Stand-Alone Datalogger with a Graphical User Interface

Final Year Electronic & Computer Engineering Project Report

Paul Boyle

Supervisor: Pat Byrne

April 2002

Acknowledgements

This project was made possible by the efforts and talents of many people. I want to express my appreciation to my project supervisor Pat Byrne for her support and advise through the course of this project.

At this point, the end of my time in the NUI Department of Electronic Engineering, I want to express my gratitude to all members of the Department who have helped me in any way during the past two years.

Also, many thanks to Aodh, Martin and Myles in helping me with the design and construction of the prototype Logger board. I also wish to thank the many contributions to this project. Without their input, this project would be next to impossible.

Lastly, but most importantly, I thank my parents and family. Whatever I have achieved, or may achieve in the future, is a result of their support and effort. I hope that some day I can begin to repay them.

Many thanks to all Paul Boyle

Abstract

This report describes a final year Electronic & Computer Engineering project entitled "Stand-Alone DataLogger with Graphical User Interface". This project involves designing and constructing a Stand-Alone Datalogger that can be used to monitor temperature inside a drinks vending machine and, which the user interacts with the DataLogger in the initializing and data-downloading modes through a Graphical User Interface (GUI).

This project is both hardware and software based with the software being developed in Java and the hardware being based around the 8051 Microcontroller chip developed by Analog Devices.

This report will describe the passage from design feasibility to final operation of the Stand-Alone Datalogger. The project was completed by Paul Boyle under the supervision of Pat Byrne. This report was submitted to the Department of Electronic Engineering, National University of Ireland Galway, in April 2002.

CONTENTS

1. Aims & Objectives

AIMS

1. To produce a Stand-Alone Datalogger which will interface with the PC though the serial port of the host computer.
2. To design the circuitry for the Stand-Alone Datalogger which will incorporate the Analog Devices ADuC812 microcontroller.
3. To design and write the software to implement the microcontroller operations.
4. To design and write the host computer software to allow the user to interface with the Datalogger during its three modes of operation.

OBJECTIVES

1. To make a single sided circuit board to the minimum of size.
2. To ensure that the circuit board will utilise the appropriate supply lines for the ICs and other discrete devices on the board.
3. To design the software to operate the microcontroller and allow it to communicate with other devices on the circuit board and with the host computer.
4. To design the necessary programs to allow the host computer communication with the Datalogger using the serial port, to be designed for and written in Java and to provide a Graphical User Interface for the end user.
5. To present a report which should contain the passage from design feasibility to final operation

2. Introduction

In environmental monitoring applications, parameters such as temperature, humidity, water levels or pollution need to be monitored continuously over long periods. A conventional personal computer based data acquisition system can be used but such a system involves a computer and a Datalogger, making it expensive. Secondly, the physical size is large. Thirdly, power assumption will be high, and this implies that a powerful battery pack is required in the application where there is no mains supply.

A Stand-Alone Datalogger is a useful device for such an application. Firstly, it is dedicated. Its only task is to acquire data and save the data into its memory. It can be connected to a computer at any time to allow its collected data to be transferred and analysed.

The Datalogger that has been designed will have one input. The input will be an analogue input that will measure temperature.

The Datalogger will be used to monitor the conditions within a drinks vending machine.

The temperature sensor will monitor the temperature within the vending machine to make sure the drinks are kept to the required temperature and any changes with the temperature will be recorded i.e. time, day, month, year, and the temperature.

At the end of each day the Datalogger may be removed, (if required) from the vending machine and connected to the host computer, and all the data is off-loaded onto the host computer. This allows the logged data to be analyzed and permanently stored.

The connection between the host computer and the Datalogger is though serial port. This is used because it is the common connection between peripherals and is more than sufficient for the system needs i.e. don't require high-speed data transfer between host computer and the Datalogger.

When all the data has been off-loaded onto the host computer the user will again initialize the Datalogger through the Graphically User Interface (GUI). This GUI will be constructed from the Java language, and will be the users only way to interact with the Datalogger.

Such a Datalogger will be made small in size and with ultra-low power consumption. Such a Datalogger's small size allows it to be placed in almost any location. It can collect data continuously over a period of time without having its battery changed.

The Datalogger will be based around Analog Devices 8051 microcontroller (`ADuC812`). This has a built-in temperature sensor and an Analogue-to-Digital Converter (ADC) with a conversion accuracy of 12-bits as well as memory, although using external memory will extend this. The `ADuC812` does not contain a Real Time Clock (RTC) therefore a dedicated RTC is used. The Datalogger will have a Liquid Crystal Display (LCD), which displays to the user the current temperature read, what percentage of memory is remaining and also informs the user what mode the Datalogger is in.

When driven by a lithium 9V PP3 sized battery it could capture data for a month or so unattended.

3. Background

3.1 ADuC812 Overview

Analog Devices ADuC812 is a fully integrated 12-bit data acquisition system incorporating a high performance self- calibrating multichannel Analogue-to- Digital Converter (ADC), two 12-bit Digital-to-Analogue Converters (DAC) and programmable 8-bit (8051-compatible) MCU on a single chip.

The programmable 8051-compatible core is supported by 8K bytes Flash/EE program memory, 640 bytes Flash/EE data memory and 256 bytes data SRAM on-chip.

Additional MCU support functions include watchdog timer, power supply monitor and ADC DMA function. 32 programmable I/O lines, I^2C-compatible, SPI and standard UART serial port I/O are provided for multiprocessor interfaces and I/O expansion.

Normal, idle and power-down operating modes for both the MCU core and analogue converters allow for flexible power management schemes suited to low power applications.

3.1.1 ADuC812 Features

Analogue I/O features:

- Eight channel, high accuracy 12-bit ADC
- On-chip, 40 ppm/^{o}C voltage reference ADC-to-RAM capture
- Two 12-bit voltage output DACs
- On-chip temperature sensor function
- High speed 200 kSPS
- DMA controller for high speed

Memory features:

- 8K bytes on-chip flash/EE program memory
- 640 bytes on-chip flash/EE data memory
- On-chip charge pump (No Ext. V_{pp} requirements)
- 256 bytes on-chip data RAM
- 16M bytes external data address space
- 64K bytes external program address space

8051-Compatible core:

- 12 MHz nominal operation (16MHz max)
- Three 16-bit timer/counters
- 32 programmable I/O lines
- High current drive capability—Port 3
- Nine interrupt sources, two priority levels

Power features:

- Specified for 3V and 5V operation
- Normal, idle and power-down modes

On-chip peripherals:

- UART serial I/O
- 2-wire (I^2C compatible) an SPI serial I/O
- Watchdog timer
- Power supply monitor

3.2 What is I^2C bus?

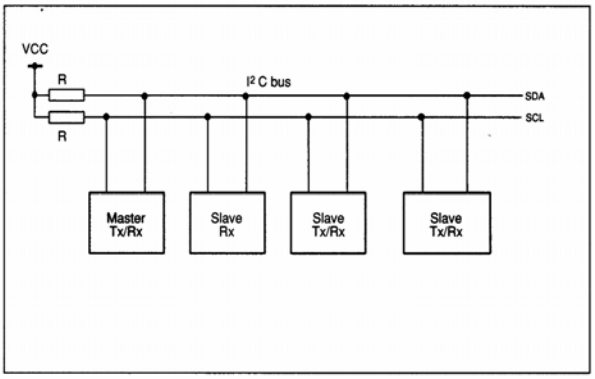
Devised by Phillips, I^2C stands for inter-IC-communication. It is a data bus that allows integrated circuits or modules to communicate with each other.

The bus allows data and instruction to be exchange between devices via only two wires. This greatly simplifies the design of a complex electronic circuits. There is a family of I^2C compatible devices available for various applications. They include I/O expansion, analogue-to-digital and digital-to-analogue conversion, time keeping, memory and frequency synthesis, etc.

3.2.1 Principle of the I2C bus

The I^2C bus consists of two lines: a bi-directional data line called SDA and a clock line called SCL. Both are pulled up to the positive power supply via resistors. An I^2C bus system is shown in **Fig. 3-1**

Fig. 3-1. An I^2C bus consists of only two dta lines: serial, SCL and serial data, SDA. I^2Ccompatible devices connect to the bus using these tow wires, making hardware design simpler.



A device generating a message is a 'transmitter' while a device receiving a message is the 'receiver'. The device controlling the bus operation is the 'master' and devices controlled by the master are 'slaves'.

The following communication protocol is defined:

- A data transfer may be initiated only when the bus is not busy
- During the data transfer, the data line must remain stable whenever the clock line is high.

Changes in the data line while the clock line is high is interpreted as control signals. The following bus conditions are defined, **Fig. 3-2**

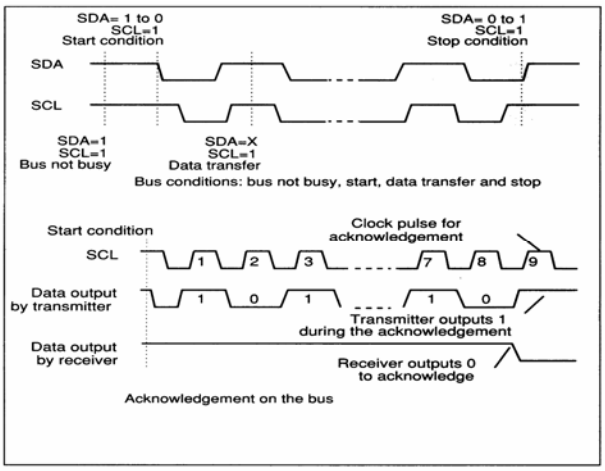


Fig. 3-2. Timing sequence for Bus Not Busy, Start, Stop and Acknowledgement.

- **Bus not busy:** both data and clock lines remain high
- **Start data transfer:** a both in the state of the data line from high to low while the clock is high, defines the start condition
- **Stop data transfer:** A change in the state of the data line from low to high while the clock is high defines the stop condition.
- **Data valid:** The state of the data line represents valid data after a start condition. The data line is stable for the duration of the high period of the clock signal. The data on the line may be changed during the period of the clock signal. There is one clock pulse per bit data. Each data transfer is initiated with start condition and terminated with a stop condition. The number of data bytes transferred between the start and stop conditions is not limited. The information is transmitted byte-wise and the receiver acknowledges with a ninth bit.
- **Acknowledge bit:** Each byte is follow by an acknowledge bit. The acknowledge bit is a high level put on the bus by the transmitter whereas the master generates an extra acknowledge bit is a low level put on the bus by the receiver. A slave receiver which is addressed is obliged to generate an acknowledge bit after the reception of each byte.

The device that acknowledges has to pull down the SDA line during the acknowledge clock pulse in such a way that the SDA line is at a stable low state during the high period of the acknowledge related clock pulse. A master receiver must signal an end to the slave transmitter by not generating an acknowledge on the last byte that has been clocked out of the slave.

3.2.2 How the bus operates.

Before any data is transmitted on the bus, the device that should respond is addressed first. This is carried out with the seven-bit address byte plus R/-W bit transmitted after a start condition. A typical address byte has the following format:

```
Fixed Address bits = Programmable address bits + R/-W bit
(in total 8 bits)
```

The fixed address depends son the IC and it cannot be changed. * The programmable address bits can be set using the address pins on the chip. The last bit is the read/write bit indicates the direction of the data flow. The byte following the address byte in the control byte, which depends on the IC, used. Following the control byte are the data bytes. The serial data has the same format shown in **Fig. 4-12**.

*Although some I^2C devices have inputs that can modify the address depending on their logic state, allowing more than one of the same IC to be used on the bus.

3.3 Serial Port

Microcontroller's have proven to be quite popular recently. Many of these have in built SCI (Serial Communications Interface) that can be used to talk to the outside world. Serial communication reduces the pin count. Only two pins are commonly used, Transmit Data (TXD) and Receive Data (RXD) compared with at least 8 pins if you use an 8-bit parallel method.

Devices that use serial cables for their communications are split into two categories. These are DCE (Data Communications Equipment) and DTE (Data Terminal Equipment.) Data Communication Equipment are devices such as modem, TA adapter, plotter etc while Data Terminal Equipment is your computer or terminal.

The electrical specifications of the serial port are contained in the EIA (Electronics Industry Association) RS232C standard. It states many parameters such as:

1. A "Space" (logic 0) will be between +3 and +25 volts
2. A "Mark" (logic 1) will be between –3 and –25 volts
3. The region between +3 and –3 volts is undefined
4. An open circuit voltage should never exceed 25 volts (in reference to GND)
5. A short circuit should not exceed 500mA. The driver should be able to handle this without damage

Above is no were near the complete list of the EIA standard. Line Capacitance, maximum baud rate etc are also included.

Serial ports come in two sizes. There are the D-Type 25 pin connector and the D-Type 9 pin connector, both of which are male at the back of the PC, thus you will require a female connector on your device.

Name	Address	IRQ
COM1	3F8	4
COM2	2F8	3
COM3	3E8	4
COM4	2E8	3

Table 3-1: Stand Port Addresses

Table 3-1 shows the standard port address. These should work for most PC's. The base addresses for the COM ports can be read from the BIOS Data Area.

Start Address	Function
0000:0400	COM1's Base Address
0000:0402	COM2's Base Address
0000:0404	COM3's Base Address
0000:0406	COM4's Base Address

Table 3-2 COM Port Addresses in the BIOS Data Area

Table 3-2 shows the address at which we can find the COM ports addresses in the BIOS Data Area. Each address will take up to 2 bytes.

3.4 Java

Java, formerly known as oak, is an object-oriented programming language developed by Sun. It shares many superficial similarities with C, C++, and Objective C (for instance `for` loops have the same syntax in all four languages); but it is not based on any of those languages.

The language was originally created because C++ proved inadequate for certain tasks. Since the designers were not burdened with compatibility with existing languages, they were able to learn from the experience and mistakes of previous object-oriented languages. They added a few things C++ doesn't have like garbage collection and multithreading; and they threw away C++ features that had proven to be better in theory than in practice like multiple inheritance and operator overloading.

Even more importantly Java was designed from the ground up to allow for secure execution of code across a network, even when the source of that code was entrusted and possibly malicious. This required the elimination of more features of C and C++. Most notably there are no pointers in Java. Java programs cannot (at least in theory) access arbitrary addresses in memory.

Furthermore Java was designed not only to be cross-platform in source form like C, but also in compiled binary form. Since this is frankly impossible across processor

architectures, Java is compiled to an intermediate byte-code that is interpreted on the fly by the Java interpreter. The interpreter must be written for your particular computer. Java is the closest thing we have to a universal computer language, which means it runs on all computers (as long as a Java virtual machine has been written for them). Thus to port Java programs to a new platform all that is needed is a port of the interpreter and a few native code libraries.

3.4.1 Java Features

- *Cross Platform:* Java is a cross platform language. The Java compiler compiles Java source code into "bytecodes". These bytecodes are then interpreted by a Java "virtual machine" that is written for the processor architecture the program is running on. This means that your Java Applet will run on any platform that has a Java interpreter. This kind of platform independence is essential for a heterogeneous platform like the Internet or even corporate Intranets.
- *Software Distribution:* In the case of applets, the Java bytecodes are downloaded at run-time, so the user is always getting the most current code. This solves all sorts of software distribution nightmares that enterprises have traditionally had to contend with.
- *Security:* Java was designed to verify and execute binary programs in a controlled environment. This protects the end-user's computer from viruses and security violations. Whenever a Java applet is transferred to the user's browser, it is subject to byte-code verification. This means that if the packet's size is changed along the way, the program will be aborted. This checking guards against Trojan horses and viruses being added to the Java bytecodes.
- *Easier to program than C++, and just as powerful!* Here are some of the advantages of Java over C++:
    - No pointers: The Java language passes all arrays and objects by reference but does not have an explicit pointer type. This prevents the programmer from constructing a reference to anonymous memory.
    - Automatic garbage collection - The Java interpreter manages the memory, it is not the programmer's responsibility.
    - C++ style Exceptions are automatically generated when dereferencing a NULL pointer, accessing outside the bounds of an array, or running out of memory.
- *Network protocol handlers:* Support for HTTP, FTP, NNTP, MIME, and Sockets make it sort of a "network programming language."

3.4.2 Java Pros

- Java allows HTML writers the ability to take their pages from static information to interactive applications.
- Java has broad industry support. It has been licensed by: Netscape, Spyglass, Wollongong, Microsoft, Oracle, Novell, Borland and Symantec.
- Java runs on many platforms. Sun distributed versions of the compiler and interpreter for the Mac, Solaris, Windows 95 and NT. IBM is working on versions for OS/2 and Windows 3.1.

3.4.3 Java Cons

- Currently, Java lacks 'persistent' objects. Let's say you go to a Web site with a Java applet and wait for several minutes for your browser to download it. The next time you visit the site, you need to wait all over again (unless the applet is still in your browser's cache). Persistence would allow you to store the applet on your PC the first time you download it, and only download it again if the applet has changed.
- Even though Java was designed to take some of the complexities out of coding in a language like C++, it did evolve from C++ and is similar in some ways. Programmers that don't want to learn C++ may not expend the time and energy to learn to code in Java.
- There is no usage metering capabilities. For example: once everyone has Java Applets on their Web pages, and anyone can run them, how do you charge people for your software? Many believe hooks for this kind of processing needs to be built into the Java language.
- Other areas of Java that need to be improved:
    - Lacks multimedia objects.
    - Lacks a Visual Development Environment like Visual Basic or Delphi - although many vendors are working on this right now, including Borland, Symantec, Powersoft, IBM, Sunsoft, and many others.
    - No imaging support.
    - Only supports Sun's audio format.

4. Circuit Description

The Datalogger has three operation modes. These are the initialisation mode, the data logging mode and the data-downloading mode.

In initialisation mode, the user specifies the start time of data logging and scanning interval – i.e. the period between two consecutive data loggings. Plugging the Datalogger to the serial port of a host computer does this.

After initialisation, the logger enters data logging mode. It can now be disconnected from the computer and placed to a designated location. The Datalogger converts analogue signal into digital data at a fixed interval and stores the data into memory with a time stamp comprising the year, month, day, of week, hour, minute, second and temperature into memory.

Data logging is terminated either by pressing the reset button on the logger or when memory is full. At this point, the logger is connected to the host computer once more for data downloading. During downloading, the data stored in the Datalogger is transferred in to the host computer.

Figure 4-1 shows the logger's block diagram. The system comprises of six units. They are:

- Central controller based on the ADuC812
- LM031L LCD unit
- 24LC64 memory unit
- DS1302 real time clock
- Power supply

The system utilises only three key ICs, namely the controller, the memory and the real time clock. The ADuC812 has a built in analogue-to-digital converter with a conversion accuracy of 12-bits, it also has a built in UART which is used to communicate to and a temperature sensor The 64Kbyte EEPROM and the real time clock communicate with the ADuC812 via an I^2C bus. The ADuC812 manages responses to incoming event signals and stores time stamps in the memory. It also controls communication with the host computer via the serial port.

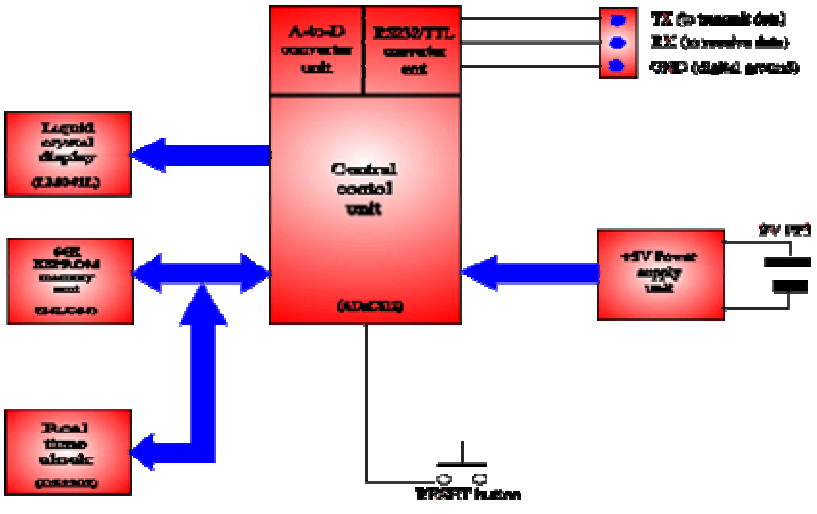
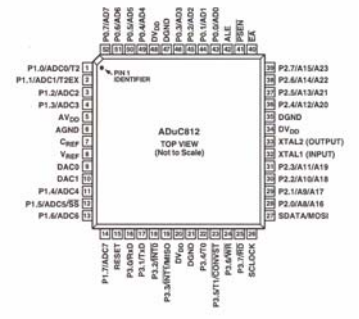


Fig. 4-1. The central control unit controls all operations of the Datalogger. Four main elements are the a-to-d converter unit, the real-time clock the memory unit and the RS232/TTL converter.

4.1 Central Control Unit



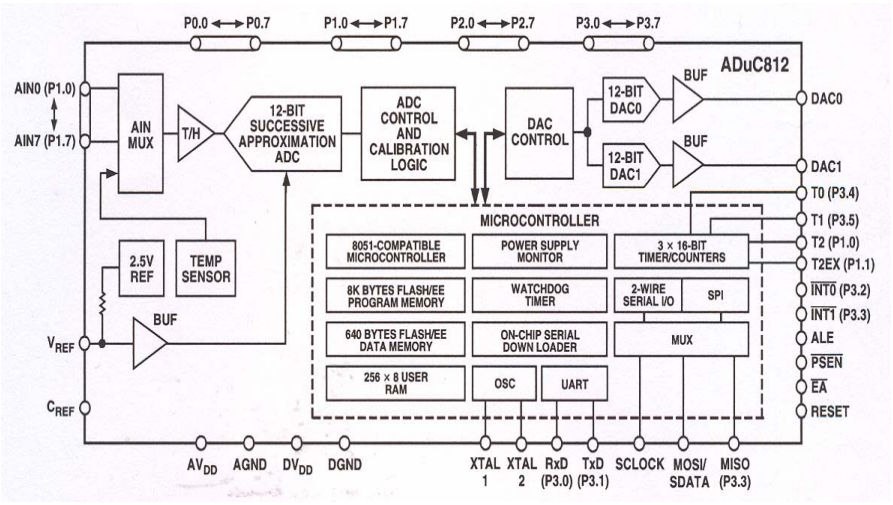
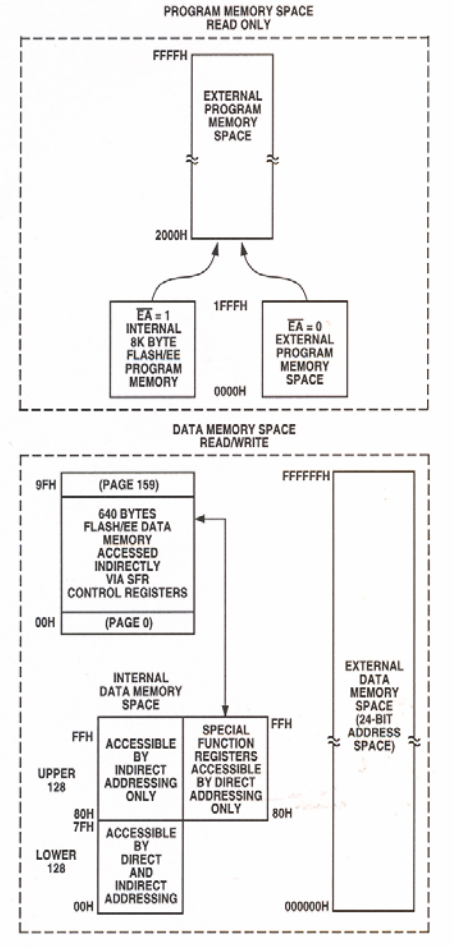


Fig. 4-2. Pin-out and the internal block diagram of the ADuC812 microcontroller. This is a 52-pin quad pack device.

The ADuC812 is a highly integrated high accuracy 12-bit data acquisition system. At its core, the ADuC812 incorporates a high performance 8-bit (8051 compatible) MCU with on-chip reprogrammable non-volatile flash/EE program memory controlling a multichannel (8-input channels), 12-bit ADC. The pin-out and the internal block diagram of the ADuC812 are shown in **fig. 4-2**.



The chip incorporates all secondary functions to fully support the programmable data acquisition core. These secondary functions include User Flash Memory, Watchdog Timer (WDT), Power Supply Monitor (PSM) and various industry standard parallel and serial interfaces.

As with all 8051 compatible devices, the ADuC812 has separate address spaces for program and data memory as shown in **fig 4-3**. Also shown in **fig 4-3**, additional 640 bytes of flash/EE data memory are available to the user. The flash/EE data memory area is accessed indirectly via a group of control registers mapped in the special Function Register (SFR) area.

The lower 128 bytes of internal data memory are mapped as shown in **fig 4-4**. The lower 32 bytes are grouped into four banks of eight registers addressed as R0 through R7. The next 16 bytes (128 bits) above the register banks form a block of bit addressable memory space at bit addresses 00H through 7FH.

Fig. 4-3. Shows the internal address space for program and data memory associated with compatible 8051 devices.

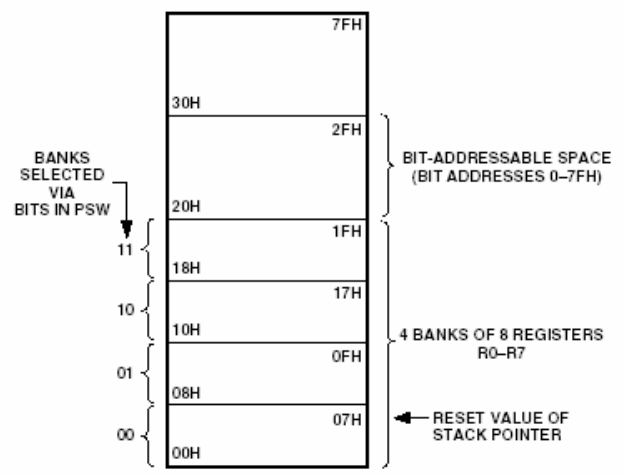


Fig 4-4. Shows the lower 128 bytes of internal memory.

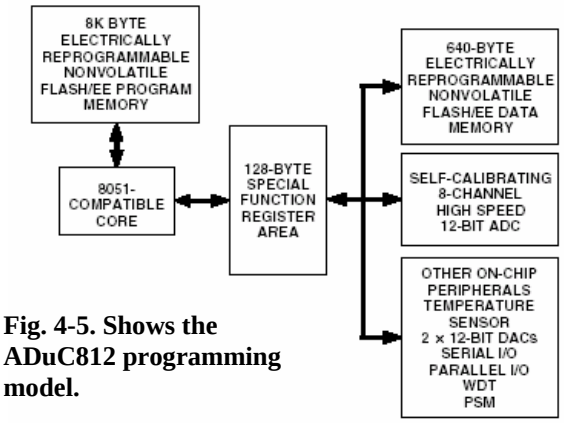


Fig. 4-5. Shows the ADuC812 programming model.

The SFR space is mapped in the upper 128 bytes of internal data memory space. The SFR area is accessed by direct addressing only and provides an interface between the CPU and all on-chip peripherals. The block diagram of the programming model for the ADuC812 via the SFR area is shown in **fig 4-5**. ACC is the Accumulator register and is used for math operations including addition, subtraction, integer multiplication and division, and Boolean bit manipulations. The mnemonics for accumulator-specific instructions refer to the Accumulator as A.
The B register is used with the ACC for multiplication and division operations. For other instructions it can be treated as a general-purpose scratchpad register. The Data Pointer is made up of three 8-bit registers, named DPP (page byte), DPH (high byte) and DPL (low byte). These are used to provide memory addresses for internal and external code access and external data access. It may be manipulated as a 16-bit register (DPTR = DPH, DPL), although INC DPTR instructions will automatically carry over to DPP, or as three independent 8-bit registers (DPP, DPH, DPL).

