UNIVERSITY OF LA VERNE

SMART THERMOSTAT

A LEARNING EXPERIENCE IN BASIC HARDWARE/SOFTWARE SYSTEM DESIGN

A SENIOR PROJECT SUBMITTED TO

THE FACULTY OF THE DEPARTMENT OF MATHMATICS AND PHYSICS

IN CANDIDACY FOT THE DEGREE OF

BACHELOR OF COMPUTER SCIENCE AND COMPUTER ENGINEERING

ENGINEERING CONCENTRATION

BY

MICHAEL WARNER II

LA VERNE, CALIFORNIA

MAY 2000

CONTENTS

ABSTRACT

This paper is the practical application of theory learned and a great amount of additional learning. The goal was a device that is capable of monitoring temperature in different areas at a given time. Then compare that data to historic stored data to determine if and which windows should be open or closed to improve the temperature in the house built at Center for Regenerative Studies at California State Polytechnic University, Pomona, California. Much of the learning was finding what chips and compatible devices are available that will perform the function desired. They can not be to expensive, since I had to buy everything, and allow me to program them to complete the task desired. The goal changed to be a learning experience in building a basic computer system. Do to inexperience, there was no feasibility study completed before beginning design and implementation. The assumption was made that almost anything is possible with digital electronic technology and software. The biggest problem was locating the hardware that was available and that would interface with other components to perform the functions desired. The resulting implementations of the project was designed as a direct result of learning theories/concepts and was implemented more from the theories/concepts than from solving a problem. Do to these factors, this paper is written mostly in a chronological order. The chronological order will help the reader understand the results.

CHAPTER 1

INTRODUCTION

The prospect of deciding what to do for a senior project was horrendous. I knew I wanted my project to involve digital hardware, since I have a hardware emphasis of the computer science and computer engineering program at the University of La Verne. I thought of many possibilities such as having a computer control a model train system or collect and route the audio signals of a stereo sound system via USB. After many such thoughts, I decided that I did not want to spend time making something that only a techno-weenie like me would want, fabulous yet barely useful. I wanted to make something that someone actually wants and could use. Working to solve a problem that someone needs solving would also motivate me better.

I made my decision for pragmatism at a good time. My opportunity came during the winter interim. This was just before it was time sign-up for my senior project. I was in an introductory biology class in which there were a few field trips. I was looking for opportunities on each field trip. During a field trip at the Center for Regenerative Studies at California State Polytechnic University, Pomona, I heard that some things there were automated and some more things were going to be automated. Automation is a killer application for computer systems. What made this a real opportunity was that the Center for Regenerative Studies was a place of experimentation, which means experimenting with a prototype would be tolerated. I asked the tour guide to talk with a person who is involved with the automation. They gave me the center’s number to call. The secretary was able to reach the person whom I needed to talk with.

The center serves as a part of many science departments. Despite the many academic majors involved, the center is focused on finding and developing ways to have humans live in ways that are less harmful on the environment. Students gain hands-on experience working on many ecological projects such as fish farming, plankton growing, and finding ways to conserve energy (placing solar panels on poles that rotate to track the sun). Students bring the expertise of their field of study to help on projects. Civil engineering students help construct things other than the buildings which the students inhabit. Biology students, work on organically growing plants and fish farming. Students live and work at the center, usually for two years, as part of an academic program. Beside the projects, students also do maintenance duties such as taking turns to cook for the group of residing students, and cleaning the buildings (Korthof).

The Center for Regenerative Studies wants as little pollution as possible, so no air conditioners or heaters are used. To compensate for the lack of active temperature regulation, their buildings are designed with superior insulation. The building, with which I became interested, was built partially into a hillside. The earth from the hill mostly covers one side of the building. The walls of the building are made of thick concrete.

The building's windows are the only way to heat and cool the building. There are rows of small glass windows in the roof to let in sunlight, which keeps the buildings from being too cold like an underground basement. In the winter, the windows stay closed and let sunlight in to create a greenhouse effect and warm up the building in the daytime. In the summer, the windows in the roof are opened to allow the hot air to rise through the roof, which is replaced, by air that has been cooled by the coolness of the concrete walls and the hillside. Normal convection causes the warmer air to rise. The building has one large main room and many small rooms. The main room has the ceiling windows and many of the lower windows. The small rooms also have windows that can open and close.

The person at the Center for Regenerative Studies whom I finally spoke with was William Korthof. He was a civil engineering student who was also an electronic hobbyist, one of many students who live at the center. A project on which he was working was automating the opening and closing of windows on a building to regulate the interior temperature. He had already begun installing motors on the windows. He appeared to me as a person of action, which gave me more confidence in collaborating with him. This was offset by the fact the center refused to give much support, financial or otherwise. The center did not make any requests in regards with that project. It was an open-ended project with the only constraints being minimal cost to the center and not interfering with other projects. The objective was to keep the building's temperature more comfortable without consuming a substantial amount of electricity.

Mr. Korthof even installed a control feature, having each motor wired to a limiter, which sets the range of a window's movement. When a window reaches a given position, the limiter cuts the power to the motor. The limiter prevents the windows from opening too wide. The windows were each originally installed with a crank that opened the windows with a set of levers. If a window opens too wide, the levers are placed in a position that makes closing the window difficult.

He also bought a small power converter to power the motors from the standard utility power. The center had made technologies such as electrical devices a low priority. The center is largely focused on more manual methods that tend to pollute less and require less development of things like power plants, long-distance power lines, etceteras. The power converter that Mr. Korthof could afford was only powerful enough to power one or two motors at a time. To operate many motors from one power converter, he devised a plan to set a chain of relays, wired in series, that would run the motors one at a time. When a limiter turns off one motor, a relay would turn on the next motor.

The limiters make controlling the motorized windows more feasible. Whatever or whoever opens or closes the windows does not have to worry about leaving the motors run after the windows have reached the open or closed position and running the risk of wearing down the motors. The limiters combined with the relays would allow a two-position mechanism (a switch, relay, and etceteras) to control a group of windows. The mechanism could be set to the given open or close position and left there until the windows need to be placed in another position.

What Mr. Korthof had not yet planned, was how to automate the opening and closing of windows. He only placed a few switches to manually switch the windows open and closed. He noted that people would tend to forget to open and close the windows until the building becomes too hot or cold. This would be unacceptable for most people who are accustom to convenient temperature regulation, and thus would not achieve the center’s goal develop a living area that can become popular with most people while using very little resources. Opening the ceiling windows to vent the accumulated heat out during a hot summer day might not be sufficient to cool the building. The proactive step of allowing the cold early-morning air drift inside the building might be needed to achieve sufficient amount of coolness for the day. One complication of taking this step would be determining how cold the building should be allowed to become to compensate for the heat of the day. This determination is as much human preference for comfort as it is technical. A regular thermostat could not perform this type control. Regular thermostats are design to regulate temperature by using heaters and air conditioners. A common configuration for older thermostats is to have a blue slider and a red slider moved to various positions to indicate the desired temperature range. When the temperature becomes too cold, the heater operates. When the temperature becomes too hot, the air conditioner operates.

Using windows to regulate temperature does not work the same way as using heaters and air conditioners. Heaters and air conditioners add and remove heat from a mass of air, respectively. Opening windows only allows air to convect heat into or out of the building, heat convection to a cooler place. Thus, a special thermostat is needed for the building. The thermostat needs to be capable of monitor the outside and inside temperatures. The thermostat needs to take proactive measures to compensate for very hot and very cold days. Neither Mr. Korthof nor I knew of any thermostats that could perform such sophisticated functions. We concluded that a smart thermostat is needed and one should be invented and built.

Mr. Korthof had the knowledge of how to wire motors and switches, but he did not know about programming and hardware design. I asked him if I could design a smart thermostat for the building. With my computer software/hardware education, I had a decent chance to successfully create a thermostat with programmable behavior. Since the window control project was a low priority, he was able to have me create the smart thermostat without any approval from the center's administration or the university's faculty.

The next thing that Mr. Korthof and I did was determined what features we would like the smart thermostat to have. To have the thermostat perform proactive measures with reasonable accuracy, the thermostat should store inside and outside temperature readings for a few days and predict the temperatures for the next 24 hours. The thermostat's controls or menus should be as simple as possible to be more compatible the center's culture. Some performance might be sacrificed to eliminate some fancy controls or menu options, which would probably not get used by people at the center. The options we definitely wanted were setting the desired high and low temperature preferences for the day and night, respectively. More controls maybe thought about as the thermostat was being designed, built, and programmed. The building's main room was the top priority for temperature control. If possible, the small rooms would also be regulated as separate zones that have their own temperature preference settings. These features were tentative and subject to the successes and failures of the hardware/software design process.

This project became a learning experience in making a complete digital computer system. Though the thermostat is incomplete and is far simpler than many existing computer systems. It has all the components of a digital computer system, input (buttons), output (LED display), input-output processor (DSP), central processing unit (DSP), and random-access memory (DSP).

CHAPTER 2

INITIAL PROJECT DESIGN

I started with the advice, often given by people who are buying a personal computer, that one should consider what they wanted a system to do (software) before they choose the appropriate components (hardware). Thus, I focused on the algorithms for determining when to close and open windows first. I first made crude assumptions: open windows on summer nights to collect cool air for summer day, open windows on winter days to release excess heat caused by greenhouse effect. These assumptions develop a general idea of what should happen. The question of figuring and predicting the optimum times to open and close windows arose. I thought that such algorithms needed to be sophisticate and involved artificial intelligence. Books that cover the topic of artificial intelligence were difficult to read and comprehend. I could only understand some general ideas. Due to my inexperience, I had to abandon artificial intelligence and focus on the hardware of my project. The process of thinking about algorithms did influence my hardware design.

