");

            resultOfPosting = sendInformation.sendData(onePoint.getDate(),
                            onePoint.getTime(),
                            onePoint.getValueAsString(),
                            onePoint.getUnits(), onePoint.getSourceID(),
                            onePoint.getChannel(), onePoint.getTiniID(),
                            onePoint.getSoftwareVersionAsString(),
                            onePoint.getTestTypeIDAsString(),
                            onePoint.getSequenceNumberAsString(),
                            onePoint.getGroupNumberAsString());

            sendInformation = null;  //free up this object for future use

            if (resultOfPosting.compareTo(" ") == 0)  //implies no confirmation from
observer that data was received
            {
              posting_flag = (byte) 1;            //observer didn't get data, so set flag
            }//if observer didn't get data
          }//try
          catch(Exception e)
          {
            //flag that posting didn't work
            posting_flag = (byte) 1;
          }//catch

          if (posting_flag == 0)  //flag is not set, so observer got the data, so clear the
buffer
          {
            deleteData.deleteAllPoints();
          }//if
        }//for posting data

        vectorDataPoints.removeAllElements();
        vectorDataPoints = null;

        ++tempCounter;

//create a new object in case the config file has been updated.

        System.out.println("writing the sequence number");
        configureTest.writeValues(sequenceNumber);

        System.out.println("Reading the configuration file");
        configure();

        System.out.println("Read, configured, and just closed the configuration file.");

        if (num_seq_limit_new == num_seq_limit) //no change in the limit
        {
          if (num_seq_limit == 0)  //we want to run indefinately
          {
            tempCounter = 0;  //since we run indefinately, need to keep this counter finite

            System.out.println("OK, now will run indefinately.  To change, enter");
            System.out.println("a nonzero number for the sequence limit in the
configuration file.");

          }//if run infinitely

          else  //not running infinitely, so check to see if reached the limit
          {
            if (tempCounter >= num_seq_limit)  //reached (or exceeded) the limit, so quit
            {
              System.out.println("Reached the sequence limit, so I'm stopping.");
          System.gc();              //run garbage collector to free up memory
          break;
        }//if reached (or exceeded) the limit
      }//else not running infinitely
    }//if no change in limit

    else  //sequence limit has changed
    {
      System.out.println("Sequence Limit has changed.");
      if (num_seq_limit_new <= tempCounter)
      {
        if (num_seq_limit_new != 0)  //don't wish to run infinitely, so quit
        {
          System.out.println("Changed the sequence limit, and it's a finite limit,");
          System.out.println("so I'm stopping.");
          System.gc();              //run garbage collector to free up memory
          break;
        }//if wish to quit
        else
        {
          //do nothing so that we can loop again
        }
      }//if new sequence limit is less than tempCounter
    }//else sequence limit has changed

    try
    {
      System.gc();              //run garbage collector to free up memory
      System.out.println("Sleeping for " + time_between_sequences + " minutes.");
      Thread.sleep(1000*60*time_between_sequences); //wait for
time_between_sequences minutes
    }//try
    catch (InterruptedException e)
    {
      System.out.println("Interrupted my sleep.");
    }//catch
    num_seq_limit = num_seq_limit_new; //update old sequence limit to the new one

  }//do
  while (true);
}//try the whole thing
```

```
  catch (Exception e)
  {
    FaultHandler fh = new FaultHandler();
    fh.handleFault(e.toString());
  }//catch
} //main

private static void configure()
                    throws Exception, iButtonException, OneWireIOException
{
  configureTest = new ConfigureTest();
  configureTest.readValues();
  arm_HV_address            = configureTest.getARM_HV_ADDRESS();
  reset_switch_address      = configureTest.getRESET_SWITCH_ADDRESS();
  fire_DSI_address          = configureTest.getFIRE_DSI_ADDRESS();
  a2dConverter_address      = configureTest.getA2D_CONVERTER_ADDRESS();
  sequenceNumber            = configureTest.get_sequenceNumber();
  testTypeID                = configureTest.get_testTypeID();
  resultChannel             = configureTest.get_resultChannel();
  num_shots_limit           = configureTest.get_num_shots_limit();
  shotRestTime              = configureTest.get_shotRestTime();
  time_between_sequences    = configureTest.get_time_between_sequences();
  num_seq_limit_new         = configureTest.get_num_seq_limit();

  configureTest = null;

  System.out.println("this is what I have:");
  System.out.println("test type id = " + testTypeID);
  System.out.println("num groups limit = " + num_shots_limit);
  System.out.println("number seqences limit = " + num_seq_limit_new);
}//configure()

private static void init()
{
  command = new String[2];
  command[0] = a2dConverter_address;
  command[1] = "time-date stamp";

  captureData      = new CaptureRawData(a2dConverter_address, testTypeID);
  processData      = new ProcessData();
  armingSwitch     = new ImplementPWorldCmds(arm_HV_address);
  resetSwitch      = new ImplementPWorldCmds(reset_switch_address);
  fireSwitch       = new ImplementPWorldCmds(fire_DSI_address);
  time_dateStamp   = new TimeDateStamp();
  storeData        = new StoreData(BUFFER_FILE_NAME);
  deleteData       = new DeleteData(BUFFER_FILE_NAME);
```

```
    retrieveData        = new RetrieveData(BUFFER_FILE_NAME);
    concentratorID      = new ConcentratorID();
    sensorStamp          = new SensorStamp();

    armingSwitch.initializeSwitch("open");
    resetSwitch.initializeSwitch("open");
    fireSwitch.initializeSwitch("open");
  }//init

  public static void createTiniDataPoint() throws Exception
  {
    onePoint = new TiniDataPoint();
    onePoint.setDate(time_dateStamp.getTotalDateString());
    onePoint.setTime(time_dateStamp.getTotalTimeString());
    onePoint.setValue(result_voltage);
    onePoint.setUnits(processData.getUnits(a2dConverter_address));
    onePoint.setSourceID(sensorStamp.getSensorStamp());
    onePoint.setChannel(captureData.getChannel(RSLT_VLT_CHNL));
    onePoint.setTiniID(concentratorID.getConcentratorID());
    onePoint.setSoftwareVersion(1.1);     //THIS IS HARDCODED IN AND SHOULD
BE FIXED
    onePoint.setTestTypeID(testTypeID);
    onePoint.setSequenceNumberString(sequenceNumberString);
    onePoint.setGroupNumber(groupNumber);
    System.out.println("Created data point");
    vectorDataPoints.add(onePoint);
    System.out.println("Added data point to vector of data points");
  }//createTiniDataPoint()

  private static void initSwitches() throws Exception
  {
    armingSwitch.initializeSwitch("open");
    resetSwitch.initializeSwitch("open");
    fireSwitch.initializeSwitch("open");
    armingSwitch.initializeSwitch("closed");
  }//initSwitches()
} //Concentrator
```

F.2 CaptureRawData.java

F.2.1 Code Description

This program retrieves raw data from the DS2450 (A/D converter) and returns this data point to **Concentrator.java**. It also returns the DS2450's channel letter associated with the passed channel number. Available routines include:

- retrieveData: Retrieves raw data from the DS2450 using the DS2450Lib class.
- getChannel: Returns the alphabetic representation of the passed channel number.

F.2.2 Actual Code

```
/*******************************************************************
 *
 *        Capture Raw Data:
 *
 *        Author:  Michael T. DePlonty
 *
 *        Date:    November 20, 2000
 *
 *******************************************************************/
import com.ibutton.iButtonException;
import com.ibutton.adapter.*;
import com.ibutton.utils.Address;
import com.dalsemi.tininet.TININet;

class CaptureRawData
{
  private      DS2450Lib          adConv;

  CaptureRawData(String passedib20Address, int passedTestTypeID)
  {
    adConv = new DS2450Lib(passedib20Address, passedTestTypeID);
  }//CaptureRawData
  public int retrieveData(int resultChannel)
             throws iButtonException, OneWireIOException
  {
    return adConv.getRawData(resultChannel);
  }//retrieveData()

  public String getChannel(int channelNumber)
  {
    return adConv.getChannel(channelNumber);
  }//getChannel(int channelNumber)

  public void cleanUpObjects()
  {
    adConv = null;
  }//cleanUpObjects()
} //CaptureRawData
```

F.3 ConcentratorID.java

F.3.1 Code Description

This program returns TINI's Ethernet address as a string and is set up to return a detailed description of TINI's responsibilities. Available routines include:

• stampData:	Places TINI's Ethernet address into the passed data vector, effectively stamping the data.
• getConcentratorID:	Retrieves and returns TINI's Ethernet address.
• getDetailedDescription:	Returns a vector containing two elements. One, all the one-wire devices TINI uses. Two, a hardcoded description of TINI's role in the DSI test.

F.3.2 Actual Code

```
/*******************************************************************
 *
 *        ConcentratorID:
 *
 *        Author:  Michael T. DePlonty
 *
 *        Date:    November 21, 2000
 *
 *******************************************************************/
import com.dalsemi.tininet.TININet;
import java.util.Vector;

class ConcentratorID
{
  private Vector description;
  private FindiButtonsConsoleCon findiButtons;

  ConcentratorID()
  {

  }//ConcentratorID()

  public Vector stampData(Vector data)
  {
    data.addElement(TININet.getEthernetAddress());
    return data;
  }//stampData(Vector data)

  public String getConcentratorID()
  {
    return TININet.getEthernetAddress();
  }//getConcentratorID()

/*******************************************************************
 *
 *  Returns a detailed description of the concentrator's responsibilities.  Includes what
 *  sensors it receives data from and what controllers it controls.  Also includes a
 *  summary of what it does.  For the DSI application, this summary would be
 *  "Measure the dielectric strength of a transformer's oil."
 *
 *******************************************************************/
```

```
  public Vector getDetailedDescription()
  {
    description = new Vector();
    findiButtons = new FindiButtonsConsoleCon();
    description = findiButtons.listiButtons();
    description.addElement("Measure the dielectric strength of a " +
                    "transformer's tap changer oil.");

    return description;
  }//getDetailedDescription()

  public void cleanUpObjects()
  {
    description = null;
    findiButtons = null;
  }//cleanUpObjects()
} //ConcentratorID
```

F.4 ConfigureTest.java

F.4.1 Code Description

This program reads **concentrator.cfg** and stores the results in the class variables. It also provides access methods that allow other classes to retrieve the results of reading the configuration file. Available routines include:

- ConfigureTest: Constructor that reads the configuration file.
- readValues: Initializes the class variables with the results from ConfigureTest.
- writeValues: Writes to the configuration file.
- Routines to access each of the class variables.

F.4.2 Actual Code

```
/*******************************************************************
 *
 *        ConfigureTest:
 *
 *        Author:  Michael T. DePlonty
 *
 *        Date:    November 20, 2000
 *
 *******************************************************************/
import com.ibutton.adapter.*;
import java.io.*;
import com.ibutton.iButtonException;
import com.dalsemi.system.TINIOS;

class ConfigureTest
{
  private static String ARM_HV_ADDRESS;
  private static String RESET_SWITCH_ADDRESS;
  private static String FIRE_DSI_ADDRESS;
  private static String A2D_CONVERTER_ADDRESS;

  private static LoadConfigFile   loadConfig;
  private static String           configFileName;
  private static long             sequenceNumber;
  private static long             startPosition;

  private static int              testTypeID;
  private static int              resultChannel;
  private static int              num_shots_limit;
  private static int              num_seq_limit_new;
  private static int              shotRestTime;
  private static int              numAddresses;
  private static int              time_between_sequences;
  private static int              num_seq_limit;
  private static int              num_config_values;
  private static String[]         configValues;

  public ConfigureTest()
  {
    configFileName = "concentrator.cfg";
    loadConfig = new LoadConfigFile(configFileName);
    loadConfig.openConfigFile();
    loadConfig.read_number_of_lines();
    num_config_values = loadConfig.get_number_of_lines();
    System.out.println("Trying to initialize configValues array.");
    System.out.println("Number of lines = " + num_config_values);
    configValues = new String [num_config_values];

    System.out.println("Reading config values.");
    configValues = loadConfig.readFile();

    System.out.println("Length of configValues array = " + configValues.length);

    System.out.println("Closing configuration file");
    loadConfig.closeInputFile();
    startPosition = loadConfig.getTallyOfBytes();
  }//ConfigureTest()

  public void readValues() throws Exception, iButtonException, OneWireIOException
  {

      System.out.println("Stuff that is passed:");
      for (int i=0; i<configValues.length; i++)
        System.out.print(configValues[i] + " ");
      System.out.println(" ");

      numAddresses = loadConfig.convertStringToInteger(configValues[0]);

      //get address of arm HV switch
      ARM_HV_ADDRESS = configValues[1];

      //get address of the reset switch
      RESET_SWITCH_ADDRESS = configValues[2];

      //get address of the fire switch
      FIRE_DSI_ADDRESS = configValues[3];

      //get address of the A/D converter
      A2D_CONVERTER_ADDRESS = configValues[4];

      //get int value of the test type ID
      testTypeID = loadConfig.convertStringToInteger(configValues[numAddresses+1]);

      //how many shots/groups we are to take (a loop constraint)
      num_shots_limit =
loadConfig.convertStringToInteger(configValues[numAddresses+2]);
```

```
        //get int value of the channel the result will appear on
        resultChannel =
loadConfig.convertStringToInteger(configValues[numAddresses+3]);

        //how long we sleep between shots (in seconds)
        shotRestTime =
loadConfig.convertStringToInteger(configValues[numAddresses+4]);