The ADuC812 provides nine interrupt sources with two priority levels. Interrupt priority within a given level is shown in descending order of priority in **fig. 4-6**, which gives a general overview of the interrupt sources and illustrates the request and control flags. The interrupt vector addresses for corresponding interrupts are also included in **Table 4-1** To use any of the interrupts on the ADuC812, the following three steps must be taken.

1. Locate the interrupt service routine at the corresponding Vector Address of that interrupt. See **Table 4-1**.
2. Set the EA (enable all) bit in the IE SFR to “1”.
3. Set the corresponding individual interrupt bit in the IE SFR to “1”.

Interrupt	Interrupt Name	Interrupt Vector Address	Priority Within Level
PSMQ	Power Supply Monitor	43H	1
IE0	External INT0	03H	2
ADCI	End of ADC Conversion	33H	3
TF0	Timer 0 Overflow	0BH	4
IE1	External INT1	13H	5
TF1	Timer 1 Overflow	1BH	6
I2CI/ISPI	Serial Interrupt	3BH	7
RI/TI	UART Interrupt	23H	8
TF2/EXF2	Timer 2 Interrupt	2BH	9

Table 4-1. Shows the Interrupt Vector Addresses.

Three SFRs are used to enable and set the priority for the various interrupts, for this project however the priority levels are kept to their default settings and only interested in the first Interrupt Enable SFR.
The bit designation for this register is shown in appendices A

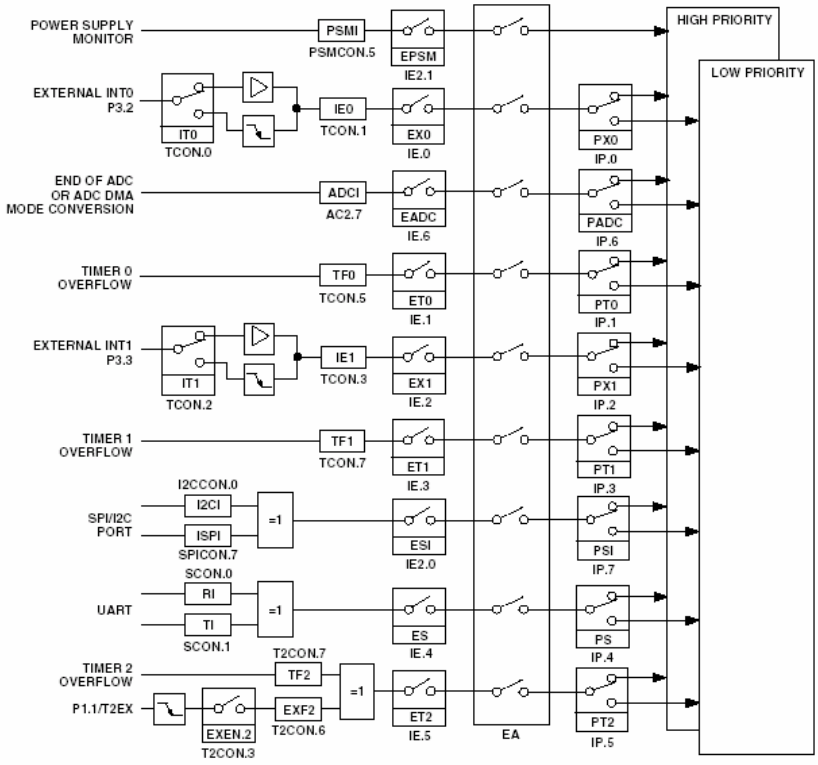


Fig. 4-6. Shows the Interrupt Request Sources.

4.2 Analogue-to-Digital Converter Unit

The analogue-to-digital converter (ADC) conversion block incorporates a 5µs, 8-channel 12-bit, single supply analogue-to-digital converter. This block provides the user with multichannel mux, track/hold, on-chip reference, calibration features and analogue-to-digital converter. All components in this block are easily configured via a 3-register SFR interface.

The analogue-to-digital converter consists of a conventional successive approximation converter and accepts an analogue input range of 0 to $+V_{REF.}$ A high precision, low drift calibrated 2.5V reference is provided on-chip. The internal reference may be overdriven via the external V_{REF} pin.

Single step or continuous conversion modes can be selected in software or, alternatively by applying a convert signal to an external pin however, the Datalogger is configured for single step conversions and will be configured to automatically start a new conversion on each overflow of the **Scan_Rate** (see 5.6), thereby allowing repetitive conversions at a user selectable sample rate.

The analogue input range for the ADC is 0v to $V_{REF.}$ For this range, the designed code transitions occur midway between successive integer LSB values. The ideal input/output transfer characteristics for the 0 to V_{REF} range is shown in **fig 4-8**.

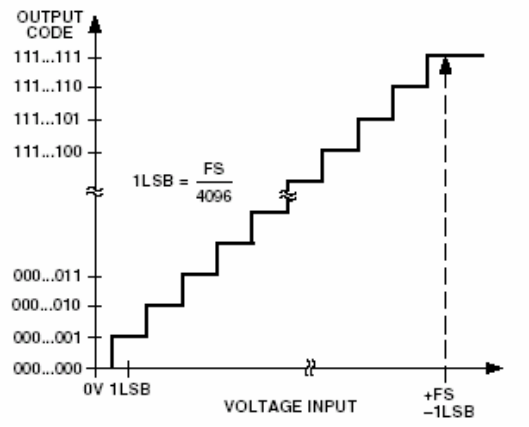


Fig. 4-8. Shows the analogue-to-digital conversion transfer function.

The ADC will convert the analogue input and provide an ADC 12-bit result word in the ADCDATAH/L SFRs. The top four bits of the ADCDATAH SFR will be written with the channel selection bits to identify the channel result. The format of the ADC 12-bit result word is shown in **fig. 4-9**.

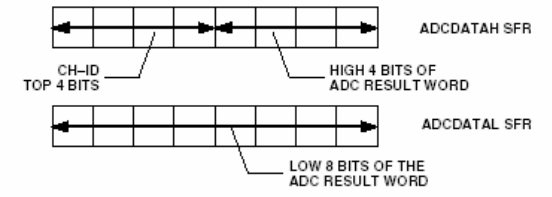


Fig. 4-9. Shows the analogue-to-digital conversion result format.

The on-chip ADC has been designed to run at a maximum speed of one sample every 5µs (i.e., 200KHz sampling rate). Therefore, in an interrupt driven routine the user software is required to service the interrupt, read the ADC result and store the result for further post processing, all within 5µs otherwise the next ADC sample could be lost. This however, will not affect the Datalogger as it is configured as single step and the minimum amount of time between consecutive conversions will be one minute.

4.3 Memory Unit

The memory unit is 24LC64 64Kbyte 2.5V *SmartSerial* EEPROM, which can be written to and erased up to 1'000'000 times. This requires a power supply 2.5V to 6V with a typical current consumption of 1mA in active mode and 1μA in standby mode.

Again, this device uses the I^2C bus for data transfer and operates as a slave device. Pin-out and the internal block diagrams of the chip are given in **Fig. 4-10 and 4-11**.

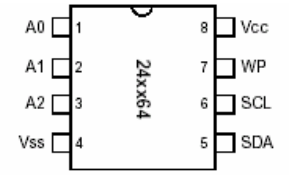


Fig. 4-10. Shows the pin-out of the 24LC64.

Lines A_0, A_1 and A_2 set the address of the chip. This allows up to eight chips to be used on the same bus, giving 512Kbyte of memory capacity. Lines designated SCL and SDA are the clock

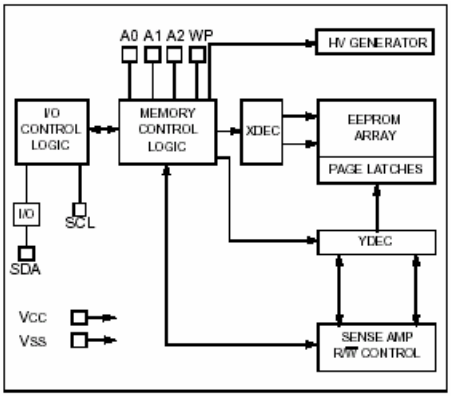
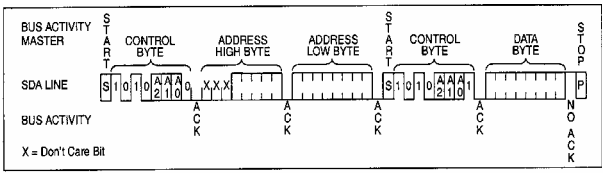


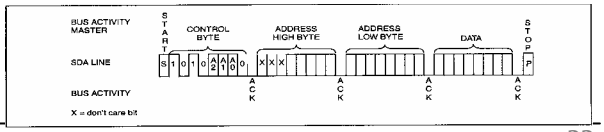
Fig. 4-11. Details of the 24LS64 64Kbyte EEPROM. It has an I^{2C} bus comprising a clock line, SCL, and a data line, SDA.

and data lines of the I^2C bus. Data can be written to and read from the ROM via the I^2C bus. The write operation has two modes: byte-write mode and page-write mode. The former writes a single byte to a memory location. Page-write mode writes 64 bytes to a block in one go.

The read operations can be carried out in one of three modes: current address read, random read and sequential read. The byte write modes and random read modes are used in this application. Their timing sequence is described below, **Fig. 4-12**.

Fig. 4-12. Timing sequence required for reading and writing 24LS64 EEPROM. The top diagram is the byte write and the bottom diagram is a random read.





The byte-write operation is as follows. Following a start condition on the I^2C bus, the control code, 1010_2, the ADuC812 places the device address on A2, A1 and A0, and the R/-W bit on to the bus. The R/-W bit should be zero to indicate a write operation. Address lines A_2, A_1 and A_0 should be the same as the hardware setting on the memory chip.

The next byte transmitted by the ADuC812 is the high-order byte of the address and will be written into the address pointer of the 24LS64. The following byte is the least significant address byte. After receiving another acknowledge signal from the 24LS64 the ADuC812 transmits the data byte into the memory.

Random-read mode allows the ADuC812 to access any memory locations in a random manner. Following a start condition on the I^2C bus, the control code, 1010_2, the ADuC812 places the device address, A2, A1 and A0 and the zero R/-W bit into the bus.

The following byte transmitted by the ADuC812 is the high-order byte of the word address and will be written into the address pointer of the 24LS64. The next is the least significant address byte.

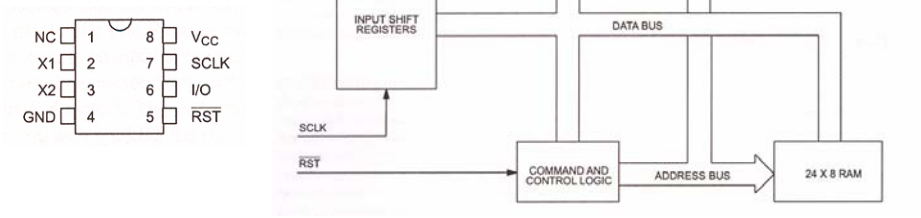
After receiving another acknowledge signal from the 24LS64 the ADuC812 generates a start condition again and then it transmits the control byte to the memory. This time the R/-W bit is 1 to indicate a read operation. The 24LS64 acknowledges and outputs the addressed byte bit by bit. The ADuC812 finally generates a stop condition.

SCL and SDA are both controlled by the ADuC812. Both lines are pulled to +5V to form the I^2C bus. The ADuC812 permanently sets the SCL line on as an output. Data line SDA is set as an input to the ADuC812 or an output according to I^2C bus operations.

4.4 Real Time Clock Unit

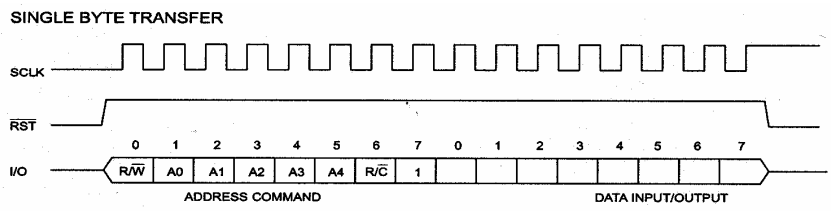
The Real Time clock is a Dallas DS1302. It also communicates to the microprocessor by I^2C bus. The real time clock provides seconds, minutes, hours, day, date, month and year information and can operate in either the 24-hour or 12-hour format with an AM/PM indicator. Data can be transferred to and from the clock/RAM one byte at a time or in burst of up to 24 bytes. The DS1202 is designed to operate on very low power and retain data and clock information on less than 1 microwatt.

Fig. 4-13. Pin-out and internal block diagram of the DS1302 timekeeper. It has and I2C bus, making the hardware design easier.



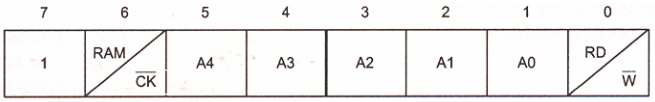
The pin-out and internal block diagram of the DS1302 is shown in **fig 4-13**. To initiate any transfer of data RST is taken high and eight bits are loaded into the shift register providing both address and command information. Data is serially input on the rising edge of the SCLK as shown in **fig 4-14**.

Fig. 4-14. Timing sequence of the DS1302 real time clock.



The first eight bits specify which of the bytes will be accessed, weather a read or write cycle will take place, and weather a byte or burst mode transfer is to occur. After the first eight clock cycles have occurred which load the command word into the shift register, additional clocks will output data for a read or input data for a write.

Fig. 4-15. Format of the command byte to initialise either a read or write operation to the timekeeper.



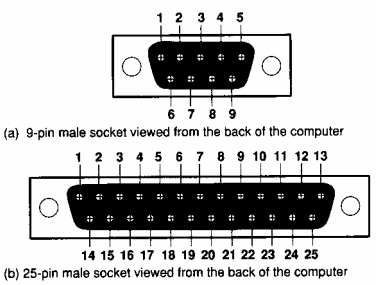
The command byte is shown in **fig 4-15**. Each data transfer is initiated by a command byte. The Most Significant Byte (bit 7) must be logic one. If it is a zero, further action will be terminated. Bit 6 specifies clock/calendar data if logic zero or RAM data if logic one. Bits one through five specify the designated registers to be input or output, and the Least Significant Byte (Bit 0) specifies a write operation (input) if logic zero or read operation (output) if logic zero. The command byte is always input starting with the LSB (bit 0).

4.5 RS232/TTL Translator Unit

The function of this unit is to perform voltage conversions between RS232 and TTL logic levels. Since the ADuC812 has a built-in UART the voltage conversions are taken care of by the ADuC812. Rx line is the line from which the logger receives data. The Tx signal is the signal output from the logger, RS232 voltage level. The pin-out of the PC's RS232 port connector and its functions are given in **Fig. 4-16**.

25 Pin	9 Pin	Name	Direction (from PC)	Description
1		**Prot**	----------	Protective ground
2	3	**TD**	Output	Transmit data
3	2	**RD**	Input	Receive data
4	7	**RTS**	Output	Request to send
5	8	**CTS**	Input	Clear to send
6	6	**DSR**	Input	Data set ready
7	5	**GND**	-------	Signal ground
8	1	**DCD**	Input	Data carrier detect
20	4	**DTR**	Output	Data terminal ready
22	9	**RI**	Input	Ring indicator
23		**DSRD**	I/O	Data signal rate detector

Fig. 4-16. Pin-out of the RS232 port on IBM compatibles. In my project, only the Tx transmits output from the PC, the Rx input and GND are used.



The serial port on the ADuC812 is full duplex, meaning it can simultaneously transmit and receive. It is also receive-buffered, meaning it can commence reception of a second byte before a previously received byte has been read from the received register. However, if the first byte still hasn't been read by the time reception of the second byte is complete, one of the bytes will be lost.

5. ADuC812 Software Development

5.1 Introduction

Section 4 established the hardware that is necessary and how it will interact with the ADuC812. This section will describe the development of the assembly code that the ADuC812 interprets to perform tasks. It is clear that the Datalogger has three main tasks:

1. To acquire data by serial transmission and to set itself up from this data received
1. To acquire temperature at some fixed interval according to the way in, which it is set up and store these temperatures recorded along with the time and date in, which they were recorded into memory
1. To obtain the data store in memory at any time and transmit it serially.

From this, it is clear that the code will have three main subrountines used to control the ADuC812. **Figure 5-1** shows an overall view at the structure of the code and the main subrountines in question. This section will describe more in depth how these main subrountines are implemented, as well as other fragments of code necessary to allow the overall Datalogger to operate properly.

5.2 Description of Main Code

The following descriptes the path in, which the program can take depending on the information it receives from the serial port. This should be read in conjunction with the flowchart in fig. 5-1. Note this is an overall view of the program. The exact coding implementation used is described later in this section.

Main: The program starts of by initializing and clearing the LCD, setting 9600 baud for the UART, setting up the ADuC812 as an I^2C master, enabling the external interrupt and all other interrupts and clearing all flags on the ADuC812. It then goes into the **Waiting** subroutine.

Waiting: The **Waiting** subroutine continually polls the serial port (SBUF) for a control word from the LoggerUI application. Once it receives control byte from the serial port it is moved into the accumulator. If the value in the accumulator is equal to AAh the program enters the **Data_Logged** subroutines. **Data_Logged** subroutine is the data-downloading mode where it obtains the data stored in memory from the data loggings and sends them up the UART to the LoggerUI application.

Next: If the value isn't equal to AAh it jumps to the **Next** subroutine were it tests the value with 55h. If it is equal the program enters initialization mode were it sets up the current time on the Real Time Clock the specific start time of data logging and the scanning interval i.e. the period between two consecutive data loggings. Once this is complete the program enters the **Scan_Rate** subroutine (see **fig. 5-12**).

Warn: If the value isn't equal to 55h it jumps to the **Warn** subroutine were it loops continually flashing the LED to indicate that a problem has occurred.

When the external interrupt is trigged at any stage during program execution the program will jump to the **Waiting** subroutine and wait once again for a control byte from the LoggerUI application. Normally this will only occur if the user wishes to cease the data logging instead of waiting until the memory becomes full.

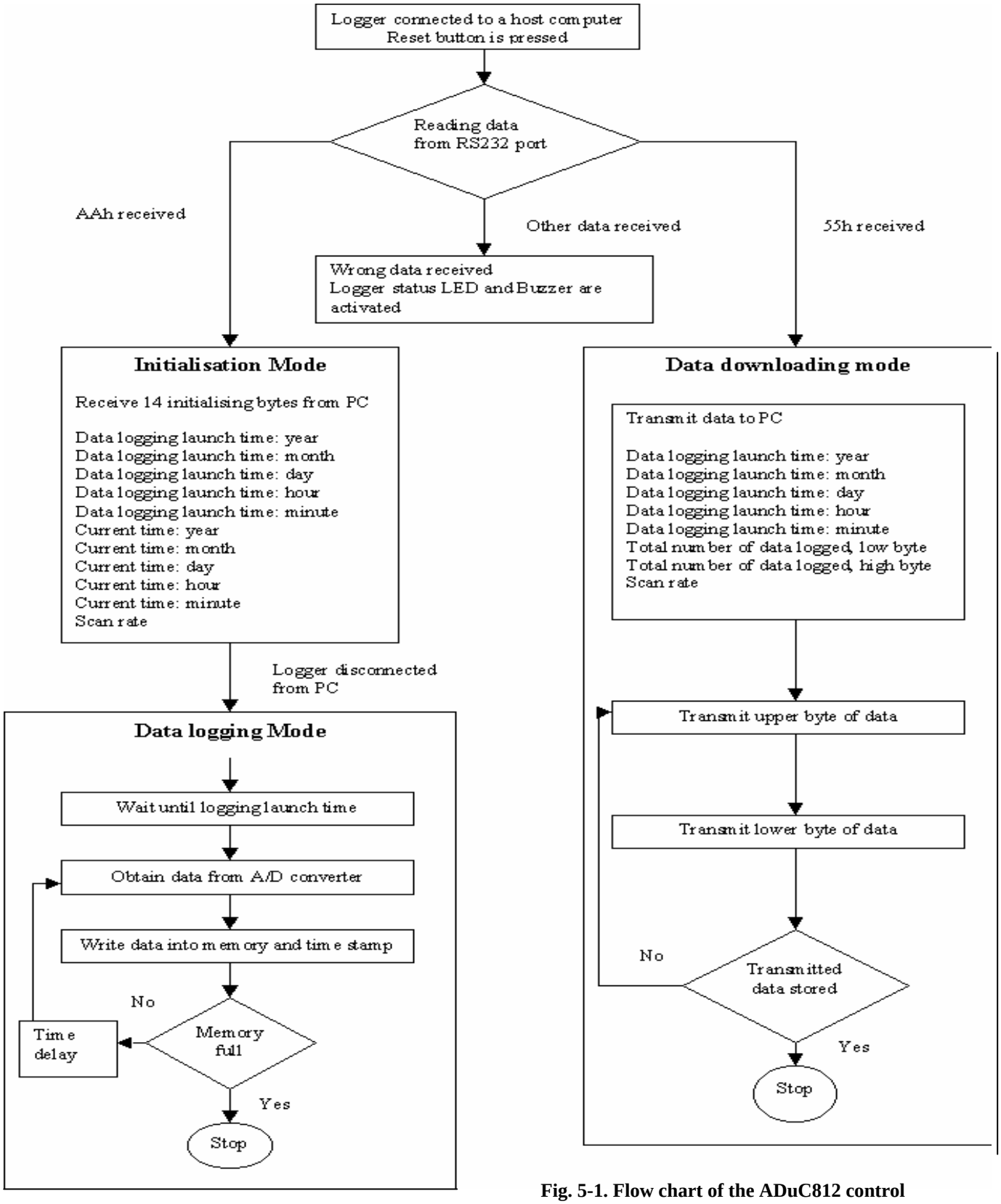


Fig. 5-1. Flow chart of the ADuC812 control program. Three main procedures are involved. The initialization procedure, the data logging procedure and the downloading procedure.

5.2 LCD Operation

A very popular standard exists that allows communication with vast majority of LCDs regardless of their manufacture. The standard is referred to as HD44780U, which refers to the controller chip that receives data from an external source (in this case, the ADuC812) and communicates directly with LCD.

The 44780 standard requires three control lines as well as either four or eight I/O lines for the data bus. In the Datalogger a 4-bit data bus is used, the LCD will require a total of seven data lines (three control lines plus a four lines for the data bus), this gives the Datalogger extra I/O pins if the Datalogger ever needed to be modified in the future to add extra functionality or just adding user friendly features like LEDs or buzzers.

The three control lines are referred to as EN, RS, and RW.

The EN line is called "Enable". This control line is used to tell the LCD that data is being sent. To send data to the LCD, the program first sets this line high (1) and then sets the other two control lines and puts data on the data bus. When other lines are completely ready, EN is brought low (0) again. This 1-0 transition tells the 44780 to take the data currently found on the other control lines and on the data bus and to treat it as a command.

The RS line is the "Register Select" line. When RS is low (0), the data is to be treated as a command or special instruction (such as clear the screen, position the cursor, etc.). When RS is high (1), the data being sent is text data that should be displayed on the screen. For example, to display the letter "T" on the screen you would set RS high.

The RW line is the "Read/Write" control line. When RW is low (0), the information on the data bus is being written to the LCD. When RW is high (1), the program is effectively querying (or reading) the LCD. Only one instruction ("Get LCD status") is a read command. All other are write commands – so RW will almost be low.

Finally, the data bus consists of four lines the only drawback of using 4-bits is that commands and data have to be sent in two nibbles (4-bit parts) to the display, this takes slightly more time. This won't be a problem in the Datalogger. Since using 4-bit mode, data bytes and command bytes are read and written in two separate 'nibbles' (4-bit parts), therefore there are two subroutines one to read two nibbles from the LCD, and the other to write two nibbles to the LCD. Furthermore, the toggling of the EN-line is also taken care of in these subroutines, as you need to toggle for each nibble.

5.3.1 Program Variables

DB4: Equates to pin 4 on port 1

DB5: Equates to pin 5 on port 1

DB6: Equates to pin 6 on port 1

DB7: Equates to pin 7 on port 1

EN: Equates to pin 7 on port 3

RS: Equates to pin 6 on port 3

RW: Equates to pin 5 on port 3

DATA: Equates to port 1

5.3.2 Description of Code

The following describes how the LCD initializes itself and how it clears and writes characters to the display. These descriptions should be read in conjunction with the flowcharts in **figures 5-2 to 5-7.**

Initialization LCD: Set and clear the appreciate control lines (`EN`,`RS` and `RW`)
Send `28h` to the LCD => 4-bit data bus, 5x8 dot character font and two-line display
Send `0Eh` to the LCD => Turn on LCD and the cursor
Send `06h` to the LCD => Every time character received the cursor position automatically moves to the right

Clear LCD: Set and clear the appreciate control lines (`EN`,`RS` and `RW`)
Send `01h` to the LCD => Command used to clear the screen on the LCD

The two above subroutines set up the LCD for when data becomes available and needs to be displayed. These subroutines use other subroutines to achieve this.

Read_Nibble: Release datalines (set output latches to '1') so we can read the LCD. Read the high nibble (4-bits) then read the low nibble and combine the two together to form 8-bits and store in the accumulator.

Write_Nibble: Push the accumulator (save the original byte) mask out the lower 4-bits and sent out the high nibble to the LCD. Pop the accumulator (restore the original byte) mask out the upper 4-bits and set out the lower nibble to the LCD.

Wait_LCD: Call **Read_Nibble** and check the most significant bit in the accumulator. If the bit is set then the LCD is still busy so call **Wait_LCD** again until the bit is clear indicating that the LCD is ready and turn off **RW** for future commands to the LCD.

Write_Text: Call **Write_Nibble** and **Wait_LCD**.

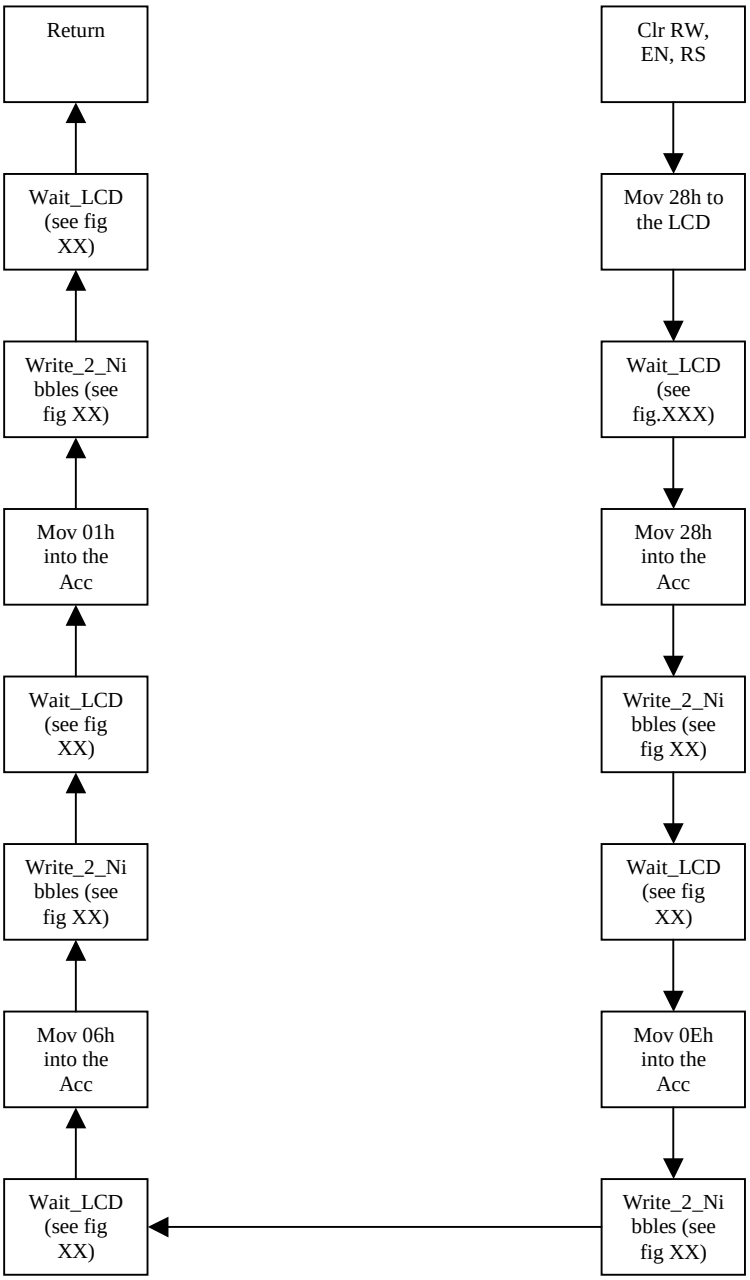


Fig. 5-2. Shows the procedure that the ADuC812 has to perform at the start of the main program in order to initialize the LCD. If this subroutine isn't called the LCD will not be able to display characters.