To perform the algorithms, the circuitry had to be designed that was capable of performing complex algorithms. The types of such circuitry that I learned in school were microprocessors, synchronous sequential circuits, and algorithmic state machines. Finite state machines and algorithmic state machines are difficult to use for complex algorithms. A, synchronous sequential circuit must be redesigned every time an algorithm needs to be changed, since its design is based on state tables (Mano 220-50). Every step in an algorithm needs to be translated into state and input conditions, a very arduous task. Algorithmic state machines are less difficult. Their designs are based on flow charts, which are more natural for implementing algorithms. They can be designed with an EPROM microchip at the core of the design. With a good design, a change in an algorithm would only require a change in the bits being stored in the EPROM. Still, each step in the algorithm needs to be translated into bits stored on the EPROM (Mano 307-36).

I was familiar with the 7400 logic series and programmable microchips, which were used in my digital logic classes. I was also becoming familiar with digital signal processors (DSPs); I found Texas Instruments' web and was able to order free information that came on CD-ROMs. The criterion for selecting the type of microchip was based on how complex the logic should be and what the hardware could handle. The thermostat needed to perform elaborate algorithms so it could manage the building's temperature.

7400 logic series microchips are a tempting choice. They are very cheap and are very well known. I had no problems looking for their specifications. Yet, I realized that it would take numerous 7400 microchips to store instructions and execute instructions. I built a simple arithmetic logic unit as a laboratory exercise of CMPN 280 class. It took twelve 7400 series microchips to build. A gigantic mess of wires and microchips would form and envelop the project with errors. I wanted to have as few parts as possible to minimize hardware assembly problems.

Programmable microchips are a better choice than 7400 series microchips (Mano 153-54). One programmable microchip could emulate several 7400 microchips. Despite that, the ones that ULV had still did not have enough integration. From my experience in using it in advanced architecture class, CMPN 480 I knew that it has limited capabilities. The microchip that I used in class did not have enough input/output lines and control the logic to operate as a traffic light controller. It would take at least a few of the microchips to build a processor, not including memory for program and data. I did not look for other programmable microchips that might have greater capabilities, since I found a better solution, a digital signal processor.

Processors are more complex than 7400 logic and programmable chips. They have a steeper learning curve as I have discovered. Their overwhelming advantage is the algorithms that they are running can be changed easily. In processors, algorithms are run as a sequence of machine language instructions, which exists simply as a series of ones and zeroes that causes the processor to perform operations. Fortunately, programmers can write algorithms in assembly language instead of machine language. Assembly provides programmer's with symbolic names and numbers in place of binary bits (Mazidi 50-77). Better still, many processors have a version of "C language", a high level language, available for them. I felt "C language" would be the best for writing the algorithms for the thermostat. It is easier to write programs with "C language" than with machine language, yet it is highly optimized. "C language" still must be compiled into machine language to have programs run; thus the "C language" compiler must be designed for the processor that is being used (Deitel 6.13).

Before I signed-up for my senior project, I searched Texas Instruments' web site and ordered their reference CD's, since they have a good reputation as a leader in technology. I wanted to see what types of hardware existed before I try to propose a project. I did not want to make a project proposal and then be unable to find the hardware for the brain. The microchips that seemed to be most promising were Texas Instrument's digital signal processors. These are complete processors that come with many features, such as built-in RAM. Using a real processor would eliminate the design task of creating a processor out of 7400 series and programmable microchips.

While I was thinking about artificial intelligence, I started to look at Texas Instruments' application notes that involved artificial intelligence and fuzzy logic. The one note that really caught my attention was "What is Fuzzy Logic? An Overview of the Latest Control Methodology" by

Tomothy A. Adcock. Fuzzy logic produces a range of output values, instead of just of two values of crisp logic: off and on (Adcock 2). Most thermostats operate with crisp logic. This creates the hit and misses situations of heaters over heating by a few degrees and air-conditioners over cooling by a few degrees. After extensive analyses, I realized that my project was significantly different from this application and I could not figure out how to apply it to my project. The application note discussed the fuzzy logic of a thermostat that operates a variable-speed fan to control temperature. My thermostat was to predict the daily temperature changes and control long-term temperatures instead of the immediate temperature. The note explained that the calculations needed for fuzzy logic could be easily handled by their DSP microprocessors, which have multiplier accumulators. I searched a month to find a source to purchase their DSP microprocessors, before I found a DSP starter kit. The kit comes with the processor already mounted to a board that contains the circuits needed to operate the processor and simple software needed to assemble and load code. This made my project much more manageable.

One of the first aspects of the thermostat would be the user interface. Inspired by a digital thermostat that is in my home, I felt having a display of alphanumeric characters was important. Users need to know what temperatures the thermostat is reading, what are the current settings of the thermostat, and view what they are setting as they press buttons. LED lamps that were used in my digital logic classes are good for reading output from relatively simple circuits. They are not good for easy reading of more complex information, such as time of day, current temperature, and etceteras. The user also needs a way to input preferences in the thermostat. The logic switches that were used in the digital logic classes were used to input binary values and to control digital circuits by sending +5 volts and ground signals that are accepted as logic values by the digital circuits. Keyboards send binary values to a computer system. One of the thermostats that are in my home has a keypad. Keypads are like keyboards in that they have a matrix, which sees a pressed key as a set of numbers. As I have mentioned earlier, the thermostat needs to have as simple an interface as possible, thus the number of buttons should be kept to a minimum. With a small number of buttons, a matrix is not necessary. This Thermostat could use an algorithm that is simpler than the algorithms that most computer systems use, since there is no matrix to scan.

When I received the kit, I was overwhelmed by the technical information that it came with. The one thing I quickly realized was the software simply converts assembly language code into machine language and loads it into the DSP microprocessor. Assembly language makes creating the complex algorithms needed to determine when to open and close the windows vary difficult. I found out that Texas Instruments does make a "C language" compiler for that microprocessor that costs $1600. Such a high level language is needed to make writing complex algorithms feasible. Do to financial constraints, I could not obtain the "C" compiler and start writing fancy algorithms thus forcing me to focus on hardware.

The DSP does not come with input and output devices. The DSP only has ports and buses with which to interface (Texas Instruments Tms320c54x Dsp Reference Set: Cpu and Peripherals). The DSP also lacks a real-time clock with which to know current day and time of day. Computer platforms, such as microcomputers, include a processor(s), system buses, video output, keyboard, real-time clock, and etceteras. The "C language" was designed for such complete systems. It was necessary to add the display, buttons, and real-time clock to the DSP system to make it more like common computer systems. If I had a "C language" compiler for the DSP, I would still have to write the drivers (low level code) for the Stdout (standard output) and Stdin (standard input), and Time (real time and processor clock count) libraries to for my LED, buttons, and RTC (real-time clock), respectively. (Kernighan 161,242,55-

56; Texas Instruments 7.4-7.11). Thus, using “C language” would not be as advantageous as it would be for a standard computer system.

When I stopped focusing on fancy programming, I realized that putting the thermostat hardware together and trying to get its parts to function correctly, would be a project onto itself. The only means of electronically measuring temperature that I found were thermistors. Thermistors are resisters that change resistance in relation to temperature. The voltage being measured depends on the circuit in which a thermistor is placed. Using a thermistor would involve using the A/D converter of the DSP starter kit, using the serial port of the DSP microprocessor, and calibrating the readings into actual temperature measurements (Rizzoni 715). I did not have the experience to perform those tasks without great effort. I was looking for thermometers with digital parallel outputs.

As I thought more about how the thermometers might be wired to the thermostat at the Center, I realized that having thermometers send temperature bits in parallel would cause problems. Parallel signal lines cannot run the long distances required placing thermometers. Parallel lines would develop different propagation delays that would cause bits to not arrive at the same time (Rizzoni 767; Mazidi 747-58). I only thought of scenarios where only the last few bits would be fluctuating (i.e. 00001100 and 00001011 might read as 00001111, a 2 bit inaccuracy). Then I realized that there are much worst scenarios where several bits can be fluctuating (i.e. 00100000 and 00011111 might read as 00111111, a 5 bit inaccuracy).

While I was looking for thermometers, I also was looking for a display. The preferred features for a display would be built-in or add-on control circuits (buffers, decoders, and etceteras), to avoid a heinous patch wire jungle that I often built in digital logic labs. Decoders are needed to select character positions. ASCII decoders are needed to convert ASCII code to LED patterns. And, Buffers/memory are needed to store the display characters while the DSP is performing operations other than outputting to the display. It is common for LCDs to come with integrated with onboard logic circuits to handle ASCII code and placing characters on various positions on the display, but their documents confused me. Much later, I realized that I misread the document that lead me to not use LCDs. Sometime later, I decided to use LEDs. My classes made me familiar with LEDs. Only a resister is needed to connect an LED to the output of a TTL microchip. LEDs rarely come with any control. I found an assembly system that mounts digit LEDs together to form a display line and mounts BCD decoders to the LEDs. That solution was expensive and only reduces wiring a little. I could not use LED bulbs for the senior project. They would not give me enough feedback information and would create an undesirable design. I was depending on the display device to see what is happening to the hardware, since the DSP kit’s monitoring program did not seem to work. I finally found a alphanumeric LED module that has the built in logic circuitry to decode ASCII code and store characters in built-in RAM to eliminate the need for extra circuitry for displaying (Hewlett Packard HDSP-2502) (Hewlett Packard).

The one of the features that distinguishes nice digital thermostats from simple thermostats, is a clock that is used to turn heaters/air-conditioners on and off at scheduled times. The DSP has a timer. This timer works in terms of processor clock cycles, not time of day. I realized I needed a real-time clock. The preferred features for a RTC would be having an output to indicate when it is a certain time, and to set alarms for any time down to the minute. I found a RTC with all the desired features plus a built-in battery to keep time/alarm settings during power outage, 50 bytes of general-purpose RAM that can protect important information from power outages (Dallas Semiconductor DS1286) (Dallas Semiconductor Ds1286 Watchdog Timekeeper).

After I found a neat display and clock, I turned my focus toward the heart of the thermostat. I realized that the design of how the buttons, display, and clock will interact with the DSP microprocessor would be a challenge. I forwent designing the connections to motors and thermometers.