        //get time to sleep between sequences
        time_between_sequences =
loadConfig.convertStringToInteger(configValues[numAddresses+5]);

        //get number of sequences to run
        num_seq_limit =
loadConfig.convertStringToInteger(configValues[numAddresses+6]);

        //get the sequence number
        sequenceNumber =
loadConfig.convertStringToInteger(configValues[configValues.length-1]);

  }//readValues()

  public static void writeValues(long passed_sequenceNumber)
  {
    sequenceNumber = passed_sequenceNumber;
    System.out.println("Opening the configuration file");
    loadConfig = new LoadConfigFile(configFileName);
    System.out.println("Opening the configuration file");
    loadConfig.openConfigFile();
    System.out.println("Writing to configuration file");
    loadConfig.writeToFile(Long.toString(sequenceNumber), startPosition);
    System.out.println("Closing the configuration file");
    loadConfig.closeInputFile();
  }//writeValues()

  public String getARM_HV_ADDRESS()
  {
    return ARM_HV_ADDRESS;
  }//getARM_HV_ADDRESS()

  public String getRESET_SWITCH_ADDRESS()
  {
    return RESET_SWITCH_ADDRESS;
  }//getRESET_SWITCH_ADDRESS()
  public String getFIRE_DSI_ADDRESS()
  {
    return FIRE_DSI_ADDRESS;
  }//getFIRE_DSI_ADDRESS()

  public String getA2D_CONVERTER_ADDRESS()
  {
    return A2D_CONVERTER_ADDRESS;
  }//getA2D_CONVERTER_ADDRESS()

  public long get_sequenceNumber()
  {
    return sequenceNumber;
  }//get_sequenceNumber()

  public int get_testTypeID()
  {
    return testTypeID;
  }//get_testTypeID()

  public int get_resultChannel()
  {
    return resultChannel;
  }//get_resultChannel()

  public int get_num_shots_limit()
  {
    return num_shots_limit;
  }//get_num_shots_limit()

  public int get_shotRestTime()
  {
    return shotRestTime;
  }//get_shotRestTime()

  public int get_time_between_sequences()
  {
    return time_between_sequences;
  }//get_time_between_sequences()

  public int get_num_seq_limit()
  {
    return num_seq_limit;
  }//get_num_seq_limit()
} //ConfigureTest
```

F.5 DeleteData.java

F.5.1 Code Description

This program provides routines to delete data from the buffer file. It assumes the buffer file has one line for each data point. With this assumption, this program can delete specific lines from a specified file. Available routines include:

• deleteAllPoints:	Deletes the entire buffer file.
• deleteFirstPoint:	Deletes the first line in the buffer file.
• deleteLastPoint:	Deletes the last line in the buffer file.
• deleteSpecificPoint:	Deletes the specified line and moves all subsequent lines up by one.

F.5.2 Actual Code

```
/******************************************************************
 *
 *        DeleteData:
 *
 *        Author:  Michael T. DePlonty
 *
 *        Date:    November 24, 2000
 *
 ******************************************************************/
import java.io.*;

class DeleteData
{
  private RetrieveData     retrieveData;
  private StoreData        storeData;
  private File             outputFile;
  private String           fileName;

  DeleteData()
  {
    this("buffer.txt");
  }//DeleteData()
```

```
DeleteData(String passedFileName)
{
  fileName = passedFileName;
}//DeleteData(String passedFileName)

public void deleteAllPoints()
{
  outputFile = new File(fileName);
  outputFile.delete();
}//deleteFile()

public void deleteFirstPoint()
{
  retrieveData = new RetrieveData(fileName);
  storeData    = new StoreData(fileName);
  String[] result = retrieveData.getAllPoints();
  deleteAllPoints();
  System.out.println("reult length = " + result.length);
  for (int i = 1; i < result.length; i++)
  {
    System.out.println(i);
    storeData.writeData(result[i], true);
  }//for  //store all but the first data point.
}//deleteFirstPoint()

public void deleteLastPoint()
{
  retrieveData = new RetrieveData(fileName);
  storeData    = new StoreData(fileName);
  String[] result = retrieveData.getAllPoints();
  deleteAllPoints();
  for (int i=0; i < (result.length - 1); i++)
  {
    storeData.writeData(result[i], true);
  }//for  //store all but the last data point.
}//deleteLastPoint()

public void deleteSpecificPoint(int pointToDelete)
{
  retrieveData = new RetrieveData(fileName);
  storeData    = new StoreData(fileName);
  String[] result = retrieveData.getAllPoints();

  if (pointToDelete < result.length)
  {
    deleteAllPoints();
```

```
      int index = 0;
      while (index != pointToDelete)
      {
        storeData.writeData(result[index], true);
        ++index;
      }//while we read the points prior to the point to delete

      for (int i=(pointToDelete + 1); i < result.length; i++)
      {
        storeData.writeData(result[i], true);
      }//for  //store the rest of the points.
    }//if want to delete a point index in the buffer length

    else //point index beyond buffer length
    {
      //currently, do nothing
    }//else
  }//deleteSpecificPoint(int pointToDelete)

public void cleanUpObjects()
  {
    storeData      = null;
    retrieveData   = null;
    outputFile     = null;
    fileName       = null;
  }//cleanUpObjects()
} //DeleteData
```

F.6 DS2450Lib.java

F.6.1 Code Description

This program provides a device library for the DS2450 (A/D converter). Available DS2450 actions include:

- getRawData: Digitize input signal, read result, then return the digitized value.
- convertToStandardUnits: Convert passed raw data from microVolts to Volts.
- unitsLabel: Returns units associated with processed data, which for the DSI test is Volts.

• getSensorID:	Returns DS2450's one-wire address.
• getChannel:	Returns DS2450's representation of the passed channel number.
• getTestTypeID:	Returns the test type ID initialized when this class was instantiated.
• channelNumberToString:	Converts the numeric representation of the DS2450 channel to a alphabetic representation found on the data sheets.

F.6.2 Actual Code

```
/*********************************************************************
 *
 *         DS2450Lib:
 *
 *         Author:  Michael T. DePlonty
 *
 *         Date:   November 24, 2000
 *
 *********************************************************************/
import com.ibutton.utils.Address;
import com.ibutton.iButtonException;
import com.ibutton.adapter.*;

class DS2450Lib
{
  private      iButtonContainer20   ib20;

  private      int[]            rawData;

  private static int             testTypeID;
  private      double            convertedData;
  private      String            channelString;

  DS2450Lib()
  {
  }//DS2450Lib()
```

```
DS2450Lib(String passedib20Address)
{
  ib20 = new iButtonContainer20(passedib20Address);
}//DS2450Lib(String passedib20Address)

DS2450Lib(String passedib20Address, int passedTestTypeID)
{
  testTypeID    = passedTestTypeID;
  ib20 = new iButtonContainer20(passedib20Address);
       //represents a container for the DS2450 A/D converter
}//DS2450Lib(String passedib20Address, passedTestTypeID)

public int getRawData(int resultChannel)
           throws iButtonException, OneWireIOException
{
  ib20.writeToMemory(resultChannel, 1);  //write channel's data to page 1
  ib20.convert();
  rawData = ib20.readConvertedData(resultChannel);
  return rawData[resultChannel];
}//getRawData(int resultChannel)

public double convertToStandardUnits(int passedRawData)
{
  convertedData = (double) passedRawData/1000000;
  return convertedData;
}//convertToStandardUnits(int passedRawData)

/***********************************************************************
 *
 *        unitsLabel():  This routine simply returns the units the DS2450
 *                       measurements occur in.
 *
 ***********************************************************************/

public String unitsLabel()
{
  return "Volts";
}//unitsLabel()

public String getSensorID()
{
  return Address.toString(ib20.getROMId());  //get DS2450 A/D converter unique ID
}//getSensorID()
```

```
 public String getChannel(int passedChannelNumber)
 {
   channelNumberToString(passedChannelNumber);
   return channelString;
 }//getChannel(int passedChannelNumber)

 public int getTestTypeID()
 {
   return testTypeID;
 }//getTestTypeID()

/*********************************************************************
 *
 * channelNumberToString(int passedChannel) converts the numeric representation
 *    of the DS2450 channel to a alphabetic representation found on the data sheets.
 *
 *********************************************************************/

 public void channelNumberToString(int passedChannel)
 {
   switch (passedChannel)
   {
     case 0:
       channelString = "A";
       break;

     case 1:
       channelString = "B";
       break;

     case 2:
       channelString = "C";
       break;

     case 3:
       channelString = "D";
       break;

     default:
       channelString = "INVALID";
   }//switch
 }//channelNumberToString(int passedChannel)
```

```
  public void cleanUpObjects()
  {
    ib20 = null;
  }//cleanUpObjects()

} //DS2450Lib
```

F.7 FaultHandler.java

F.7.1 Code Description

This program handles all faults that are passed to it by other classes. Available actions include:

- handleFault: Reads the number of previous faults that were generated. If the number is less than twenty, this routine writes the passed error message to a log file **ErrorLog.txt** then reboots TINI. If the number of previous faults is greater than or equal to twenty, the routine first deletes the log file, writes the passed error message to the log file, and then reboots TINI.

F.7.2 Actual Code

```
/******************************************************************
 *
 *        FaultHandler:
 *
 *        Author:  Michael T. DePlonty
 *
 *        Date:    November 30, 2000
 *
 ******************************************************************/
import com.dalsemi.system.TINIOS;

class FaultHandler
{
  private static final int num_Fault_limit  =  20;
  public StoreData        faultNumFileStore;
  public StoreData        errorLogFile;
  public RetrieveData      faultNumFileRetrieve;
  public DeleteData        errorLogFileDelete;

  FaultHandler()
  {
    faultNumFileStore      = new StoreData("faultnum.cfg");
    errorLogFile          = new StoreData("ErrorLog.txt");
    faultNumFileRetrieve   = new RetrieveData("faultnum.cfg");
  }//FaultHandler()

  //read number of times error generated
  //if not too big, write error to log file and time and reboot
  //else start at top of file, write error to log file, and reboot

  public void handleFault(String fault)
  {
    System.out.println("Rebooting Tini.");
    System.out.println("Error, handling fault.");
    //produces a radix 10 integer from the string read from the file
    int num_Faults = Integer.parseInt(faultNumFileRetrieve.getFirstPoint());
    num_Faults++;

    if (num_Faults < num_Fault_limit)
    {
      errorLogFile.writeData(fault, true);
      errorLogFile = null;
    }//if we haven't exceded the number of faults limit
    else
    {
      num_Faults = 0;
      errorLogFileDelete = new DeleteData("ErrorLog.txt");
      errorLogFileDelete.deleteAllPoints(); //for now, just delete the entire file
                                            //and start over
      errorLogFileDelete = null;
      errorLogFile.writeData(fault, true);
      errorLogFile = null;
    }//else generated too many faults so start at the top of the file

    faultNumFileStore.writeData(Integer.toString(num_Faults), false);
    TINIOS.reboot();
  }//handleFault
} //FaultHandler
```

F.8 FindButtonsConsoleCon.java

F.8.1 Code Description

This program is a modified version of Dallas Semiconductor's FindiButtonsConsole.java. It determines and lists all of the one-wire devices connected on the one-wire bus. Available actions include:

- listiButtons: Returns a vector containing all the present one-wire devices. Each vector entry contains the devices name, which port it is connected on, its part name, its address, and its description.

F.8.2 Actual Code

```
// FindiButtonsConsoleCon.java
/*---------------------------------------------------------------------------
 * Copyright (C) 1998 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 java.util.*;
import com.ibutton.*;
import com.ibutton.adapter.*;
import com.ibutton.container.*;

//---------------------------------------------------------------------------
/** FindiButtonsConsole is a console application to view all of the iButtons
 *  on the currently available adapters.
 *
 *  @version    0.00, 1 January 1999
 *  @author     DS
 */
class FindiButtonsConsoleCon
{
  Vector presentiButtons;
  FindiButtonsConsoleCon()
  {

  }//FindiButtonsConsoleCon()

  public Vector listiButtons()
  {
    presentiButtons = new Vector();

    // enumerate through each of the adapter classes
    for(Enumeration adapter_enum = iButtonAccessProvider.enumerateAllAdapters();
                adapter_enum.hasMoreElements(); )
    {
      // get the next adapter DSPortAdapter
      DSPortAdapter adapter = (DSPortAdapter)adapter_enum.nextElement();

      // get the port names we can use and try to open, test and close each
      for(Enumeration port_name_enum = adapter.getPortNames();
                  port_name_enum.hasMoreElements(); )
      {
        // get the next packet
        String port_name = (String)port_name_enum.nextElement();

        /* You could check to make sure that it is a valid TINI port here */
```

```
            try
            {
              // select the port
              adapter.selectPort(port_name);

              // verify there is an adapter detected
              if (adapter.adapterDetected())
              {
                // clear any previous search restrictions
                adapter.setSearchAlliButtons();
                adapter.targetAllFamilies();