Fig. 5-3. Shows the Write_Nibble routine used to write a single character to the LCD.

Fig. 5-4. Shows the Read_Nibble routine used in the checking of the busy status of the LCD.

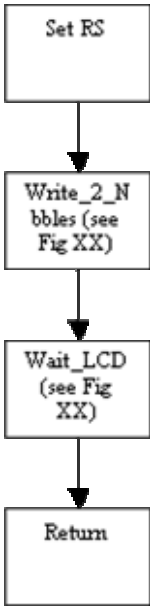


Fig. 5-5. This is the Write_Text rountine, which is called when data in the Acc needs to be displayed on the LCD.

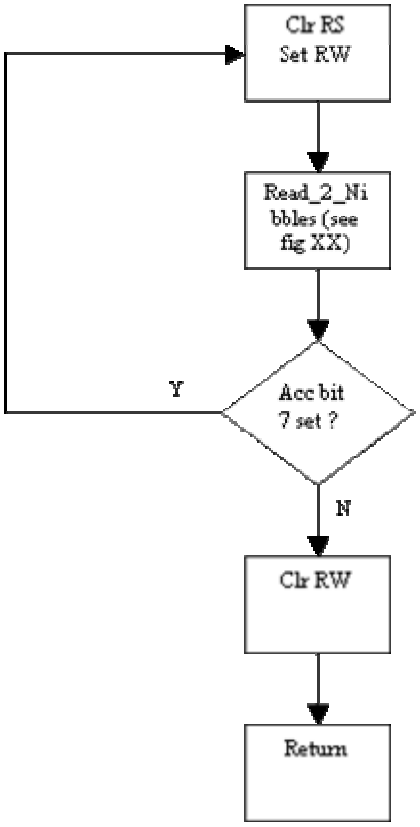


Fig. 5-6. This is the Wait_LCD routine, which is used to determine if the LCD is ready to receive another character.

5.4 Temperature Operation

The temperature sensor on the ADuC812 outputs a voltage that is *inversely* proportional to the chip temperature. At 25°C, this voltage is approximately 600mV. As temperature changes, the voltage changes by –3mV/°C. So an operating temperature *rises*, the temperature sensor output voltage *decreases*. And of course the ADC is used to convert the voltage on the temperature sensor to a digital value. It should be noted that the two figures quoted above of 600mV @ 25 degrees and –3mV/°C are typical figures.

5.4.1 Description of Code

The following describes how the ADC on the ADuC812 is set up for a single conversion and how the 16-bits obtained from the conversion is converted into a decimal value. This description once again should be read in conjunction with the flowchart in **fig. 5-7.**

Configuration: The ADC is configured for normal mode, ADC clock divide bits is set to four, ADC acquisition select bits is set to zero, timer mode and external event bits disabled. The ADC interrupt bit is set.

ADC: Initiate a single AD conversion; the ADC Interrupt Service Routine (ISR) is called upon completion, which calls **CONVERT_TEMP** subroutine.

Convert_Temp: ADCDATAH/L are the registers that the ADuC812 stores the result from an ADC. The upper 4-bits of the ADCDATAH indicate the channel that the conversion was taken from, therefore they are of no use and are masked out. These are combined with upper 4-bits of the ADCDATAL and are stored in the accumulator as they are more accurate that the lower 4-bits of the ADCDATAL. Before the lower 4-bits of the ADCDATAL are masked out they are stored in register three for future use.

The value that is in the accumulator has a value of 58h subtracted from it. This value is a value that has been calculated to correspond to zero Celsius. Since the ADuC812 on board temperature sensor is inverted, meaning that if the temperature raises the value from the ADC becomes smaller. Unless the temperature is zero Celsius the resulting subtraction will always produce a minus answer e.g. 10 – 15 = -5. Therefore the sign bit will always be set and from this the **Convert_Temp** subroutine can take two different paths of execution, one to indicate that it is above zero Celsius and other to indicate that it is below zero Celsius.

Hundreds: The result from the subtraction is held in the accumulator and is divided by 100. The accumulator receives the integer quotient and B receives the remainder. If accumulator contains zero the remainder that is in B is moved into accumulator otherwise whatever value that is in the accumulator is stored in memory and sent out to the LCD and then the value from B is moved into the accumulator.

Tens: The value that is in the accumulator is now divided by 10. After the second division the accumulator receives the integer quotient once again and B receives the remainder. The value in the accumulator is stored in memory and sent out to the LCD once again.

Units: The remainder that is in B is moved into accumulator and is stored in memory and sent out to the LCD. The value that is in register three is now moved into the accumulator (the ordinary value from ADCDATAL). The upper 4-bits are masked out and has Fh subtracted from it. This is now divided by 10 and the value in the accumulator is once again stored in memory and sent out to the LCD. Before the result from the subtraction is stored in memory and sent out to the LCD the value A5h is stored in memory and also sent out to the LCD which represents '.' character in ASCII. Finally after the result from the subtraction the characters ^{0}C are stored in memory and sent out to the LCD.

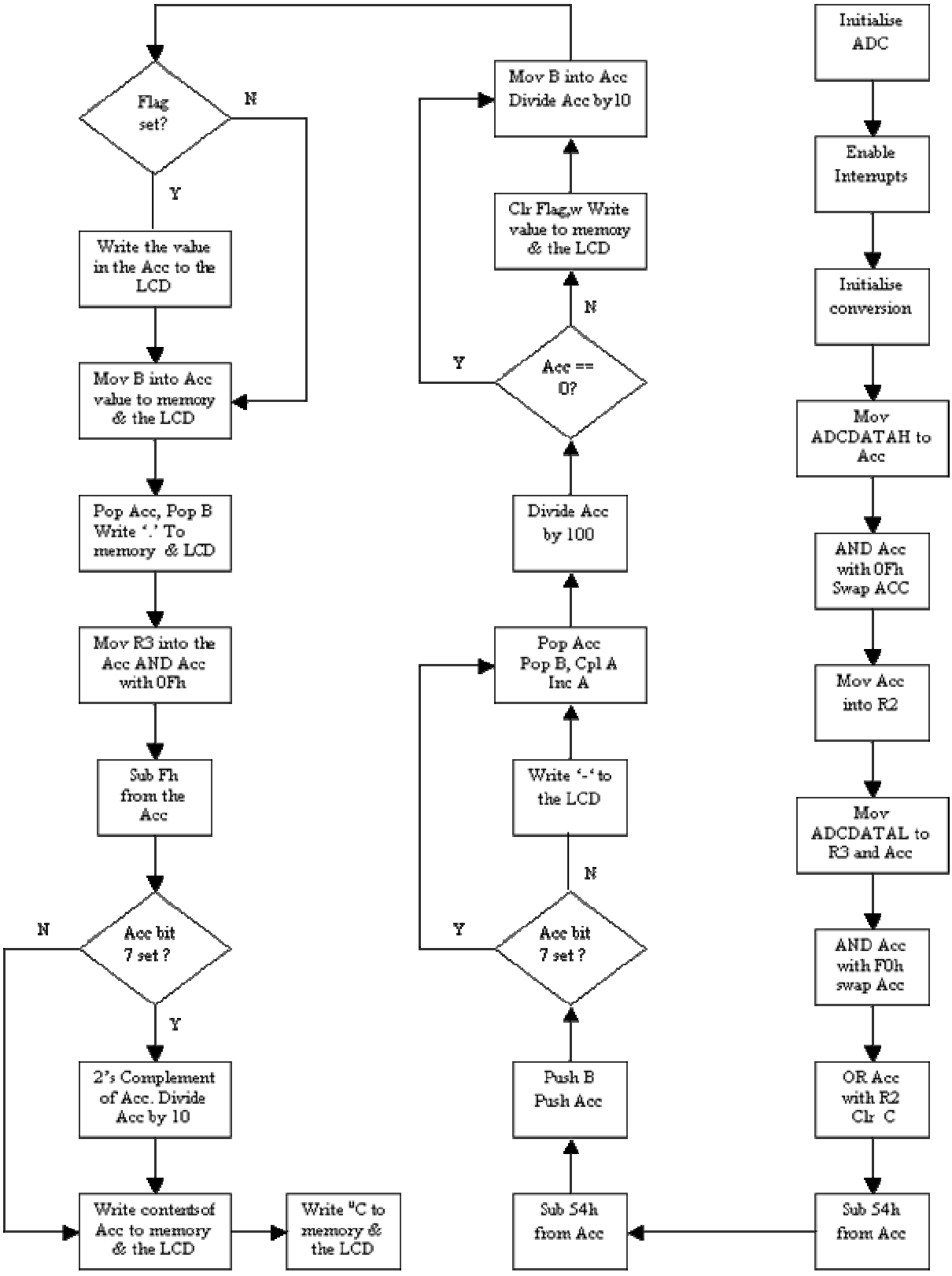


Fig. 5-7. This shows the procedure that the ADuC812 nees to perform in order to convert the 16-bit value received from the ADC and convert it into a decimal temperature value and store it in memory.

5.5 ADuC812 I²C Operation

The ADuC812 acts an I²C software master and is programmed to 'bit bang' the SDATA and SCLOCK lines. Master mode is selected by setting the I2CM bit in the I2CCON register on the ADuC812.

To transmit data on the SDATA line, the MDE bit must first be set to enable the output driver on the SDATA pin. The MDO bit in the I2CCON register is the Data Out bit. The output driver on the SDATA pin will either pull the SDATA line high or low depending on whether the MDO bit is set or cleared.

The MCO bit in the I2CCON register is the clock output bit. The output driver on the SCLOCK pin is always enabled in master mode and will either pull the SCLOCK line high or low depending on the whether the bit MCO is set or cleared.

On the ADuC812 there is no pull-up on the SDATA/SCLOCK output driver hence external pull-ups are implemented to pull this line high. To receive data, the MDE bit must be cleared to disable the output driver on SDATA. The software is used to toggle the MCO bit (send out a clock pulse) and read the status of the SDATA line via the MDI bit. The MDI bit is set if SDATA is high and cleared if SDATA is low (provided MDE is cleared). The software must also control the MDO, MCO and MDE bits appropriately to generate the START condition, slave address, acknowledge bits, data bytes and STOP conditions appropriately. The data is latched into MDI on a rising edge SCLOCK only if MDE is cleared.

5.5.1 Program Variables

SLAVEADDH: SLAVEADDH holds the high byte address of the slave.

SLAVEADDL: SLAVEADDL holds the low byte address of the slave.

OUTPUT: Output holds the value to be transmitted to the slave.

INPUT: Input is the value received from the slave. This value is sent out the UART.

NOACK: This is set if a NACK is received when an ACK is expected.

ERR: The ERR flag is set if the NOACK is set anywhere in the program and allows the NOACK flag to be cleared in the software.

5.5.2 Description of Code

The following description should be read in conjunction with the main flowcharts in **fig. 5-8** and **fig. 5-9** and also the two separate flowcharts for **RCVDATA** and **SENDDATA** in **fig. 5-10** and **fig. 5-11**.

When the Datalogger is in data logging mode the program sends the START condition, the slave control byte, R/*W* bit (cleared to indicate a transmission) to the slave and the slave high and low address bytes. The ADuC812 transmits eight bytes to the slave (year, month, day, hour, minute, second and temperature recorded). After the transmission of each byte it examines the ACK which indicates to the ADuC812 that the slave has received the last byte and when all eight bytes are transmitted the ADuC812 sends the STOP condition.

When the Datalogger enters data-downloading mode the ADuC812 program sends the START condition, the slave control byte, the R/*W* bit (cleared to indicate a transmission) to the slave and the slave high and low address bytes. After the word address is sent, the ADuC812 generates a START condition following the ACK from the slave. This terminates the write operation, but not before the internal address pointer is set in the slave. Then the ADuC812 issues the control byte again but with the R/W bit set to initiate a master-reception to the slave. The ADuC812 then receives a single byte from the slave and sends back a ACK and the STOP condition. The ACK indicates to the slave that the master has received the last byte to be transmitted by the slave. The ADuC812 then transmits the received byte up the UART to the PC where it is used by the LoggerUI application. The master program jumps back to the start and receives another byte from the slave again.

Configuration: The UART is configured for 9600 baud.

Initialization: The I2C registers and flags are initialized.

I2CCON = A8h	=>	Master Mode
		Disables the output driver on SDATA
		SCLOCK float high
OUTPUT = 0		Initial byte to be transmitted is ‘0’.

Note: Since the I^2C interface is in master mode there is no need to enable the I^2C interrupt, or put a value into I2CADD.

Transmission: Transmission of a byte is done as follows (see **SENDDATA fig. 5-11**)

1. Send the START bit.
2. Send the Slave control byte (manipulated with R/*W* bit clear for reception).
3. Send the Slave high and low byte addresses.
4. Check the ACK.

5. If an NACK is received send a STOP bit and set the ERR flag.
6. If an ACK is received then send 64 clocks to the slave device. With each clock the MDO bit in I2CCON should be loaded with the appropriate value from the data byte in the accumulator reading the MDI bit after each clock is transmitted.
7. Check the ACK.
8. If a NACK is received then set the ERR flag.
9. Send STOP bit.

Reception: Reception of a byte is done as follows (see RCVDATA figure 7a)

1. Send the START bit.
2. Send the Slave control byte (manipulated with R/*W* bit clear for reception).
3. Send the Slave high and low byte addresses.
4. Check the ACK.
5. If an NACK is received send a STOP bit and set the ERR flag.
6. Send the START bit.
7. Send the Slave control byte (manipulated with R/*W* bit set for reception).
8. If an ACK is received then send 8 clocks to the slave device reading the MDI bit after each clock is transmitted. After 8 clocks the received byte is saved in the accumulator.
9. Send NACK to indicate that this is the last byte to be received.
10. Send STOP bit.
11. If a NACK is received then set the ERR flag.

Check ERR: Check the ERR flag to see if an error occurred. If an error occurred send an error message up the UART to the PC.

Delay: This delay (approx five machine cycles) is only used to slow the program down and give the slave time.

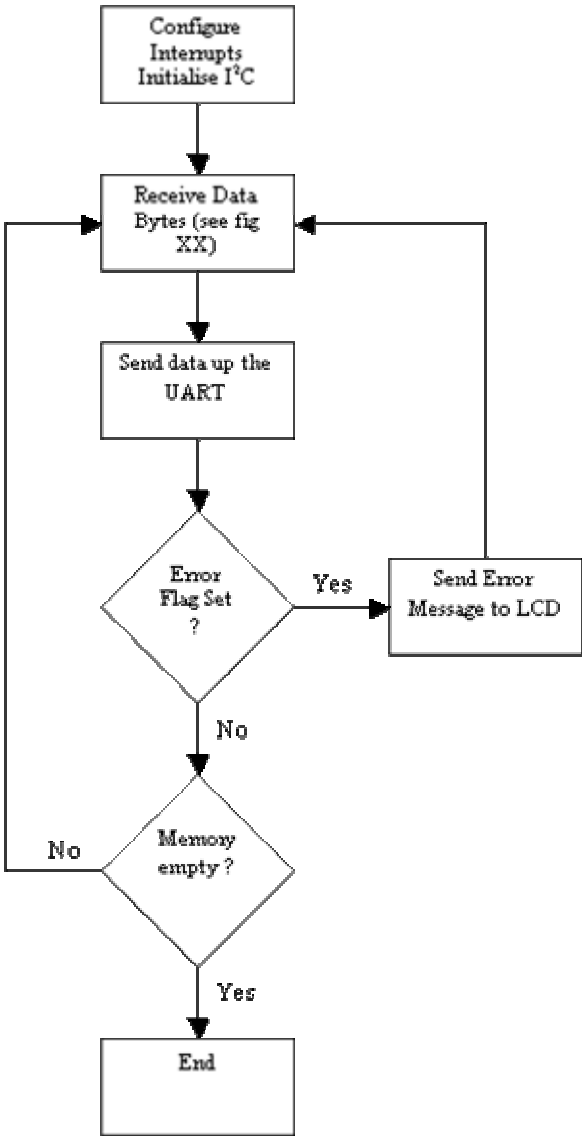


Fig. 5-8. Shows the procedure the ADuC812 performs when it wishes to obtain data from the memory chip during download mode.

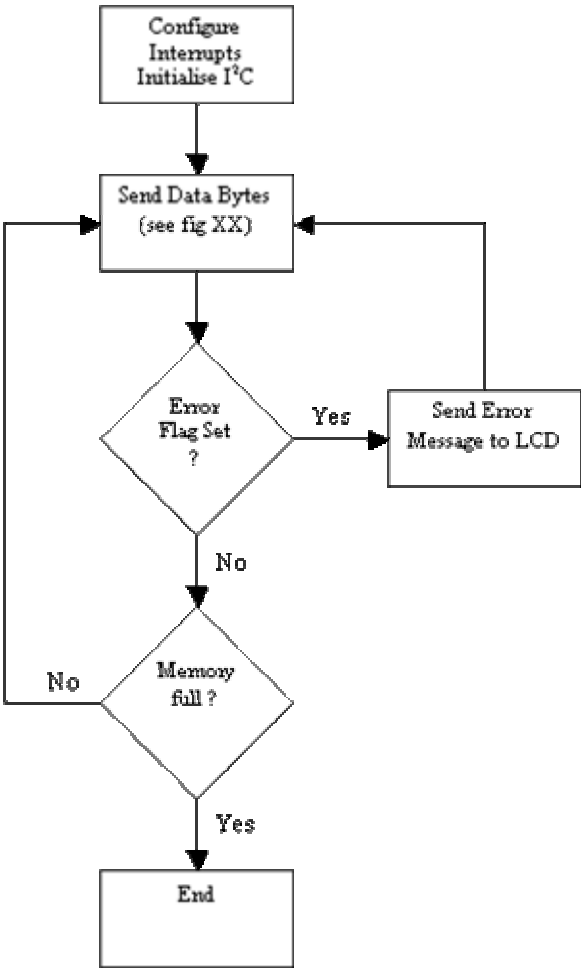


Fig. 5-9. Shows the procedure the ADuC812 performs when it wishes to write data to the memory chip during data logging mode.

Fig. 5-10. RCVDATA subroutine used to obtain data from memory chip

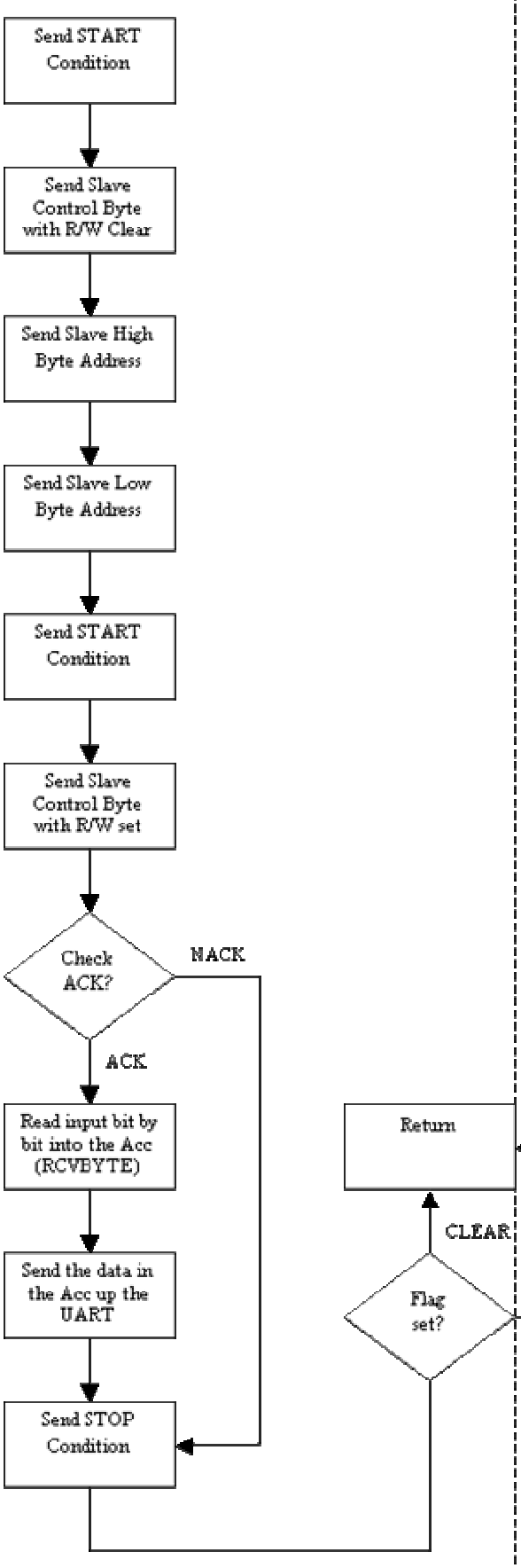
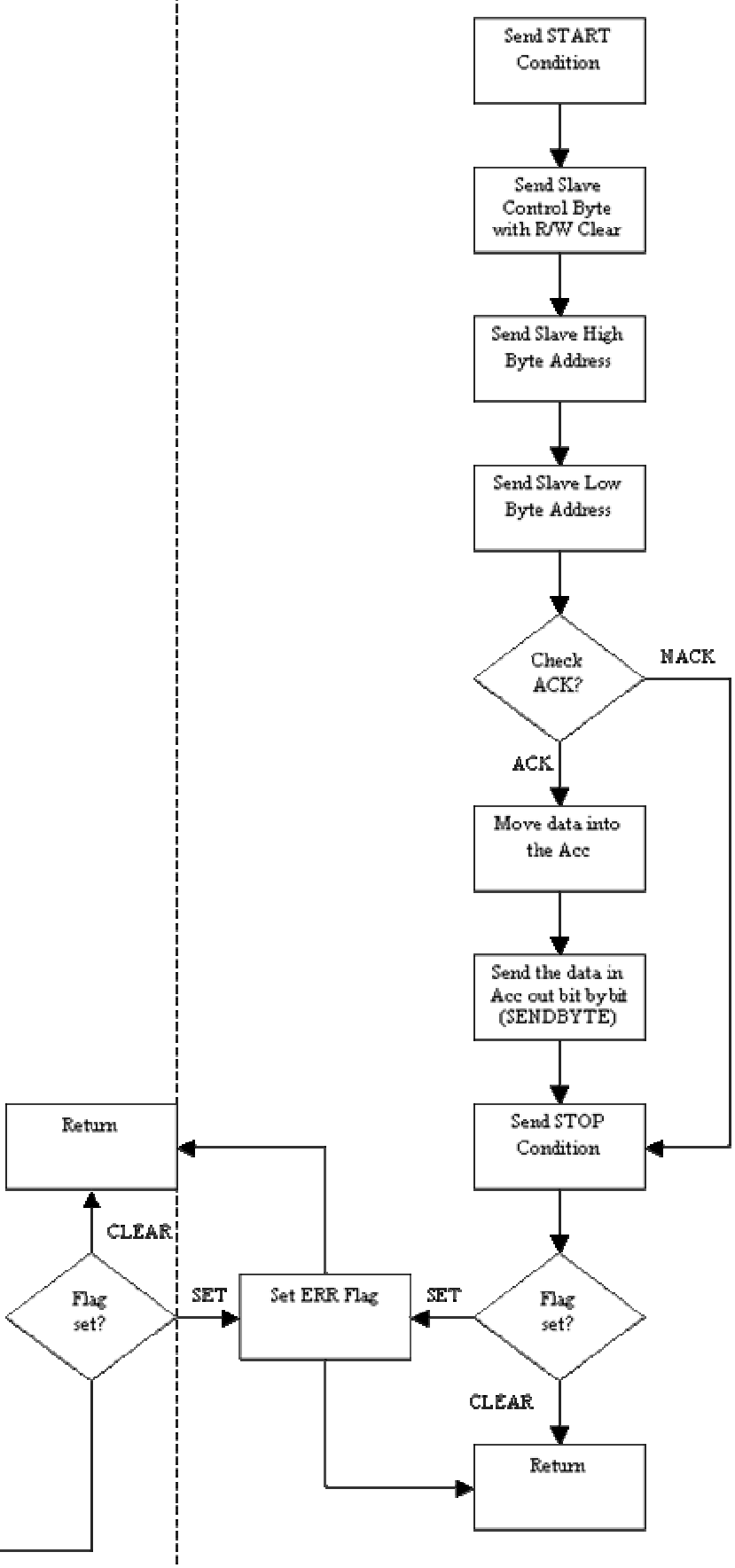


Fig. 5-11. SENDDATA subroutine used to send data from memory chip



5.6 Scan Interval Loop Operation

This scan interval loop is basically a large delay depending on the value that was received on intialisation. One way to approach this is to develop a software real time clock, though this involved some calculations e.g. overflow, frequency of the Timer and also involved using some interrupts.

Instead, since there is a delay of 100 milli-seconds being used on the LCD code this was used and was called ten consecutive times producing a delay of one second. This produces the building blocks of the scan interval loop.

5.6.1 Program Variables

TICKS: This holds the number of time the **Delay** subroutine has to be executed.

DELAYSEC: This is the number of seconds that have passed.

SCANRATE: This holds the scanning interval - i.e. the period between two consecutive data loggings. This value obtained during initialization.

MINUTES: This is the number of minutes that have passes

5.6.2 Description of Code

The following description should be read in conjunction with the flowchart in **fig 5-12**

Scan_Rate: This subroutine determines when an AD conversion should be called. On entry to the **Scan_Rate** subroutine the **Delay** subroutine is called. The **Delay** subroutine is a subroutine that delays for 100 milli-seconds. Once the **Delay** subroutine is completed the **TICKS** variable is decremented. If **TICKS** variable is decremented to zero then the **TICKS** variable is reset to its default value of 10 and the **DELAYSEC** variable is incremented indicating that one second has passed, otherwise it loops back to the **Delay** subroutine. Therefore every time **TICKS** decrements to zero the **DELAYSEC** variable is incremented by one. **DELAYSEC** is then checked against the value 60 to see if a minute has passed if so the **MINUTES** variable is incremented by one. The **MINUTES** variable is then checked against the **SCANRATE** variable and once these match an AD conversion is called. Once the AD conversion is completed it checks to see if memory is full on the Logger. If memory is full the program jumps to the **Waiting** subroutine (see **fig. 5-1**) otherwise it enters the **Scan_R**ate subroutine once again. When any of the checks fail the **Delay** subroutine is called.