CHAPTER 3

PROJECT IMPLEMENTATION

I drew a simple block diagram of the thermostat to brainstorm what kind of functional parts are needed. In all the excitement, I overlooked the obvious need to think of the basic design of the whole thermostat. Fortunately, the components that I have collected were robust and designed for reasonably easy implementation.

Computer systems have three types of lines: address, data, and control (Mazidi 882; Mano 385-91). Sometimes, a line will fall into more than one category. Intel 8088 has eight lines that function as data and address lines 217-222. Many peripherals, such as Hantronix LCDs and Motorola MC146818 RTC have similar lines (Hantronix "Commands for Character Modules" ; Hantronix "Processor Interfacing" ; Motorola Semiconductor 10,17-19). Address lines select a location in memory or a peripheral. For most part, address lines connect directly to the address lines of memory and other chips. High address lines often are connected via a decoder to chip enable pins of memory and other chips (Mano 297). Data lines transfer actual data between chips. Data lines are connected directly from the processor to peripherals. Control lines handle interrupt requests, indicate read or write operations, indicate the address space being used, and indicate when valid data or address is on the lines. Many control lines connect directly from the processor to peripherals, but due to semantics, control lines often have to be combined via logic circuits to form new control lines that match a group of peripherals (Mazidi 222-23,26-29). I understood the logical implications of this, but not the timing aspects.

With the key components collected, the actual designing process could begin. One of first things was figuring what and where the connections are on the DSP kit. I roughly knew what the kit’s features were, but not the details. I ran into a problem figuring what each hole is on the DSP kit circuit board. These holes are where the other microchips would be connected to the DSP. I looked up the board’s diagram in the manuals that came with the kit. Finally, I understood what a set of circuit board diagrams were showing (Texas Instruments A.1-A.8). The diagrams were schematic diagrams that are like pin diagrams done in digital logic classes, with a few important differences that confused me. The orientation of components in relation to each other is shown and is important, since the DSP kit board was one solid piece that could not be re-arranged at-will. Also the DSP has multiple rows of holes instead of lines of pins.

One of first problems was figuring how to have the DSP processor respond to the pressing of buttons quickly. If DSP does not respond within a fraction a second, a user would let go of the button, and the signal that the button made would disappear without being captured for processing. The DSP also needed to respond to the RTC’s alarm signal to keep time current or to perform a scheduled function promptly. I knew of two methods for monitoring peripherals. In one method, called polling, the DSP could periodically check the status of the peripherals. The RTC has a bit that indicates if an interrupt has occurred. The problem with this method is the intervals for checking must be short enough to assure reading each button being press. In the other method, called interrupting, for each peripheral that needs to send the DSP notices, one of the DSP’s interrupt lines is wired to the peripheral’s line that will send the notice signal (Hennessy 567-70). The RTC has two interrupts lines for its alarms. The buttons would need logic gates to form an interrupt line. I realized that the buttons need to be connected to an interrupt pin on the processor and an interrupt service program will determine which key is being pressed and give that information to the program that was running. A feature, such as interrupting is another reason to choose a full-blown processor system.

The DSP has three address spaces: program, data and I/O. Program and data spaces cover on-chip RAM and ROM. If external memory is connected to the DSP, they can also cover the external memory (Texas Instruments 10.1-10.20). In contrast, I/O space can only cover devices that are external to the DSP. That is one reason that I chose to use I/O space for the LED and RTC. There is no possibility of confusion between program/data and external peripherals. The DSP’s built-in peripherals are accessed via registers that are mapped in the beginning of data space, so they are also not covered by I/O space (Texas Instruments 8.2-8.9).

The DSP uses separate control lines for I/O access and external memory access. Each of the three addressing spaces has a space select pin that changes to ground voltage when its associated space is in use. There are two strobe pins, one for program/data memory access and one for I/O access. Thus, using the I/O strobe line to enable the LED and RTC operations prevents any command other than PORT from affecting them, even if the DSP is set to access external memory for program code and data. I looked at the CPU’s I/O timing diagrams to figure out what signal I will use to enable the LED and RTC for data transfer. After a long series of looks, I spotted the strobe signal as being ideal for that usage. I have seen the word strobe used in pin diagrams for parallel ports and expansion slots, but I only then realized what it does (Mazidi 313-503).

Another reason for choosing I/O space for the LED and RTC, is the lack of choices for timing. Program and data spaces operate as memory access. External memory can exist in multiple banks. The DSP has bank-switching capabilities that can efficiently handle multiple banks. This feature changes the timing characteristics of memory access, creating potential problems. I/O space only has the option of using wait-states that lengthen the times from when the strobe, space select, and read/write lines change values to when the data values is transmitted. This should have no negative effects for the LED and RTC.

The DSP’s wait states feature only allows more time for address and control signals to set up before data is transmitted. The data transmission is not lengthened. The address and control lines are not given more time after the data transmission. This would help with the fist half of the LED’s and RTC’s read/write cycle and with the tri-state buffers if they delay button inputs too long. Yet, the second half of the LED’s and RTC’s read/write cycle would not be helped. See Figure 11 in appendix C.

In a processor system, timing becomes an issue due having many types of lines. Each of these lines has a timing relationship to each other. Address lines activate first to select an I/O component or a region in memory. Control lines activate at various times during a cycle to control the type of action (read, write, enable a component, indicate data or address line status, and etceteras). Data lines activate last when everything else is ready and only then does valid data transfers occur. A special control line, called strobe, indicates the exact moment when the signals on the data lines are reliable and are captured by the device that is reading. After that, most or generally all lines de-activate in reverse order from which they activated. Many standard components follow this timing principle, which allowed the LED and RTC to be connected to the DSP with only simple logic circuits mitigating some lines (Mano 60-61).

The principle mentioned is applicable to systems using separate address and data lines. The timing principle for systems that have address and data multiplexed together onto the same lines is different. That principle need not be explored, since the LED, RTC, and DSP have separate data and address lines.

Compared to processor issues, what I learned in college was simpler. In the digital logic lab classes, the most complex sequential circuit assembled had several shift registers that were connected together via a few buses. In class the issue of timing was discussed simply in terms of the time taken for a signal to travel, not dependent on relationship. The control lines were manually operated via logic switches. Data and control switches are set and then the clock pulse switch is manually pushed. The registers constantly output data except during a transition, and during an operation the registers would simply receive input from a bus.

The LED and the RTC comply with a popular standard for memory interface. There are no dedicated strobe lines. The read and write lines also act as strobe lines. This standard applies to read/write lines that are one line or two separate lines (Intel 2; Mano 294-95; Mano 60-61). The DSP does not completely comply with this standard make wiring more difficult.

To connect the LED's and RTC's control lines to the DSP, their timing and other characteristics must first be analyzed. Unlike the DSP, the LED and RTC have no strobe lines. Unlike the DSP, they have separate read and write enable lines. These facts prevent straight connections of control lines from the DSP to the LED and RTC. Fortunately, the LED's and RTC's lines are very similar, thus allowing a single design solution to operate both. The read and write enable lines of the LED and RTC have the same timing characteristics as the DSP's I/O strobe line (Texas Instruments 10.16; Hewlett Packard 6-7; Dallas Semiconductor 9). See Figures 9 and 10 in Appendix C.

I wondered if the LED and RTC needed tri-state buffers to protect the data pins. If I do need tri-state buffers, would I be able to select them and still have enough time to transmit the data?

All the hardware issues confused me. Luckily, I finally found an experienced digital/analog hardware engineer, Mr. Richard Burrows, to look at the technical documents of the CPU, DSP, and RTC. The engineer pointed out several important things to me. Since the LED and RTC have data pins that act as input and output, they must have built-in tri-state function. The strobe signal of the CPU is prolonged when the CPU is instructed to lengthen its I/O timing. With those facts known, he concluded that I could wire the LED's and RTC's data pins directly to the CPU's data pins. The CPU's strobe pin can be wired to a decoder with strobe input (Texas Instruments 2). Given that a decoder can route a strobe signal, the DSP's address lines can select the read and write enable lines of the LED and RTC (Burrows). See Figure 8 in Appendix C. Just as with standard computer systems, the DSP cannot operate the LED and RTC concurrently nor perform read operations and write operations concurrently. Thus, this is a suitable design solution. The only disadvantage is that the address to read from a location in the LED or RTC is different from the address to write to the same location in the LED or RTC.

Given that a decoder can route a strobe signal, the DSP's address lines can select the read and write enable lines of the LED and RTC. Just as with standard computer systems, the DSP cannot operate the LED and RTC concurrently nor perform read operations and write operations concurrently. Thus, this is a suitable design solution. The only disadvantage is that the address to read from a location in the LED or RTC is different from the address to write to the same location in the LED or RTC.

As I examined the timing information for the CPU, LED, and RTC; I realized that the CPU operates significantly faster than the LED and RTC. There is a potential problem for the LED and RTC to be unable to successfully transmit and receive data from the CPU.

Synchronous circuits have a signal, clock, which controls the holding of bits in flip-flops. A flip-flop is a collection of logic gates that can hold a logic signal (Mano 210-18). By changing the frequency of the clock signal, the rate at which bits are captured and held can be control. It is this concept, which gave me the idea of attempting to change the clock speed of the DSP to solve my timing problem of my different components' clock speeds.

To solve the problem of the DSP running faster than the LED and RTC, I seek to manipulate the clock signals that regulate each of them. The LED and RTC can output their clock signals or be overridden by external clock signals. The RTC has the lowest clock frequency of all the components, and thus the other components could run from it. Unfortunately, the DSP came soldered to a crystal oscillator. To override its clock, I would need to break the oscillator circuit on the DSP's board, which could result in damaging the DSP's board, since the board's micro-size parts makes such an alteration extremely difficult. I would only have implemented this solution after other options failed.