                // enumerate through all the iButtons found
                for(Enumeration ibutton_enum = adapter.getAlliButtons();
                            ibutton_enum.hasMoreElements(); )
                {
                  // get the next ibutton
                  iButtonContainer ibutton = (iButtonContainer)ibutton_enum.nextElement();
                  presentiButtons.addElement(adapter.getAdapterName() + "\t" +
                              port_name + "\t" +
                              ibutton.getiButtonPartName() + "\t" +
                              ibutton.getAddressAsString() + "\t" +
                              ibutton.getDescription().substring(0,25) +
                              "...");
                }//for enumerate through all iButtons found
              }//if adapter is detected
              // free this port
              adapter.endExclusive();
            }//try
            catch(Exception e) {};
          }//for opening all port names
        }//for enumerate through all adapter classes
          return presentiButtons;
    }//listiButtons()
}//FindiButtonsConsoleCon
```

F.9 iButtonContainer05.java

F.9.1 Code Description

This program is the device driver for the DS2405 addressable switch. Available actions include:

• findSensor:	Finds the first DS2405 on the one-wire bus, either looking for any switch or looking for a specified address. If the switch is found, this routine returns its one-wire address. Else it notifies the software that the switch was not found.
• toggleSwitch:	Causes the switch to change state, either from open to closed or from closed to open. Does not check the switch's current state.
• determineSwitchState:	Determines and returns the switch's current state, either open or closed.
• initializeSwitchState:	Initializes the switch to a defined state. Calling routine must specify if the switch is to be open or closed.

F.9.2 Actual Code

```
/*****************************************************************
 *
 *        iButtonContainer05 contains methods designed to communicate with the
 *        DS2405 addressable switch (family code 05 hex).
 *
 *        Author:  Michael T. DePlonty
 *
 *        Date:    August 24, 2000
 *
 *****************************************************************/
import com.ibutton.adapter.*;
import com.ibutton.container.*;
import com.ibutton.iButtonException;
import java.lang.*;

class iButtonContainer05 extends iButtonContainer
{
  private static final byte DS2405_FAMILY_CODE     = (byte) 0x05;

  private static final byte MATCH_ROM              = (byte) 0x55;
  private static final byte SEARCH_ROM             = (byte) 0xF0;
  private static final byte CONDITIONAL_SEARCH     = (byte) 0xEC;
private TINIAdapter ds2405;
private byte[] romId;
private byte[] block = new byte[9];
private byte[] switch_state = new byte[5];

public iButtonContainer05()
{
  ds2405 = new TINIExternalAdapter();
  findSensor();
}//iButtonContainer05()

public iButtonContainer05(String address)
       throws iButtonException, OneWireIOException
{
  ds2405 = new TINIExternalAdapter();
  findSensor(address);
}//iButtonContainer05(String address)

public boolean findSensor()
{
  boolean foundDS2405 = false;
  romId = null;

  try
  {
    int resetStatus = ds2405.reset();
    if(resetStatus == 1)
    {
      if(ds2405.findFirstiButton())
      {
        boolean       first = true;
        iButtonContainer ib;

        do
        {
          if (first)
          {
            ib = ds2405.getFirstiButton();
            first = false;
          }//if(first)
          else
          {
            ib = ds2405.getNextiButton();
          }//else
          System.out.println(ib.getAddressAsString());
          romId = ib.getAddress();
          if(romId[0] == DS2405_FAMILY_CODE)
          {
            foundDS2405 = true;
            break;
          }//if(romID)
        }//do
        while(ds2405.findNextiButton());
      }//if(findFirstiButton())
    }//if(reset)
    else
      System.out.println("Still not ready, now leave me alone");
  }//try

  catch(Exception e)
  {
    e.printStackTrace();
    return false;
  }//catch
  return foundDS2405;
}//findSensor()

public boolean findSensor(String address)
                 throws iButtonException, OneWireIOException
{
  int resetStatus = ds2405.reset();
  boolean foundDS2405 = false;
  iButtonContainer ib;

  if(resetStatus == 1)
  {
    if (ds2405.isPresent(address))
    {
      ib = ds2405.getContainer(address);
      romId = ib.getAddress();
      System.out.println(ib.getAddressAsString());
      foundDS2405 = true;
    }//if
    else
    {
      System.out.println("Device with Specified address is not present.");
      foundDS2405 = false;
    }//else
  }//if(resetStatus == 1)
  if(resetStatus == 0)
  {
    System.out.println("DS2405 not present.");
    foundDS2405 = false;
  }//if
  if(resetStatus == 2)
  {
    System.out.println("DS2405 is in alarm status.");
    foundDS2405 = false;
  }//if
  return foundDS2405;
}//findSensor(String address)

public void toggleSwitch() throws iButtonException, OneWireIOException
{
  int resetStatus = ds2405.reset();
  if(resetStatus == 1)
  {
    block[0] = MATCH_ROM;
    System.arraycopy(romId, 0, block, 1, romId.length);
    ds2405.dataBlock(block, 0, 9);
  }//if
  if(resetStatus == 0)
      System.out.println("DS2405 not present.");
  if(resetStatus == 2)
      System.out.println("DS2405 is in alarm status.");
  ds2405.reset();

}//toggleSwitch()

public String determineSwitchState() throws iButtonException, OneWireIOException
{
  String switch_state_string = null;
  switch(switch_state[0])
  {
    case 0:
    {
      switch_state_string = "closed";
      break;
    }//case 0

    case -1:
    {
      switch_state_string = "open";
      break;
    }//case 1
    default:
    {
      switch_state_string = "invalid";
      break;
    }//else
  }//switch
  return switch_state_string;
}//determineSwitchState()

public void initializeSwitchState(String switch_state_desired)
              throws iButtonException, OneWireIOException
{
  System.out.println("entered initState");

  byte[] current_switch_state = new byte[1];
  boolean[] tempRomID = new boolean[2];
  byte buffer;

  if (switch_state_desired.compareTo("open") == 0)  //passed in "open"
    switch_state[0] = (byte) -1;
  else if (switch_state_desired.compareTo("closed") == 0)  //passed in "closed"
    switch_state[0] = (byte) 0;

  int resetStatus = ds2405.reset();
  if(resetStatus == 1)
  {
    block[0] = SEARCH_ROM;
    ds2405.putByte(block[0]);

    for (int j = 0; j < romId.length; j++)
    {
      for (int i = 0; i < 8; i++)
      {
        tempRomID[0] = ds2405.getBit();  //get bit
        tempRomID[1] = ds2405.getBit();  //get bit's complement
        buffer = (byte) (romId[j] >>> i);
        buffer = (byte) (buffer & (byte) 0x01);
        if (buffer == 0)
          ds2405.putBit(false);  // bit is a zero
        else
          ds2405.putBit(true);  // bit is a one
      }//for (i)
    }//for (j)
    ds2405.getBlock(current_switch_state, 0, 1);

    System.out.println("current switch state = " + current_switch_state[0]);
```

```
      if (current_switch_state[0] != switch_state[0])
        toggleSwitch();

    }//if
    if(resetStatus == 0)
        System.out.println("DS2405 not present.");
    if(resetStatus == 2)
        System.out.println("DS2405 is in alarm status.");
    ds2405.reset();

/*****************************************************************
 determine what the switch should be set at
 determine what the switch is currently set at
 if current and should are equal, do nothing
 else toggle switch
******************************************************************/
    current_switch_state = null;
    tempRomID = null;
  }//initializeSwitchState()
} //iButtonContainer05
```

F.10 iButtonContainer20.java

F.10.1 Code Description

This program is the device driver for the DS2450 A/D converter. Available actions include:

• getROMId:	Returns DS2450's one-wire address.
• getiButtonPartName:	Returns the A/D converter's name in iButton terminology. Namely, returns "DS2450."
• getAlternateNames:	Returns colloquial name of DS2450.
• getDescription:	Returns description of the device, as hardcoded in.
• iButtonContainer20:	Instantiates an iButtonContainer20 object and initializes all important variables.
• init:	Initializes the memory array that corresponds to the DS2450's memory pages.

• findSensor:	Finds first DS2450 and returns its one-wire address.
• readMemory:	Reads the values stored in the DS2450's memory. It starts at a given address, reads that byte, reads the sequentially next byte, and stores the result in an array.
• writeMemory:	Writes a byte to the specified memory location, then writes the next byte in the sequentially next memory location. The data to write must first be stored in an array with the first byte stored as the array's first element and the second byte stored as the array's second element. Also must previously specify which channel the data is associated with.
• convert:	Digitizes the analog signal at the previously specified channel. The result is stored in the DS2450's memory.
• scaleValue16:	Converts the sixteen bit result of the A/D conversion into microVolts. The sixteen bit result must first be stored in the memory array. Returns an array with each element corresponding to a specific channel's result.
• getResolution:	Returns the DS2450's bit resolution used to perform the A/D conversion. This resolution is read from the memory array. Use readMemory to find out the resolution the DS2450 has in its memory.
• setResolution:	Sets the DS2450's bit resolution in the software's memory array. Use writeMemory to set the resolution at the DS2450.
• readConvertedData:	Uses readMemory and scaleValue16 to read a conversion result.

• makeAnalogOutputHigh:	Forces the specified input channel to behave as an output channel with a logic high as its output.
• makeAnalogOutputLow:	Forces the specified input channel to behave as an output channel with a logic low as its output.
• disableAnalogOutput:	Forces the specified channel that was behaving as an output channel to behave as an input channel.
• getInputVoltageRange:	Returns voltage range the DS2450 will tolerate as input, as stored in the memory array.
• setInputVoltageRange:	Sets the allowable input voltage range for a specified channel and stores this range in the software's memory array. Use writeMemory to make this specification on the DS2450.
• getuSecConversionTime:	Calculates and returns the number of microSeconds needed to perform an A/D conversion.
• setuSecConversionTime:	Sets the microSecond converstion time used for the specified channel.
• writeToMemory:	Writes the values of the memory array to the corresponding locations in the DS2450's memory.

F.10.2 Actual Code

```
/*******************************************************************
 *
 *      iButtonContainer20 contains methods designed to communicate with the
 *      DS2450 A/D converter (family code 20 hex).
 *
 *      Author:  Michael T. DePlonty
 *
 *      Date:    July 14, 2000
 *
 *******************************************************************/

import com.ibutton.adapter.*;
import com.ibutton.container.*;
import com.ibutton.utils.CRC16;
import com.ibutton.iButtonException;

class iButtonContainer20 extends iButtonContainer
{
  private static final byte DS2450_FAMILY_CODE          = (byte) 0x20;

  private static final byte READ_MEMORY                 = (byte) 0xAA;  //used to
read conversion results
  private static final byte WRITE_MEMORY                = (byte) 0x55;  //used to
write to memory pages 1 and 2 (to set channel control and alarm settings)
  private static final byte CONVERT                     = (byte) 0x3C;  //used to
start A/D conversion

  private static final byte READ_ROM                    = (byte) 0x33;
  private static final byte MATCH_ROM                   = (byte) 0x55;
  private static final byte SKIP_ROM                    = (byte) 0xCC;
  private static final byte SEARCH_ROM                  = (byte) 0xF0;
  private static final byte CONDITIONAL_SEARCH          = (byte) 0xEC;
  private static final byte OVERDRIVE_SKIP_ROM          = (byte) 0x3C;
  private static final byte OVERDRIVE_MATCH_ROM         = (byte) 0x69;

  private static final byte INPUT_RANGE_MASK            = (byte) 0xfe;
  private static final byte OUTPUT_CONTROL_MASK         = (byte) 0x3f;
  private static final byte RESOLUTION_MASK             = (byte) 0x0F;
  private static final byte ALARM_HIGH_MASK             = (byte) 0xdf;
  private static final byte ALARM_HIGH_ENABLE_MASK = (byte) 0xf7;
  private static final byte ALARM_LOW_MASK              = (byte) 0xef;
  private static final byte ALARM_LOW_ENABLE_MASK  = (byte) 0xfb;
  private static final byte POR_FLAG_MASK               = (byte) 0x7f;

  private static final byte OUTPUT_HIGH_ENABLE          = (byte) 0xC0;
  private static final byte OUTPUT_LOW_ENABLE           = (byte) 0x80;

  private static final byte[] [] CHANNEL_ADDRESSES      = new byte[3][8];

  private static final int BASE_CONVERT_TIME            = 160;   //max offset time
needed and
  private static final int CONVERT_TIME_PER_BIT         = 80;    //max time per bit
needed to do an A/D conversion (in micro seconds)
  private static final int[] UVOLT_RANGES               = {2560000, 5120000};

  private static int resolution;
  private static int uSecConversionTime;

  private int startChannel;
  private int startPage;
  private byte startAddress;
  private byte input_select_mask;
  private byte read_out_control_byte;
  private int[] scaledValue;
  private int range;          //range of voltages the DS2450 handles (in uVolts)
  private byte[] romId;
  private byte[] block = new byte[14];
  private byte[][] memory = new byte[4][10];

  private TINIAdapter ds2450;

/**********************************************************************
 *
 *  Informative methods, used to give user information about DS2450.
 *
 **********************************************************************/

  public byte[] getROMId()
  {
    return romId;
  }//getROMId()

  public String getiButtonPartName()
  {
    return new String ("DS2450");
  }//getiButtonPartName()

  public String getAlternateNames()
  {
    return new String ("1-wire Quad A/D Converter");
  }//getAlternateNames()

  public String getDescription()
  {
    return new String( "1-Wire Quad A/D Converter to measure four " +
                       "high impedence inputs with a user selectable " +
                       "range of 2.56V and 5.12V with a resolution " +
                       "of 1 to 16 bits.");
  }//getDescription()

/**********************************************************************
 *
 *  Initialization methods, used set-up iButton object.
 *
 **********************************************************************/
```