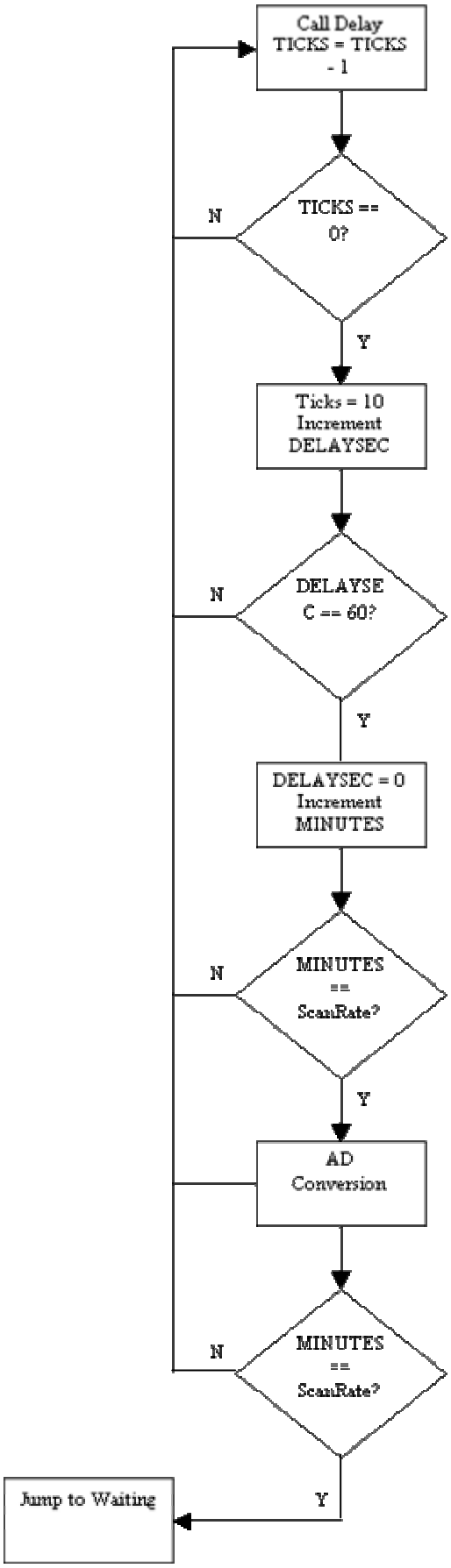


Fig. 5-12. Shows the flowchart of the Scan Interval Loop. SCANRATE value is defined on initialization by the user.

5.7 Real Time Clock Operation

The Real Time Clock is connected to port 0 of the ADuC812 and interacts by using synchronous serial communication. All data transfers are initiated by driving the RST input high. The RST input serves two functions. First, it turns on the control logic, which allows access to the shift register for the address/command sequence. Second the RST signal provides a method of terminating either single byte or multiple byte data transfer.

5.7.1 Program Variables

RST:	Equates to pin 7 on port 0
IO:	Equates to pin 5 on port 0
SCLK:	Equates to pin 3 on port 0
CENABLE:	This preset value starts the Real Time Clock
WENABLE:	This is a preset value that write enables the Real Time Clock
WSECOND:	This is a preset value that enables writing to the seconds register
RSECOND:	This is a preset value that enables reading from the seconds register
WMINUTE:	This is a preset value that enables writing to the minutes register
RMINUTE:	This is a preset value that enables reading from the minutes register
WHOUR:	This is a preset value that enables writing to the hour register
RHOUR:	This is a preset value that enables reading from the hour register
WDAY:	This is a preset value that enables writing to the day register
RDAY:	This is a preset value that enables reading from the day register
WMONTH:	This is a preset value that enables writing to the month register
RMONTH:	This is a preset value that enables reading from the month register
WYEAR:	This is a preset value that enables writing to the year register
RYEAR:	This is a preset value that enables reading from the year register
WCONTROL:	This is a present value that enables writing to the control register
RCONTROL:	This is a present value that enables reading from the control register
WCB:	This is a present value that enables writing to the Real Time Clock in burst mode.
RCB:	This is a present value that enables reading from the Real Time Clock in burst mode.

5.7.2 Description of Code

The following describes the sysnchronous serial communication between the ADuC812 and the Real Time Clock and should be read in conjunction with the flowcharts in **figures 5-13 to 5-15.**

RTC_Setup: This subroutine sets up the current time on the Real Time Clock according to the values that were sent down from the LoggerUI application. It gets these values from the memory locations that they were stored during initialization. Before it does this it sends out the **WCONTROL** value to the Real Time Clock through the **Write_RTC** subroutine, which disables the write protection register and allows the Real Time Clock to be written to. After this it sequentially sends out **WYEAR** value followed by the value that was sent down from the LoggerUI application for the year value which is written into the year register on the Real Time Clock, next it sends out the **WMONTH** value followed by the month value to be written into the month register and so on until all the registers are set up. Next, it enables the write protection register on Real Time Clock and starts the Real Time Clock by sending out the **CENABLE** byte.

Write_RTC: This sets the reset pin on the Real Time Clock high which allows all data transfers to be initiated. Once the reset pin on the Real Time Clock is low all data transfers are terminated and the IO pin goes high impedance state. It then sends out the byte in the accumulator bit by bit to the Real Time Clock. This is done by clearing and setting the SCLK pin.

Read_RTC: This does exactly the same as the **Write_RTC** subroutine only that It calls a different routine to set and clear the SCLK pin. This routine first sets the SCLK pin and then clears it as opposed to clearing and then setting it. This is done because data inputs must be valid during the rising edge of the clock and data bits are output on the falling edge of the clock. The bits obtained from the Real Time Clock are read into the accumulator and are then stored in memory.

–

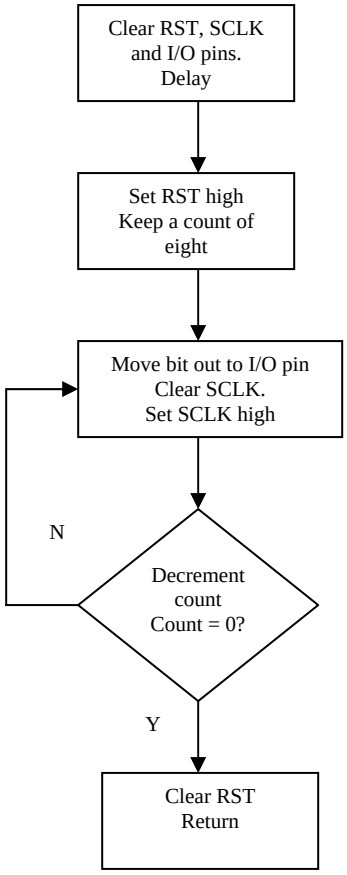


Fig. 5-13. This is the routine used by the ADuC812 to send a byte to the Real Time Clock, i.e. setting up the time of the Real Time Clock

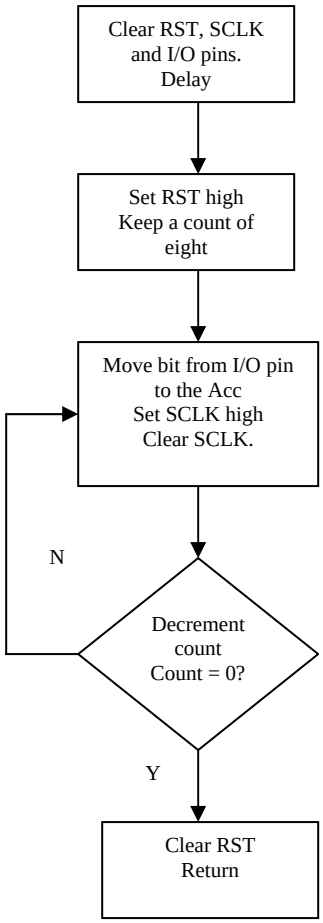


Fig. 5-14. This is the routine used by the ADuC812 to read a byte from the Real Time Clock, i.e. obtaining the time of the Real Time Clock.
Note the differences between the setting and clearing of the SCLK with fig. 5-13

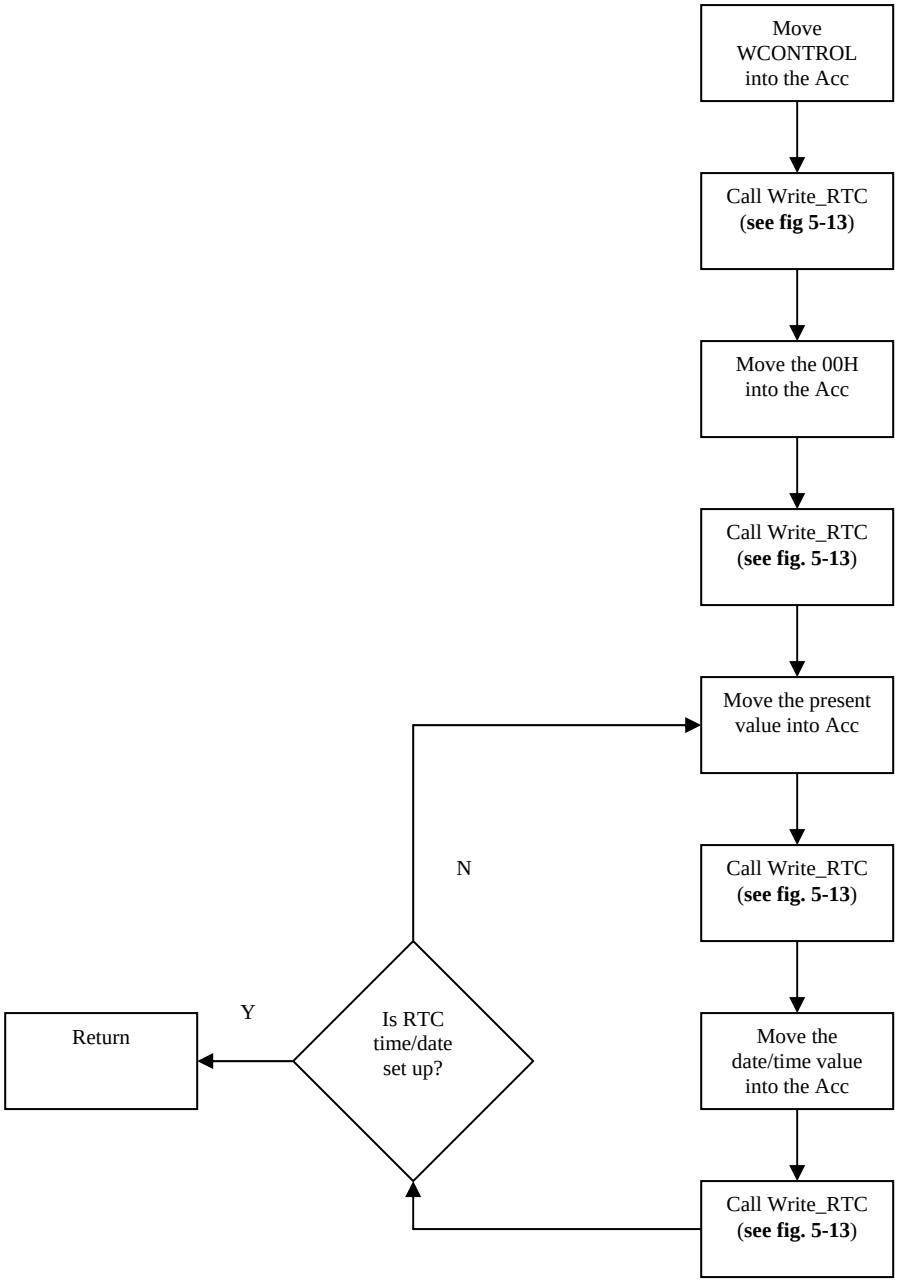


Fig. 5-15. This is the procedure that the ADuC812 has to perform in order to set up the time and date on the Real Time Clock. This routine executes until all the registers on the Real Time Clock are written to. There are eight registers in all to write to in order to set up the time and date on the Real Time Clock.

6. Java Software Development

6.1 Introduction

Designing object-oriented software is hard, and designing reusable object-oriented software is even harder. You must find pertinent objects, factor them into classes at the right granularity, define class interfaces and inheritance hierarchies, and establish key relationships among them. The design should to be specific to the problem at hand but also general enough to address future problems and requirements. You want to avoid redesign, or at least minimize it.

Design patterns help designers get a design "right" faster. Design patterns make it easier to reuse successful designs and architecture. Expressing proven techniques as design patterns make them more accessible to developers of new systems. Design patterns help choose design alternatives that make a system reusable and avoid alternatives that compromise reusability. Design patterns can even improve the documentation and maintenance of existing systems by furnishing an explicit specification of class and object interactions and their underlying intent.

6.1.1 What is a Design Pattern?

In general, a pattern has four essential elements:

1. The **pattern** name is a handle you can use to describe a design problem, its solutions, and consequences in a word or two. Naming a pattern immediately increases the design vocabulary. Having a vocabulary for patterns lets you to talk about them to other people, in the documentation and even yourself. It makes it easier to think about designs and t communicate them and their trade-offs to others.
2. The **problem** describes when to apply the pattern. It explains the problem and its context. It might describe specific design problems such as how to represent algorithms as objects. It might describe class or object structures that are symptomatic of an inflexible design.
3. The **solution** describes the elements that make up the design, their relationships, responsibilities and collaborations. The solution doesn't describe a particular concrete design or implementation, because a pattern is like a template that can be applied in many different situations. Instead, the pattern provides an abstract description of a design problem and how a general arrangement of elements solves it.
4. The **consequences** are the results and trade-offs of applying the pattern. The consequences for software often concern space and time trade-offs. They may address language and implementation issues as well. Since reuse is often a factor in object-oriented design, the consequences of a pattern include its impact on a system's flexibility, extensibility or portability.

`

6.2 LoggerUI Application

In the development of the LoggerUI application a number of Design patterns have been used. The patterns that have been used are classified as Creational, Structural and Behavioral Patterns. From the Creational Patterns the Abstract Factory, Singleton and Prototype are used for object creation and the Factory method is used for class creation.

The Structural Pattern, Decorator is used for object structural and the Template method is used from the Behavioral Pattern for class behavioral. The following is a description of the patterns used:

Abstract Factory: Provides an interface for creating families of related or dependent objects without specifying their concrete classes.

Factor Method: Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory method lets a class defer instantiation to subclasses.

Singleton: Ensure a class only has one instance, and provide a global point of access to it.

Prototype Method: Specify the kinds of objects to create using a prototypical instance and create new objects by copying this prototype.

Decorator: Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

The host computer controls the Datalogger through the LoggerUI application that has been designed and developed in Java. Also the link code that is used to communicate to the serial port and in return communicates to the ADuC812 microcontroller on the Datalogger is also done in Java.

The following is the specifications of the LoggerUI application and the link code that is needed in order for it to interact with the Datalogger correctly.

6.2.1 Specifications

- The user must be able to set the start date and time of data logging and the scanning interval i.e. the period between two consecutive temperature samples, though the LoggerUI application.
- If a date or time value is exceed then the system should produce an error e.g. you can't have 8 days in the week or 70 minutes in an hour.
- The code must convert the date and time into byte form in order for the ADuC812 to recognise it. It must also send the current time and date to the ADuC812 in order for it to count down to the start date and time.
- The user must be able to select a serial port of his/her choice and may initialize the Datalogger or download data from the Datalogger though that COM port that they have selected. This includes setting the buffer sizes, the baud rate etc. If the

serial port that is selected is in use by another application on the host computer an error should be displayed to the user telling them so.

- The LoggerUI application must be able to allow the user to download data from the Datalogger to the host computer by providing the user with the option of choosing the file path and file name for which they want to store their data.
- Data stored must be in text format and should include at the start of the file the start time and date of downloading, start time and date of which the logging mission started, the scanning interval that was chosen and the total number of temperatures recorded.
- The LoggerUI application will provide user with the option of viewing the temperature samples in graph form, with the option of printing them if the user wishes to.
- In addition to the above specification, addional features will be incorporated into the LoggerUI application that will make it more user friendly to the end user. These included help files, system check, system set-up and about forms.

6.2.2 Dataflow Diagrams

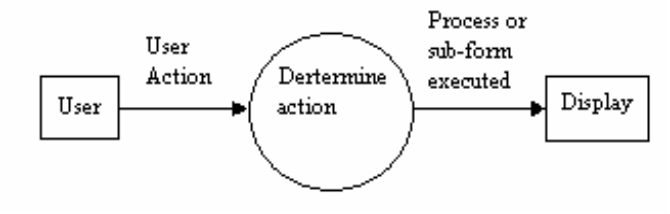


Fig 6-1. Top level Dataflow diagram showing the users input to select an action and displaying the action selected by the user

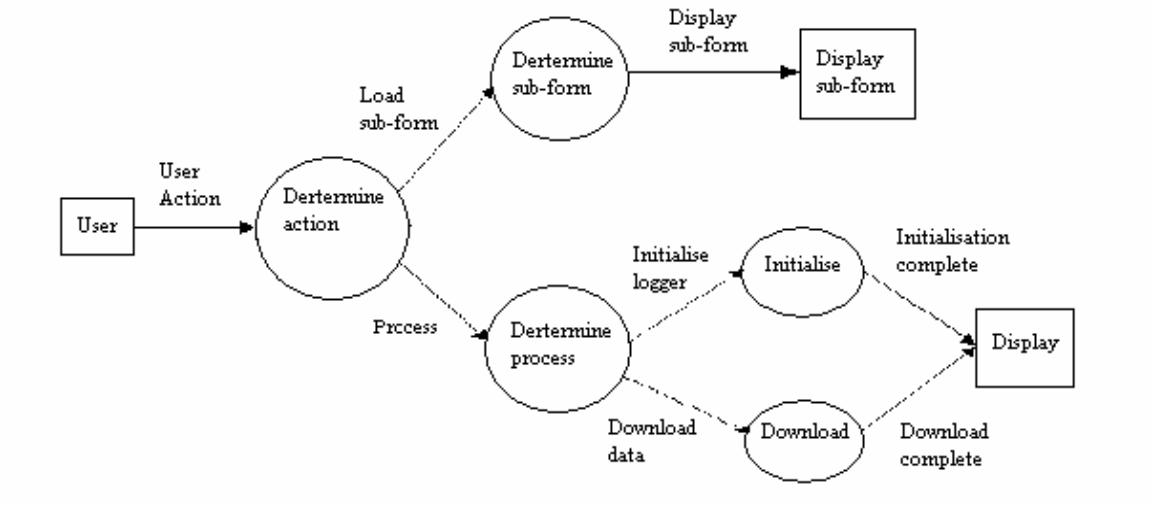


Fig. 6-2. The intial level 1 Dataflow diagram, indicating that there is a choice between what type of action is activated. Either a sub-dialog box is loaded to the screen or there is a process chosen, that is either initialize or download data with the necessary information being displayed to the screen.

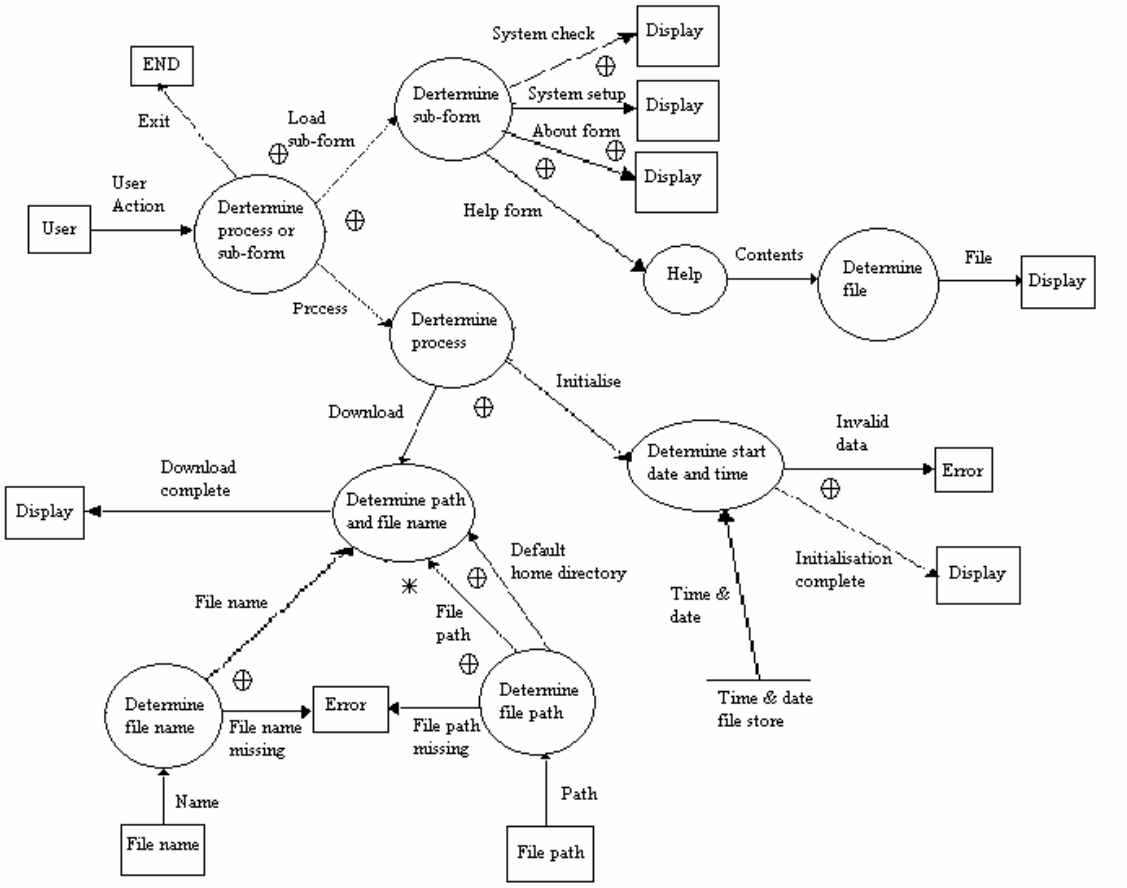


Fig. 6-3. The complete level 1 Dataflow diagram has been refined as much as possible. It shows that the user has the option of displaying four sub-dialog boxes to the screen, with the help dialog being able to provide the user with the choice of diplaying certain help files to the screen. The initialization process obtains data from the time and date store that has already been tested before being stored in this file. In the actual system this is not so but this is doe in order to make the Dataflow diagram earier to read and also simplifies the diagram. The file path and file name are both required to store the datadownloaded and an error is generated if either is missing.

6.2.3 Class Diagram

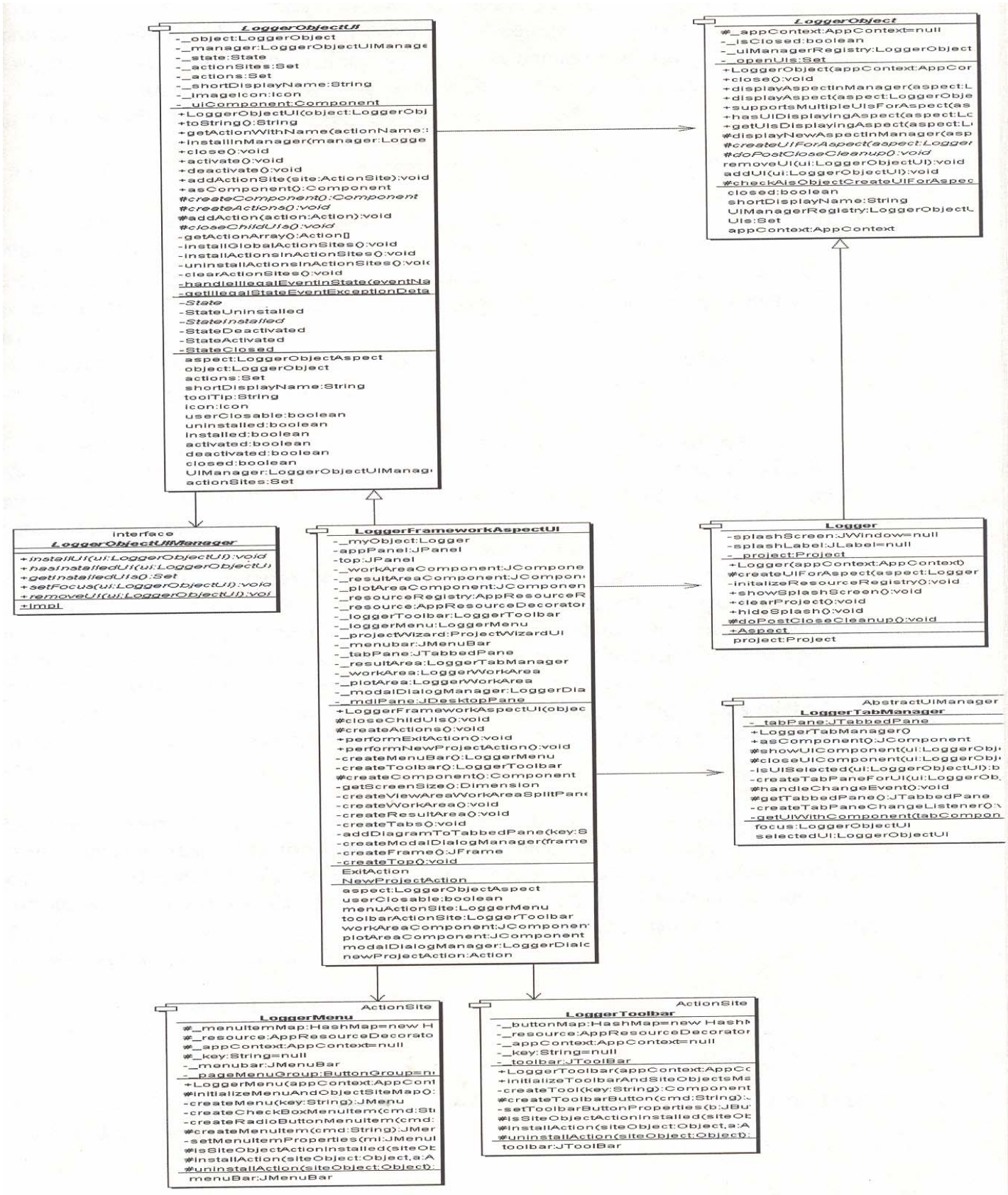


Fig. 6-4. Shows the class diagram of the LoggerUI application's superclasses, which all the framework derives of

6.2.4 LoggerUIs Superclsses

`LoggerObject` is an abstract class and implements the Abstract factory pattern providing most of the framework for domain-level objects in LoggerUI.
`LoggerObject` defines some of the behaviour for domain-level Logger objects. The primary behaviour a `LoggerObject` provides is the ability to display an aspect of itself as a user-interface (`LoggerObjectUI`) using the method:

```
displayAspect(aspect : LoggerObjectAspect) : LoggerObjectUI
```

LoggerObject's may support various aspects (`LoggerObjectAspect`) but all objects must at LEAST support a default aspect (`LoggerObjectAspect.DEFAULT`).
`LoggerObject` provides a simple implementation of `displayAspect`, which returns a LoggerObjectUI that display's, the LoggerObject's `toString()` method in as a `JLabel`.
A `LoggerObject` may be asked to close itself. When a `LoggerObject` is closed in this way, all its open LoggerObjectUIs will also be told to close themselves.

Subclasses of `LoggerObject` must override certain methods of `LoggerObject`. Subclasses must override the public static method `getTypeAspects`. This method returns as Set of LoggerObjectAspects that represent the aspects that instances of the subclass may display.
The LoggerObject Aspect class implements the prototype pattern. A LoggerObject creates a prototypical instance for each supported Aspect. In most cases the prototype aspect is all that is needed. However if you want to associate instance specific information that has a particular aspect then the prototype should be cloned before adding the instance specific information. An example of this would be a warning message Aspect that contains run time information in the warning message.
Subclasses should also override the `createUIForAspect` method. This method used to return the `LoggerObjectUI` appropriate for displaying a particular aspect of the object.

`LoggerObjectUI` (UI) is one of the LoggerUI Framework classes and also implements the Abstract factory pattern. Its role in the framework is to provide a graphical representation of a specific aspect of a framework object. This class should be subclassed for each aspect that a framework object wants to make visible to a user. Subclasses of `LoggerObjectUI` should provide a constructor, which takes an instance of the object whose aspect the UI is displaying.

```
public ResultAreaUI(ResultArea object) {
     super(object);
}
```

The subclass should override the `getAspect()`method to return the actual aspect that this ui is displaying.

The subclass should completely override the `createComponent()` method. The implementation of this method should create the graphical component required to display the appropriate aspect of the underlying object.

```
protected Component createComponent() {
      JPanel result;
      result = new JPanel;
      result.add(new JLabel(getShortDisplayName()));
      result.add(new JLabel(getFooProperty()));
      return result;
}
```

(Note: `LoggerObjectUI` provides a default implementation of `createComponent()` which returns a `JLabel` displaying the `toString()` method of its object. The subclass may also override the `createActions()` method. The implementation of this method should contain a call to `addAction(action)` for every Action that this UI supports.

```
protected void createActions() {
      addAction(getFooAction());
      addAction(getBarAction());
}
```

Any action sites added in this way are automatically handled by the framework (i.e. they are installed/uninstalled in external ActionSites as required).