Another way to make data transfer between the CPU and peripherals reliable, is to have a handshaking process. The LED and RTC that I have chosen do not have any pins that will give feedback, so I can't perform handshaking (Hewlett Packard ; Dallas Semiconductor <u>Ds1286 Watchdog Timekeeper</u>). As my computer architecture book indicates, asynchronous serial transfer would be the simplest, since it has the strobe signal merged with the data and there is only on data line instead of several. I found LEDs and RTCs that have serial data interfaces in my catalog. I can't do that too, since the CPU's serial port operates synchronously, thus it does not produce start and stop bits. I could connect a UART to the CPU, but that would probably be as complicated as connecting the LED and RTC directly with the CPU.

Many Intel processors (486 and 586) can operate at multiples of an external clock speed (Mazidi 638-39). The DSP has a similar feature. The DSP has three clock mode pins that determine what multiple of its internal or an external oscillator. These pins are accessible from the solder holes in the board. When I started to wire the thermostat, I tried to change the voltages that appear at the DSP’s clock mode pins. The unfortunately, one of the pins would not change to voltage that I apply to the corresponding soldering hole.

The next piece of hardware I needed to add would was the buttons that would be used for user input. As with wiring the LED and RTC to the DSP, there are design considerations for interfacing buttons into the thermostat. I connected the buttons to DSP data lines that are not being used by the LED or RTC. I figured this would eliminate the need for adding tri-state buffers to control the buttons’ connection to the data bus.

Unlike a common logic gate, tri-state buffers have a high-impedance state that effectively acts as a broken circuit. When many components share a common line, they use tri-state buffers to have only one component connect to the line at a time (Mano 420).

I used a group of “AND” gates to effectively merge the data lines to one interrupt signal, the button interrupt.

In a computer repair class that I took, I found out that mechanical switches/keys bounce that causes voltage fluctuations. Debouncing circuits will be needed for my thermostat’s keypad. I looked at the breadboard that I borrowed from ULV, and I noticed that a 7414 microchip is next to the pulse buttons. Ironically, I did not realized what it really does until two months later. 7414 is just a schmitt-trigger version of 7404 (Texas Instruments 1.2). Schmitt-trigger is different from any microchip shown in my classes. It avoids the ambiguity that exist between logic 0 voltage and logic 1 voltage by not changing its output until the input crosses the ambiguous voltage range and reach one of the two trigger voltages (Rizzoni 752).

I forgot that the data lines are also outputs. When the DSP is outputting data, data signals from the DSP reach the inputs of the “AND” gates that emit the button interrupt signal, and the “AND” gates act as if the buttons were being pressed. I had to put a tri-state buffer in the button circuits. I bought the microchips that comes closes to the AHCT (advanced high-speed CMOS, TTL compatible) family that I am using. The electronics suppliers that will sell the microchips that I prefer only sell it as a special order and require me to buy over a thousand.

I bought 74126, 74125, 74240, and 74244. These four microchips are all tri-state buffers. I preferred to use 74240 or 74244, since one pin operates the control of all the tri-state buffers. I was not sure what microchip to use, since DSP documentation is not clear on whether "read" is incoming data and "write" is outgoing data or visa versa. I determined input or output only by machine instructions: port(address)=data and data=port(address). I discovered that 74126 is the one that works the best. It has control pins that enable the tri-state buffers when they receive a high signal.

Documentation for the DSP shows that the read/write pin is high for read operation (Texas Instruments 10.10-10.17). This means that the DSP documents assign the term "read" to receiving data from external sources. The other three microchips have the negative type of control pins, which made them useless for my use.

CHAPTER 4

HARDWARE FEATURES

The DSP’s address lines connected the RTC’s and LED’s address lines and is decoded to control their read and write operations. A memory map can be constructed to indicate all the LED’s and RTC’s functions. I found making this table to be a very useful way of getting more aquatinted with the LED and RTC. It also serves as a quick reference for programming the thermostat. See Figures 4, 5, 6, and 7 in Appendix B.

The LED and RTC are very self-contained components. They each have a built-in crystal to synchronize their circuits. They operate more like many sophisticated I/O devices than common microchips. Control pins do not designate most of their functions. Instead, commands are given by loading values into their various memory/register locations. To make an action or a configuration change, one needs to address the appropriate memory/register and give the settings or values as data. I am using the term “memory/register”, since there is a combination of memory and registers that are addressed as part of the memory space.

The RTC’s memory map is relatively simple. All its functions and values are stored in a contiguous block of memory/register. The only thing that is complex about the map is some control bits are in the current month memory location instead of the command register. Unlike the hour format bits, these bits have nothing to do with their location. The LED’s memory map is far more complex. It nearly drove me insane! The map is not contiguous. It is a hodgepodge collection of memories and registers. Unlike the RTC, the LED’s address lines do not act as standard address lines. The LED has a pin named “FL” that has the timing behavior of an address line, but is actually function pin for flash RAM operation. The address range is riddled with doesn’t cares that prevent a straight forward memory map. The way I choose to deal with this problem, is to simply wire the address lines of the LED and RTC straight to the RTC’s address lines, and the same for the data lines. Deal with the complexities in with software code.

CHAPTER 5

SYNTAX AND SOFTWARE CODE

The DSP’s machine language is very different from the IBM PC compatible machine language with which I am experienced. PCs run software in segments. Program, data, and the stack are loaded into separated segments and are indicated by their respective segment registers. Program means the actual sequence of instructions that tell the computer system what to do. Data is the collection of variables used by the program. Stack is the stack that temporary values are stored, especially when subroutines are executed. These three elements also exist in the DSP, but in a different form. Data and program are stored in separate address spaces. Data and program are stored in two separate memory systems that have separate addressing. There is address 100 in program space and another address 100 in data space. Stack is placed in the program space and is indicated by a stack pointer. PCs also have addressing spaces. The Program, data, and stack go into memory space. PCs and the DSP have separate addressing space input/output peripherals (Texas Instruments 9.30-9.42).

I found that I could perform consecutive readings from the LED without problems. The problem was how I defined the variables used to hold the read values. I used “.space” assembler directive. It is documented for use with setting aside memory for variables. Its uniqueness is that it can define the allocated space in bits and not give an initial data value for the variable. I accidentally used an odd value that made the apparent addresses of my variables deceptive. Due to a confusing selection of assembler variable/data directives, I thought that “.word” directive, which initializes a word of memory with a data value, would have a fixed data value that can’t be changed by the processor during execution. However the “.word” directive was actually the perfect directive for all my variables and data locations.

Input/output operations operate with a word of data (Texas Instruments 4.1-4.86).

Besides the hardware connections, using interrupts also involve software implementation. When a processor receives an interrupt signal, it must note that an interrupt has occurred and, if “more than” exists, which interrupt it is. This information is stored in the DSP’s interrupt flag register. There are two other registers that DSP software code uses to control the handling of interrupts. The interrupt mask register is used to control the monitoring and ignoring of individual interrupts. To quickly ignore all interrupts, the interrupt mode (INTM) bit of status register zero (ST0). Fortunately, DSP assembly language offers directly access to register bits, such as INTM. I set this bit to disable interrupts during the execution of important code that most not is interrupted. I do not nee to manipulate INTM when calling interrupt service routines, since the DSP automatically set INTM to disable interrupts when calling the routine and enables interrupts when leaving the routine (Texas Instruments 6.26-42).

Again reading the interrupt section of one of my DSP manuals. This time, things started to click in my mind. I went back to the samples of software code that came with the kit and finally matched what the manuals say with the sample codes. I copied the sample interrupt vector table and changed the pointer for "INT0" (the button input interrupt). I then copied my "HELLO" routine to my time display program and inserted the label to which interrupt points. Nothing appeared to have changed. I tested the switch circuits with a digital multi-meter and I found one problem, which I fixed quickly. Nothing still changed. I guessed that the interrupt was masked. I cleared the interrupt mask register. The display then showed "HELLO" and the time simultaneously without pushing any buttons. I suspected oscillation was occurring somewhere in the button circuit. The logic probe confirmed it. I realized that the oscillation was coming from the data lines to which the buttons are connected. I disconnected the data lines, and the interrupt routine then behaved as I planned.

The DSP kit's user manual boasts that it has an easy to use algebraic assembly language (Texas Instruments 1.7). Months earlier, I wondered what the difference was between mnemonic and algebraic assembly languages. When I saw a reference manual for each listed in a related document list in the kit's user manual I thought that it was not important to understand the deference, since I only need to use the algebraic assembler software (Texas Instruments vi-vii). I was wrong. When I looked for the I/O instructions that I had use for programming. The DSP CPU architecture reference manual shows the I/O instructions, but I could not find them in the algebraic assembly reference manual (Texas Instruments <u>Tms320c54x Dsp Reference Set: Algebraic Instruction Set</u>). I ran into confusion. I discovered that the architecture manual gives information in terms of mnemonic instructions (Texas Instruments 2.14,3.12,5.4-5.7). When I downloaded the mnemonic assembly reference manual and found the I/O instructions that I saw in the architecture reference manual (Texas Instruments <u>Tms320c54x Dsp Reference Set: Mnemonic Instruction Set</u>). Then I thought that there were differences in the capabilities of the two assemblers and that I will need the mnemonic assembler to perform I/O operations. I finally found a statement in the kit user's manual that illustrates and clarifies the differences between the two assemblers. They do have the same capabilities. The algebraic instructions are in the form equations for people are not familiar with assembly languages. Mnemonic instructions are like traditional assembly instructions (Texas Instruments 1.7).

This explains the difficulty I had understanding the algebraic instructions. I was trying to compare algebraic instructions to the assembler instructions I learned in my college courses. I finally found the algebraic instructions I was looking for by searching for the description headlines that both assembler reference manuals have in common. I do not know what Texas Instruments was thinking when they made two assemblers with drastically different syntax for the same CPU. Algebraic reads more naturally like “C language”, but it still has as many syntax restrictions and bulkiness of traditional assembly language.

I successfully ran code to have the clock (RTC), send an interrupt once per second, and have the DSP respond by running an interrupt service routine (ISR) that reads the time and displays it on the LED. I also have successfully ran the ISR code that reads the button inputs and displayed them on the LED in response to flipping a button. I had both ISRs (time and input) run in the same execution. Then I could begin to write the actual thermostat code.