```
  public iButtonContainer20()
  {
    init();
    try
    {
      ds2450 = new TINIExternalAdapter();
      startAddress = CHANNEL_ADDRESSES[0][6]; //Arbitrarily chosen
      resolution = 8;                //Arbitrarily chosen 8 bit resolution
      range = (int) 5120000;              //range in uVolts to give us more precision
      startChannel = 0;               //Default start at Channel A
      startPage = 0;                 //Default start at page 0 of DS2450 memory, which is
the result of the A/D conversion
      input_select_mask = (byte) 0x08;       //select only channel D
      read_out_control_byte = (byte) 0x80;
      uSecConversionTime = BASE_CONVERT_TIME +
CONVERT_TIME_PER_BIT*resolution;
      scaledValue = new int[4];
      findSensor();
    }//try

    catch(Exception e)
    {
      e.printStackTrace();
    }//catch
  }//iButtonContainer20()

  public iButtonContainer20(String address)
  {
    init();
    try
    {
      ds2450 = new TINIExternalAdapter();
      startAddress = CHANNEL_ADDRESSES[0][6]; //Arbitrarily chosen
      resolution = 8;                //Arbitrarily chosen 8 bit resolution
      range = (int) 5120000;              //range in uVolts to give us more precision
      startChannel = 0;               //Default start at Channel A
      startPage = 0;                 //Default start at page 0 of DS2450 memory, which is
the result of the A/D conversion
      input_select_mask = (byte) 0x0E;       //select channels B, C, and D
      read_out_control_byte = (byte) 0x00;   //preset all channels
      uSecConversionTime = BASE_CONVERT_TIME +
CONVERT_TIME_PER_BIT*resolution;
      scaledValue = new int[4];
      findSensor(address);
    }//try
    catch(Exception e)
    {
      e.printStackTrace();
    }//catch
  }//iButtonContainer20()

  public void setupContainer(DSPortAdapter adapter, byte[] iBId)
  {
    super.setupContainer(adapter, iBId);
    init();
  }//setupContainer

  public void init()
  {
    //init memory array to power-on values corresponding to memory pages
    //in the DS2450's memory.

    for (int i = 0; i<8; i += 2)
    {
      memory[0][i]     = (byte) 0x00;
      memory[0][i+1] = (byte) 0x00;
      memory[1][i]     = (byte) 0x08;
      memory[1][i+1] = (byte) 0x8d;
      memory[2][i]     = (byte) 0x00;
      memory[2][i+1] = (byte) 0xff;
      memory[3][i]     = (byte) 0x00;
      memory[3][i+1] = (byte) 0x00;
/*********************************************************************
 *
 *  Now to initialize the Memory address numbers.
 *  General format is: CHANNEL_ADDRESSES[memory page][row number] = address
 *  Since the loop variable is incrementing by two, this initializes
 *  two rows per page.  And since there are eight rows per page, starting
 *  at zero, there is an offset of eight between pages.
 *
 *********************************************************************/

      CHANNEL_ADDRESSES[0][i]   = (byte) (0x00 + i);    //these are hex numbers
      CHANNEL_ADDRESSES[0][i+1] = (byte) (0x01 + i);    //not decimal numbers
      CHANNEL_ADDRESSES[1][i]   = (byte) (0x08 + i);
      CHANNEL_ADDRESSES[1][i+1] = (byte) (0x09 + i);
      CHANNEL_ADDRESSES[2][i]   = (byte) (0x10 + i);
      CHANNEL_ADDRESSES[2][i+1] = (byte) (0x11 + i);
    }//for
  //init CRCs
  memory[0][8]      = (byte) 0xdc;
  memory[0][9]      = (byte) 0x25;
  memory[1][8]      = (byte) 0x66;
  memory[1][9]      = (byte) 0xe8;
  memory[2][8]      = (byte) 0x94;
  memory[2][9]      = (byte) 0x94;
  memory[3][8]      = (byte) 0xff;
  memory[3][9]      = (byte) 0xff;
}//init

public boolean findSensor()
{
  boolean foundDS2450 = false;
  romId = null;

  try
  {
    int resetStatus = ds2450.reset();
    if(resetStatus == 1)
    {
      if(ds2450.findFirstiButton())
      {
        boolean         first = true;
        iButtonContainer ib;

        do
        {
          if (first)
          {
            ib = ds2450.getFirstiButton();
            first = false;
          }//if(first)
          else
            ib = ds2450.getNextiButton();

          romId = ib.getAddress();
          if(romId[0] == DS2450_FAMILY_CODE)
          {
            foundDS2450 = true;
            break;
          }//if(romID)
        }//do
        while(ds2450.findNextiButton());
      }//if(findFirstiButton())
    }//if(reset)
```

```
      else
        System.out.println("Still not ready, now leave me alone");
    }//try

    catch(Exception e)
    {
      e.printStackTrace();
      return false;
    }//catch
    return foundDS2450;
  }//findSensor()

  public boolean findSensor(String address) throws iButtonException,
OneWireIOException
  {
    int resetStatus = ds2450.reset();
    boolean foundDS2450 = false;
    iButtonContainer ib;

    if(resetStatus == 1)
    {
      if (ds2450.isPresent(address))
      {
        ib = ds2450.getContainer(address);
        romId = ib.getAddress();
        foundDS2450 = true;
      }//if
      else
      {
        System.out.println("Device with Specified address is not present.");
        foundDS2450 = false;
      }//else
    }//if(resetStatus == 1)
    if(resetStatus == 0)
    {
      System.out.println("DS2450 not present.");
      foundDS2450 = false;
    }//if
    if(resetStatus == 2)
    {
      System.out.println("DS2450 is in alarm status.");
      foundDS2450 = false;
    }//if
    return foundDS2450;
  }//findSensor(String address)
```

```
/**********************************************************************
 *
 *  readMemory(): Reads the values stored in the DS2450's memory.  This starts at a
 *             given address and reads two bytes of data, the byte at the given address and
 *             the byte at the next address (value of the address + 1).  These results are
 *             then stored in an array memory[][].
 *
 *             Needed to run this routine -
 *                  romId         - the unique ID of a DS2450
 *                  startAddress - the address of the first byte of
 *                                   data in memory to be read
 *                  startPage     - page of memory to start read at
 *                  startChannel - the channel the desired info corresponds to.
 *
 **********************************************************************/

 public void readMemory() throws iButtonException, OneWireIOException
 {
   byte[] data_buffer = new byte[10];
   int resetStatus = ds2450.reset();
   if(resetStatus == 1)
   {
     block[0] = MATCH_ROM;
     System.arraycopy(romId, 0, block, 1, romId.length);
     block[9] = READ_MEMORY;
     block[10] = startAddress;
     block[11] = (byte) 0x00;

     ds2450.dataBlock(block, 0, 12);
     ds2450.getBlock(data_buffer, 0, 2);

         //2 bytes of data
         //the offset option for this command does not work

     memory[startPage][2*startChannel] = data_buffer[0];
     memory[startPage][2*startChannel+1] = data_buffer[1];

         //2*startChannel since there are two rows per channel

   }//if
   if(resetStatus == 0)
       System.out.println("DS2450 not present (in readMemory).");
   if(resetStatus == 2)
       System.out.println("DS2450 is in alarm status (in readMemory).");
   ds2450.reset();
 }//readMemory()
```

```
/*********************************************************************
 *
 *  writeMemory():  Writes bytes to a specified memory location on the DS2450.
 *              The data to write needs to be stored in the memory array before
 *              running this rountine.  It then writes the two bytes associated with the
 *              desired channel.
 *
 *              Needed to run this routine -
 *                  romId        - the unique ID of a DS2450
 *                  startAddress - the address of the first byte of data in memory
 *                                  to be read
 *                  startPage     - page of memory to start read at
 *                  startChannel - the channel the desired info corresponds to.
 *
 *********************************************************************/

  public void writeMemory() throws iButtonException, OneWireIOException
  {
    byte[] data_buffer1 = new byte[10];
    byte[] data_buffer2 = new byte[3];
    int resetStatus = ds2450.reset();
    if(resetStatus == 1)              //device present
    {
      block[0] = MATCH_ROM;
      System.arraycopy(romId, 0, block, 1, romId.length);
      block[9] = WRITE_MEMORY;
      block[10] = startAddress;   //CHANNEL_ADDRESSES[1][2*startChannel];
      block[11] = (byte) 0x00;
      block[12] = memory[startPage][2*startChannel];  //data to write to the control
register
      block[13] = memory[startPage][2*startChannel+1];  //data to write to
D_Control_H_Address

                        //make highest possible range 5.10V
      ds2450.dataBlock(block, 0, 13);
      ds2450.getBlock(data_buffer1, 0, 3);  //2 bytes of CRC16, 1 byte of data written to
memory
      ds2450.putByte(block[13]);
      ds2450.getBlock(data_buffer2, 0, 3);  //2 bytes of CRC16, 1 byte of data written to
memory
      for(int index = 0; index <3; index++)
        data_buffer1[3+index] = data_buffer2[index];
    }//if
    if(resetStatus == 0)
        System.out.println("DS2450 not present (in writeMemory).");
  if(resetStatus == 2)
      System.out.println("DS2450 is in alarm status (in writeMemory).");
 }//writeMemory()

/*********************************************************************
 *
 *  convert(): Converts the analog data at the desired channel to digital data.
 *             The result is stored in the DS2450's memory.  To retrieve the result,
 *             call readMemory() routine.
 *
 *             Needed to run this routine    -
 *                  romId                    - the unique ID of a DS2450
 *                  input_select_mask        - selects which channel(s) participate in the
 *                                             conversion process
 *                  read_out_control_byte  - presets selected channels to all 0's or 1's
 *                                             or leaves as is.  Used to distinguish between
 *                                             previous result and new value.
 *
 *********************************************************************/

 public void convert() throws iButtonException, OneWireIOException
 {
   try
   {
     int resetStatus = ds2450.reset();
     if(resetStatus == 1)
     {
       block[0] = MATCH_ROM;
       System.arraycopy(romId, 0, block, 1, romId.length);
       block[9] = CONVERT;
       block[10] = input_select_mask;
       block[11] = read_out_control_byte;

       ds2450.dataBlock(block, 0, 12);
       ds2450.getBlock(block, 0, 2);       //receive CRC16

       ds2450.setPowerDuration(5); //deliver power till told to stop
       ds2450.startPowerDelivery(0); //deliver power now!

       Thread.sleep(1); //delay for 1 ms

       ds2450.setPowerNormal(); //turn off power delivery

       ds2450.reset();
     }//if
```

```
      if(resetStatus == 0)
          System.out.println("DS2450 not present (in convert).");
      if(resetStatus == 2)
          System.out.println("DS2450 is in alarm status (in convert).");
    }//try

    catch (InterruptedException e)
    {
      System.out.println("interrupted Exception");
      ds2450.setPowerNormal();
    }//catch
  }//convert

/*****************************************************************
 *
 *  scaleValue16(): Converts the 16 bit result of the A/D conversion into a "normal"
 *              result in microVolts.  The result of the A/D conversion must be stored
 *              in the memory array before running this routine.  Returns an integer array,
 *              with each array entry corresponding to a specific channel's result.
 *
 *              Needed to run this routine -
 *                  memory[][]   - array of data; each element corresponds to an
 *                              element on a specific memory map page found
 *                              on the DS2450.
 *                  resolution   - bit resolution of the A/D conversion.
 *                  range        - output range of the DS2450, in microVolts.
 *                              Either 2550000 or 5100000 microVolts.
 *                  startChannel - the channel the desired info corresponds to.
 *
 *****************************************************************/

  public int[] scaleValue16()
  {
    int value16 = ((memory[0][2*startChannel+1] & 0xff) << 8)
             + (memory[0][2*startChannel] & 0xff);
    scaledValue[startChannel] = value16 >>> (16 - resolution);
    scaledValue[startChannel] = (int)(((long)scaledValue[startChannel]
                      * range) / (1 << resolution));
    return scaledValue;
  }//scaleValue16
```

```
/*********************************************************************
 *
 *  The following two routines retrieve current A/D resolution and allows the resolution
 *     to be set.  IMPORTANT, if 16 bit resolution is desired, then resolutionToSet
 *     must equal 0.
 *
 ********************************************************************/

/*********************************************************************
 *
 *  getResolution():  Returns the bit resolution used to make the A/D conversion as
 *                 stored in the memory[][] array.  To get the resolution stored on the
 *                 DS2450, readMemory() needs to be called.
 *
 *                 Needed to run this routine -
 *                      channel          - desired channel number
 *
 ********************************************************************/

  public int getResolution(int channel)
  {
    resolution = memory[1][2*channel] & RESOLUTION_MASK;
    if( resolution == 0)
    {
      resolution = 16;
    }//if
    return resolution;
  }//getResolution()

/*********************************************************************
 *
 *  setResolution():  Sets the bit resolution used to make the A/D conversion, stored
 *                 in the memory[][] array.  To set the resolution on the DS2450,
 *                 writeMemory() needs to be called.
 *
 *                 Needed to run this routine -
 *                      channel          -  the channel on the DS2450 who's
 *                                          resolution that is being set
 *
 *                      resolutionToSet   -  the value the resolution will be set at
 *
 ********************************************************************/

  public void setResolution(int channel, int resolutionToSet)
  {
    if (resolutionToSet == 16)
    resolutionToSet = 0;
   memory[1][2*channel] = (byte) ((memory[1][2*channel] & (byte) 0xF0)
                          | (resolutionToSet & RESOLUTION_MASK));
  }//setResolution()