6.2.5 LoggerUIs Framework Details

LoggerObjectUIs are created by a `LoggerObject` in response to a `displayAspect(aspect)` message. This is an Abstract factory method, subclasses implement this method and create appropriate LoggerObjectUI subclasses. Any subclass of LoggerObject can be asked to display an aspect and the caller doesn't have t care what concrete subclass of LoggerObjectUI is created. The `LoggerObject` is responsible
for working out which particular `LoggerObjectUI` to instantiate for the given aspect.

UIs provide a graphical representation of the object's aspect as an `AWT Component`. The installation of a UI in a `LoggerObjectUIManager` (through the `installUI (ui)` method **see fig. 6-5**) usually causes the UI's component to be displayed. The object is responsible for finding an appropriate UI manager and for calling the `installUI(ui)` method. This all happens in LoggerObject's `displayAspect(aspect)` method.

A UI maintains a collection of `Action` objects, which perform some operation against the UI or underlying object. Subclasses initialise the collection of actions by calling `addAction(action)` in the `createActions()`method. A UI may make these actions available through its own graphical representation but will more commonly make them available through an external `ActionSite` (e.g.: a menu bar or

toolbar). A UI may be associated with zero or more ActionSites through the `addActionSite` method. When a UI is activated it automatically installs any actions it knows about into its ActionSites and when it is deactivated, those actions are uninstalled from its ActionSites. UIs do not need to be told explicitly about ActionSites, the framework handles this automatically through the UI manager.
The `LoggerObjectUIManager` manages LoggerObjectUIs and is (usually) responsible for displaying them in some sort of UI container.

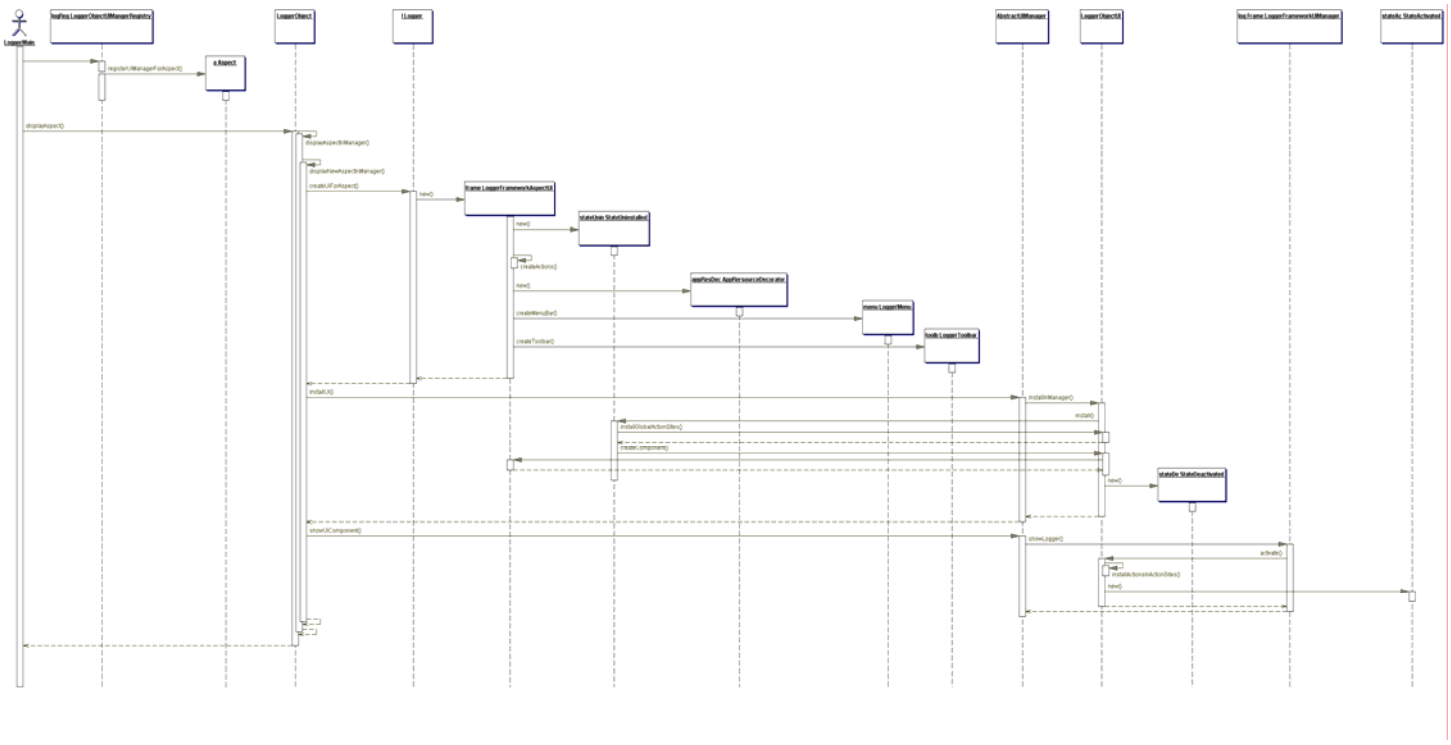


Fig. 6-5. Shows the sequence diagram of the template method that all LoggerObjects call when they want to display an aspect of themselves. This particular sequence diagram is the Logger object displaying an aspect of itself – the LoggerFrameworkAspectUI. This creates the LoggerMenu and LoggerToolbar respectively and installs whatever actions are available to itself at that time. The result of this is shown in fig. 6-6.

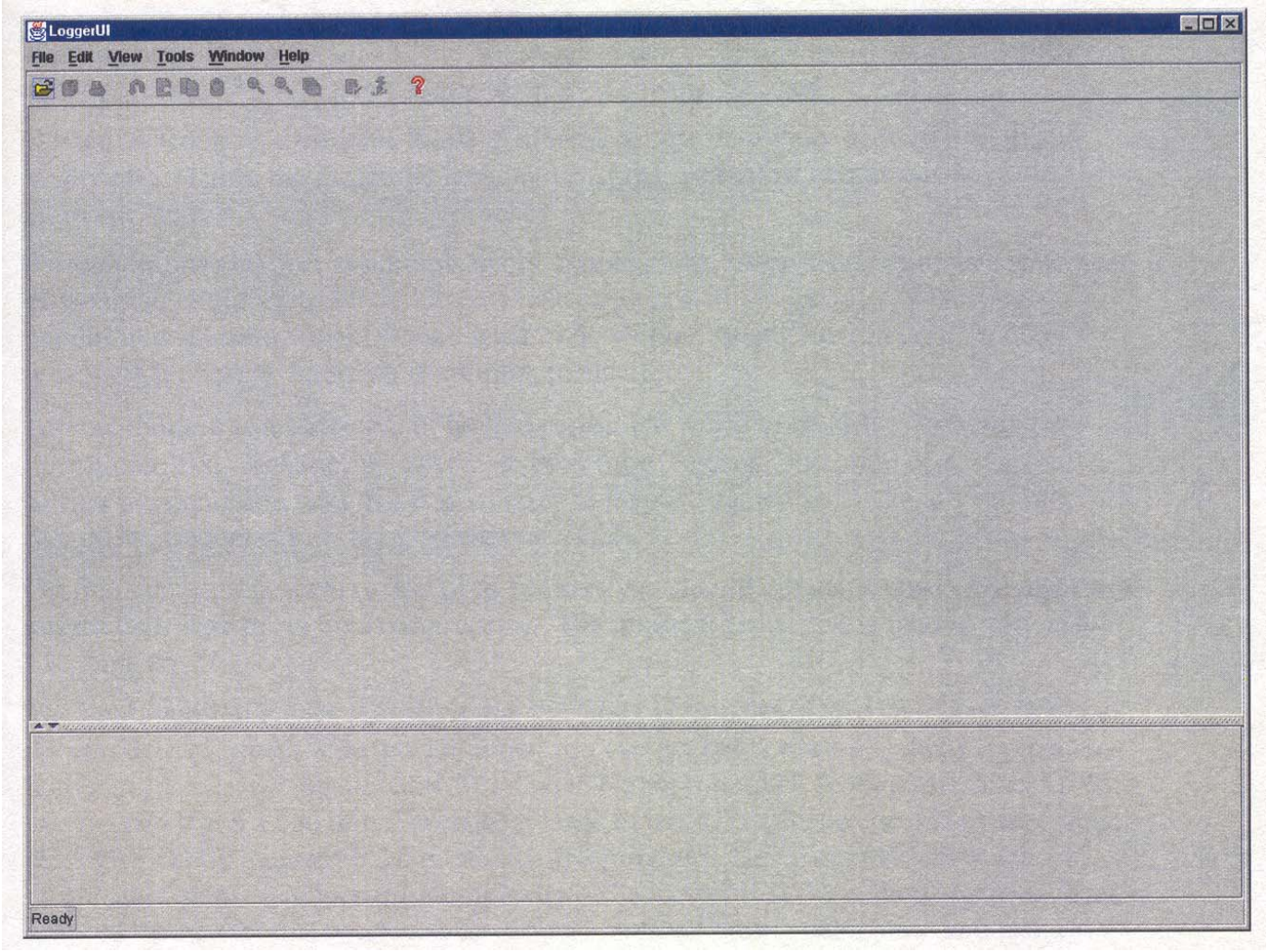


Fig. 6-6. This is the end result from calling the displayAspect on the Logger class. From the screen shot above the menu bar and toolbar are both visible and so is the status bar (bottom of the application with the default text set to "Ready"). The area above this is called the result area were some of the exception are displayed if they should occur. Above that area is the work area. When the user performs a new project action or opens an existing project this is were the Work Area and Plot Area tabs are placed. Note that the only actions that are available to the user at this stage on the toolbar, is to open an existing project or display the help dialog box.

6.3 Design by Contract

Design by contract is a design technique developed by Bertrand Meyer. At the heart of Design by Contract is the assertion. An assertion is a `Boolean` statement that should never be `false` and, therefore only `false` because of a bug. Typically, assertions are checked only during debug and are not checked during production execution. Indeed, a program should never assume that assertions are being checked.

Design by Contract uses three kinds of assertions: pre-condition, post-condition and invariants. Within the LoggerUI application only the pre/post conditions are used although there is a function for invariants.

Pre-conditions and post-conditions apply to operations. A post-condition is a statement of what the world should look like after the execution of an operation. The post-condition is a useful way of saying what we do without saying how we do it, in other words, of separating interface from implementation.

A pre-condition is a statement of how we expect the world to be before we execute an operation. We might define a pre-condition for a "square" operation of `this>=0`. Such a pre-condition says that it is an error to invoke "square" on a negative number and that the consequences of doing so are undefined.

On first glance, this seems a bad idea, because we should put some check somewhere to ensure that "square" is invoked properly. The important question who is responsible for doing so.

The pre-condition makes it explicit that the caller is responsible for checking. Without this explicit statement of responsibilities, you can get either too little checking (because both parities assume that the other is responsible) or too much (both parties check). Too much checking is a bad thing, because it leads to lots of duplicate checking code, which can significantly increase the complexity of a program. Being explicit about who is responsible helps to reduce this complexity. The danger that the caller forgets to check is reduced by the fact that assertions are usually checked during debugging and testing.

From these definitions of pre-condition and post-condition, you can see a strong definition of the term **exception**, which occurs when an operation is invoked with its pre-condition satisfied, yet cannot return with its post condition satisfied.

One of the dangers of polymorphism is that you could redefine a subclass's operation to be inconsistent with the superclass's operations. The post-conditions must apply to all subclasses. The subclasses can choose to strengthen these, but they cannot weaken them. The pre-condition, on the other hand, cannot be strengthened but may be weakened.

This looks odd at first, but it is import to allow dynamic binding. You should always be able to treat the subclass object as if it were an instance of the superclass (pre the principle of substitutability). If a subclass strengthened its pre-condition, then a superclass operation could fail when applied to the subclass. Pre-conditions are a statement of passing a responsibility on to the caller; you increase the responsibilities of a class by weakening a pre-condition. In practice, all of this allows much better control of subclassing and helps to ensure that subclasses behave properly. Pre-conditions often give the best chances of catching errors for the least amount of processing overhead.

Design by Contract is a valuable technique that should be used whenever you program. It is particular helpful in building clear interfaces.

6.3.1 LoggerUncheckedException

From this technique I developed a two classes called `LoggerUncheckedException` and `Contract`. The class definition for `LoggerUncheckedException` is that this class allows you to set the message in the exceptions that were being thrown. Overriding the `getMessage` method does this. There are three constructors in the`LoggerUncheckedException` class but all take a `String exceptionType`, which identifies the exception type thrown. The other constructors allow exceptions to be created from the exception resource bundle.

`Contract` defines two exception types `PreconditionFailureException` and `PostconditionFailureException` which are subclasses of `LoggerUncheckedException`. The exceptions define their own exception types ("PreconditionFailureException" and "PostconditionFailureException" respectively) and their Detail is set by whatever you put in the call to `Contract.Precondition` or `Contract.Postcondition`.

Another class developed was the `Message` class. This class is used to generate error messages, be they exceptions, warnings or expected conditions. The class reads the text from a file, the correct message is found using a key. Parameters are then substituted in this message to build the context sensitive message.

The sequence diagram in **fig. 6-7** shows the interactions if an exception occurred in the LoggerUI application. `Object1` (for example) calls the `displayAspect` method, which starts to get its pre-conditions through the static `Contract.Precondition` method. This method takes a `String` message and a `boolean` condition. In this instance the condition is `false` therefore, a `PreconditionException` is thrown with the message as its parameter. This then calls the constructor of its superclass (`LoggerUncheckedException`) which creates a new `Message` object and substitutes the necessary parameters into a message using the `setParam` method.

Once again depending on the constructor called the message could be generated from the exception resource bundle but for this sequence diagram the `exceptionType` and `exceptionDetails` are hard coded.

The `Contract.Precondition and Contract.Postcondition are` mostly used in the framework classes when the framework wants to make an aspect object visible to a user. The other place that the `LoggerUncheckedException` is used is in the initialisation of the serial port with the exceptions being thrown/displayed in the result area of the LoggerUI application.

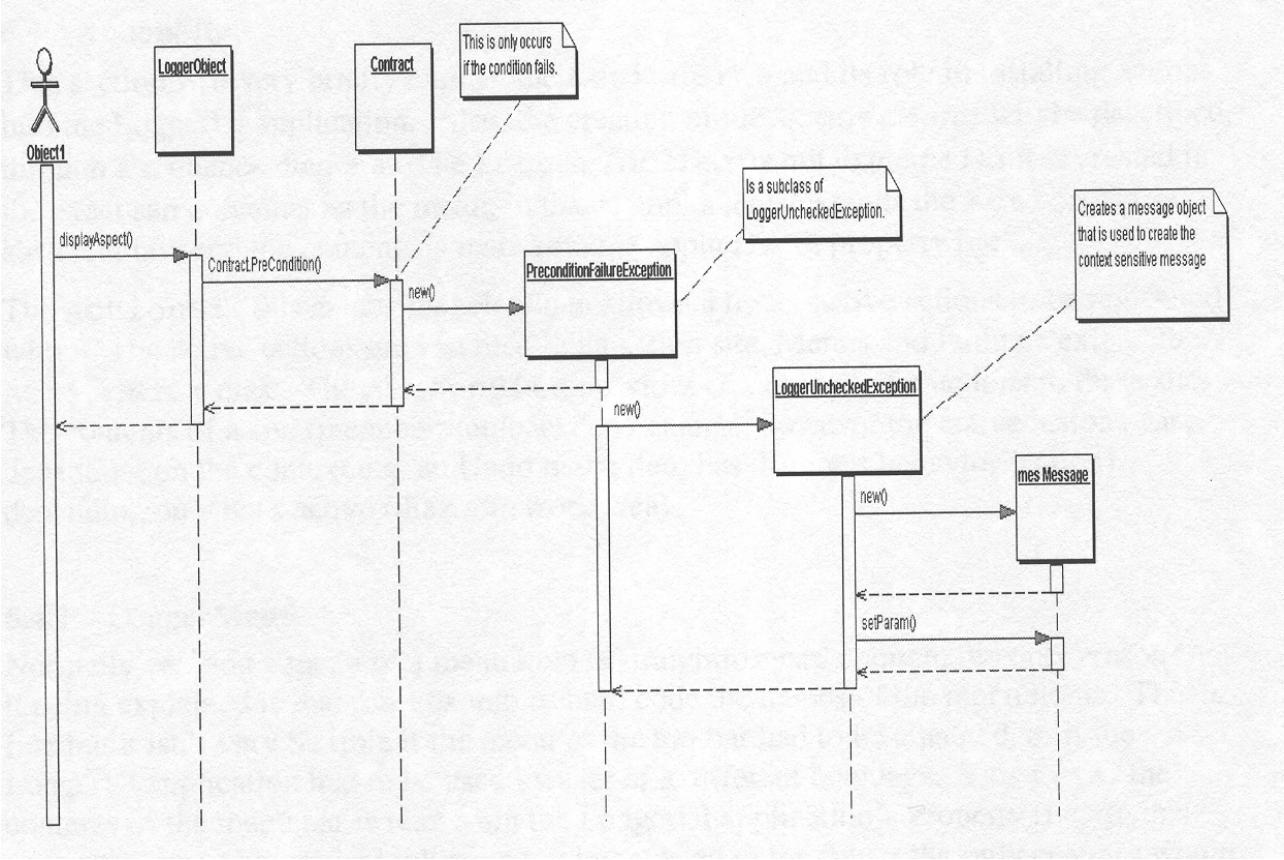


Fig. 6-7. Shows the sequence diagram of a pre-condition exception been thrown within the LoggerUI application. As stated in the diagram this will occur if the Boolean expression is equal to false.

6.4 ActionSite

This section will very briefly explain the `ActionSite` and its role in installing actions into the LoggerUI application. Also, the creation of the `LoggerMenu` will be described, through a sequence diagram. The `LoggerToolbar` is not explained as it is created in the exact same manner as the menu, in that implements the `ActionSite` abstract class and the contents is read from the application's property file.

The `ActionSite` is an abstract class that allows a list of active actions to be registered with it. The active actions are installed in an action site. Menus and toolbars extend the `ActionSite` class. The `ActionSite` provides context specific actions to these sites. The contents of a site (menubar, toolbar) don't change, however the active actions vary depending on the context e.g. an Undo menu item has different behavior (action) depending on what's active (diagram, work area).

6.4.1 LoggerMenu

Normally creating a menu or a menu item is straightforward enough, the only reason for it being explained is that most examples hard code the names of the menu items. This is fine but it isn't very flexible if the menu or the toolbar had to be changed, or if the LoggerUI application had to be used by user of a different language. This is why the contents of the menu bar are read from the LoggerUI application's property file. In this sense if the menu bar or the toolbar had to be updated in the future the only changes would need to be made to the property file and not the `LoggerMenu` or the `LoggerToolbar` for that matter. The same gos if it needed to be put into a different language a new property file would have to be created the `LoggerMenu` would not need to be reworked. This is a technique known as internationalization.

The Decorator pattern is implement on the `AppResource` by the `AppResourceDecorator` class. This provides a flexible alternative to subclassing for extending functionality. The `AppResourceDecorator` forwards requests to the `AppResource` after it has attached the appropriate suffixes used to look information up in the property files using a resource bundle. On creation of the `LoggerMenu` the method `setMenuItemProperties` calls the `getLablStr` method which ends up calling the `getStringforkey` method on the `AppResourceDecorator.` This attaches the appropriate suffix to the string passed in and then this calls the `getStringForKey` method on `AppReource` which receives the string from the resource bundle that matches the sting being passed, see **fig. 6-8**.

The same approach is used then to retrieve the image, tool tip description , mnemonic and the accelerator key for that menu item. This then repeated for all menu items that were ordinary retrieved from the resource bundle (*LoggerUI applications property file*).

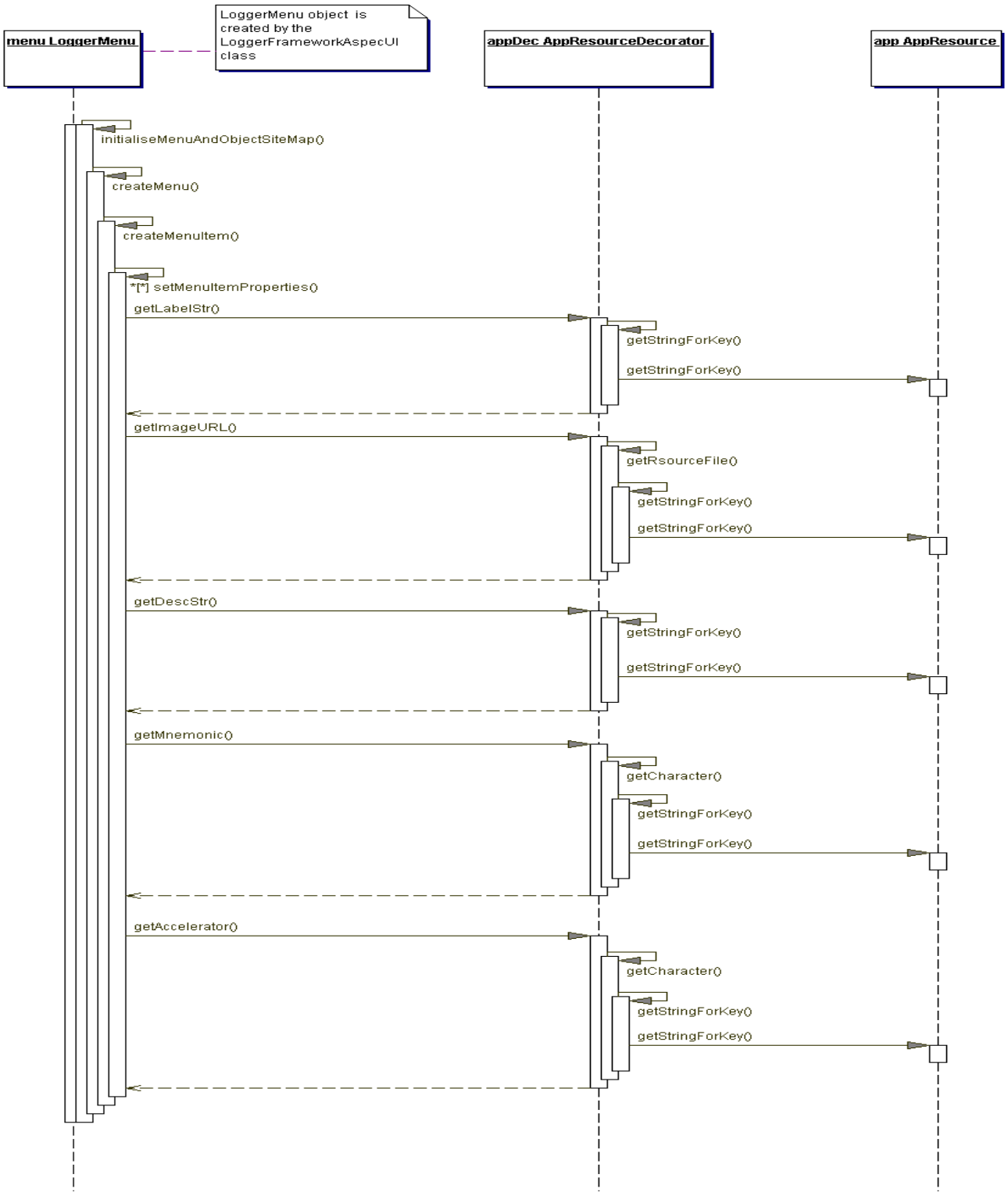


Fig. 6-8. Shows the sequence diagram for creating the menu bar within the LoggerUI application. The items that need to be added to the menu bar are obtained from the LoggerUI application property file.

6.5 Logger Check

This Logger check interface is used to give the user information such as when was the last time data was downloaded from the Datalogger and when was the last time the Datalogger was initialised including the scan rate that was selected for the period between two consecutive data loggings.

The reason I choose to put this interface into the LoggerUI application was that this project has the capability of measuring temperature over a long period of time depending on the scan rate that was selected by the user. Therefore if the Datalogger were placed in a remote place that is not easily accessibly then the user would only wish to retrieve it when he/she is sure that the memory is nearly full as it wouldn't be convenient for them otherwise. So for this reason I decided to add this to the LoggerUI application so that when the user loads up a previous project it will tell the user when he/she last downloaded data or when he/she last initialised the system, and from this they could calculate when the on board memory of the Datalogger is near its full capacity and therefore retrieve it from its location.

The interface provides no other functionality it is a user feature that has been incorporated into the LoggerUI application.

However in designing this form absolute positioning was used as this is faster than using any of the swing layout managers that are in a sense pretty limited in laying out graphically component. Absolute positioning is were you give the component the exact x and y coordinate which corresponds to the top left hand corner of that component and you also specify the width and height of the component.

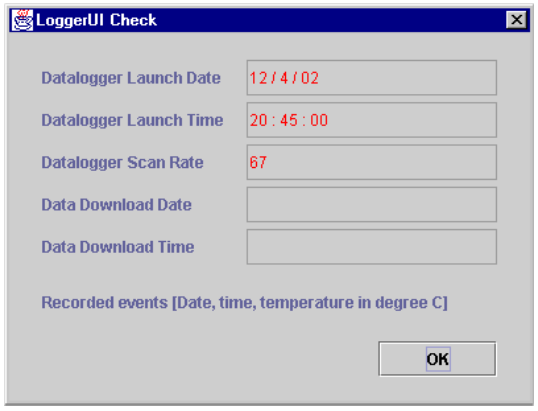


Fig. 6-9. Screen shot of the Logger check when loaded by the LoggerUI application to the screen.

Table 6-1. The following line of code is how the components are placed in their respective places using absolute positioning. The first two parameters represent the x and y coordinates of the component and the last two parameters represent the width and height of the component.

```
_launchTime.setBounds(new
Rectangle(25, 52, 150, 27));
```

6.6 Logger Setup

This Logger Setup interface is used to tell the user what the specifications of the Datalogger are in that it tells the user what microcontroller is used, what type of input is used on the Datalogger i.e. analogue or digital, how many inputs are there on the Datalogger, what its memory capacity is and how many records can be stored in the memory, this would be depending on the input used. In the case of the my Datalogger which uses an analogue input with a 12-bit analogue-to-digital converter which means that every record recorded would take up two bytes within the memory (excluding the time stamp), if the analogue-to-digital was changed to a 8-bit then each record recorded would take up one byte within memory and therefore double the number of records would be able to be stored in memory (excluding the time stamp) but they wouldn't be as actuate as the 12-bit analogue-to-digital converter.

It informs the user what type of features are on the Datalogger to tell them if the Datalogger is initialised properly or if a record has been recorded, such as an LED flashing indicates the control byte received is invalid or a LCD indicating what the current mode of the Datalogger is. The interface also indicates what type of transmission speed the overall system is set up for such as 9600 bits per second.

The only recommendation I would make for this interface is that if this project ever happen to be mass produced and the LoggerUI application happen to remain basically the same but there were different types of Dataloggers produced i.e. different types of microcontrollers used, multiple inputs, etc then every time the LoggerUI application was loaded up and the Datalogger was connected to the COM port for downloading data, the Datalogger would have its details permanent stored in its on board memory so when it starts downloading data it would first send its details which would update the form letting the user know what type of Datalogger they were using and they would then be able to determine if the Datalogger was compatible with the software they were using so they could initialise the Datalogger.