Now its time to make seriously detailed design decisions. I look upon this project as an opportunity to express my design preference for rock-solid reliability that too many of today’s cheaply made systems do not have. The thermostat should always retrieve information from its source, instead of indirect methods that would be simpler or more efficient but more unreliable. Each minute, the RTC will send an interrupt to the CPU; the CPU then copies the new time from the RTC a few times; the CPU then verifies the new time reading by comparing copies to each other. This is done instead of just incrementing the CPU’s copy of the time after each interrupt. When the user sets the time, the CPU write the user’s time to its RAM while they are inputting; the CPU copies the whole time to the RTC; the CPU then reads the RTC’s time to verify that the RTC is set to the user’s time. When the display’s contents need to be changed, the CPU updates display data in its RAM and copy the whole display data to the LED; the CPU then copies display data from the LED’s RAM to verify a successful transmission. This done instead of transferring and verifying only the display data characters that actually change. The chosen method is less efficient, especially if the display is updating from user input, but the software routine for the display is simplified. If a transmission has failed, it will be repeated. One could be transmitted for satiability reasons, however for software reasons all of it must me retransmitted.

I believe that the architecture of the thermostat’s software should be based on the architecture of the DOS operating system. A DOS boot disk contains free system files (IO.SYS, MSDOS.SYS, and COMMAND.COM). IO.SYS is the set of routines that adapt the operating system’s core to the system’s hardware and contains service routines for file operations, etceteras. MSDOS.SYS is the operating system’s core or kernel that is coordinates activities and manage resources of the system. COMMAND.COM is the command interpreter, which translates the user’s commands into instructions that the kernel can understand (Forney 240-41,318-19,47; Mazidi 650-55). The DSP chip has an interrupt vector table that handles hardware and software interrupts. That will allow me to have my kernel and other routines call over routines via software interrupts. Secondary operations such as setting the correct time could be handled as hybrid of application programs (the kernel will give control over to the operation) and service routines (operation will be requested by a known software interrupt and known parameters) (Mazidi 822-61).

Later, I came up with a different scheme for planning my thermostat's software. I was trying to come up with a scheme that follows MS-DOS software architecture (what I understood of it). I now came up with a scheme based on 'modes'. As I look at my basic draft plan, I noticed that it resembles DOS more accurately than my earlier scheme and that my earlier scheme has a few mistakes in it. Sometimes re-inventing the wheel is better.

I have three modes (menu, decision, and control equivalent to MS-DOS's COMMAND.COM, MSDOS.SYS, and IO.SYS, respectively). Decision mode is the default mode in which decisions are made and actions are called. Menu mode is the mode, which is triggered when a user presses a button and is responsible for users inputting the desired temperature-time settings. Control mode is the mode that is activated by decision mode when it decides that a task of motor controlling or temperature reading is needed. For simplicity, control and menu modes are uninterruptible. This is like in DOS when applications and the command prompt were not pre-empted by anything and ended only when their tasks were finished. Time updates occur and button input interrupts are acknowledge only when decision mode is running. The reasoning is time updates are only needed to trigger decision mode to call for actions and the buttons will be read every second or less when menu mode is running. At the end of the control and menu modes, time will be updated. See Figure 12 in Appendix C.

I was inspired by how time and alarms are set in digital watches. They only use a few buttons to navigate all their functions, and the display flashes characters and digits to indicate what is being changed. Fortunately, the LED has built-in flashing (blinking of selected characters). I was going to use the LED's flashing in the same manner as a digital watch. Characters are selected for flashing by changing the contents of the LED's flash RAM.

Making use of LED's flashing capability to indicate a cursor position, there are two variables for the cursor. "curpos" represents the current cursor place, while "curflsh" represents the current LED positions that should be flashing. Having both provides abilities to turn off flashing without forgetting the cursor position and have LED positions flash that do not represent the cursor position.

The PORT command requires an immediate address, thus changing I/O addresses cannot be automated by using a variable (Texas Instruments 1-4,3.65-3.66). To over come this problem for the proposed thermostat software, the characters are placed in a location in DSP's RAM instead of being sent directly to the LED. A subroutine is used to automatically to copy all eight characters from the location in DSP’s RAM to the location in LED’s RAM that displays ASCII characters.

As I feared, the DSP assembly language has proven itself to be very difficult.

The code I made to test the hardware only used a few types of instructions and a few simple addressing modes. As soon as I started to try using more instructions and addressing mode, I ran into the language's restrictive syntax numerous times. Auxiliary registers can't be used in many arithmetic-logic operations, accumulator registers can't be used in stack operations, and etceteras(Texas Instruments 1.2,1.3,2.11,2.15).

As soon as I tried to make procedures, the assembler crashes. I had to change to clean booting an old version of MS-DOS to assemble.

After eliminating errors and tracing the problem down to a few instructions. I received unexpected garbage on the LED display. I tried to trace the problem by means of elimination with no success. The software to bring the thermostat closer to functionality (proposed software) is not completed.

I have decided to submit my project with my primitive code that I made to fully test the hardware, including interrupts. It has the DSP interrupt table that contains my time and button-input interrupts, and the corresponding interrupt routines. The code illustrates the hardware functionality by reading the time from the RTC, changing the data format, writing to the time to the LED, reading button pressing, and display the button pattern on the LED. I also provide the proposed code of my thermostat in Appendix A to further illustrate how the thermostat should and can operate.

CHAPTER 6

CONCLUSION AND FUTER EXPANSION

There is one thing that I really could have done better. R/W' (read/write) pin would probably have worked correctly in place of one of the address lines that went to the select pins of the decoder, thus simplifying my I/O addressing scheme. The address to read from a location in the LED or RTC would be the same as the address to write to the same location in the LED or RTC. The thermostat's I/O address map would be half its current size. R/W' pin has timing characteristics that are nearly identical to the addressing lines. Small differences in timing should not make a difference, since it is the IOSTRB (I/O strobe) pin that actually triggers the decoder into sending a signal. I try to use as few different types of pins as possible, because I was uncomfortable with timing.

In the above mention implementation, the decoder would effectively combine the strobe and R/W' lines to form a more conventional read and write signals. This is similar to Early IBM Personal Computers that had the Intel 8086 processor mix its read and write signals with its address space select (IO/MEM) line to form hybrid signals, read and writes for memory and for I/O. (Mazidi 222-23)

This project has taught me some valuable lessons. As a customer of numerous computer hardware products, I complain when I buy a new hardware product that has less logic circuitry and thus requires more external processing power to run its larger software. One example is a new printer that I bought. Its driver (software) does much of the computations for the printer and consumes large amounts of RAM and CPU time. My older printer's driver simply acted as an adapter for the printer and Window's printer manager or a DOS program. As a designer, I can now appreciate the trend toward software design. I try to make the hardware design of my project to be as simple as possible. As a direct result, the software became more complicated. Hardware is more difficult to design than software. Software has crisp digital logic. From the software side, a machine instruction is clearly defined. From the hardware side, a machine instruction is defined as a collection of electronic signals that have timing and other electronic characteristics. Those characteristics are analog and can fluctuate based on different conditions (quality of wiring, environment conditions, and etceteras).

The thermostat obviously needs temperature sensors and window motor controls to complete its hardware. Using another of the DSP's interfaces to connect the sensors and controls would be a good idea. This avoids altering the core thermostat hardware that has already been proven to function properly. Also, this provides increased reliability. Unlike the core hardware that can be placed in a case, the sensors and controls would be strung around the building. The DSP still has a buffered serial port and a time-division serial port available (Texas Instruments 1.4). Only the DSP will control the sensors and controls, thus the time-division serial port is inappropriate.

One set of products that show potential are Dallas Semiconductor's 1-wire, line-powered "MicroLAN" series of switches, digital temperature sensors, and other items such as real time clocks. There web site is http://www.dalsemi.com. The advantage of "MicroLAN" is that it only uses one wire to transfer information (Dallas Semiconductor Application Note 104: Minimalist Temperature Control Demo). This reduces wiring. Conceivably, a set of only four wires would be needed to run around the building. One wire would be used as the ground for the "MicroLAN" devices. Another wire would be used as the data line for the "MicroLAN" devices. "MicroLAN" devices also use the data line as its power line. The other two wires would be used for the window motors. This scheme separates the ground of the motors from the ground of the “MicroLAN”. When motors start and stop, the voltage that appears across them fluctuates. This is acceptable for the motors, but would corrupt logic values for the “MicroLAN”. Another advantage is one wire is both control and data which means there are no timing issues. Dallas Semiconductor makes other serial products with 2-wire and three-wire interfaces. Those products are not compatible with the DSP’s interface timing characteristics.

The consequence of having a one-wire interface is that the protocol of the devices is relatively complex. To provide an easier design solution, Dallas Semiconductor makes adapters/controllers that can connect “MicroLAN” to the serial ports on computers, which are asynchronous. Using such adapters might be difficult, since the DSP serial port’s design is proprietary, synchronous, and is designed to connect to similar DSPs. Wiring the “MicroLAN” directly to the DSP serial port could also be difficult. Without an adapter, the DSP’s buffered serial port’s pin for output and pin for input (Texas Instruments 9.33) would have to be wired to MicroLAN’s single input/output wire. Tri-state buffers would need to control the directions of signals. Fortunately, since there are no hardware timing issues, software could emulate the “MicroLAN” signals by using the continuous mode of the DSP serial port (Texas Instruments 9.25).

If the Center for Regenerative Studies ever decides to fund this thermostat, the “C language” compiler for TMS320C54x could be purchased. With the “C language”, most computer scientists could program the temperature management routines of the thermostat as needed without being familiar with the hardware. This is the theoretical part of the thermostat, and would require a term of trial and error. Other students could possibly make parts of the thermostat, in their projects. I could finish designing many of the hardware control routines and provide function headers for linking to “C language” code to minimize the need to know the thermostat’s hardware design for other students. A student who specializes in network protocols could write the code to emulate the “MicroLAN” signals. A student who specializes in artificial intelligence could design the temperature management of the thermostat. Due to the need for such specialties, the student probably needs to be graduate students.