/*********************************************************************
 *
 *  Buffer to give user access to converted data stored on DS2450.
 *
 *        Channel          Associated Integer
 *          A                  0
 *          B                  1
 *          C                  2
 *          D                  3
 *
 *********************************************************************/

  public int[] readConvertedData(int channel)
              throws iButtonException, OneWireIOException
  {
   startChannel = channel;
   startPage = 0;                  //read data on memory page 0
                               //hardcoded since the results always appear there.
   startAddress = CHANNEL_ADDRESSES[startPage][2*startChannel];
                               //Multiply channel by 2 since
                               //there are two rows per channel
   readMemory();                   //Read the DS2450's memory and store in array
   scaleValue16();                 //Convert result to useful number
   return scaledValue;             //pass back the usefull number

  }//readConvertedData()

/*********************************************************************
 *
 *  The following three methods control the four channels' output capabilities.  The first
 *    two cause the DS2450 input channels to act as outputs.  IMPORTANT, the routine
 *    getResolution() must have been called sometime before these methods are called
 *    else incorrect data will be written to the memory registers.  The third method
 *    restores the channels to their regular input capabilities.
 *
 *********************************************************************/

  public void makeAnalogOutputHigh(int channel)
          throws iButtonException, OneWireIOException
  {
   startChannel = channel;
```

```
    startPage    = 1;    //this will always be memory page 1
    startAddress = CHANNEL_ADDRESSES[startPage][2*startChannel];
    memory[startPage][2*channel] = (byte) (OUTPUT_HIGH_ENABLE | resolution);
    writeMemory();
  }//makeAnalogOutputHigh()

  public void makeAnalogOutputLow(int channel)
          throws iButtonException, OneWireIOException
  {
    startChannel = channel;
    startPage    = 1;    //this will always be memory page 1
    startAddress = CHANNEL_ADDRESSES[startPage][2*startChannel];
    memory[startPage][2*channel] = (byte) (OUTPUT_LOW_ENABLE | resolution);
    writeMemory();
  }//makeAnalogOutputLow()

  public void disableAnalogOutput(int channel)
          throws iButtonException, OneWireIOException
  {
    startChannel = channel;
    startPage    = 1;    //this will always be memory page 1
    startAddress = CHANNEL_ADDRESSES[startPage][2*startChannel];
    memory[startPage][2*channel] = (byte) (0x00 | resolution);
    writeMemory();
  }//disableAnalogOutput()

/*********************************************************************
 *
 *  getInputVoltageRange():
 *
 *********************************************************************/

  public int getInputVoltageRange(int channel)
  {
    return memory[1][2*channel + 1] & (byte) 0x01;
  }//getInputVoltageRange(int channel)

/*********************************************************************
 *
 *  setInputVoltageRange():
 *
 *********************************************************************/

  public void setInputVoltageRange(int channel, int rangeToSet)
  {
    memory[1][2*channel + 1] =
            (byte) ((memory[1][2*channel + 1] &
                  (byte) 0x00) | (rangeToSet & 0x01));
  }//setInputVoltageRange()

/***********************************************************************
 *
 *  getuSecConversionTime() calculates and returns the number of microSeconds needed
 *     to perform an A/D conversion.
 *
 ***********************************************************************/

  public int getuSecConversionTime()
  {
    return uSecConversionTime;
  }//getuSecConversionTime()

  public void setuSecConversionTime(int channel)
  {
    uSecConversionTime = BASE_CONVERT_TIME +
CONVERT_TIME_PER_BIT*resolution;
  }//setuSecConversionTime()

/***********************************************************************
 *
 *  writeToMemory():  Sets up necessary variables before calling writeMemory().  The
 *                data to be written must be stored in the memory[][] array before
 *                calling this routine.
 *
 *                Needed to run this routine -
 *                   channel    - channel corresponding to the data that is to be written
 *                   page       - specifies which memory page to write to
 *
 ***********************************************************************/

  public void writeToMemory(int channel, int page)
              throws iButtonException, OneWireIOException
  {
    startChannel = channel;
    startPage = page;
    startAddress = CHANNEL_ADDRESSES[startPage][2*startChannel];  //memory
address to start writing at
    writeMemory();
  }//writeMemoryBuffer
} //iButtonContainer20
```

F.11 ImplementPWorldCmds.java

F.11.1 Code Description

This program accepts physical world control commands and then operates the DS2405 switch accordingly. Available actions include:

• ImplementPWorldCmds:	Constructor that creates an object of the switch's device driver. The switch's address must be specified.
• toggleSwitch:	Causes switch specified in ImplementPWorldCmds above to switch state from open to closed or from closed to open.
• initializeSwitch:	Forces switch specified in ImplementPWorldCmds above to be in a specified state. Must pass in switch state, either open or closed.

F.11.2 Actual Code

```
/*******************************************************************
 *
 *      ImplementPWorldCmds: Accepts commands from the concentrator and
 *      converts them to controller instructions. For the DSI test, the only controllers
 *      are the iButton switches.
 *
 *      Author: Michael T. DePlonty
 *
 *      Date:   November 30, 2000
 *
 *******************************************************************/

class ImplementPWorldCmds
{
  private iButtonContainer05 switch2toggle;

  ImplementPWorldCmds(String address)
  {
    try
    {
      switch2toggle = new iButtonContainer05(address);
    }//try

    catch (Exception e)
    {
      System.out.println("Exception in ImplementPWorldCmds(String address)");
      System.out.println(e.toString());
    }//catch
  }//ImplementPWorldCmds()

/*******************************************************************
 *
 *    toggleSwitch(String address, int numToggles):
 *      Toggles the DS2405 iButton a specified number of times.
 *
 *      address: the address of the DS2405 iButton
 *      numToggles: the number of times the switch should toggle
 *
 *******************************************************************/

  public void toggleSwitch(int numToggles)
  {
    try
    {
      for (int index = 0; index < numToggles; index++)
      {
        switch2toggle.toggleSwitch();
      }//for
    }//try

    catch (Exception e)
    {
      System.out.println("Exception in ImplementPWorldCmds.");
      System.out.println(e.toString());
    }//catch

    System.out.println("Toggled " + numToggles + " times.");

  }//toggleSwitch(int numToggles)
  public void initializeSwitch(String switchState)
  {
    try
    {
      switch2toggle.initializeSwitchState(switchState);
    }//try

    catch (Exception e)
    {
      System.out.println("Exception in initializeSwitch(String switchState)");
      System.out.println(e.toString());
    }//catch
  }//initializeSwitch(String switchState)
} //ImplementPWorldCmds
```

F.12 LoadConfigFile.java

F.12.1 Code Description

This program reads a configuration file and passes results in an array. Elements in the array are ordered in the order they appear in the configuration file. It also provides a method to write to the configuration file at a specified location. Available actions include:

• LoadConfigFile:	Constructor that initializes the file name.
• openConfigFile:	Opens the configuration file.
• closeInputFile:	Closes the configuration file.
• readFile:	Reads the configuration file and stores results into an array.
• convertStringToInteger:	Converts passed string into an integer.
• writeToFile:	Writes passed data to the specified file location. This location represents the number of bytes in the file the software will skip over.
• getTallyOfBytes:	Returns the total number of bytes this class has read from the configuration file.

- read_number_of_lines: Determines the number of uncommented lines in the configuration file and stores the result in a class variable.
- get_number_of_lines: Returns the number of uncommented lines in the configuration file. read_number_of_lines determines this number.

F.12.2 Actual Code

```
/*******************************************************************
 *
 *       LoadConfigFile: Reads a configuration file and passes results in an array.
 *       Elements in array are ordered in the order they appear in the configuration file.
 *       Also provides a method to write to the configuration file at a specified location.
 *
 *       Author: Michael T. DePlonty
 *
 *       Date:   August 23, 2000
 *
 *******************************************************************/
import java.io.*;

class LoadConfigFile
{
  private String [] data;
  private String fileName;
  private RandomAccessFile configFile;
  private long tallyOfBytes;
  private int number_of_lines;

  public LoadConfigFile()
  {
    tallyOfBytes = (long) 0;
    fileName = "test1.cfg";
  }//LoadConfigFile()

  public LoadConfigFile(String passedFileName)
  {
    tallyOfBytes = (long) 0;
    fileName = passedFileName;
  }//LoadConfigFile(String passedFileName)
```

```
public void openConfigFile()
{
  try
  {
    System.out.println("In the open file routine");
    configFile = new RandomAccessFile(fileName, "rw");
    System.out.println("configFile length = " + configFile.length());
  }//try
  catch (IOException exception)
  {
    System.out.println("IO error");
  }//catch
}//openInputFile()

public void closeInputFile()
{
  try
  {
    configFile.close();
  }//try
  catch (IOException exception)
  {
    System.out.println("IO error");
  }//catch
}//closeInputFile()

public String[] readFile()
{
  String dataString = null;
  int index = 0;
  int fileLinePointer = 0;
  int testSeqLnNumber = 0;
  long runningTally_of_Bytes = (long) 0;
  long temp_runningTally_of_Bytes = (long) 0;
  number_of_lines = 0;  //represents the number of uncommented lines in the
configuration file

  try
  {
    configFile.seek(0);  //start at the begining of the file
    while (true)
    {
      dataString = configFile.readLine();
      if (dataString == null)       //no more left, reached end of the file
      {
        break;
        }//if
        temp_runningTally_of_Bytes = (long) dataString.length() + 2; //2 for carriage
return and line feed
        runningTally_of_Bytes = runningTally_of_Bytes + temp_runningTally_of_Bytes;
        dataString = dataString.trim();
        if (dataString.startsWith("#"))
        {
          continue;     //we don't care what follows since it is commented out
        }//if
        else
        {
          ++fileLinePointer;
          if (fileLinePointer == 1) //we are reading the first two lines and wish to
configure the length of the array
          {
            System.out.println("Reading the number of lines.");
            number_of_lines = convertStringToInteger(dataString);
            System.out.println("Number of lines = " + number_of_lines);
            data = new String[number_of_lines];
            testSeqLnNumber = number_of_lines+1;
          }//if (fileLinePointer == 1)
          else if (fileLinePointer == testSeqLnNumber) //we have read the test sequence
number and reached the end of the file
          {
            System.out.println("Reading the sequence number");
            tallyOfBytes = runningTally_of_Bytes - temp_runningTally_of_Bytes;
            //subtract what we just read since we want to write over it
            data[index] = dataString;
            index++;
          }//else if
          else   //we are beyond the test sequence number
          {
            data[index] = dataString;
            index++;
          }//else we are beyond the test sequence number
        }//else
        dataString = null;
      }//while
      closeInputFile();
    }//try
    catch (IOException exception)
    {
      System.out.println("IO error");
    }//catch
    return data;
  }//readData()
  public static int convertStringToInteger(String passed_data)
  {
    Integer            bufferIntegerObj;

    bufferIntegerObj = Integer.valueOf(passed_data);
    return bufferIntegerObj.intValue();
  }//convertStringToInteger(String passed_data)

  public void writeToFile(String dataToWrite, long startPosition)
  {
    try
    {
      System.out.println("File Pointer = " + configFile.getFilePointer());
      configFile.seek(startPosition);  //skip to this position in the file
      System.out.println("start position = " + startPosition);
      configFile.writeBytes(dataToWrite);  //write the data
      System.out.println("made it here, and finished writing the data.");
    }//try
    catch (IOException exception)
    {
      System.out.println("IO error");
    }//catch

  }//writeToFile(String dataToWrite, long startPosition)

  public long getTallyOfBytes()
  {
    return tallyOfBytes;
  }//getTallyOfBytes()
  public void read_number_of_lines()
  {
    number_of_lines = 0;  //represents the number of uncommented lines in the
configuration file
    String dataString = null;
    int fileLinePointer = 0;
    try
    {
      System.out.println("File Pointer = " + configFile.getFilePointer());
    }//
    catch (IOException exception)
    {
      System.out.println("IO error in reading number of lines");
    }//catch
    try
    {
      while (true)
      {
        dataString = configFile.readLine();
        if (dataString == null)        //no more left, reached end of the file
        {
          closeInputFile();
          break;
        }//if
        dataString = dataString.trim();
        if (dataString.startsWith("#"))
        {
          dataString = null;
          continue;     //we don't care what follows since it is commented out
        }//if
        else
        {
          ++fileLinePointer;
          if (fileLinePointer == 1) //we are reading the first two lines and wish to
configure the length of the array
          {
            number_of_lines = convertStringToInteger(dataString);
            dataString = null;
            closeInputFile();
            break;
          }//if (fileLinePointer == 1)
          else //error and return a value that will cause an exception
          {
            number_of_lines = 0;
            dataString = null;
            closeInputFile();
            break;
          }//else we have an error
        }//else line did not begin with #
      }//while
    }//try

    catch (IOException exception)
    {
      System.out.println("IO error");
    }//catch
  }//read_number_of_lines()
  public int get_number_of_lines()
  {
    return number_of_lines;
  }//get_number_of_lines()
} //LoadConfigFile
```

F.13 PostDataLib.java

F.13.1 Code Description

This program is a library for posting data to a web page. Available actions include:

• PostDataLib:	Constructor that initializes the url that TINI will post to.
• postDataPoint:	Returns the result of posting the passed data point to the web page.
• buildQueryString:	Builds string that will be posted to the web page.
• postDataPoint_GET:	Posts passed data point to the web page.

F.13.2 Actual Code