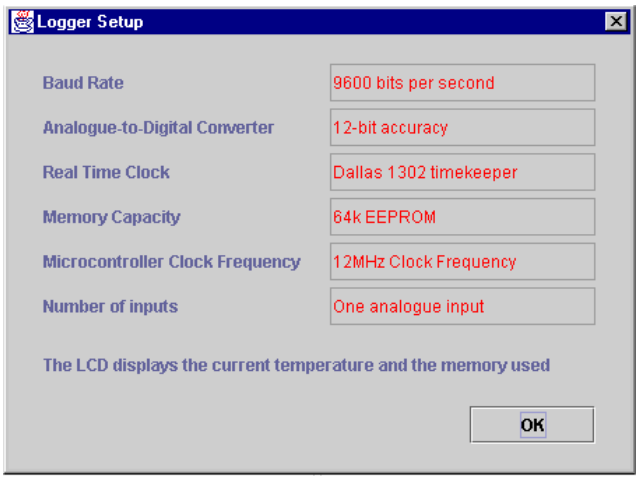


Fig. 6-10. Screen shot of the Logger Setup interface when loaded to the screen by the LoggerUI application.

6.7 Serial Port Introduction

One of the major functionalities of the LoggerUI application is its ability to communicate with the serial port(s) on the host computer. With out this ability the LoggerUI application is of no real use as it can neither initialize nor download data from the Datalogger. The JDK doesn't provide functionality to support this, but Java does have an extension package called Java communication API.

The Java communications API, is a standard extension to the Java platform. Like all Java standard extensions, the Java communications API is intended to be implementable from specification by third parties.

6.7.1 `javax.comm` extension package

There are three levels of classes in the Java communications API:

- High-level classes like `CommPortIdentifier` and `CommPort` manage access and ownership of communication ports.
- Low-level classes like `SerialPort` and `ParallelPort` provide an interface to physical communications ports. The current release of the Java communications API enables access to serial (RS-232) and parallel (IEEE 1284) ports.
  - Driver-level classes provide an interface between the low-level classes and the underlying operating system. Driver-level classes are part of the implementation but not the Java communications API. Application programmers should not use them.

The `javax.comm` extension package provides the following basic services:

- Enumerate the available ports on the system. The static method `CommPortIdentifier.getPortIdentifiers` returns an enumeration object that contains a `CommPortIdentifier` object for each available port. This `CommPortIdentifier` object is the central mechanism for controlling access to a communications port.
- Open and claim ownership of communications ports by using the high level methods in their `CommPortIdentifier` objects.
- Resolve port ownership contention between multiple Java applications. Events are propagated to notify interested applications of ownership contention and allow the port's owner to relinquish ownership. `PortInUseException` is thrown when an application fails to open the port.
- Perform asynchronous and synchronous I/O on communications ports. Low-level classes like `SerialPort` and `ParallelPort` have methods for managing I/O on communications ports.
- Receive events describing communication port state changes. For example, when a serial port has a state change for Carrier Detect, Ring Indicator, DTR, etc. the

`SerialPort` object propagates a `SerialPortEvent` that describes the state change.

```
java.lang.Object
    |
    +----javax.comm.CommPortIdentifier
```

public class `CommPortIdentifier`

extends `Object`

Communications port management. `CommPortIdentifier` is the central class for controlling access to communications ports. It includes methods for:

- Determining the communications ports made available by the driver.
- Opening communications ports for I/O operations.
- Determining port ownership.
- Resolving port ownership contention.
- Managing events that indicate changes in port ownership status.

The LoggerUI application first uses methods in `CommPortIdentifier` to negotiate with the driver to discover which communication ports are available and then select a port for opening. It then uses methods in the class `SerialPort` to communicate through the port selected.

```
java.lang.Object
    |
    +----javax.comm.CommPort
            |
            +----javax.comm.SerialPort
```

public abstract class `SerialPort`

extends `CommPort`

An RS-232 serial communications port. `SerialPort` describes the low-level interface to a serial communications port made available by the underlying system. `SerialPort` defines the minimum required functionality for serial communications ports.

6.7.2 InitialisePort

When the user opens a new project the `workarea` panel constructs a drop down menu (`JComboBox`) that contains all the available Serial ports available on the underlying host computer. If user now wants to either initialize or download data from the Datalogger all they have to do is connect the Datalogger to the desired serial port on the host computer and select that port from the drop down menu and hit either the initialization or download button, depending on their required need.

The state diagram in **fig. 6-11** describes the behavior of the `InitialisePort` object as it sets up the port for either initialization or data downloading. Once the `openPort()` action is performed the Port Open state is entered into. This checks to see that no other port is opened by the LoggerUI application, i.e. if the user was initializing the Datalogger and accidentally hit the download button. On the LoggerUI application an error will be thrown to the result area because the `isOpen` flag is set to `true` by the initialization action. If this check passes then the `openConnection()` action is performed and the `InitialisePort` object enters into Open Port state and the `isOpen` flag is set to `true`. Here, the LoggerUI application checks to see if the port selected to open exists on the host computer. If it doesn't the `NoSuchPort` exception is throw to the result area. However, this will never occur in the LoggerUI application as whatever ports that are available on the host computer are the only ones added to the drop down menu. Once it has checked the port selected the `InitialisePort` object still stays in the Open Port state but it now try's to open and claim ownership of the serial port that is selected. This is done by notifying interested applications of ownership contention and allows the port's owner to relinquish ownership. If the owner doesn't relinquish ownership after 30 seconds then the `PortInUseException` is thrown when the LoggerUI application fails to open the port.

If the port is opened successfully then the `setConnectionParameters()` action is performed and the `InitialisePort` object enters the PortSetup state. In this state the LoggerUI application sets up the selected port to the desired parameters i.e. 9600 baud, no parity, 8 data bits and one stop bit. These are standard enough settings for most serial ports so there won't be a problem but if there is an `UnSupportedParameter` will be thrown indicating to the user which parameter that is selected is unsupported in the host computer. Again this will not happen in the LoggerUI application, as the user hasn't the option to select the ports parameters, they are hard coded into the `SerialParameters` class. Once the parameters are set up sucessfully the `InitialisePort` object will either have `openInputStream()` or `openOutputstream()` action performed on it, depending on what action the user has performed i.e. initialization or download data.

The `openOutputstream()` opens the output stream of the serial port selected to allow the initialisation data to be sent out to the Datalogger.

The `openInputstream()` opens the input stream of the serial port selected to allow data being sent by the Datalogger to be received asynchronously by the LoggerUI application and to be stored in a text file. When both of these actions are completed they both close the serial port that was opened and set the `InitialisePort` object back to its ordinary state by setting the `isOpen` flag to `false.`

This code snippet below finds the serial ports available on the host computer and assigns them to the drop down menu.

```
/**
 * Sets the elements for the portChoice from the ports available on the
 * system. Uses an emuneration of comm ports returned by
 * CommPortIdentifier.getPortIdentifiers.
 */
  public void listPortChoices() {
    _comPorts = new JComboBox();
    _comPorts.addItemListener(this);
    CommPortIdentifier portId;

    Enumeration en = CommPortIdentifier.getPortIdentifiers();

    // iterate through the ports.
    while (en.hasMoreElements()) {
      portId = (CommPortIdentifier) en.nextElement();
      if (portId.getPortType() == CommPortIdentifier.PORT_SERIAL) {
        comPorts.addItem(portId.getName());
      }
    }
  }
```

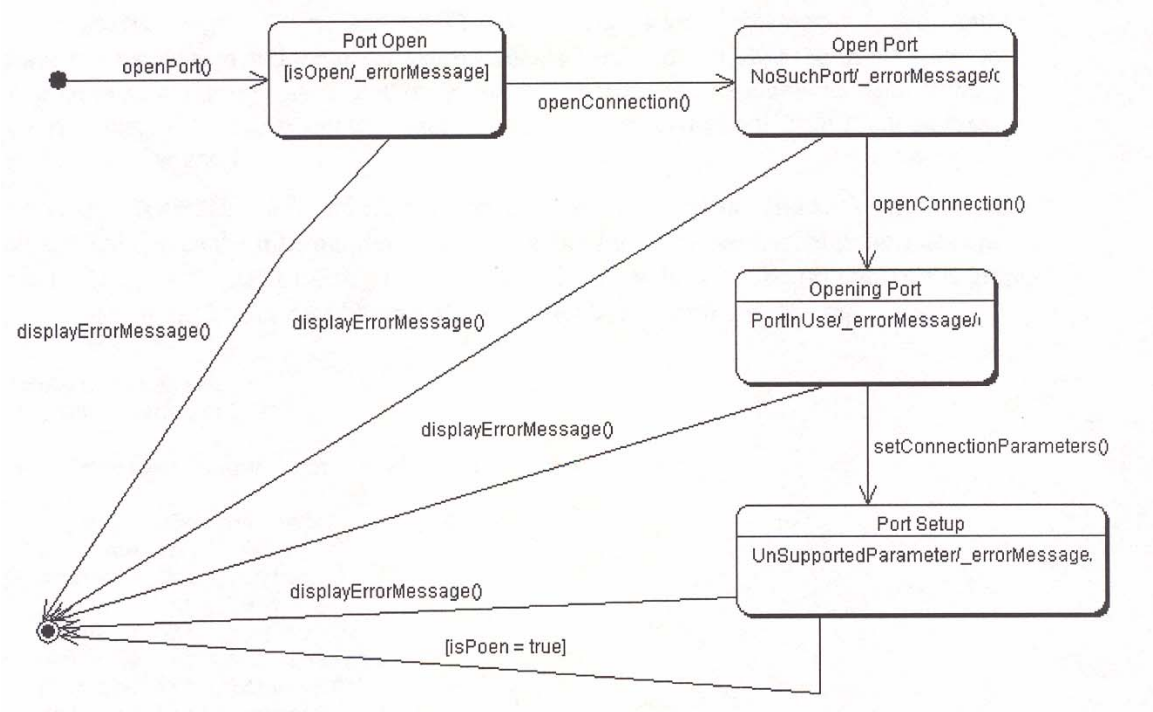


Fig. 6-11. Shows the state diagram for the InitialisePort class. The openPort() action is invoked by the user either pressing the Initialisation or Download button on the Work Area of the LoggerUI application.

6.8 Graph Introduction

Once the Datalogger has obtained it's temperature recordings over a period of time and the LoggerUI application has downloaded these recordings to the host computer's hard-drive, or were ever the user has specific for the results to be stored, they can now be analyzed. However, this information is stored in a text file and there could up to 8000 temperatures recordings. This would make it very tedious to read through these results and to find were any major changes in temperature occurred over that period of time. For this reason a graph was implemented into the LoggerUI application to make life easier for the user and to be able to see at a glance were any major changes in temperature occurred.

Unfortunately, the Java JDK doesn't come with any specific graph elements, but with a little pixel twiddling a graph component can be created. This requires manual drawing, therefore the `Canvas` class is subclassed. This is the standard component that provides direct graphics manipulation. The technique used is to override the `paint` method of `Canvas` with the custom drawing that is required. The `Graphics` object is used, which is automatically passed into the `paint` method of all components, to access colors and drawing methods.

6.8.1 LoggerLineGraphUI

To create the `LoggerLineGraphUI` class, `Canvas` was subclassed. A font could have been specified and the pixel measurements hard-coded in, but the user would be unable to resize the graph. A better approach is to measure the elements against the *current* size of the component, so that resizing the application will result in a correct resizing of the graph.

The constructor takes a `String` title, an `int` minimum value, and an `int` maximum value. This gives the information needed to layout the framework. Four variables are kept -- the `_top`, `_bottom`, `_left`, and `_right` values for the borders of the graph-drawing region. These variables are used to calculate positioning of graph items.

```
import java.awt.*;
import java.util.*;

public class LoggerLineGraphUI extends Canvas {

  // variables needed
  private int _top;
  private int _bottom;
  private int _left;
  private int _right;
  private int _titleHeight;
  private int _labelWidth;
  private int increment;
  private int position;
  private FontMetrics _fm;
  private int _padding = 4;
  private String _title;
```

```
  private int _min;
  private int _max;
  private Vector _items;
```

To calculate the correct placement of graph elements, first need to calculate the regions in our `LoggerLineGraphUI` layout that make up the framework. To improve the appearance of the component, a 4-pixel padding is added to the outer edges. The _title is added and centered at the top, taking into account the padding area. To make sure that the graph is not drawn on top of the text, the height of the text is subtracted from the _title region. The same is done for the `_min` and `_max` value range labels. The width of this text is stored in the variable `_labelWidth`. A reference to the font metrics is needed in order to do the measurements. The `items` vector is used to keep track of all the individual items that have been added to the `LoggerLineGraphUI` component. A class used to hold variables related to graph items is included (and explained) after the `LoggerLineGraphUI` class.

```
 public LoggerLineGraphUI(String title, int min, int max) {
      this._title = title;
      this._min = min;
      this._max = max;
      _items = new Vector();
    } // end constructor
```

The constructor takes the graph title and the range of values, and creates an empty vector for the individual graph items.

6.8.1.1 The Overridden reshape method

```
public void reshape(int x, int y, int width, int height) {
      super.reshape(x, y,width, height);
      _fm = getFontMetrics(getFont());
      _titleHeight = _fm.getHeight();
      _labelWidth = Math.max(_fm.stringWidth(new
                 Integer(_min).toString()),
                 _fm.stringWidth(new Integer(_max).toString())) + 2;
      _top = _padding + _titleHeight;
      _bottom = size().height - _padding;
      _left = _padding + _labelWidth;
      _right = size().width - _padding;
} // end reshape
```

Note: In JDK 1.1, the `reshape` method is replaced with `public void setBounds(Rectangle r)`. See the API documentation for details.

The `reshape` method is overridden; this is inherited down the chain from the `Component` class. The `reshape` method is called when the component is resized and when it is laid out the first time. This method is used to collect measurements, so that they will always be updated if the component is resized. The font metrics are obtained for the current font and assigned to the `_titleHeight` variable the maximum height

of that font. The maximum width of the labels is obtained, testing to see which one is bigger and then using that one. The `_top`, `_bottom`, `_left`, and `_right` variables are calculated from the other variables and represent the borders of the center graph-drawing region. Note that all of the measurements take into account a *current* size of the component so that redrawing will be correct at any size or aspect. If hard-coded values are used, the component could not be resized.

6.8.1.2 The Overridden paint method

```
public void paint(Graphics g) {
     // draw the title
     _fm = getFontMetrics(getFont());
     g.drawString(_title, (size().width - _fm.stringWidth(_title))/2,
                  _top);
     // draw the max and min values
     g.drawString(new Integer(_min).toString(), _padding, _bottom);
     g.drawString(new Integer(_max).toString(), _padding, _top +
                  _titleHeight);
     // draw the vertical and horizontal lines
     g.drawLine(_left, _top, _left, _bottom);
     g.drawLine(_left, _bottom, _right, _bottom);
} //end paint
```

The framework is drawn in the `paint` method. The _title and labels are drawn in their appropriate places. A vertical line is drawn at the left border of the graph-drawing region, and a horizontal line at the bottom border.

In this next snippet we set the preferred size for the component by overriding the `preferredSize` method. The `preferredSize` method is also inherited from the `Component` class. Components can specify a preferred size and a minimum size. I have chosen a preferred width of 925 and a preferred height of 450. The layout manager will call this method when it lays out the component.

```
public Dimension preferredSize() {
      return(new Dimension(925, 450));
   }

} // end LoggerLineGraphUI
```

Note: In JDK 1.1, the `preferredSize` method is replaced with `public Dimension getPreferredSize()`

6.8.1.3 Adding and Removing item to be graphed

```
   public void addItem(String name, int value, Color col) {
      items.addElement(new GraphItem(name, value, col));
   } // end addItem

   public void addItem(String name, int value) {
      items.addElement(new GraphItem(name, value, Color.black));
   } // end addItem
```

```
   public void removeItem(String name) {
      for (int i = 0; i < items.size(); i++) {
         if (((GraphItem)items.elementAt(i)).title.equals(name))
            items.removeElementAt(i);
      }
   } // end removeItem

} // end LoggerLineGraphUI
```

The `addItem` and `removeItem` methods after similar methods in the `Choice` class, so the code will have a familiar feel. Notice that two `addItem` methods are used; this allows items to be added with or without a color. When an item is added, a new `GraphItem` object is created and added to the items vector. When an item is removed, the first one in the vector with that name will be removed.

6.8.2 GraphItem Class

The `GraphItem` class is very simple; here is the code:

```
import java.awt.*;

class GraphItem {

   String title;
   int value;
   Color color;

   public GraphItem(String title, int value, Color color) {
      this._title = title;
      this._value = value;
      this._color = color;
   } // end constructor
} // end GraphItem
```

The `GraphItem` class acts as a holder for the variables relating to graph items. This strategy is quite convenient; don't have to go to the trouble of measuring the pixels for the framework again, and focus on filling in the graph drawing region.

6.8.2.1 Plotting the GraphItems

Since the items need to be spaced evenly a _increment variable is used to indicate the amount to shift to the right for each item. The `_position` variable is the current position, and the `_increment` variable is added to it each time.

```
  public void paint(Graphics g) {
      // The code from the above snippet is here first
      increment = (right - left)/(items.size() - 1);
      position = left;
      Color temp = g.getColor();
      GraphItem firstItem = (GraphItem)items.firstElement();
```

```
      int firstAdjustedValue = bottom - (((firstItem.value -
                                    min)*(bottom - top))/(max - min));
      g.setColor(firstItem.color);
      g.drawString(firstItem.title, position –
              fm.stringWidth(firstItem.title), firstAdjustedValue - 2);
      g.fillOval(position - 2, firstAdjustedValue - 2, 4, 4);
      g.setColor(temp);
      for (int i = 0; i < items.size() - 1; i++) {

         GraphItem thisItem = (GraphItem)items.elementAt(i);
         int thisAdjustedValue = bottom - (((thisItem.value - min)*
                                 (bottom - top))/(max - min));
         GraphItem nextItem = (GraphItem)items.elementAt(i+1);
         int nextAdjustedValue = bottom - (((nextItem.value - min)*
                                 (bottom - top))/(max - min));
         g.drawLine(position, thisAdjustedValue,
                    position+=increment, nextAdjustedValue);
         g.setColor(nextItem.color);
         if (nextAdjustedValue < thisAdjustedValue)
            g.drawString(nextItem.title, position –
                  fm.stringWidth(nextItem.title),nextAdjustedValue
                  + titleHeight + 4);
         else
            g.drawString(nextItem.title, position -
            fm.stringWidth(nextItem.title),nextAdjustedValue - 4);
            g.fillOval(position - 2, nextAdjustedValue - 2, 4, 4);
            g.setColor(temp);
      }
   } // end paint
```

First the framework is drawn, then the custom graph drawing is implement. The value of the `increment` is found by measuring the difference between the left and right edges of the graph region and then by dividing the result by the number of elements minus 1. This formula will produce the correct increment value. Because there may be colors associated with the graph items, the original color is kept in a `temp` variable, then set the color to be the first item's color. A small circle is drawn and the name of the first item in the correct position. The color is set back to its original color.

In the `for` loop, the correct pixel values are found of the current and next element in the vector, adjusted for the actual size of the component. A reference is needed for the current and next item so that connecting lines can be drawn.

The next value is checked to see if it is less than the current value to decide where to draw the label. If the line will go up, the label is drawn under the point, and if the line will go down, the label is drawn above the point. This technique ensures that the lines won't cross the labels.

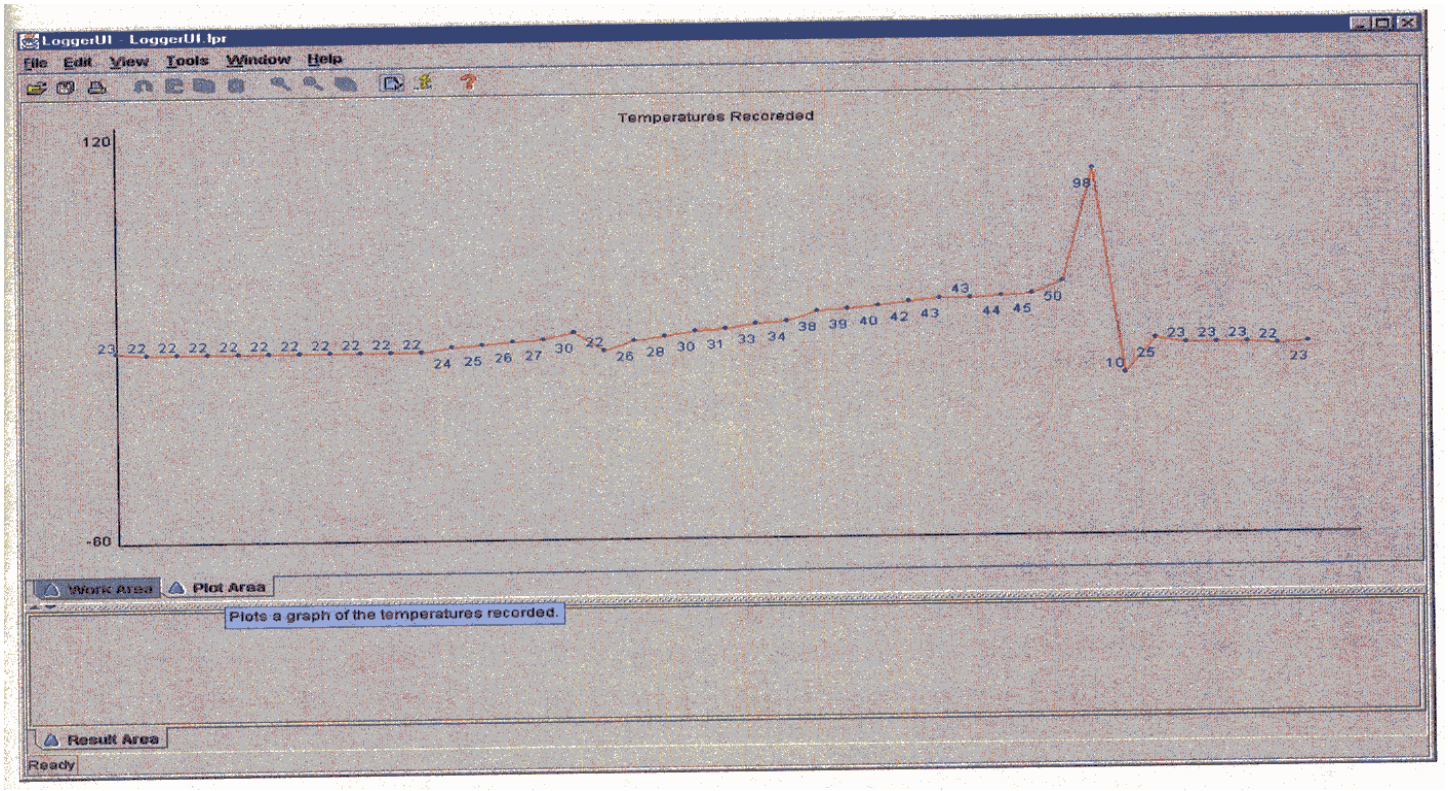


Fig. 6-12. Shows the screen shot of the line graph plotted in the Plot Area tab after the temperatures have been downloadedfrom the Datalogger.

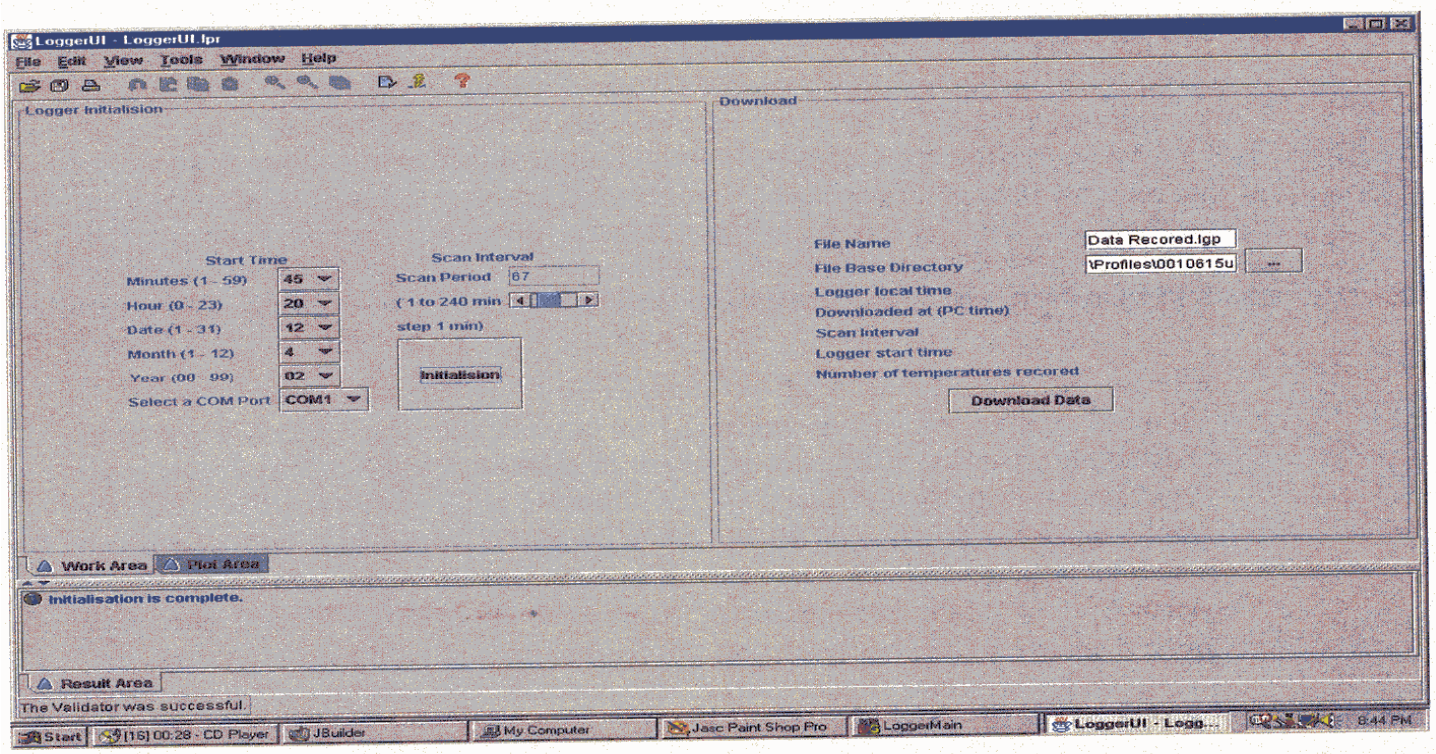


Fig. 6-13. Shows a screen shot of the LoggerUI application after the user has initialized the Datalogger. Note the message being displayed in the Result Area telling the user that the Datalogger is initialized. Also the logger Initialization and Download panels, which are displayed in the Work Area tab.

7. Problems Encountered

Like most projects or even things that take time to organize over a period of time they never or very rarely go smoothly. This project is no exception to this, but it can be put down to mistakes done and lessons learn. From this, some of the procedures that have been taken in the duration of this project if it was to be done again from scratch there would be things that would be certainly changed.

In this section of the report, some of the problems that have been encountered during the project will be documented. These problems have either hindered the project in the sense of time or even in the functionality. It will be broken into two parts; one dealing with the ADuC812 development and the interaction with the other hardware components that made up the Datalogger. The other section will describe the problems that were encountered during the development of the LoggerUI application.

7.1 ADuC812 Problems Encountered

One of the problems that resisted through out the project and at the start when the spec was being drawn up it seemed straight forward enough. This was the analogue-to-digital conversion. Getting the ADuC812 to perform an analogue-to-digital conversion wasn't the problem. The problem was that once the conversion a done two-byte value respented the current temperature. Since it was inverted didn't make life any easier, but trying to convert it into a decimal value and store it in memory was the hassle. A lookup table could have been used but this very efficient as was found out. Also trying to calculate what zero Celsius corresponded to on the chip was another problem but was eventually over come. This was a more time consuming problem but it was one that was overcome.