APPENDIX A

PROJECT SOURCE CODES

Proposed Software – Main Body

```
; Diagnostic Thermostat Software
; This File was copied from a Texas Instruments sample file,
;     and then altered by Michael Warner II.
;
; ~ in comments indicates continuation of comment from previous the line
;
; LED stores characters as ASCII ("23" = 01100010b 01100011b)
; RTC stores time values as packed BCD (23d = 0010 0011 b)
;
; ***************************************************************************
; File: FirstApp.ASM -> First Application program for the 'C54x DSKplus
;
; ***************************************************************************

        .width   80
        .length  55
      .title "Test input"

        .mmregs                     ;associate register names
                              ;~with their address
      .setsect ".text",   0x500,0   ;loads program section
      .setsect "vectors", 0x180,0   ;loads interrupt vector table
      .setsect ".data",   0x700,1   ;loads data section

      .sect "vectors"               ;beginning of interrupt vector table
      .copy "d:\personal\senior\vector~2.asm"   ;copies from another file


      .data                   ;beggining of data section

; variables used as constants for DSP ouput
ledb  .word ' '
ledt  .word ':'
logic0      .word 0000h
logic1      .word 0001h
leds  .word 000Ah ;initial command values for LED
rtcs  .word 0097h ;initial command values for RTC
wdt   .word 0001h ;RTC watchdog # of seconds

; variables
read1 .word 0000h
read2 .word 0000h
read3 .word 0000h
cursor      .word 0040h
```

```
      .text

      pmst = #01a0h                   ; set up iptr
      sp = #0ffah                       ; init stack pointer.

      IMR = #0ffffh                   ;un-mask all interrupt

; initialization
      port(0070h) = *(leds)           ;initialize LED commands
      port(00CCh) = *(logic0)         ;set watchdog timer
      port(00CDh) = *(wdt)            ;~to 1 second intervals
      port(00CBh) = *(rtcs)           ;initialize RTC commands
      port(0040h) = *(logic0)         ;turn off LED flashing
      port(0041h) = *(logic0)         ;~
      port(0042h) = *(logic0)         ;~
      port(0043h) = *(logic0)         ;~
      port(0044h) = *(logic0)         ;~
      port(0045h) = *(logic0)         ;~
      port(0046h) = *(logic0)         ;~
      port(0047h) = *(logic0)         ;~

; infinite loop to keep DSP running
place
      goto place


; button interrupt service - visually indicate which button is pressed
KEYIN
      *(read3) = port(0001h)          ;read data lines
      B = *(read3)
      B = B <<C -12                   ;shift button data bits down
      B &= #000Fh               ;clear extra bits
      B += #0030h               ;add 30h to form ASCII character
      *(read3) = B
      port(007Fh) = *(read3)          ;output chracter to LED
      return_enable                   ;end button interupt service


; RTC interrupt service - displaying the time
TIMEIN
      port(007Dh) = *(ledt)           ;output colons to LED
      port(007Ah) = *(ledt)           ;~

; displaying the hours
      *(read1) = port(0084h)          ;read hours from RTC
      B = *(read1)                    ;copy for ones digit
      *(read1) &= #000Fh              ;clear extra bits
      *(read1) += #0030h              ;add 30h to form ASCII character
      B = B <<C -4                    ;shift tens digit down four bits
      *(read2) = B                    ;copy for tens digit
      *(read2) &= #000Fh              ;clear extra bits
      *(read2) += #0030h              ;add 30h to form ASCII character
      port(0079h) = *(read1)          ;output numbers to LED
      port(0078h) = *(read2)          ;~

; displaying the minutes
      *(read1) = port(0082h)          ;read minutes from RTC
      B = *(read1)
      *(read1) &= #000Fh
      *(read1) += #0030h
      B = B <<C -4
      *(read2) = B
      *(read2) &= #000Fh
      *(read2) += #0030h
      port(007Ch) = *(read1)
      port(007Bh) = *(read2)

; displaying the seconds
      *(read1) = port(0081h)        ;read seconds from RTC
      B = *(read1)
      *(read1) &= #000Fh
      *(read1) += #0030h
      B = B <<C -4
      *(read2) = B
      *(read2) &= #000Fh
      *(read2) += #0030h
      port(007Fh) = *(read1)
      port(007Eh) = *(read2)

      return_enable                 ;end RTC interrupt service

      .end
```

Test Code – Interrupt Vector Table “Vectors~1.asm”

```
; Interrupt Vector Table for Thermostat Diagnostic Software
; This File was copied from a Texas Instruments sample file,
;     and then altered by Michael Warner II.
;
; ~ in comments indicates continuation of comment from previous the line
;
; Only int0 and int1 are used by thermostat.
;
; ***********************************************************************
; File: VECTORS.ASM -> Vector Table for the 'C54x DSKplus           10.Jul.96
;
; ***********************************************************************
; The vectors in this table can be configured for processing external and
; internal software interrupts. The DSKplus debugger uses four interrupt
; vectors. These are RESET, TRAP2, INT2, and HPIINT.
;   *  DO NOT MODIFY THESE FOUR VECTORS IF YOU PLAN TO USE THE DEBUGGER  *
;
; All other vector locations are free to use. When programming always be sure
; the HPIINT bit is unmasked (IMR=200h) to allow the communications kernel and
; host PC interact. INT2 should normally be masked (IMR(bit 2) = 0) so that the
; DSP will not interrupt itself during a HINT. HINT is tied to INT2 externally.
;
;
;
      .width   80
      .length  55
       .title "Vector Table"
      .mmregs                  ;associate register names
                               ;~with their address

reset  goto #80h    ;00; RESET  * DO NOT MODIFY IF USING DEBUGGER *
      nop
      nop
nmi    return_enable      ;04; non-maskable external interrupt
      nop
      nop
      nop
trap2  goto #88h    ;08; trap2  * DO NOT MODIFY IF USING DEBUGGER *
      nop
      nop
      .space 52*16        ;0C-3F: vectors for software interrupts 18-30

; Points to RTC routine.
int0   goto KEYIN   ;40; external interrupt int0
      nop
      nop
      nop

; Points to button routine.
int1   goto TIMEIN        ;44; external interrupt int1
      nop
      nop
      nop
```

```
int2   return_enable      ;48; external interrupt int2
      nop
      nop
      nop
tint   return_enable      ;4C; internal timer interrupt
      nop
      nop
      nop
brint  return_enable      ;50; BSP receive interrupt
      nop
      nop
      nop
bxint  return_enable      ;54; BSP transmit interrupt
      nop
      nop
      nop
trint  return_enable      ;58; TDM receive interrupt
      nop
      nop
      nop
txint  return_enable      ;5C; TDM transmit interrupt
      nop
      nop
int3   return_enable      ;60; external interrupt int3
      nop
      nop
      nop
hpiint       dgoto #0e4h        ;64; HPIint  * DO NOT MODIFY IF USING
DEBUGGER *
      nop
      nop
      .space  24*16       ;68-7F; reserved area
```