```
/*******************************************************************
 *
 *        PostDataLib:
 *
 *        Author:  Will Johnson, Michael T. DePlonty
 *
 *        Date:    November 28, 2000
 *
 *******************************************************************/
import java.io.*;
import java.net.*;
import com.dalsemi.system.TINIOS;

public class PostDataLib
{
  private static String url2post;
```

```
public PostDataLib(String s) throws Exception
{
  url2post = s;
}//PostDataLib(String s)

public String postDataPoint(String date_stamp, String time_stamp, String data_val,
                    String data_unit, String sensor_id, String channel_id,
                    String concentrator_id, String sw_ver, String test_type_id,
                    String seq_num, String group_num) throws Exception
{
  return postDataPoint_GET(date_stamp, time_stamp, data_val, data_unit, sensor_id,
                    channel_id, concentrator_id, sw_ver, test_type_id,
                    seq_num, group_num);
}//postDataPoint(filler)

private String buildQueryString(String date_stamp, String time_stamp, String data_val,
                     String data_unit, String sensor_id, String channel_id,
                     String concentrator_id, String sw_ver, String test_type_id,
                     String seq_num, String group_num) throws Exception
{
  String s = "";
  s =     "date_stamp=" + URLEncoder.encode(date_stamp);
  s = s + "&time_stamp=" + URLEncoder.encode(time_stamp);
  s = s + "&data_val=" + URLEncoder.encode(data_val);
  s = s + "&data_unit=" + URLEncoder.encode(data_unit);
  s = s + "&source_id=" + URLEncoder.encode(sensor_id);
  s = s + "&channel_id=" + URLEncoder.encode(channel_id);
  s = s + "&tini_id=" + URLEncoder.encode(concentrator_id);
  s = s + "&sw_ver=" + URLEncoder.encode(sw_ver);
  s = s + "&test_type_id=" + URLEncoder.encode(test_type_id);
  s = s + "&session_num=" + URLEncoder.encode(seq_num);
  s = s + "&group_num=" + URLEncoder.encode(group_num);

  return s;
}//buildQueryString(filler)

private String postDataPoint_GET(String date_stamp, String time_stamp,
                      String data_val, String data_unit, String sensor_id,
                      String channel_id, String concentrator_id, String sw_ver,
                      String test_type_id, String seq_num, String group_num)
                      throws Exception
{
  String       inputLine;
  String       returnval = "";
  URL           url;
  URLConnection    urlConn;
  BufferedReader   in;

  try
  {
    System.out.println("Entered PostDataPoint_GET");

    url2post = url2post.concat("?");
    url2post = url2post.concat(buildQueryString(date_stamp, time_stamp, data_val,
                                   data_unit, sensor_id, channel_id,
                                   concentrator_id, sw_ver, test_type_id,
                                   seq_num, group_num));
    url = new URL(url2post);
    System.out.println("Trying to open URL connection.");
    urlConn = url.openConnection();

    System.out.println("Now trying to read from the Input Stream.");
    in = new BufferedReader(new InputStreamReader(urlConn.getInputStream()));

    System.out.println("Reading from the input stream");
    while ((inputLine = in.readLine()) != null)
    {
      System.out.println("inputLine = " + inputLine);
      returnval = returnval + inputLine;
    }//while

    System.out.println("Closing the buffered reader.");
    in.close();
    in = null;
    url = null;
    urlConn = null;
    inputLine = null;
    System.gc();
    return returnval;
  }//try

  catch (Exception e)
  {
    System.out.println(e.toString());
    System.out.println("Doing a system reboot");
    FaultHandler fh = new FaultHandler();
    fh.handleFault(e.toString());
    return returnval;
  }//catch
 }//postDataPoint_GET(filler)
}//PostDataLib
```

F.14 ProcessData.java

F.14.1 Code Description

This program uses the DS2450 library to convert raw data into the specified units. Available actions include:

• ProcessData:	Constructor that creates an object of the DS2450 library.
• convertToStandardUnits:	Converts passed raw data into Volts and returns result in the double format.
• getUnits:	Returns the units the DS2450 associates with the processed raw data.

F.14.2 Actual Code

```
/*******************************************************************
 *
 *      ProcessData: Ideally, this would look at sensor stamp, look on a table to find
 *      what units it needs to convert the data to, then do the conversion. Right now it
 *      just does the DSI test's conversion.
 *
 *      Author: Michael T. DePlonty
 *
 *      Date:   November 24, 2000
 *
 *******************************************************************/

class ProcessData
{
  private DS2450Lib ds2450_lib;

  ProcessData()
  {
    ds2450_lib = new DS2450Lib();
  }//ProcessData()
  ProcessData(String sensorID)
  {
//this is where this module would get the proper sensor object based
//on the sensor ID that was passed.

    ds2450_lib = new DS2450Lib();

  }//ProcessData(String sensorID)

  public double convertToStandardUnits(int passedRawData)
  {
    return ds2450_lib.convertToStandardUnits(passedRawData);
  }//convertToStandardUnits(int passedRawData)

  public String getUnits(String sensorID)
  {
    return ds2450_lib.unitsLabel();
  }//getUnits(String sensorID)

  public void cleanUpObjects()
  {
    ds2450_lib    = null;
  }//cleanUpObjects()
} //ProcessData
```

F.15 RetrieveData.java

F.15.1 Code Description

This program retrieves data from the buffer. The buffer must have each data point on separate lines. Available actions include:

• getFirstPoint:	Retrieves first data point by reading the first line in the buffer.
• getLastPoint:	Retrieves last data point in the buffer.
• getSpecificPoint:	Retrieves specific data point. The desired index passed to this routine corresponds to the line number in the buffer, index at 0.

• getFirst_N_Points:	Retrieves first N points in the buffer, where N is passed to this routine.
• getLast_N_Points:	Retrieves last N points in the buffer, where N is passed to this routine.
• getBlockPoints:	Retrieves a block of data points, where the beginning and ending indices are passed to this routine.
• getAllPoints:	Retrieves all the data in the buffer.
• displayAllBufferContents:	Displays the buffer's contents.
• openFile:	Private routine that opens the buffer file.
• closeFile:	Private routine that closes the buffer file.
• readFile:	Private routine that reads the buffer file.

F.15.2 Actual Code

```
/*********************************************************************
 *
 *      RetrieveData:
 *        Retrieves data from the buffer.  The data in the buffer is assumed to have been
 *      written on separate lines in the buffer file.  Thus the index refers to the line in
 *      the file, starting at line 0.
 *
 *      Author:  Michael T. DePlonty
 *
 *      Date:    November 25, 2000
 *
 *********************************************************************/
import java.io.*;
import java.util.Vector;

class RetrieveData
{
  private RandomAccessFile     bufferFile;
  private Vector               result;
  private String               fileName;
  private int                  number_of_lines;
```

```
RetrieveData()
{
  this("buffer.txt");
}//RetrieveData()

RetrieveData(String passedFileName)
{
  result = new Vector(20);
  fileName = passedFileName;
}//RetrieveData(String passedFileName)

public String getFirstPoint()
{
  result = readFile();
  return (String) result.firstElement();
}//getFirstPoint()

public String getLastPoint()
{
  result = readFile();
  return (String) result.lastElement();
}//getLastPoint()

public String getSpecificPoint(int index)
{
  result = readFile();
  if (index > result.size())  //want a point that does not exist
  {
    System.out.println("Index for specific point too big.");
    return null;
  }//if
  return (String) result.elementAt(index);
}//getSpecificPoint(int index)

public String[] getFirst_N_Points(int limit)
{
  result = readFile();

  if (limit > result.size())  //if desire more points than exist in buffer
  {
    System.out.println("Limit larger than number of points in buffer.");
    limit = result.size();
  }//if

  String[] dataPoints = new String[limit];
  for (int i = 0; i < limit; i++)
  {
    dataPoints[i] = (String) result.elementAt(i);
  }//for
  return dataPoints;
}//getFirst_N_Points(int limit)

public String[] getLast_N_Points(int desiredNumber)
{
  result = readFile();

  if (number_of_lines < desiredNumber)  //desire more points than exist
                                        //in buffer
  {
    System.out.println("Desired number too big.");
    desiredNumber = number_of_lines;  //just return all the points in the buffer
  }//if starting point larger than number of lines in the file

  String[] dataPoints = new String[desiredNumber];

  for (int i = (number_of_lines - desiredNumber); i < number_of_lines; i++)
  {
    dataPoints[i-number_of_lines+desiredNumber] = (String) result.elementAt(i);
  }//for
  return dataPoints;
}//getLast_N_Points(int startPoint)

public String[] getBlockPoints(int startIndex, int endIndex)
{
  result = readFile();

  if (endIndex < startIndex)
  {
    System.out.println("Invalid, end index must be larger than start index");
    String[] dummy = new String[1];
    dummy[0] = null;
    return dummy;
  }//if

  if (startIndex > result.size())  //desired to get a block of points outside the buffer
  {
    System.out.println("Invalid, must start within the buffer.  Start index too large.");
    String[] dummy = new String[1];
    dummy[0] = null;
    return dummy;
  }//if
  if (endIndex > result.size())  //desire to end outside the buffer
  {
    System.out.println("Invalid, wish to end outside the buffer.  End index too large.");
    endIndex = result.size();
  }//if

  String[] dataPoints = new String[endIndex - startIndex];

  for (int i = startIndex; i < (endIndex); i++)
  {
    dataPoints[i-startIndex] = (String) result.elementAt(i);
  }//for
  return dataPoints;

}//getBlockPoints(int startIndex, int endIndex)

public String[] getAllPoints()
{
  result = readFile();
  String[] resultString = new String[result.size()];
  for (int i=0; i<result.size(); i++)
  {
    resultString[i] = (String) result.elementAt(i);
  }//for
  return resultString;
}//getAllPoints()

public void displayAllBufferContents()
{
  String data[] = getAllPoints();

  for (int i=0; i < data.length; i++)
  {
    System.out.println(data[i]);
  }//for
}//displayAllBufferContents()

private void openFile()
{
  try
  {
    bufferFile = new RandomAccessFile(fileName,"rw");
  }//try
```

```
   catch (Exception e)
   {
     System.out.println(e.toString() + " in Retrieve Data");
   }//catch
 }//openFile()

 private void closeFile()
 {
   try
   {
     bufferFile.close();
   }//try

   catch (Exception e)
   {
     System.out.println(e.toString() + " in Retrieve Data");
   }//catch
 }//closeFile()

 private Vector readFile()
 {
   String buffer;
   number_of_lines = 0;

   openFile();

   try
   {
     bufferFile.seek(0);

     while ((buffer = bufferFile.readLine()) != null) //while we haven't reached the end
of file
     {
       result.addElement(buffer);
       number_of_lines++;
     }//while not at end of file
   }//try

   catch (Exception e)
   {
     System.out.println(e.toString());
   }//catch

   closeFile();
   return result;
 }//readFile()
```

```
  public void cleanUpObjects()
  {
    bufferFile = null;
    fileName = null;
    result.removeAllElements();
    result = null;
    System.gc();
  }//cleanUpObjects()
} //RetrieveData
```

F.16 SendInformation.java

F.16.1 Code Description

This program sends processed data, with all appropriate stamps, to the observer. It uses PostDataLib. Available actions include:

- SendInformation: Constructor that initializes the url to which TINI will post.
- sendData: Uses PostDataLib to post data to the specified web page.

F.16.2 Actual Code

```
/*******************************************************************
 *
 *        SendInformation sends concentrator information to the observer.
 *                Information is a String.
 *
 *        Author:  Michael T. DePlonty
 *
 *        Date:   November 21, 2000
 *
 *******************************************************************/
import java.io.*;
import java.net.*;

class SendInformation
{
  private static String url2Open;  //this is static as a concentrator
                                   //can report to only one ovserver
  private       String resultOfPosting;

/*******************************************************************
 *
 *  Default Constructor, observer's url (location) passed in.
 *
 *******************************************************************/

  SendInformation(String desiredURL) throws Exception
  {
    url2Open = desiredURL;
  }//SendInformation()

  public String sendData(String dateStamp, String timeStamp, String dataValue,
                   String dataUnit, String sensorID, String channelID,
                   String concentratorID, String concentrator_sw_ver,
                   String test_type_ID, String sequenceNum, String groupNum)
      throws Exception
  {
    PostDataLib pd;
    pd = new PostDataLib(url2Open);
    resultOfPosting = pd.postDataPoint(dateStamp, timeStamp,
                              dataValue, dataUnit, sensorID,
                              channelID, concentratorID, concentrator_sw_ver,
                              test_type_ID, sequenceNum,
                              groupNum);
    System.out.println(resultOfPosting);
    pd = null;
    return resultOfPosting;
  }//sendData()
  public void cleanUpObjects()
  {
    resultOfPosting = null;
    System.gc();
  }//cleanUpObjects()
} //SendInformation
```

F.17 SensorStamp.java

F.17.1 Code Description

This program stamps data with the appropriate sensor ID. Available actions include:

- acceptCommand: Accepts a Capture Data Command that contains the desired sensor address and the data's module destination.
- getSensorStamp: Uses the DS2450 library to retrieve the DS2450's one-wire address.
- stampData: Adds sensor ID to the data vector, effictively stamping the data. This routine is available for future expansion and is not used in the current software version.

F.17.2 Actual Code