The next problem ended up affecting the functionality of the project. This problem was to do with the I^2C bus. This problem was the biggest disappointment of the project if there is one. The problem was that you could see the ADuC812 sent out the necessary control word to the memory chip and receive an acknowledge back from the memory chip. It then sent out the high and low byte address of were to write to in the memory chip and got another acknowledge back and finally sent out the necessary data to the memory chip. But, the memory chip would never acknowledge the data being sent and from this it would never be written into the memory of the chip. I spent a lot of time trying to figure out what was causing the problem but to no avail.

Other problems encountered in the development of the ADuC812 software were the interrupts and CALL and JMP statements. Using any of the Timers for the Scan Interval delay wasn't possible since an analogue-to-digital conversion was interrupt based and so was the overflow of each of the Timers. The CALL and JMP statements ended up causing a problem in the main method of the ADuC812 code. It was more of a programming error on my part were I was using a CALL statement instead of a JMP statement. The difference being that a CALL statement needs a RET (return) statement to jump back to were it has been called from. A JMP doesn't, you jump to the required location and the program executes sequentially from there. This error caused the ADuC812 to jump back to main (start of the program) when the memory was full

instead of jumping to were it was supposed to wait for the control word to start downloading the data logged.

7.3 Java Problems Encountered

The first problem encountered in the development of the LoggerUI application was the resource bundle. The problem was that JBuilder (the developing package that was used to write the `.java` files) wasn't placing the `Logger.property` file in the resource folder in the source output. Therefore every time the program ran it looked in the source output path for the resource folder, which contained the `.property` file, which didn't exist, and therefore one of the Pre-conditions was being thrown, indicating that there was an unexpected exception. Once this problem was fixed the menu and toolbar could be created and hence the LoggerUI application created.

From here most of the development involved using Java `swing` components and in particular the `GridBagLayout` manager which is the most flexible layout managers in `swing` but also one of the most difficult to work with as most of the time it has a mind of its own and places components in locations that you haven't specified. This manager was used in the layout of the Logger Initialization and Download panels. Using this manager is more of a case of trial and error.

When it came to the stage to start implementing the operations of the serial port a problem was encounter that resulted in the LoggerUI application no being platform independent anymore i.e. working on `Windows` and `UNIX` operating systems. The problem is due to the fact that `Window` additions above 95 don't allow easy access to the serial ports of the underlying system. There is a work around this problem, which means instanitaineing an object called a `Win32Driver`, which is found in `com.sun.comm.*`. Once this object is created an `initialize` method has to be called on it. This method doesn't take any parameters. Once this is done the ports on the underlying system become available to the user.

Serial Port alos introducted another problem in the download of data from the Datalogger. The problem is that the LoggerUI application reads in data asynchronously from the serial port and this is event triggered. Therefore, this operates on a separate thread of execution that the one that calls the download data action. This resulted in the LoggerUI application only reading in one byte of data and then closing the port. Therefore to overcome this problem a delay of a minute was introduced into the code before actual close port action was executed. This fixed the problem but it isn't a very efficient way of doing it but since time wasn't allowing me to research more appropriate methods of overcoming this problem this was the only solution.

.

8. Conclusion

In the early part of this report, the background technologies outlined how this project would be developed. The central part of this report has been concerned with how these technologies were implemented in achieving the aims and the objectives that were set out in the early part of this report also.

The overall goal was to develop a device that was dedicated to measuring temperatures over a period of time and that wasn't large in size. The system developed in section 4 achieved this goal. It was compact in size consumed little power and its only function was to acquire temperature.

I am quite happy with the outcome of this project. On the microcontroller end of the project i.e. `ADuC812` most of if not all, the aims have been met for it. The only real let down on this wasn't been able to talk to the external memory chip over the I^2C bus. But if anything could come out of this was that I seen that the code that had been implemented was sending out the right signals over I^2C bus as they could be seen on the oscilloscope. `ADuC812` was however, capable of reading temperatures storing them in memory, taking a time stamp an also storing it in memory (the `ADuC812` internal memory) and was able to communicate with the host computer through the serial port for both initialization and data downloading.

On the Graphical User Interface all of the aims were achieved an even a few additional features but not all were added into the interface. It is capable of initializing and downloading data from the Datalogger through the serial port. It can represent the data retrieved from the Datalogger in a line graph and also save the data to a file on the host computer of were ever the user specifies. Additional features that were able to be implemented were saving each new initialization and downloading of data as new projects.

If time had permitted the zoom and cut and paste functionality could have been implemented. As with many projects it is sometimes difficult to know when to stop improving and modifying the design but here is a few improvements that could make the overall system better as a whole.

- The communication between the host computer and the Datalogger could use wireless communication. This would involve using Radio Packet Controllers (RPC) that are able to both transmit and receive data over wireless communication link. One of the controllers would be connected to the serial port of the host computer and the other would be on the Datalogger board. This means that the LoggerUI application would still communicate to the serial port of the host computer but the need for the cable between it and the Datalogger wouldn't be necessary.
- An observer pattern could have been implemented in the design for checking the constraints of a date select by the user. The way in which it works at the moment is that the date isn't checked until the user try's to initialize the Datalogger. The `runValidator` method is called which checks the constraints of the date selected. If an observer pattern were implemented then if the date selected was inaccurate an error message would inform the user in the usual way (message

being displayed in the Result Area) but if the user changed the date then the message would instantly disappear. The way in which it is done at the moment is the constraint is only checked when the user hits the initialization button on the Logger initialization panel in the Work Area.

This project was quite large in size than originally anticipated, with the final LoggerUI application having over 50 classes. Overall I am happy with what has been achieved. This project has been challenging, interesting and most of all a valuable experience in the domain of both Java and microcontroller programming.

Appendix A : ADuC812 Control Register Settings

Bit	Name	Description
ADCCON1.7 ADCCON1.6	MD1 MD0	The mode bits (MD1, MD0) select the active operating mode of the ADC as follows: MD1 MD0 Active Mode 0 0 ADC powered down 0 1 ADC normal mode 1 0 ADC powered down if not executing a conversion cycle 1 1 ADC standby if not executing a conversion cycle Note: In power-down mode the ADC V_{REF} circuits are maintained on, whereas all ADC peripherals are powered down, thus minimizing current consumption.
ADCCON1.5 ADCCON1.4	CK1 CK0	The ADC clock divide bits (CK1, CK0) select the divide ratio for the master clock used to generate the ADC clock. A typical ADC conversion will require 17 ADC clocks. The divider ratio is selected as follows: CK1 CK0 MCLK Divider 0 0 1 0 1 2 1 0 4 1 1 8
ADCCON1.3 ADCCON1.2	AQ1 AQ0	The ADC acquisition select bits (AQ1, AQ0) select the time provided for the input track-and-hold amplifier to acquire the input signal, and are selected as follows: AQ1 AQ0 #ADC Clks 0 0 1 0 1 2 1 0 4 1 1 8
ADCCON1.1	T2C	The Timer 2 conversion bit (T2C) is set by the user to enable the Timer 2 overflow bit be used as the ADC convert start trigger input. ADC conversions are initiated on the second Timer 2 overflow.
ADCCON1.0	EXC	The external trigger enable bit (EXC) is set by the user to allow the external CONVST pin to be used as the active low convert start input. This input should be an active low pulse (minimum pulsewidth >100 ns) at the required sample rate.

Table A-1. ADCCON1 SFR Bit Designations

Location	Name	Description
ADCCON2.7	ADCI	The ADC interrupt bit (ADCI) is set by hardware at the end of a single ADC conversion cycle or at the end of a DMA block conversion. ADCI is cleared by hardware when the PC vectors to the ADC Interrupt Service Routine.
ADCCON2.6	DMA	The DMA mode enable bit (DMA) is set by the user to enable a preconfigured ADC DMA mode operation. A more detailed description of this mode is given in the ADC DMA Mode section.
ADCCON2.5	CCONV	The continuous conversion bit (CCONV) is set by the user to initiate the ADC into a continuous mode of conversion. In this mode, the ADC starts converting based on the timing and channel configuration already set up in the ADCCON SFRs; the ADC automatically starts another conversion once a previous conversion has completed.
ADCCON2.4	SCONV	The single conversion bit (SCONV) is set to initiate a single conversion cycle. The SCONV bit is automatically reset to "0" on completion of the single conversion cycle.
ADCCON2.3 ADCCON2.2 ADCCON2.1 ADCCON2.0	CS3 CS2 CS1 CS0	The channel selection bits (CS3–0) allow the user to program the ADC channel selection under software control. When a conversion is initiated, the channel converted will be the one pointed to by these channel selection bits. In DMA mode, the channel selection is derived from the channel ID written to the external memory. CS3 CS2 CS1 CS0 CH# 0 0 0 0 0 0 0 0 1 1 0 0 1 0 2 0 0 1 1 3 0 1 0 0 4 0 1 0 1 5 0 1 1 0 6 0 1 1 1 7 1 0 0 0 Temp Sensor 1 1 1 1 DMA STOP All other combinations reserved.

Table A-2. ADCCON2 SFR Bit Designations

ADC Control Register #1	Description	Status
ADCCON1.7	ADC power control bits	1
ADCCON1.6	(SHTDN, NORM, AUTOSHTDN, AUTOSTBY)	0
ADCCON1.5	Conversion time =16/ADCCLK	1
ADCCON1.4	ADCCLK = MCLK/[1, 2, 4, 8]	0
ADCCON1.3	Acquisition time select bits	0
ADCCON1.2	ACQ time = [1, 2, 3, 4] / ADCCLK	0
ADCCON1.1	Timer2 convert enable	0
ADCCON1.0	External CONVST enable	0

ADC Control Register #2	Description	Status
ADC1	ADC interrupt flag	0
DMA	DMA mode enable	0
CCONV	Continuous conversion mode	0
SCONV	Single conversion start bit	1
CS3	Input channel select bits	1
CS2	0000 – 0001 = ADC0-ADC7	0
CS1	1XXX = temperature sensor	0
CS0	1111 = “halt” command (in DMA mode only)	0

The tables above show the configurations that the ADCCON1 and ADCCON2 SFRs are set to in the Datalogger.

A-1 ADuC812 TMOD, TCON & SCON Register Settings

Bit	Name	Description
7	Gate	Timer 1 Gating Control. Set by software to enable Timer/Counter 1 only while $\overline{INT1}$ pin is high and TR1 control bit is set. Cleared by software to enable Timer 1 whenever TR1 control bit is set.
6	C/T	Timer 1 Timer or Counter Select Bit. Set by software to select counter operation (input from T1 pin). Cleared by software to select timer operation (input from internal system clock).
5	M1	Timer 1 Mode Select Bit 1 (used with M0 Bit).
4	M0	Timer 1 Mode Select Bit 0. M1 M0 0 0 TH1 operates as an 8-bit timer/counter. TL1 serves as 5-bit prescaler. 0 1 16-Bit Timer/Counter. TH1 and TL1 are cascaded; there is no prescaler. 1 0 8-Bit Autoreload Timer/Counter. TH1 holds a value that is to be reloaded into TL1 each time it overflows. 1 1 Timer/Counter 1 Stopped.
3	Gate	Timer 0 Gating Control. Set by software to enable Timer/Counter 0 only while $\overline{INT0}$ pin is high and TR0 control bit is set. Cleared by software to enable Timer 0 whenever TR0 control bit is set.
2	C/T	Timer 0 Timer or Counter Select Bit. Set by software to select counter operation (input from T0 pin). Cleared by software to select timer operation (input from internal system clock).
1	M1	Timer 0 Mode Select Bit 1.
0	M0	Timer 0 Mode Select Bit 0. M1 M0 0 0 TH0 operates as an 8-bit timer/counter. TL0 serves as 5-bit prescaler. 0 1 16-Bit Timer/Counter. TH0 and TL0 are cascaded; there is no prescaler. 1 0 8-Bit Autoreload Timer/Counter. TH0 holds a value that is to be reloaded into TL0 each time it overflows. 1 1 TL0 is an 8-bit timer/counter controlled by the standard timer 0 control bits. TH0 is an 8-bit timer only, controlled by Timer 1 control bits.

Table A-3. TMOD SFR Bit Designations

Bit	Name	Description
7	TF1	Timer 1 Overflow Flag. Set by hardware on a Timer/Counter 1 overflow. Cleared by hardware when the Program Counter (PC) vectors to the interrupt service routine.
6	TR1	Timer 1 Run Control Bit. Set by user to turn on Timer/Counter 1. Cleared by user to turn off Timer/Counter 1.
5	TF0	Timer 0 Overflow Flag. Set by hardware on a Timer/Counter 0 overflow. Cleared by hardware when the PC vectors to the interrupt service routine.
4	TR0	Timer 0 Run Control Bit. Set by user to turn on Timer/Counter 0. Cleared by user to turn off Timer/Counter 0.
3	IE1	External Interrupt 1 ($\overline{INT1}$) Flag. Set by hardware by a falling edge or zero level being applied to external interrupt pin $\overline{INT1}$, depending on bit IT1 state. Cleared by hardware when the when the PC vectors to the interrupt service routine only if the interrupt was transition-activated. If level-activated, the external requesting source controls the request flag, rather than the on-chip hardware.
2	IT1	External Interrupt 1 (IE1) Trigger Type. Set by software to specify edge-sensitive detection (i.e., 1-to-0 transition). Cleared by software to specify level-sensitive detection (i.e., zero level).
1	IE0	External Interrupt 0 ($\overline{INT0}$) Flag. Set by hardware by a falling edge or zero level being applied to external interrupt pin $\overline{INT0}$, depending on bit IT0 state. Cleared by hardware when the PC vectors to the interrupt service routine only if the interrupt was transition activated. If level activated, the external requesting source controls the request flag, rather than the on-chip hardware.
0	IT0	External Interrupt 0 (IE0) Trigger Type. Set by software to specify edge-sensitive detection (i.e., 1-to-0 transition). Cleared by software to specify level-sensitive detection (i.e., zero level).

Table A-4. TCON SFR Bit Designations

Bit	Name	Description
7	SM0	UART Serial Mode Select Bits.
6	SM1	These bits select the Serial Port operating mode as follows: SM0 SM1 Selected Operating Mode 0 0 Mode 0: Shift Register, fixed baud rate (Core_Clk/2) 0 1 Mode 1: 8-bit UART, variable baud rate 1 0 Mode 2: 9-bit UART, fixed baud rate (Core_Clk/64) or (Core_Clk/32) 1 1 Mode 3: 9-bit UART, variable baud rate
5	SM2	Multiprocessor Communication Enable Bit. Enables multiprocessor communication in Modes 2 and 3. In Mode 0, SM2 should be cleared. In Mode 1, if SM2 is set, RI will not be activated if a valid stop bit was not received. If SM2 is cleared, RI will be set as soon as the byte of data has been received. In Modes 2 or 3, if SM2 is set, RI will not be activated if the received ninth data bit in RB8 is 0. If SM2 is cleared, RI will be set as soon as the byte of data has been received.
4	REN	Serial Port Receive Enable Bit. Set by user software to enable serial port reception. Cleared by user software to disable serial port reception.
3	TB8	Serial Port Transmit (Bit 9). The data loaded into TB8 will be the ninth data bit that will be transmitted in Modes 2 and 3.
2	RB8	Serial Port Receiver Bit 9. The ninth data bit received in Modes 2 and 3 is latched into RB8. For Mode 1, the stop bit is latched into RB8.
1	TI	Serial Port Transmit Interrupt Flag. Set by hardware at the end of the eighth bit in Mode 0, or at the beginning of the stop bit in Modes 1, 2, and 3. TI must be cleared by user software.
0	RI	Serial Port Receive Interrupt Flag. Set by hardware at the end of the eighth bit in Mode 0, or halfway through the stop bit in Modes 1, 2, and 3. RI must be cleared by software.

Table A-5. SCON SFR Bit Designations

Parameter		12 MHz Min	12 MHz Typ	12 MHz Max	Variable Clock Min	Variable Clock Typ	Variable Clock Max	Unit
UART TIMING (Shift Register Mode)								
t_{XLXL}	Serial Port Clock Cycle Time		1.0			$12t_{CK}$		µs
t_{QVXH}	Output Data Setup to Clock	700			$10t_{CK} - 133$			ns
t_{DVXH}	Input Data Setup to Clock	300			$2t_{CK} + 133$			ns
t_{XHDX}	Input Data Hold after Clock	0			0			ns
t_{XHQX}	Output Data Hold after Clock	50			$2t_{CK} - 117$			ns

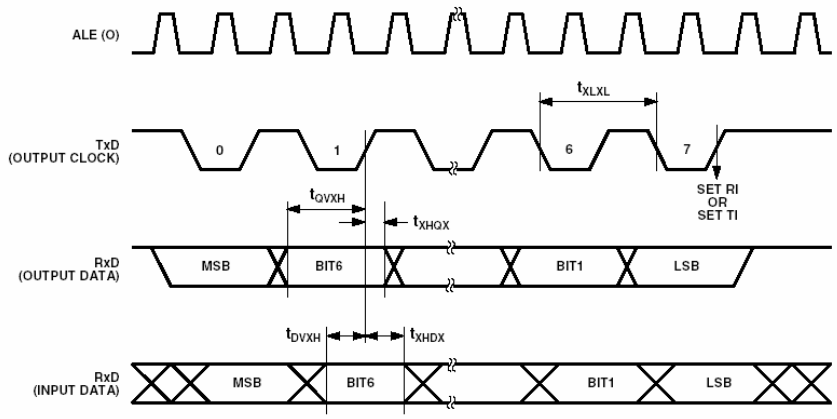


Fig. A-1. Shows the UART Timing in Shift Register Mode

Timer/ Counter 0 & 1 Mode Register	Description	Status
Gate	Timer 1 Gating Control	0
C/T	Timer 1 Timer or Counter Select Bit	0
M1	Timer 1 Mode Select Bit 1 (Used with Mo Bit)	1
M0	Timer 1 Mode Select Bit 0	0
Gate	Timer 0 Gating Control	0
C/T	Timer 0 Timer or Counter Select Bit	0
M1	Timer 0 Mode Select Bit 1 (Used with Mo Bit)	0
M0	Timer 0 Mode Select Bit 0	0

Timer/ Counter 0 & 1 Control Register	Description	Status
TF1	Timer 1 Overflow Flag	0
TR1	Timer 1 Run Control Flag	1
TF0	Timer 0 Overflow Flag	0
TR0	Timer 0 Run Control Flag	0
IE1	External Interrupt 1 Flag	0
IT1	External Interrupt 1 Trigger Type	0
IE0	External Interrupt 0 Flag	0
IT0	External Interrupt 0 Trigger Type	1

UART Serial Port Control Register	Description	Status
SM0	UART Serial Mode Select Bits	0
SM1	Serial Port Operating Mode	1
SM2	Multiprocessor Communication Enable Bit	0
REN	Serial Port Receive Enable Bit	1
TB8	Serial Port Transmit Bit 9	0
RB8	Serial Port Receive Bit 9	0
TI	Serial Port Transmit Interrupt Flag	0
R1	Serial Port Receive Interrupt Flag	0

The tables above show the configurations that the TMON, TCON and SCON SFRs are set to in the Datalogger.

A-2 I²C Implementation on the ADuC812

This section describes the I²C implementation on the ADuC812. The ADuC812provides both hardware and software master operating modes. Three SFRs are used to control the I²C interface.

I2CADD: Holds the 7-bit address of the MicroConverter device (default value = 55H). This SFR is only used in slave mode.

I2CDAT: In slave-receiver mode the received data from the SDATA line is latched into this SFR. Hence, after a successful reception the received data can be read from this SFR. E.g.

```
MOV A,  I2CDAT
```

reads the received data into the accumlator. In slave transmitter mode a write to this SFR will make the data available for transmission on the SDATA line under control of the master. E.g.

```
MOV I2CDAT,  #60h
```

writes 60h out to the SDATA line when clocked by the master.

Note: For the ADuC812, a write or a read of the I2CDAT SFR automatically clears the I2CI interrup flag. Clearing this flag for a second time will cause the I²C controller to get 'lost'. For the ADuC812 the I2CI interrupt flag must be cleared in software.

I2CCON: Holds configuration/control bits for master/slave mode operation.

Bit Mneumonic	Description
MDO	**Software Master Data Out Bit (MASTER ONLY)** This bit is used to implement a master I^2C interface transmitter in software. Data written to this bit will be outputted on the SDATA pin if the data output enable (MDE) is set.
MDE	**Software Master Data Out Enable Bit (MASTER ONLY)** This bit is used to implement a master I^2C interface in software. Setting this bit enable the SDATA pin as an output (TX). Clearing this bit enables SDATA as an input (RX)
MCO	**Software Master Clock Out Bit (MASTER ONLY)** This bit is used to implement a master I^2C receiver interface in software. Data written to MCO will be outputted on the SCLOCK pin.
MDI	**Software Master Data In Bit** This bit is used to implement a master I^2C receiver interface in software. The data on the SDATA pin is latched in here on SCLOCK if data output enable (MDE) is clear.
I2CM	**I^2C Mode Bit** Setting this bit enables software master mode, clearing this bit enables hardware slave mode.
I2CRS	**I^2C Serial Port Reset (SLAVE ONLY)** Setting this bit will cause a reset of the I^2C interface
I2CTX	**I^2C Transmission Direction Status (SLAVE ONLY)** This bit indicates the direction of transfer. The bit is set if the master is reading from the slave. This bit is cleared if the master is writing data to the slave. This bit is automatically loaded with the R/W bit after the slave address and start condition.
I2CI	**I^2C Interrupt Flag (SLAVE ONLY)** This is the interrupt flag for the I^2C serial port. This bit is set after a byte has been transmitted or received. It must be cleared in software.

Table A-6: Bit Definition of I2CCON

Parameter		Min	Max	Unit
I^2C COMPATIBLE INTERFACE TIMING				
t_{LOW}	SCLOCK Low Pulsewidth	1.3		µs
t_{HIGH}	SCLOCK High Pulsewidth	0.6		µs
$t_{HD;\ STA}$	Start Condition Hold Time	0.6		µs
$t_{SU;\ DAT}$	Data Setup Time	100		µs
$t_{HD;\ DAT}$	Data Hold time	0	0.9	µs
$t_{SU;\ STA}$	Setup time for Repeated Start	0.6		µs
$t_{SU;\ STO}$	Stop Condition Setup Time	0.6		µs
t_{BUF}	Bus Free Time between a STOP Condition and a START Condition	1.3		µs
t_R	Rise Time for Both SCLOCK and SDATA		300	ns
t_F	Fall Time for Both SCLOCK and SDATA		300	ns
t_{SUP}[1]	Pulsewidth of Spike Suppressed		50	ns

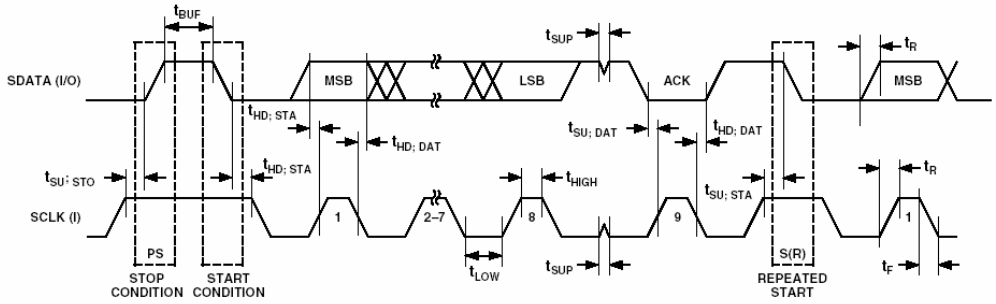


Fig. A-2. Shows the I^2C-Compatible Interface Timing

I^2C Control Register	Description	Status
MDO	Software Master Data Out Bit (MASTER ONLY)	1
MDE	Software Master Data Out Enable Bit (MASTER ONLY)	0
MCO	Software Master Clock Out Bit (MASTER ONLY)	1
MDI	Software Master Data In Bit (MASTER ONLY)	0
I2CM	I^2C Mode Bit	1
I2CRS	I^2C Serial Port Reset (SLAVE ONLY)	0
I2CTX	I^2C Transmission Direction Status (SLAVE ONLY)	0
I2CI	I^2C Interrupt Flag (SLAVE ONLY)	0

The table above show the configurations that the I2CCON SFRs are set to in the Datalogger.