Proposed Software – Main Body

```
; Proposed Thermostat Software
; This File was copied from a Texas Instruments sample file,
;     and then altered by Michael Warner II.
;
; ~ in comments indicates continuation of comment from previous the line
;
; ***************************************************************************
; File: FirstApp.ASM -> First Application program for the 'C54x DSKplus
;
; ***************************************************************************

      .width  80
      .length 55
      .title "Test input"

      .mmregs                        ;associate register names
                              ;~with their address
      .setsect ".text",    0x500,0   ;loads program section
      .setsect "vectors", 0x180,0   ;loads interrupt vector table
      .setsect ".data",    0x700,1   ;loads data section

      .sect "vectors"                ;beginning of interrupt vector table
      .copy "d:\personal\senior\Code\Vectors.asm"
                              ;copies from another file

      .data
;spoint     .word 0     ;points to current temperature setting
;sched      .word 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
;10 temperature settings (day,hour,minute)

curtemp     .word 0070h
curtime     .word 0,0,0,0,0          ;(day, hour 10s/1s, minute 10s/1s)
curdspl     .word 0,0,0,0,0,0,0,0   ;eight LED characters
curkey      .word 0000h       ;current key read
curflsh     .word 0000h       ;current flash locations
curpos      .word 0000h       ;current cursor position
                        ;~(each bit = LED character position)

; LED values
ledb  .word ' '         ;variables used as constants for output
ledc  .word ':'         ;~
ledinit     .word 000Ah       ;initial command values for LED
daydd .word 'S','u','M','o','T','u','W','e','T','h','F','r','S','a'

; RTC values
rtcinit     .word 0097h       ;initial command values for RTC
rtckey      .word 0050h       ;RTC watchdog # of .01 seconds for key read
rtctick     .word 0060h       ;RTC watchdog # of seconds for time update
rtcstr      .word 0041h
rtcintm     .word 0000h       ;RTC watchdog interrupt context
                        ;~(1 = real-time, 2 = key rate)

; variables used as constants for DSP output
```

```
logic0      .word 0000h
logic1      .word 0001h

; variables
read1 .word 0000h
read2 .word 0000h
read3 .word 0000h

      .text

;Initialization BEGIN

;     Initialize DSP BEGIN
      pmst = #01a0h                   ;set up iptr
      sp = #0ffah               ;init stack pointer.
      imr = #0ffffh                   ;enable all interrupts
      intm = #0001h                   ;globally disable interrupts for now
;     Initialize DSP END

;     Initialize LED BEGIN
      port(0070h) = *(ledinit)        ;set LED settings

      port(0040h) = *(logic0)         ;clear flashing
      port(0041h) = *(logic0)         ;~
      port(0042h) = *(logic0)         ;~
      port(0043h) = *(logic0)         ;~
      port(0044h) = *(logic0)         ;~
      port(0045h) = *(logic0)         ;~
      port(0046h) = *(logic0)         ;~
      port(0047h) = *(logic0)         ;~

      port(0078h) = *(ledb)           ;blank LED
      port(0079h) = *(ledb)           ;~
      port(007Ah) = *(ledb)           ;~
      port(007Bh) = *(ledb)           ;~
      port(007Ch) = *(ledb)           ;~
      port(007Dh) = *(ledb)           ;~
      port(007Eh) = *(ledb)           ;~
      port(007Fh) = *(ledb)           ;~
;     Initialize LED END

;     Initialize RTC BEGIN
      port(00CCh) = *(logic0)         ;set watchdog interval
      port(00CDh) = *(rtctick)        ;~to 60.00 seconds
      port(00C9h) = *(rtcstr)         ;start RTC's clock
      *(rtcintm) = #0001h             ;set watchdog context to real time
      *(read1) = port(0082h)          ;set RTC to 24 hour time
      *(read1) &= #00BFh              ;~
      port(00C2h) = *(read1)          ;~
      port(00CBh) = *(rtcinit)        ;enable watchdog on intA
;     Initialize RTC END

      intm = 0                  ;ready to receive interrupts

;Initialization END
;==================================================
;Decision Mode BEGIN
```

```
; infinite loop to keep DSP running
dm1   nop
      goto dm1
; Temperature prediction
;Decision Mode END
;---------------------------------------------------
;Menu Mode BEGIN
; THIS PROCEDURE IS NOT FUNCTIONAL.
; NON-UNDERSTANDABLE PARTS WERE BEING CHANGED FOR DEBUGGING.
KEYIN
; back-up registers contents
      AR7 = B
      push(AR7)
      AR7 = A
      push(AR7)

; change from clock read mode to key read mode
      port(00CCh) = *(rtckey)       ;set watchdog interval
      port(00CDh) = *(logic0)       ;~to 00.50 seconds for key rate
      *(rtcintm) = #0002h           ;set watchdog context to key rate

      *(curkey) = #0000h
      *(curpos) = #0007h
ki1
      B = *(curkey)
      B -= #0001h
      if (bneq) goto ki2
      A = *(curpos)
      A -= #0007h
      if (aeq) goto ki2
      *(curpos) += #0001h
      *(curkey) = #0000h
      call chnpos
      call toled
ki2
      B = *(curkey)
      B -= #0002h
      if (bneq) goto ki3
      A = *(curpos)
      if (aeq) goto ki3
      *(curpos) -= #0001h
      *(curkey) = #0000h
      call chnpos
      call toled

;
ki3   B = *(curkey)
      B -= #0004h
      if (bneq) goto ki1
      goto ki4

      goto ki1
ki4
; change from key read mode to clock read mode
      *(rtcintm) = #0001h           ;set watchdog context to real time
      port(00CCh) = *(logic0)       ;set watchdog interval back
      port(00CDh) = *(rtctick)      ;~to 60.00 seconds for real time
```

```
; restore register contents
      AR7 = pop()
      A = AR7
      AR7 = pop()
      B = AR7
      return_enable
;Menu Mode END
;--------------------------------------------------
; Control Mode BEGIN
; Thermometers and relays control main procedure goes here.
; Control Mode END
;Decision Procedures================================
;     Time Update BEGIN
timeupd
; back-up registers contents
      AR7 = B
      push(AR7)

; update day of week
      *(curtime) = port(0086h)      ;read day from RTC
      *(curtime) &= #000Fh          ;clear extra bits and store

; update hour of day
      *(curtime + 2) = port(0084h)  ;read hour fromRTC
      B = *(curtime + 2)            ;copy for ones digit
      *(curtime + 2) &= #000Fh      ;clear extra bits and store
      B = B <<C -4                  ;shift tens digit down four bits
      *(curtime + 1) = B            ;copy for tens digit
      *(curtime + 1) &= #000Fh      ;clear extra bits and store

; update minute of hour
      *(curtime + 4) = port(0082h)
      B = *(curtime + 4)
      *(curtime + 4) &= #000Fh
      B = B <<C -4
      *(curtime + 3) = B
      *(curtime + 3) &= #000Fh

; restore register contents
      AR7 = pop()
      B = AR7
      return
;     Time Update END
;-----------------------------------------------------
;     Display Time BEGIN
distime
; back-up registers contents
      AR7 = B
      push(AR7)

; find string for current day
      B = #daydd        ;get address of string list of days
      B -= #0002h       ;get address of today's characters
      B += *(curtime)         ;~
      B += *(curtime)         ;~
      AR7 = B
```

```
; copy string to DSP's display space
      B = *AR7+
      *(curdspl) = B
      B = *AR7
      *(curdspl + 1) = B

; copy hours and minutes to DSP's display space
      data(curdspl + 2) = *(ledb)         ;blank
      data(curdspl + 3) = *(curtime + 1)  ;hours tens
      *(curdspl + 3) += #0030h            ;convert to ASCII
      data(curdspl + 4) = *(curtime + 2)  ;hours one
      *(curdspl + 4) += #0030h
      data(curdspl + 5) = *(ledc)         ;":"
      data(curdspl + 6) = *(curtime + 3)  ;minutes tens
      *(curdspl + 6) += #0030h
      data(curdspl + 7) = *(curtime + 4)  ;minutes ones
      *(curdspl + 7) += #0030h

; restore register contents
      AR7 = pop()
      B = AR7
      return
;     Display Time END
;Menu Procedures====================================
;     Key Read BEGIN
; THIS SUBROUTINE IS NOT FUNCTIONAL.
; NON-UNDERSTANDABLE PARTS WERE BEING CHANGED FOR DEBUGGING.
keyrd
      *(curkey) = port(0001h)             ;port address is a dummy
      B = *(curkey)                       ;keys are the highest 4 bits
      B = B <<C -12                       ;~of the 16 bit data line
      B &= #000Fh                   ;cleared unused bits
      B ^= #0FFFFh                        ;invert
      *(curkey) = B

      return
;     Key Read END

;     Change Position BEGIN
chnpos
; backup register contents
      AR7 = B
      push(AR7)
      AR7 = A
      push(AR7)

      B = *(curpos)
      A = #0080h
      *(curflsh) = #0000h
cp1
      if (beq) goto cp2
      B += #0001h
      A = A <<C -1
      goto cp1
      *(curflsh) = A
cp2
```

```
; restore register contents
      AR7 = pop()
      A = AR7
      AR7 = pop()
      B = AR7
      return
;     Change Position END
;Control Procedures===================================
; Thermometers and relays control subroutines go here.
;General Procedures===================================
;     RTC Watchdog Interrupt BEGIN
TIMEIN
; backup register contents
      AR7 = B
      push(AR7)

      B = *(rtcintm)          ;read the RTC watchdog interrupt context
      B = B <<C -1
      if (bneq) goto ti1      ;if context was not real-time
      call timeupd            ;update DSP's copy of time
      call distime            ;copy time to DSP's display space
      call toled        ;update LED contents
      goto ti2
ti1   call keyrd        ;read buttons

; restore register contents
ti2   AR7 = pop()
      B = AR7
      Return_enable
;     RTC Watchdog Interrupt END
;------------------------------------------------------
;     Transfer to LED BEGIN
toled:
; backup register contents
      AR7 = B
      push(AR7)

; set LED flashing according to cursur position
      B = *(curflsh)                ;read cursor position

      B = B <<C -1
      if (c) goto ttl1        ;if cursor is position 1
      port(0040h) = *(logic0)       ;clear flashing at position 1
      goto ttl2
ttl1  port(0040h) = *(logic1)       ;set flashing at position 1
ttl2
      B = B <<C -1
      if (c) goto ttl3        ;if cursor is position 2
      port(0041h) = *(logic0)
      goto ttl4
ttl3  port(0041h) = *(logic1)
ttl4
      B = B <<C -1
      if (c) goto ttl5        ;if cursor is position 3
      port(0042h) = *(logic0)
      goto ttl6
ttl5  port(0042h) = *(logic1)
```

```
ttl6
      B = B <<C -1
      if (c) goto ttl7        ;if cursor is position 4
      port(0043h) = *(logic0)
      goto ttl8
ttl7  port(0043h) = *(logic1)
ttl8
      B = B <<C -1
      if (c) goto ttl9        ;if cursor is position 5
      port(0044h) = *(logic0)
      goto ttl10
ttl9  port(0044h) = *(logic1)
ttl10
      B = B <<C -1
      if (c) goto ttl11       ;if cursor is position 6
      port(0045h) = *(logic0)
      goto ttl12
ttl11 port(0045h) = *(logic1)
ttl12
      B = B <<C -1
      if (c) goto ttl13       ;if cursor is position 7
      port(0046h) = *(logic0)
      goto ttl14
ttl13 port(0046h) = *(logic1)
ttl14
      B = B <<C -1
      if (c) goto ttl15       ;if cursor is position 8
      port(0047h) = *(logic0)
      goto ttl16
ttl15 port(0047h) = *(logic1)

; copy from DSP's display space to LED
ttl16
      port(0078h) = *(curdspl)
      port(0079h) = *(curdspl + 1)
      port(007Ah) = *(curdspl + 2)
      port(007Bh) = *(curdspl + 3)
      port(007Ch) = *(curdspl + 4)
      port(007Dh) = *(curdspl + 5)
      port(007Eh) = *(curdspl + 6)
      port(007Fh) = *(curdspl + 7)

; restore register contents
      AR7 = pop()
      B = AR7
      return
;     Transfer to LED END

      .end
```