```
/******************************************************************
 *
 *       SensorStamp:
 *         Stamps data with the sensor ID it came from. Ideally, one passes in address
 *         of sensor, this module looks in a table for the object representing the sensor,
 *       returns that object, then uses it to retreive the desired ID. Currently, the only
 *       "sensor" is the A/D converter, so no table is used.
 *
 *       Author:  Michael T. DePlonty
 *
 *       Date:   November 24, 2000
 *
 ******************************************************************/
import com.ibutton.utils.Address;
import java.util.Vector;

class SensorStamp
{
  private static DS2450Lib  ds2450;
  private String sensorAddress;
  private String dataDestination;
  SensorStamp()
  {
    sensorAddress = "";
    dataDestination = "";
  }//SensorStamp()

/**********************************************************************
 *
 *          acceptCommand(String[] command):
 *            This routine accepts a "Capture Data command" and stores the sensor's
 *          address and the data's destination in the appropriate fields.
 *
 **********************************************************************/

  public void acceptCommand(String[] command)
  {
    if (command.length != 2)  //passed in an illegal command
    {
      System.out.println("Illegal command.");
//currently, do nothing
    }//if passed in illegal command
    else
    {
      sensorAddress = command[0];
      dataDestination = command[1];
    }//else passed a legal command
  }//acceptCommand(String[] command)

  public String getSensorStamp()
  {
    if (sensorAddress.equals(""))
    {
      System.out.println("Did not receive a sensor address.");
      return "";
    }//if didn't receive a sensor address
    else
    {
    //get sensor's ID, add it to the end of the vector, then return
    //the new vector
      ds2450 = new DS2450Lib(sensorAddress);
      return ds2450.getSensorID();
    }//else did receive a sensor address
  }//getSensorStamp()
  public Vector stampData(Vector data)
  {
    //ideally, this would match the sensor address with a sensor object
    //and get the sensor's ID from that object.

    if (sensorAddress.equals(""))
    {
      System.out.println("Did not receive a sensor address.");
      data.addElement(null);
      return data;
    }//if didn't receive a sensor address
    else
    {
    //get sensor's ID, add it to the end of the vector, then return
    //the new vector
      ds2450 = new DS2450Lib(sensorAddress);
      System.out.println("Created ds2450");
      System.out.println("Adding sensor ID");
      data.addElement(ds2450.getSensorID());
      return data;
    }//else did receive a sensor address

  }//stampData(Vector data)

  public void sendDataOut()
  {
/*
  look at data destination, create new object of destination module,
then pass this object the vector of data. Currently, this is not
necessary as the stampData method returns the data vector.
*/
  }//sendDataOut()

  public void cleanUpObjects()
  {
    ds2450 = null;
  }//cleanUpObjects()
} //SensorStamp
```

F.18 StoreData.java

F.18.1 Code Description

This program accepts passed data and writes it to the buffer file. Available actions include:

• writeData:	Writes data to the buffer file. Data may be a TiniDataPoint, a string, or an array of Strings. Calling routing must also specifiy if the data will be appended to the buffer or if the data will overwrite data already present in the buffer.
• openOutFile:	Private method that opens the buffer file.
• closeFile:	Private method that closes the buffer file.

F.18.2 Actual Code

```
/*******************************************************************
 *
 *        StoreData accepts passed data and writes it to a file.  Data may be a
 *        Tini Data Point or it may be a string.
 *
 *        Author:  Michael T. DePlonty
 *
 *        Date:    November 24, 2000
 *
 ********************************************************************/
import java.io.*;

class StoreData
{
  private String        fileName;
  private String        dataString;
  private File          outputFile;
  private FileWriter    out;

  private static final String TAB       = "\t";
  private static final String RETURN    = "\r\n";
 StoreData()
 {
   this("buffer.txt");
 }//StoreData()

 public StoreData(String passedFileName)
 {
   fileName = passedFileName;
   dataString = null;
 }//StoreData(String fileName)

 public void writeData(TiniDataPoint thingToWrite, boolean append)
 {
   dataString = thingToWrite.getDate() + TAB;
   dataString = dataString.concat(thingToWrite.getTime()).concat(TAB);
   dataString = dataString.concat(thingToWrite.getValueAsString()).concat(TAB);
   dataString = dataString.concat(thingToWrite.getUnits()).concat(TAB);
   dataString = dataString.concat(thingToWrite.getSourceID()).concat(TAB);
   dataString = dataString.concat(thingToWrite.getChannel()).concat(TAB);
   dataString = dataString.concat(thingToWrite.getTiniID()).concat(TAB);
   dataString =
dataString.concat(thingToWrite.getSoftwareVersionAsString()).concat(TAB);
   dataString =
dataString.concat(thingToWrite.getTestTypeIDAsString()).concat(TAB);
   dataString =
dataString.concat(thingToWrite.getSequenceNumberAsString()).concat(TAB);
   dataString =
dataString.concat(thingToWrite.getGroupNumberAsString()).concat(RETURN);
   try
   {
     openOutFile(append);
     out.write(dataString);
     dataString = null;
     closeFile();
   }//try
   catch (IOException exception)
   {
     closeFile();
     System.out.println("IO error");
   }//catch
 }//writeData(TiniDataPoint thingToWrite)
```

```
public void writeData(String thingToWrite, boolean append)
{
  try
  {
    openOutFile(append);
    out.write(thingToWrite.concat(RETURN));
    closeFile();
  }//try

  catch (IOException exception)
  {
    System.out.println("IO error");
    closeFile();
  }//catch
}//writeLine(String thingToWrite)

public void writeData(String[] thingToWrite, boolean append)
{
  try
  {
    openOutFile(append);
    for (int i=0; i<thingToWrite.length; i++)
    {
      out.write(thingToWrite[i].concat(RETURN));
    }//for
    closeFile();
  }//try

  catch (Exception e)
  {
    System.out.println("StoreData: " + e.toString());
    closeFile();
  }//catch
}//writeData(String[] thingToWrite)

private void openOutFile(boolean append)
{
  try
  {
    outputFile = new File(fileName);
    out = new FileWriter(fileName, append); //open fileName and allow appending of
data
  }//try
  catch (IOException exception)
  {
    System.out.println("IO error");
    }//catch
  }//openOutFile()

  private void closeFile()
  {
    try
    {
      out.close();
    }//try
    catch (IOException exception)
    {
      System.out.println("IO error");
    }//catch
  }//closeFile()

  public void cleanUpObjects()
  {
    fileName     = null;
    dataString   = null;
    outputFile   = null;
    out          = null;
  }//cleanUpObjects()
} //StoreData
```

F.19 TimeDateStamp.java

F.19.1 Code Description

This program puts a time-date stamp on the data. IMPORTANT, this only works for the years 2000 to 2099 and should be fixed in the future. Available actions include:

• stamp:	Retrieves time and date values from TINI's internal clock.
• convertDay:	Converts integer day of week representation into string representation.
• convertMonth:	Converts integer month representation into string representation.
• convertYear:	Converts year that is modulus 100 into a yyyy format.

- getTotalDateString: Produces string for date in mm/dd/yyyy format.
- getTotalTimeString: Produces string for time in hh:mm:ss format.
- getLapsedMilliseconds: Gives the number of milliseconds since Jan. 1, 1970.
- addStampToData: Adds time-date stamp to passed data vector.
- Access methods to retrieve the time-date stamp fields. Fields include: day, month, year, date, hout, minute, second, hundreth of a second.

F.19.2 Actual Code

```
/*******************************************************************
 *
 *        TimeDateStamp puts a time/date stamp on the TINI readings.
 *                This only works for 2000 to 2099.
 *
 *        Author:  Michael T. DePlonty
 *
 *        Date:    July 25, 2000
 *
 *******************************************************************/
import com.dalsemi.system.*;
import java.util.Vector;

class TimeDateStamp
{
  private Clock        timeDateStamp;
  private String       day;
  private String       month;
  private int          year;
  private long         lapsedMilliseconds;

  TimeDateStamp()
  {
    timeDateStamp = new Clock();
  }//TimeDateStamp()
```

```
/**********************************************************************
 *
 *  stamp():  Reads TINI's real time clock values from hardware clock and places
 *          them into Clock instance fields.  After stamping, available fields include:
 *                  date      - day of month
 *                  day       - day of week
 *                  hour      - hour
 *                  hundredth - hundreths of seconds
 *                  is12Hour  - 12/24 hour flag
 *                  minute    - minute
 *                  month     - month
 *                  pm        - PM/AM flag
 *                  second    - second
 *                  year      - year mod 100
 *
 **********************************************************************/

  public void stamp()
  {
    timeDateStamp.getRTC();
  }//stamp()

/**********************************************************************
 *
 *  convertDay():  converts integer day of week into string representation
 *
 **********************************************************************/

  public void convertDay()
  {
    switch (timeDateStamp.day)
    {
      case 1:
      {
        day = "Sun";
        break;
      }//case 1
      case 2:
      {
        day = "Mon";
        break;
      }//case 2
      case 3:
      {
        day = "Tues";
        break;
      }//case 3
      case 4:
      {
        day = "Wed";
        break;
      }//case 4
      case 5:
      {
        day = "Thr";
        break;
      }//case 5
      case 6:
      {
        day = "Fri";
        break;
      }//case 6
      case 7:
      {
        day = "Sat";
        break;
      }//case 7
    }//switch (day)
  }//convertDay()

/*************************************************************************
 *
 *  convertMonth():  converts integer month into string representation
 *
 *************************************************************************/

  public void convertMonth()
  {
    switch (timeDateStamp.month)
    {
      case 1:
      {
        month = "Jan";
        break;
      }//case 1
      case 2:
      {
        month = "Feb";
        break;
      }//case 2
```

```
case 3:
{
  month = "Mar";
  break;
}//case 3
case 4:
{
  month = "Apr";
  break;
}//case 4
case 5:
{
  month = "May";
  break;
}//case 5
case 6:
{
  month = "Jun";
  break;
}//case 6
case 7:
{
  month = "Jul";
  break;
}//case 7
case 8:
{
  month = "Aug";
  break;
}//case 8
case 9:
{
  month = "Sep";
  break;
}//case 9
case 10:
{
  month = "Oct";
  break;
}//case 10
case 11:
{
  month = "Nov";
  break;
}//case 11
```

```
      case 12:
      {
        month = "Dec";
        break;
      }//case 12
    }//switch (day)
  }//convertMonth()

/*********************************************************************
 *
 *  convertYear():  converts year mod 100 into a yyyy format.  timeDateStamp.year
 *              returns a number between 0 and 99, so this routine only gives years
 *              between 2000 and 2099 then will loop back to year 2000.
 *
 *        KNOWN BUG HERE.  NEEDS TO BE FIXED  TINI ONLY ALLOWS
 *        YEARS UP TO 2099, SO ONCE THIS IS IMPROVED, THIS BUG WILL
 *        EXIST.
 *
 *********************************************************************/

  public void convertYear()
  {
    year = timeDateStamp.year + 2000;
  }//convertYear()

  public String getDay()
  {
    convertDay();
    return day;
  }//getDay()

  public String getMonth()
  {
    convertMonth();
    return month;
  }//getMonth()

  public int getYear()
  {
    convertYear();
    return year;
  }//getYear()
  public int getDate()
  {
    return timeDateStamp.date;
  }//getDate()

  public int getHour()
  {
    return timeDateStamp.hour;
  }//getHour()

  public int getMinute()
  {
    return timeDateStamp.minute;
  }//getMinute()

  public int getSecond()
  {
    return timeDateStamp.second;
  }//getSecond()

  public int getHundredth()
  {
    return timeDateStamp.hundredth;
  }//getHundredth()

/**********************************************************************
 *
 *  getTotalDateString(): produces string for date in mm/dd/yyyy format.
 *                Note, day may be a single integer (5 instead of 05).
 *
 **********************************************************************/

  public String getTotalDateString()
  {
    String currentDateStamp = timeDateStamp.month + "/"
                + timeDateStamp.date
                + "/" + getYear();
    return currentDateStamp;
  }//getTotalDateString()

/**********************************************************************
 *
 *  getTotalTimeString(): Produces string for time in hh:mm:ss format.
 *                Note, hour is in 24 hour format, so 1pm is 13 (no am/pm ID).
 *
 **********************************************************************/
```

```
  public String getTotalTimeString()
  {
    String currentTimeStamp = timeDateStamp.hour + ":"
                  + timeDateStamp.minute + ":"
                  + timeDateStamp.second;
    return currentTimeStamp;
  }//getTotalTimeString()

/***********************************************************************
 *
 *  This method gives the number of milliseconds since Jan. 1, 1970
 *
 ***********************************************************************/

  public long getLapsedMilliseconds()
  {
    lapsedMilliseconds = timeDateStamp.getTickCount();
    return lapsedMilliseconds;
  }//getLapsedMilliseconds()

/***********************************************************************
 *
 *    This method assumes the time and date has already been stamped into this modules
 *  variables.  This is done so that the stamp can be formed as close as possible to the
 *  time the data is taken from the sensor.
 *
 ***********************************************************************/

  public Vector addStampToData(Vector data)
  {
    data.addElement(getTotalDateString());
    data.addElement(getTotalTimeString());
    return data;
  }//stampData(Vector data)

  public void cleanUpObjects()
  {
    timeDateStamp = null;
    day = null;
    month = null;
  }//cleanUpObjects()
} //TimeDateStamp
```

F.20 TiniData.java

F.20.1 Code Description

This program creates a vector of TiniDataPoints and usefull methods to manipulate this vector. Available actions include:

• add:	Adds TiniDataPoint at a specified location or at the end of the vector.
• removeElementAt:	Removes vector element at the specified location.
• removeAllElements:	Removes all elements from the vector.
• getDataPoint:	Retrieves and returns TiniDataPoint at specified index.
• setElementAt:	Replaces the vector element at the specified index with the passed TiniDataPoint.
• size:	Returns the vector's size.
• printContents:	Displays the vector's contents to the computer monitor.

F.20.2 Actual Code