Appendix B : ADuC812 Code

```
;*****************************************************************************
;
; Author:               Paul Boyle  4th Yr NUI Galway, Electronic & Computer Engineering
;
; Date:                 05 Apr 2002
;
; File:                 demo2.asm
;
; Hardware:             ADuC812
;                       HD66100 (Hitachi LCD)
;                       MICROCHIP 24LC64 64K EEPROM
;                       DALLAS DS1302 REAL TIME CLOCK
;
; Description   :       Reads a temperature from the internal temperature sensor of the
;                       ADuC812 and stores the temperature in internal memory along with
;                       a time stamp taken from RTC.  The program will continue this
;                       until either memory is full or the user presses the INT0 button
;                       on the development board were the program will wait to send the
;                       data logged up the UART to the LoggerUI application.
;                       Each temperature recorded is displayed on the LCD in 4-bit mode,
;                       therefore all commands and data have to be sent in two nibbles
;                       (4bit parts) to the display.
;                       The value stored in SCANRATE is the number minutes between each
;                       temperature value that is sampled.
;
;*****************************************************************************
;                                               EQUATES
;*****************************************************************************

 $MOD812

 DB4            EQU     P2.4
 DB5            EQU     P2.5
 DB6            EQU     P2.6
 DB7            EQU     P2.7
 EN             EQU     P3.7
 RW             EQU     P3.6
 RS             EQU     P3.5
 LCD_DATA       EQU     P2

 RST            EQU     P0.7
 IO             EQU     P0.5
 SCLK           EQU     P0.3

 LED            EQU     P3.4        ; P3.4 is red LED on eval board

 CENABLE        DATA    055h        ; Clock Start
 YEARVAL        EQU     018h
 MONTHVAL       EQU     019h
 DATEVAL        EQU     01Ah
 HOURVAL        EQU     01Bh
 MINUTEVAL      EQU     01Ch

 IOYEAR EQU     EQU     01Dh
 IOMONTH        EQU     01Eh
 IODATE EQU     EQU     01Fh
 IOHOUR EQU     EQU     020h
 IOMINUTE       EQU     021h
 SCANRATE       EQU     022h         ; Scan Rate variable
 AMPM           EQU     IOHOUR.5 ; 0 = AM, 1 = PM
 TMODE          EQU     IOHOUR.7 ; 0 = 24, 1 = 12

 WENABLE        DATA    00h
 WSECOND        DATA    080h
 RSECOND        DATA    081h
 WMINUTE        DATA    082h
 RMINUTE        DATA    083h
```

```
WHOUR           DATA    084h
RHOUR           DATA    085h
WDAY            DATA    086h
RDAY            DATA    087h
WMONTH          DATA    088h
RMONTH          DATA    089h
WYEAR           DATA    08Ch
RYEAR           DATA    08Dh
WCONTROL        DATA    08Eh
RCONTROL        DATA    08Fh
WCB             DATA    0BEh    ; Write Clock Burst
RCB             DATA    0BFh    ; Read Clock Burst

DELAYSEC        EQU     07Dh    ; Second variable
TICKS           EQU     07Eh    ; 20th of a second countdown timer
MINUTES         EQU     07Fh    ; Minute variable

ORG 0000h
JMP MAIN

ORG 03h
PUSH    PSW
MOV     R1, #036h
MOV     DELAYSEC, #59
MOV     A, SCANRATE
SUBB    A, #01h
MOV     MINUTES, A
POP     PSW
RETI

ORG 0033h                       ; ADC ISR
RETI


;*********************************************************************************
;                                READ TWO NIBBLES FROM THE LCD
;*********************************************************************************

READ_2_NIBBLES:

ORL   LCD_DATA,#0F0h            ; Be sure to release datalines (set outputlatches
                                ; to '1') so we can read the LCD
SETB  EN
MOV   A,LCD_DATA                ; Read first part of the return value (high nibble)
CLR   EN
ANL   A,#0F0h                   ; Only high nibble is usable
PUSH ACC
SETB  EN
MOV   A,LCD_DATA                ; Read second part of the return value (low nibble)
CLR   EN
ANL   A,#0F0h                   ; Only high nibble is usable
SWAP  A                         ; Last received is actually low nibble
MOV   R7,A
POP   ACC
ORL   A,R7                      ; And combine it with low nibble
RET


;*********************************************************************************
;                                WRITE TWO NIBBLES TO THE LCD
;*********************************************************************************

WRITE_2_NIBBLES:

PUSH  ACC                       ; Save A for low nibble
;ORL   LCD_DATA,#0F0h           ; Bits 4..7 <- 1
ANL   A,#0F0h                   ; Don't affect bits 0-3
MOV   LCD_DATA,A                ; High nibble to display
SETB  EN
CLR   EN
```

```
 POP   ACC                      ; Prepare to send
 SWAP  A                        ; ...second nibble
 ;ORL   LCD_DATA,#0F0h          ; Bits 4...7 <- 1
 ANL   A,#0F0h                  ; Don't affect bits 0...3
 MOV   LCD_DATA,A               ; Low nibble to display
 SETB  EN
 CLR   EN
 RET


;*********************************************************************************
;                              CHECKING THE BUSY STATUS OF THE LCD
;*********************************************************************************

 WAIT_LCD:

 CLR RS                         ; It's a command
 SETB RW                        ; It's a read command
 LCALL READ_2_NIBBLES           ; Take two nibbles from LCD in A
 JB ACC.7,WAIT_LCD              ; If bit 7 high, LCD still busy
 CLR RW                         ; Turn off RW for future commands
 RET


;*********************************************************************************
;                                  INITIALIZING THE LCD
;*********************************************************************************

 INIT_LCD:

 CLR   RS
 CLR   RW
 CLR   EN
 SETB  EN
 MOV   LCD_DATA,#28h
 CLR   EN
 LCALL WAIT_LCD
 MOV   A,#28h
 LCALL WRITE_2_NIBBLES          ; Write A as two separate nibbles to LCD
 LCALL WAIT_LCD
 MOV   A,#0Eh
 LCALL WRITE_2_NIBBLES
 LCALL WAIT_LCD
 MOV   A,#06h
 LCALL WRITE_2_NIBBLES
 LCALL WAIT_LCD
 RET


;*********************************************************************************
;                                  CLEARING THE DISPLAY
;*********************************************************************************

 CLEAR_LCD:

 CLR   RS
 MOV   A,#01h
 LCALL WRITE_2_NIBBLES
 LCALL WAIT_LCD
 RET


;*********************************************************************************
;                                  WRITING TEXT TO THE LCD
;*********************************************************************************

 WRITE_TEXT:

 SETB  RS
 LCALL WRITE_2_NIBBLES
```

```
 LCALL WAIT_LCD
 RET


;*******************************************************************************
;                                   TEXT TO BE WRITTEN TO THE LCD
;*******************************************************************************

 INIT:
 DB 'INITIALISING...'
 DB 000h

 DOWN:
 DB 'DOWNLOADING...'
 DB 000h

 WAIT:
 DB 'WAITING...'
 DB 000h

 ERROR:
 DB 'ERROR!!!'
 DB 000h

 TITLE:  DB 'Demo Program by Paul Boyle'
 DB 000h


;*******************************************************************************
;                                   DELAY
;*******************************************************************************

 DELAY:                          ; delay 100ms

 MOV     R7,#200                 ; 200 * 500us = 100ms

 DLY1:
 MOV   R6,#229                   ; 229 * 2.17us = 500us
 DJNZ  R6,$                      ; sit here for 500us
 DJNZ    R7,DLY1                 ; repeat 200 times (100ms delay)
 RET


;*******************************************************************************
;                                   CONVERT TEMPERATURE
;*******************************************************************************

 CONVERT_TEMP:

 MOV   ADCCON1,#060h             ; power up ADC
 MOV   ADCCON2,#8h               ; select channel to convert
 ;SETB    EA                     ; enable interrupts
 SETB  EADC                      ; enable ADC interrupt

 LCALL DELAY
 SETB    SCONV                   ; innitiate single ADC conversion
                                 ; ADC ISR is called upon completion
 LCALL DELAY
 MOV   A, ADCDATAh
 ANL     A, #00Fh
 SWAP    A
 MOV     R2,A
 MOV     R3, ADCDATAL
 MOV   A, ADCDATAL
 ANL     A, #0F0h
 SWAP  A
 ORL     A, R2
 CLR   C
 SUBB  A, #58H                   ; convert to 2's comp
                                 ; 54h=0, FFh=+171, 00h=-54
```

```
SENDDECs:                       ; SENDs the signed decimal number in Acc to the LCD
                                ; -54->171
 PUSH     B
 PUSH  ACC
 JBC   ACC.7, HUNDREDS
 MOV   A, #'-'                  ; transmit minus sign
 MOV   @R1, A                   ; move R0 into memory location R1
 INC   R1                       ; increment memory location and data so
 CALL  Write_text

 HUNDREDS:                      ; check #hundreds
 POP       ACC                  ; restore original value of A
 PUSH  ACC                      ; remember original value of A
 CPL   A
 INC   A
 MOV   B, #100                  ; divide remainder by 100
 DIV   AB                       ; A receives integer quotient
                                ; B receives the remainder
 SETB  F0
 JZ    TENS                     ; if ACC=0 then num=0xx
 CLR   F0
 ADD   A, #'0'
 MOV   @R1, A                   ; move R0 into memory location R1
 INC   R1                       ; increment memory location and data so
 LCALL Write_text

 TENS:                          ; check tens
 MOV       A,B
 MOV   B,#10
 DIV   AB                       ; divide remainder by 10
 JNB   F0, SEND0                ; if F0 is cleared the a number
                                ; exists in the 100s
 JZ    UNITS

 SEND0:
 ADD       A, #'0'              ; only send a zero if number
 MOV   @R1, A                   ; move R0 into memory location R1
 INC   R1                       ; increment memory location and data so
 CALL  Write_text               ; existed in the 100s

 UNITS:
 MOV       A,B                  ; send remainder (even if 0)
 ADD   A, #'0'
 MOV   @R1, A                   ; move R0 into memory location R1
 INC   R1                       ; increment memory location and data so
 CALL  Write_text
 POP   ACC
 POP   B

 MOV       A,#0F9h
 MOV   @R1, A                   ; move R0 into memory location R1
 INC   R1                       ; increment memory location and data so
 MOV       A,0A5h
 CALL      WRITE_TEXT
 MOV       A,R3
 ANL       A,#00Fh
 SUBB      A,#00Fh
 JNB       ACC.7, UNIT_TEMP
 CPL   A
 INC   A
 MOV   B, #10                   ; divide remainder by 10
 DIV   AB                       ; A receives integer quotient
 MOV   A,B                      ; B receives the remainder
 UNIT_TEMP:
 ADD   A,#'0'
 MOV   @R1, A                   ; move R0 into memory location R1
 INC   R1                       ; increment memory location and data so
 CALL      WRITE_TEXT
 MOV   A,#0F8h
 MOV   @R1, A                   ; move R0 into memory location R1
```

```
 INC   R1                          ; increment memory location and data so
 MOV   A,#0DFh
 CALL  WRITE_TEXT
 MOV      A, #'C'
 MOV   @R1, A                      ; move R0 into memory location R1
 INC   R1                          ; increment memory location and data so
 CALL     WRITE_TEXT
 CALL  DELAY
 RET


;*******************************************************************************
;                                SENDDATA TO I2C BUS
;*******************************************************************************

 SENDDATA:
                                   ; send start bit
 CALL     STARTBIT                 ; acquire bus and send slave address


 MOV      A, SLAVECON              ; send slave address
 CALL  SENDBYTE                    ; sets NOACK if NACK received
 JB    NOACK, STOPSEND             ; if no acknowledge send stop
 MOV      A, SLAVEADDH
 CALL     SENDBYTE
 JB    NOACK, STOPSEND
 MOV      A, SLAVEADDL
 CALL     SENDBYTE
 JB    NOACK, STOPSEND

 MOV   A, OUTPUT                   ; send OUTPUT byte
 CALL  SENDBYTE

 STOPSEND:
 CALL     STOPBIT                  ; sends stop bit
 INC      SLAVEADDL
 JNB   NOACK, SENDRET              ; if slave sends no-acknowedge send error
 SETB  ERR                         ; sets the error flag
 SETB     I2CRS

 SENDRET:
 CALL DELAY5
 RET


;*******************************************************************************
;                                RCVDATA TO I2C BUS
;*******************************************************************************

 RCVDATA:

 ; send start bit
 CALL     STARTBIT                 ; acquire bus and send slave address
 MOV   A, SLAVECON
 CALL  SENDBYTE                    ; sets NOACK if NACK received
 JB    NOACK, POLL                 ; if no acknowledge send stop
 MOV      A, SLAVEADDH
 CALL     SENDBYTE
 JB    NOACK, STOPSEND
 MOV      A, SLAVEADDL
 CALL     SENDBYTE
 JB    NOACK, STOPSEND
 INC   SLAVECON                    ; put slave back in transmit mode
 CALL  STARTBIT                    ; acquire bus and send slave address

 MOV      A, SLAVECON
 CALL  SENDBYTE                    ; sets NOACK if NACK received
 DEC      SLAVECON
 JB    NOACK, STOPRCV              ; Check for slave not responding.
 CALL     DELAY5                   ; this lets slave get data ready
```

```
 CALL  RCVBYTE                  ; Receive next data byte.
 MOV   INPUT,A                  ; Save data byte in buffer.

 STOPRCV:
 CALL  STOPBIT
 INC       SLAVEADDL
 JNB   NOACK, RCVRET            ; if slave sends NACK send error
 SETB  ERR                      ; sets the error flag
 SETB  I2CRS                    ; this resets the I2C interface

 RCVRET:
 RET


;*********************************************************************************
;                                SEND STARTBIT
;*********************************************************************************

 STARTBIT:

 SETB      MDE                  ; enable SDATA pin as an output
 CLR   NOACK
 CLR   MDO                      ; low O/P on SDATA
 CALL  DELAY5                   ; delay 5 Machine cycles
 CLR   MCO                      ; start bit
 RET


;*********************************************************************************
;                                SEND STOPBIT
;*********************************************************************************

 STOPBIT:

 SETB      MDE                  ; to enable SDATA pin as an output
 CLR   MDO                      ; get SDATA ready for stop
 SETB  MCO                      ; set clock for stop
 CALL  DELAY5                   ; delay 5 Machine cycles
 SETB  MDO                      ; this is the stop bit
 RET


;*********************************************************************************
;                                  SENDBYTE
;*********************************************************************************

 SENDBYTE:

 MOV       BITCNT,#8            ; 8 bits in a byte
 SETB  MDE                      ; to enable SDATA pin as an output
 CLR   MCO                      ; make sure that the clock line is low

 SENDBIT:
 RLC       A                    ; put data bit to be sent into carry
 MOV   MDO,C                    ; put data bit on SDATA line
 SETB  MCO                      ; clock to send bit
 CLR   MCO                      ; clear clock
 DJNZ  BITCNT,SENDBIT           ; jump back and send all eight bits
 CLR   MDE                      ; release data line for acknowledge
 SETB  MCO                      ; send clock for acknowledge
 CALL  DELAY5                   ; delay 5 Machine cycles
 JNB   MDI,NEXT                 ; this is a check for acknowledge
 SETB  NOACK                    ; no acknowledge, set flag

 NEXT:
 CLR       MCO                  ; clear clock
 RET
```

```
;*******************************************************************************
;                                    RCVBYTE
;*******************************************************************************

 RCVBYTE:

 MOV      BITCNT,#8              ; Set bit count.
 CLR   MDE                       ; to enable SDATA pin as an input
 CLR   MCO                       ; make sure the clock line is low

 RCVBIT:
 SETB      MCO                   ; clock to recieve bit
 CLR   MCO                       ; clear clock
 MOV   C,MDI                     ; read data bit into carry.
 RLC   A                         ; Rotate bit into result byte.
 DJNZ  BITCNT,RCVBIT             ; Repeat until all bits received.
                                 ; recieved byte is in the accumulator
 SETB  MDE                       ; Data pin of the master must be an..
                                 ; ..output for the acknowledge
 SETB  MDO                       ; Send no acknowledge, last byte.
 SETB  MCO                       ; Send no-acknowledge clock.
 CALL  DELAY5                    ; delay 5 Machine cycles
 CLR   MCO                       ; clear clock
 RET


;*******************************************************************************
;                                    DELAY5
;*******************************************************************************
; Short delay (5 machine cycles incl CALL time) for the main signals (SCLOCK , SDATA)

 DELAY5:
 NOP
 RET


;*******************************************************************************
;                                READING FROM THE RTC
;*******************************************************************************

 READ_RTC:

 CLR   RST
 CLR   SCLK
 SETB  RST
 LCALL DELAY_RTC
 MOV   B,#8
 CLR   C

 BYTERD1:
 RRC   A
 MOV   IO,C
 LCALL SCLKW
 DJNZ  B,BYTERD1
 MOV   B,#8

 BYTERD2:
 LCALL SCLKR
 MOV   C,IO
 RRC   A
 DJNZ  B,BYTERD2
 CLR       RST
 LCALL DELAY_RTC
 RET
```

```
;*******************************************************************************
;                                   WRITING TIME TO RTC
;*******************************************************************************

 WRITE_RTC:

 BYTEWR:
 CLR   RST
 CLR   SCLK
 SETB  RST
 LCALL DELAY_RTC
 MOV   B,#8
 CLR   C

 BYTEWR1:
 RRC   A
 MOV   IO,C
 LCALL  SCLKW
 DJNZ  B,BYTEWR1
 MOV   B,#8
 CLR   C
 RET

 BYTEWR2:
 RRC   A
 MOV   IO,C
 LCALL SCLKW
 DJNZ  B,BYTEWR2
 CLR   RST
 LCALL DELAY_RTC
 RET

 SCLKW:

 CLR   SCLK
 LCALL DELAY_RTC
 SETB  SCLK
 LCALL DELAY_RTC
 ;CLR      SCLK
 RET

 SCLKR:

 SETB  SCLK
 LCALL DELAY_RTC
 CLR   SCLK
 LCALL DELAY_RTC
 RET

 DELAY_RTC:
 nop
 RET


;*******************************************************************************
;                                   DISPLAY ROUTINE
;*******************************************************************************

 LCD_MSG:

 CLR       A
 MOVC      A,@A+DPTR
 INC       DPTR
 JZ        LCD_MSG9
 CALL      WRITE_TEXT
 JMP       LCD_MSG

 LCD_MSG9:
 RET
```

```
;*******************************************************************************
;                        SENDS ASCII VALUE CONTAINED INT Acc TO UART
;*******************************************************************************

 SENDCHAR:

 JNB    TI,$                        ; wait til present char gone
 CLR    TI                          ; must clear TI
 MOV    SBUF,A
 RET


;*******************************************************************************
;                                    SENDVAL
;*******************************************************************************
                                ;
 SENDVAL:                           ; converts the hex value of A into two ASCII chars,
                                    ; and then spits these two characters up the UART.
 PUSH   ACC
 SWAP   A                           ; does not change the value of A.
 CALL   HEX2ASCII
 CALL   write_text                  ; send high nibble
 POP    ACC
 PUSH   ACC
 CALL   HEX2ASCII
 CALL   write_text                  ; send low nibble
 POP    ACC
 RET


;*******************************************************************************
;                                    HEX2ASCII
;*******************************************************************************

 HEX2ASCII:                         ; converts A into the hex character representing the
                                    ; value of A's least significant nibble
 ANL    A,#00Fh
 CJNE   A,#00Ah,$+3
 JC     IO0030
 ADD    A,#007h

 IO0030:
 ADD    A,#'0'
 RET


;*******************************************************************************
;                                    DOWN_TIMES
;*******************************************************************************

 DOWNL_TIMES:

 MOV     A, @R0
 MOV  B, #100                       ; divide remainder by 100
 DIV  AB                            ; A receives integer quotient
                                    ; B receives the remainder
 SETB  F0
 JZ   TNS                           ; if ACC=0 then num=0xx
 CLR  F0
 ADD  A, #'0'
 mov SBUF, A
 CALL SENDVAL
 CALL DELAY
 CALL DELAY

 TNS:                               ; check tens
 MOV     A,B
 MOV  B,#10
 DIV  AB                            ; divide remainder by 10
 JNB  F0, SENDX0                    ; if F0 is cleared the a number
```

```
                                        ; exists in the 100s
 JZ   UNTS

 SENDX0:
 ADD      A, #'0'                       ; only send a zero if number
 mov SBUF, A
 CALL SENDVAL
 CALL DELAY
 CALL DELAY                             ; existed in the 100s

 UNTS:
 MOV      A,B                           ; send remainder (even if 0)
 ADD   A, #'0'

 MOV SBUF, A
 CALL SENDVAL
 CALL DELAY
 CALL DELAY

 INC   R0                               ; Increment address
 CJNE  R0, #23H, DOWNL_TIMES
 RET


;*******************************************************************************
;                                        DOWN_DATA
;*******************************************************************************

 DOWNL_DATA:

 MOV   A, @R0
 mov       SBUF, A
 CALL  SENDVAL
 CALL     DELAY
 CALL     DELAY

 INC   R0                               ; Increment address
 CJNE  R0, #03CH, DOWNL_DATA
 RET


;*******************************************************************************
;                                        MAIN PROGRAM
;*******************************************************************************

 ORG 0430h

 MAIN:

 LCALL INIT_LCD
 LCALL CLEAR_LCD
 mov TMOD, #20h
 mov TCON, #41h
 mov TH1, #0Fdh
 mov SCON, #50h
 MOV SLAVECON,#0A0h                     ; clear RW bit
 MOV SLAVEADDH, #00h                    ; set the high address in external memory
 MOV SLAVEADDL, #00h                    ; set the low address in external memory
 MOV I2CCON,#0A8h                       ; sets SDATA & SCLOCK, and
                                        ; selects master mode
 CLR NOACK
 CLR ERR
 SETB EA
 SETB EX0

 WAITING:
 MOV      DPTR, #WAIT
 CALL     LCD_MSG                       ; flash (complement) the red LED
 CPL   LED
 CALL     DELAY
```

```
LCALL CLEAR_LCD
jnb RI, WAITING                 ; Wait until character recevied
mov a, SBUF                     ; get character
CALL    DELAY
CALL  DELAY
clr     RI
CJNE    A,#55h,NEXT
LCALL CLEAR_LCD

DATA_LOGGED:
MOV     DPTR, #DOWN
CALL    LCD_MSG
MOV     DPTR, #DOWN
CALL    LCD_MSG
MOV   R0, #18h                  ; move value at address 40 into R2
CALL    DOWNL_TIMES
CALL    DOWNL_DATA
RET

NEXT:
CJNE    A,#0AAh,WARN
LCALL CLEAR_LCD
MOV     DPTR,#INIT
CALL    LCD_MSG
MOV     DELAYSEC,#00
MOV     TICKS,#10
MOV     MINUTES,#00
MOV     R0,#18h
MOV   R1,#24h                   ; initialise R1 to 40 to store the
                                ; input data from memory location 40
CALL    DELAY
CALL    DELAY

INITDATA:
JNB     RI, INITDATA
MOV     A, SBUF
CALL    DELAY
CALL  DELAY
clr     RI
MOV     @R0, A
INC     R0
CALL  DELAY
CALL  DELAY
CJNE    R0,#23h,INITDATA
LCALL CLEAR_LCD

SCAN_RATE:

CALL    DELAY
DJNZ    TICKS, SCAN_RATE
MOV     TICKS, #10
INC     DELAYSEC
MOV     A, DELAYSEC
CJNE    A, #60, SCAN_RATE
MOV     DELAYSEC, #00
INC     MINUTES

MOV     A, MINUTES
CJNE    A, SCANRATE, SCAN_RATE
MOV     MINUTES, #00
CALL    CONVERT_TEMP
CALL  DELAY
CALL  DELAY

CJNE  R1, #03CH, SCAN_RATE      ; reset memory location to 40h
                                ; when memory location reaches 50h
                                ; saving 16 bytes of data
CALL    DATA_LOGGED
LCALL CLEAR_LCD
JMP     WAITING
```

```
WARN:
LCALL CLEAR_LCD
MOV     DPTR,#ERROR
CALL    LCD_MSG

FLASH:
CPL    LED                      ; flash (complement) the red LED
CALL   DELAY                    ; call software delay
JMP    fLASH                    ; repeat indefinately

FINISH:
NOP
NOP
NOP

END
```

Appendix C : Java Help

C.1 HelpMap.JHM File

```
<?xml version="1.0"?>
 <map>

   <mapID target="helpintro" url="helpfiles\intro.html" />
   <mapID target="crtloverview" url="helpfiles\controls.html" />
   <mapID target="aboutbox" url="helpfiles\about.html" />
   <mapID target="closeproject" url="helpfiles\close.html" />
   <mapID target="cutobject" url="helpfiles\cut.html" />
   <mapID target="copyobject" url="helpfiles\copy.html" />
   <mapID target="exitprogram" url="helpfiles\exit.html" />
   <mapID target="openproject" url="helpfiles\open.html" />
   <mapID target="pagesetup" url="helpfiles\pagesetup.html" />
   <mapID target="pasteobject" url="helpfiles\paste.html" />
   <mapID target="printpage" url="helpfiles\print.html" />
   <mapID target="savefile" url="helpfiles\save.html" />
   <mapID target="savefileas" url="helpfiles\saveas.html" />
   <mapID target="menu" url="helpfiles\menu.html" />
   <mapID target="toolbar" url="helpfiles\toolbar.html" />
   <mapID target="systemcheck" url="helpfiles\systemcheck.html" />
   <mapID target="systemsetup" url="helpfiles\systemsetup.html" />
   <mapID target="reset" url="helpfiles\reset.html" />
   <mapID target="initialisebutton" url="helpfiles\initialisationbutton.html" />
   <mapID target="downloadbutton" url="helpfiles\downloadbutton.html" />
   <mapID target="downloadmode" url="helpfiles\download.html" />
   <mapID target="initialisationmode" url="helpfiles\initialisation.html" />
   <mapID target="dataloggingmode" url="helpfiles\datalogging.html" />
   <mapID target="resultarea" url="helpfiles\resultarea.html" />
   <mapID target="zooming" url="helpfiles\zooming.html" />

 </map>
```

C.2 LoggerUIIndex.xml File

```
<?xml version="1.0"?>
 <index>

    <indexitem text="About LoggerUI" target="aboutbox" />
    <indexitem text="Buttons">
        <indexitem text="Download Button" target="downloadbutton" />
        <indexitem text="Initialise Button" target="initialisebutton" />
        <indexitem text="Reset Button" target="reset" />
    </indexitem>

    <indexitem text="Closing a Project" target="closeproject" />
    <indexitem text="Copying" target="copyobject" />
    <indexitem text="Cutting" target="cutobject" />
    <indexitem text="Exiting a Program" target="exitprogram" />
    <indexitem text="Major Controls" target="crtloverview" />
    <indexitem text="Menu" target="menu" />

    <indexitem text="Modes of Operation">
        <indexitem text="Logger Initialisation" target="initialisationmode" />
        <indexitem text="Logger Download" target="downloadmode" />
        <indexitem text="Data Logging" target="dataloggingmode" />
    </indexitem>

    <indexitem text="Opeing a Project" target="openproject" />
    <indexitem text="Page Setup" target="pagesetup" />

    <indexitem text="Panels">
        <indexitem text="Result Area Panel" target="resultarea" />
    </indexitem>

    <indexitem text="Pasting" target="pasteobject" />
    <indexitem text="Printing" target="printpage" />
    <indexitem text="Saving a File" target="savefile" />
    <indexitem text="Saving a File As" target="savefileas" />
    <indexitem text="System Check" target="systemcheck" />
    <indexitem text="System Setup" target="systemsetup" />
    <indexitem text="Toolbar" target="toolbar" />
    <indexitem text="Zooming" target="zooming" />

 </index>
```

C.3 LoggerUItoc.xml File

```
<?xml version="1.0"?>
 <toc>

  <tocitem text="LoggerUI Application">
   <tocitem text="Introduction" target="helpintro" />
   <tocitem text="Major Controls" target="crtloverview">
     <tocitem text="Initialisation" target="initialisebutton" />
     <tocitem text="Download " target="downloadbutton" />
     <tocitem text="Reset" target="reset" />
   </tocitem>
   <tocitem text="Display Panels" target="crtloverview">
     <tocitem text="Initialisation Panel" target="initialisationmode" />
     <tocitem text="Download Panel" target="downloadmode" />
     <tocitem text="Result Area Panel" target="resultarea" />
   </tocitem>
   <tocitem text="New Project">
    <tocitem text="Open a Project" target="openproject" />
    <tocitem text="Close a Project" target="closeproject" />
    <tocitem text="Saving a File" target="savefile" />
     <tocitem text="Saving a File As" target="savefileas" />
   </tocitem>
   <tocitem text="Working with Graph">
     <tocitem text="Copying an object" target="copyobject" />
      <tocitem text="Cutting an Object" target="cutobject" />
     <tocitem text="Pasting an Object" target="pasteobject" />
     <tocitem text="Zooming" target="zooming" />
   </tocitem>
   <tocitem text="Printing">
    <tocitem text="Print a Graph" target="printpage" />
    <tocitem text="Page Setup" target="pagesetup" />
   </tocitem>
   <tocitem text="LoggerUI Menu and Toolbar">
    <tocitem text="Menu" target="menu" />
    <tocitem text="Toolbar" target="toolbar" />
   </tocitem>
   <tocitem text="Logger System Controls">
    <tocitem text="About LoggerUI" target="aboutbox" />
    <tocitem text="Logger Check" target="systemcheck" />
    <tocitem text="Logger Setup" target="systemsetup" />
   </tocitem>
   <tocitem text="Exit a Program" target="exitprogram" />
  </tocitem>

 </toc>
```

C.4 HelpSet.hs File

```
<?xml version="1.0"?>
 <helpset>
  <title>LoggerUI Application Help</title>

  <maps>
   <homeID>helpintro</homeID>
   <mapref location="HelpMap.jhm" >
  </maps>

  <view>
   <name>Table Of Contents</name>
   <label>LoggerUI TOC</label>
   <type>javax.help.TOCView</type>
   <data>LoggerUItoc.xml</data>
  </view>

  <view>
   <name>Index</name>
   <label>LoggerUI Index</label>
   <type>javax.help.IndexView</type>
   <data>LoggerUIindex.xml</data>
  </view>

  <view>
   <name>Search</name>
   <label>LoggerUI Search</label>
   <type>javax.help.SearchView</type>
   <data
engine="com.sun.java.help.search.DefaultSearchEngine">JavaHelpSearch</data>
  </view>

 </helpset>
```

Bibliography

UML Development

Developing Software with UML
Bernard Oestereich (1999), Addison-Westley Pub Co

Enterprise Java with UML
CT Arrington (2001), John Wiley & Sons

Object Oriented Designs

Design Patterns: Elements of Reusable Object-Oriented Software,
Erich Gamma, Richard Helm, Ralph Johnson & John Vlissides (1995), Addison-Westley Pub Co

Java Development

Professional Java Programming,
Brett Spell (2000), Wrox Press Inc

Beginning Java 2 – JDK 1.3 Edition,
Ivor Horton (200), Wrox Press Inc

Web Sites

http://www.java.sun.com

Sun Microsystems is the organization that developed Java. Their web site contains all the APIs, plug-ins, many tutorials and a comprehensive archive of commonly asked questions.

http://www.javaworld.com

This is IDG's weekly Web-based magazine for Java technology programmers. Java World provides news on new product information, a Java Developer Tools Guide, Java tips and tricks, cut-and-paste code, live applets and links to various Java resources on the web.

http://www.8052.com

This is an online resource, a free service provided courtesy of Vault Information Services. It contains many tutorials, free source code and links to other 8051-based web sites.