```
/*******************************************************************
 *
 *  TiniData creates a vector of TiniDataPoints and usefull methods to manipulate
 *    this vector.  Each TiniDataPoint is a vector that contains 10 identifying fields,
 *    such as Date, Time, value, etc.
 *
 *        Author:  Michael T. DePlonty
 *
 *        Date:    July 21, 2000
 *
 *******************************************************************/
import java.util.Vector;

class TiniData
{
  private static final int VECTOR_CAPACITY = 0;  //how much the vector can initially
hold

  private static Vector dataPoints;
```

```
/*********************************************************************
 *
 *  class constructor
 *
 *********************************************************************/

  public TiniData()
  {
    dataPoints = new Vector(VECTOR_CAPACITY);
    for (int i=0; i<VECTOR_CAPACITY; i++)
      dataPoints.addElement(null);
  }//TiniData()

/*********************************************************************
 *
 *  add(TiniDataPoint thingToAdd) appends a TINI data object to the
 *    end of the vector of TINI data objects.
 *
 *********************************************************************/

  public static void add(TiniDataPoint thingToAdd)
  {
    dataPoints.addElement(thingToAdd);
  }//add()

/*********************************************************************
 *
 *  add(TiniDataPoint thingToAdd, int index) inserts a TINI data object
 *    into the vector of TINI data objects at the location index.
 *
 *********************************************************************/

  public static void add(TiniDataPoint thingToAdd, int index)
  {
    dataPoints.insertElementAt(thingToAdd, index);
  }//add()

/*********************************************************************
 *
 *  removeElementAt(int index) removes a TINI data object at the
 *    location specified by index.
 *
 *********************************************************************/
 public static void removeElementAt(int index)
 {
   dataPoints.removeElementAt(index);
 }//removeElementAt()

 public static void removeAllElements()
 {
   dataPoints.removeAllElements();
 }//removeAllElements()

 public static TiniDataPoint getDataPoint(int index)
 {
   return (TiniDataPoint) dataPoints.elementAt(index);
 }//getDataPoint()

/*****************************************************************
 *
 *  setElementAt(TiniDataPoint thingToAdd, int index) replaces a TINI
 *    data object at the index with the passed TINI data object.
 *
 *****************************************************************/

 public static void setElementAt(TiniDataPoint thingToAdd, int index)
 {
   dataPoints.setElementAt(thingToAdd,index);
 }//setElementAt(int index)

/*****************************************************************
 *
 *  size() returns the number of TINI data objects in the vector
 *
 *****************************************************************/

 public static int size()
 {
   return dataPoints.size();
 }//size()

 public void printContents()
 {
   for(int i=0; i<size(); i++)
     System.out.print( dataPoints.elementAt(i) + "\t");
   System.out.println(" ");
 }//printContents()
} //TiniData
```

F.21 TiniDataPoint.java

F.21.1 Code Description

This program creates a vector that contains the processed data point plus all the necessary identification tags. It also creates usefull methods for manipulating the vector. Available actions include:

- Routines used to set and retrieve specific values in the vector. The following vector indicies contain the specified data field.

Vector Index	**Data Field**
0	Date
1	Time
2	Processed Data Value
3	Units
4	DS2450 ID
5	Channel Letter
6	TINI ID
7	Software Version
8	Test Type ID
9	Sequence Number
10	Group Number

- size: Returns the vector's size.
- printContents: Displays the vector's contents to the computer monitor.

F.21.2 Actual Code

```
/*********************************************************************
 *
 *        TiniDataPoint creates a vector dataPoint that contains the value plus nine
 *        other fields.  Each element in the vector is a String.
 *             dataPoint(0)    =    Date
 *             dataPoint(1)    =    Time
 *             dataPoint(2)    =    Value
 *             dataPoint(3)    =    Units
 *             dataPoint(4)    =    Source (chip ID of DS2450 A/D converter)
 *             dataPoint(5)    =    Channel
 *             dataPoint(6)    =    TINI ID
 *             dataPoint(7)    =    Software Version
 *             dataPoint(8)    =    Test type ID
 *             dataPoint(9)    =    Sequence Number
 *             dataPoint(10)   =    Group Number
 *
 *        Author:  Michael T. DePlonty
 *
 *        Date:   July 25, 2000
 *
 *********************************************************************/

import java.lang.*;
import java.util.Vector;

class TiniDataPoint
{
  private static final int VECTOR_CAPACITY = 11;  //how much the vector can initially
hold
  private Vector dataPoint;

/*********************************************************************
 *
 * class constructor
 *
 *********************************************************************/

  public TiniDataPoint()
  {
    dataPoint = new Vector(VECTOR_CAPACITY);
    for (int i=0; i<VECTOR_CAPACITY; i++)
      dataPoint.addElement(null);
  }//TiniDataPoint()
  public String getDate()
  {
    return (String) dataPoint.elementAt(0);
  }//getDate()

  public void setDate(String date)
  {
    dataPoint.setElementAt(date, 0);
  }//setDate()

  public String getTime()
  {
    return (String) dataPoint.elementAt(1);
  }//getTime()

  public void setTime(String time)
  {
    dataPoint.setElementAt(time, 1);
  }//setTime()

  public double getValueAsDouble()
  {
    Double buffer = Double.valueOf((String) dataPoint.elementAt(2));
    return buffer.doubleValue();
//this converts the String representation to a double and returns the value as a double
  }//getValueAsDouble()

  public String getValueAsString()
  {
    return (String) dataPoint.elementAt(2);
  }//getValueAsString()

  public void setValue(double value)
  {
    dataPoint.setElementAt(Double.toString(value), 2);
  }//setValue()

  public String getUnits()
  {
    return (String) dataPoint.elementAt(3);
  }//getUnits()
```

```
 public void setUnits(String units)
 {
   dataPoint.setElementAt(units, 3);
 }//setUnits()

 public String getSourceID()
 {
   return (String) dataPoint.elementAt(4);
 }//getSourceID()

 public void setSourceID(String sourceID)
 {
   dataPoint.setElementAt(sourceID, 4);
 }//setSourceID()

 public String getChannel()
 {
   return (String) dataPoint.elementAt(5);
 }//getChannel()

 public void setChannel(String channel)
 {
   dataPoint.setElementAt(channel, 5);
 }//setChannel()

 public String getTiniID()
 {
   return (String) dataPoint.elementAt(6);
 }//getTiniID()

 public void setTiniID(String tiniID)
 {
   dataPoint.setElementAt(tiniID, 6);
 }//setTiniID()

 public double getSoftwareVersionAsDouble()
 {
   Double buffer = Double.valueOf((String) dataPoint.elementAt(7));
   return buffer.doubleValue();
//this converts the String representation to a double and returns the value as a double
 }//getSoftwareVersionAsDouble()

 public String getSoftwareVersionAsString()
 {
   return (String) dataPoint.elementAt(7);
 }//getSoftwareVersionAsString()
```

```
  public void setSoftwareVersion(double softwareVersion)
  {
    dataPoint.setElementAt(Double.toString(softwareVersion), 7);
  }//setSoftwareVersion()

  public int getTestTypeIDAsInt()
  {
    return Integer.parseInt((String) dataPoint.elementAt(8));
  }//getTestTypeIDAsInt()

  public String getTestTypeIDAsString()
  {
    return (String) dataPoint.elementAt(8);
  }//getTestTypeID()

  public void setTestTypeID(int testTypeID)
  {
    dataPoint.setElementAt(Integer.toString(testTypeID), 8);
  }//setTestTypeID()

  public long getSequenceNumberAsLong()
  {
    return Long.parseLong((String) dataPoint.elementAt(9));  //takes string
representation, converts it to an int, then returns this value
  }//getSequenceNumberAsInt()

  public String getSequenceNumberAsString()
  {
    return (String) dataPoint.elementAt(9);
  }//getSequenceNumberAsString()

  public void setSequenceNumber(long sequenceNumber)
  {
    dataPoint.setElementAt(Long.toString(sequenceNumber), 9);
  }//setSequenceNumber()

  public void setSequenceNumberString(String sequenceNumber)
  {
    dataPoint.setElementAt(sequenceNumber,9);
  }//setSequenceNumberString()
  public long getGroupNumberAsLong()
  {
    return Long.parseLong((String) dataPoint.elementAt(10));  //takes string
representation, converts it to an int, then returns this value
  }//getGroupNumberAsLong()

  public String getGroupNumberAsString()
  {
    return (String) dataPoint.elementAt(10);
  }//getGroupNumberAsString()

  public void setGroupNumber(long groupNumber)
  {
    dataPoint.setElementAt(Long.toString(groupNumber), 10);
  }//setGroupNumber(long groupNumber)

  public int size()
  {
    return dataPoint.size();
  }//size()

  public void printContents()
  {
    for(int i=0; i<size(); i++)
      System.out.print( (String) dataPoint.elementAt(i) + "\t");
    System.out.println(" ");
  }//printContents()

  public void cleanUpObjects()
  {
    dataPoint = null;
  }//cleanUpObjects()
} //TiniDataPoint
```

Bibliography

1. Wilkinson, B. and Rees, D. "Performance Evaluation of a 'Pattern Recognition' Adaptive Controller." Intelligent Process Control and Scheduling: Discrete Event Systems: Proceedings of the 1990 European Simulation Symposium. Ed. Ghislain C. Vansteenkiste, et al. San Diego: Society for Computer Simulation, 1990. 85-90.

2. Jager, R., Verbruggen, H. B., Bruijn, P. M., and Krijgsman, A. J. "Direct Real-Time Control Using Knowledge-Based Techniques." Intelligent Process Control and Scheduling: Discrete Event Systems: Proceedings of the 1990 European Simulation Symposium. Ed. Ghislain C. Vansteenkiste, et al. San Diego: Society for Computer Simulation, 1990. 101-105.

3. Masmoudi, M. and Vansteenkiste, C.G. "Netman: An Integrated Environment for Hydraulic Networks Management." Intelligent Process Control and Scheduling: Discrete Event Systems: Proceedings of the 1990 European Simulation Symposium. Ed. Ghislain C. Vansteenkiste, et al. San Diego: Society for Computer Simulation, 1990. 51-55.

4. Bauman, R., et al. "The Control and the Command of the Vivitron, a System Managed with an OODBMS." IEEE Transactions on Nuclear Science. 45.4 August 1998: 2020-2025.

5. Cargol, Timothy L. "A Non-Destructive Transformer Oil Tester." Master's Thesis. MIT 2000.

6. Cooke, Chathan M and Hagman, Wayne H. A Non-Destructive Breakdown Measurement for Oil Dielectric Strength Testing: Final Technical Report. Cambridge MA: MIT Laboratory for Electromagnetic and Electronic Systems and Electric Utility Program, 1994.

7. McCabe, Aaron R. "Event-Driven, Asynchronous Control and Monitoring." Master's Thesis. MIT 1998.

8. Polak, T. A. and Pande, C. Engineering Measurements: Methods and Intrinsic Errors. Suffolk, UK: St. Edmundsbury Press Limited, 1999.

9. Task Committee on Data Acquisition Systems of the Hydraulics Division of the American Society of Civil Engineers. Guidelines for PC-Based Data Acquisition Systems for Hydraulic Engineering. New York: American Society of Civil Engineers, 1993.

10. Taylor, H. Rosemary. Data Acquisition for Sensor Systems. London: Chapman and Hall, 1997.

11. Young, R. E. Telemetry. London: Temple Press, 1963.

12. Gruenberg, Elliot L., ed. Handbook of Telemetry and Remote Control. New York: McGraw-Hill, 1967.

13. Borer, J. Microprocessors in Process Control. London: Elsevier Applied Science, 1991.

14. Cohen, Jonathan. Automatic Identification and Data Collection Systems. London: McGraw-Hill, 1994.

15. Pradhan, Dhiraj K. Fault-Tolerant Computer System Design. Upper Saddle River, New Jersey: Prentice Hall PTR, 1996.

16. Harold, Elliotte Rusty. Java I/O. Sebastopol, California: O'Reilly, 1999.

17. Jamsa, Kris. DOS: Secrets, Solutions, Shortcuts. Berkeley: Osborne McGraw-Hill, 1988.

Biography

Michael T. DePlonty was born in Escanaba, Michigan in 1975. He received a Bachelor of Science degree in Electrical Engineering from the University of Michigan, Ann Arbor in 1998.

During his undergraduate studies he held four separate summer internships. In 1995 he worked for Electronic Data Systems (EDS) documenting dataflow for General Motors' Small Car platform. From 1996 to 1998 he worked for Delphi – Delco Electronics. He verified assembly code for the Audio Electronics Software group in 1996, developed software for the Power Electronics Advanced Development group in 1997, and characterized analog to digital (A/D) converters for the Audio Systems Advanced Development group in 1998. During his graduate studies he served as a teaching assistant in the MIT Department of Electrical Engineering and Computer Science from the Fall of 1999 to the Spring of 2000 and served as a Research Assistant in the Summer and Fall of 2000.