Proposed Code – Interrupt Vector Table "Vectors.asm"

```
; Interrupt Vector Table for Proposed Thermostat Software
; This File was copied from a Texas Instruments sample file,
;     and then altered by Michael Warner II.
;
; ~ in comments indicates continuation of comment from previous the line
;
; Only int0 and int1 are used by thermostat.
;
; ***********************************************************************
; File: VECTORS.ASM -> Vector Table for the 'C54x DSKplus           10.Jul.96
;
; ***********************************************************************
; The vectors in this table can be configured for processing external and
; internal software interrupts. The DSKplus debugger uses four interrupt
; vectors. These are RESET, TRAP2, INT2, and HPIINT.
;   *  DO NOT MODIFY THESE FOUR VECTORS IF YOU PLAN TO USE THE DEBUGGER  *
;
; All other vector locations are free to use. When programming always be sure
; the HPIINT bit is unmasked (IMR=200h) to allow the communications kernel and
; host PC interact. INT2 should normally be masked (IMR(bit 2) = 0) so that the
; DSP will not interrupt itself during a HINT. HINT is tied to INT2 externally.
;
;
;
      .width   80
      .length  55
       .title "Vector Table"
      .mmregs                 ;associate register names
                              ;~with their address

reset goto #80h     ;00; RESET  * DO NOT MODIFY IF USING DEBUGGER *
      nop
      nop
nmi   return_enable       ;04; non-maskable external interrupt
      nop
      nop
      nop
trap2  goto #88h    ;08; trap2  * DO NOT MODIFY IF USING DEBUGGER *
      nop
      nop
      .space 52*16        ;0C-3F: vectors for software interrupts 18-30

; Points to button routine
int0   goto KEYIN   ;40; external interrupt int0
      nop
      nop

; Points to RTC routine
int1   goto TIMEIN        ;44; external interrupt int1
      nop
      nop

int2   return_enable      ;48; external interrupt int2
```

```
      nop
      nop
      nop
tint   return_enable      ;4C; internal timer interrupt
      nop
      nop
      nop
brint  return_enable      ;50; BSP receive interrupt
      nop
      nop
      nop
bxint  return_enable      ;54; BSP transmit interrupt
      nop
      nop
      nop
trint return_enable       ;58; TDM receive interrupt
      nop
      nop
      nop
txint  return_enable      ;5C; TDM transmit interrupt
      nop
      nop
int3   return_enable      ;60; external interrupt int3
      nop
      nop
      nop
hpiint      dgoto #0e4h   ;64; HPIint  * DO NOT MODIFY IF USING DEBUGGER *
       nop
       nop
       .space  24*16      ;68-7F; reserved area
```

APPENDIX B

PROJECT DIAGRAMS

Figure 1: Block Diagram

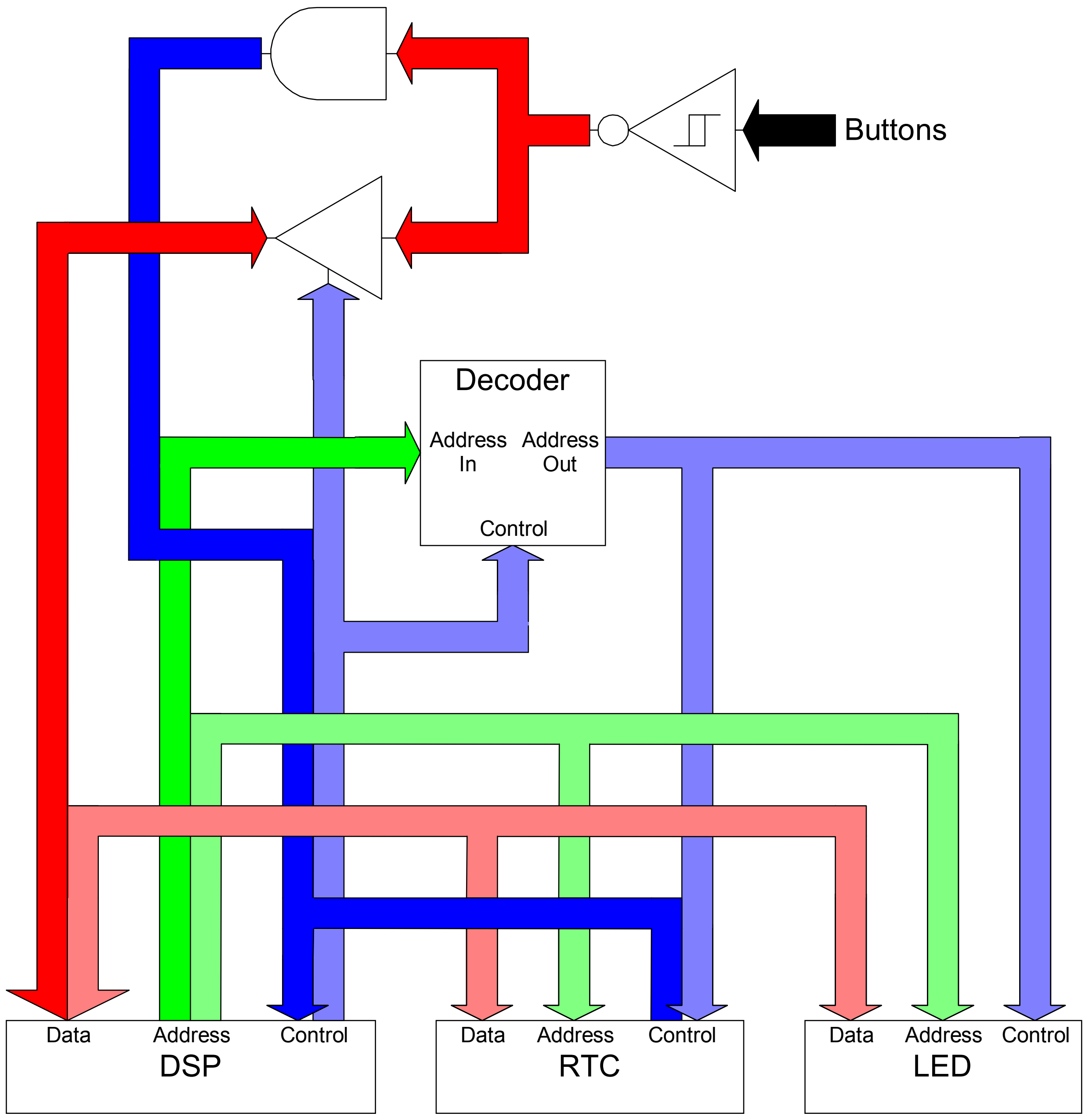


Figure 2: Logic Diagram

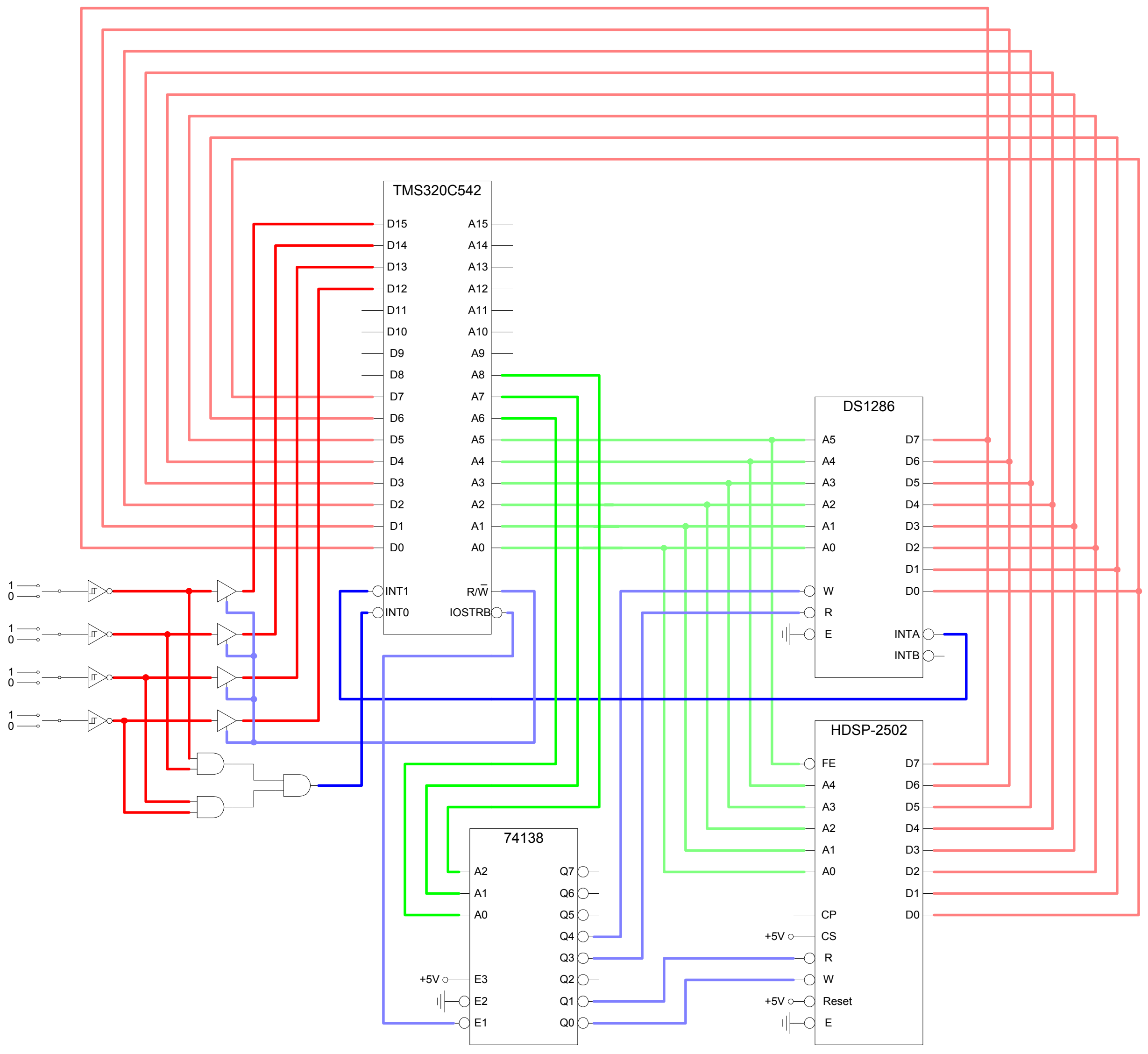
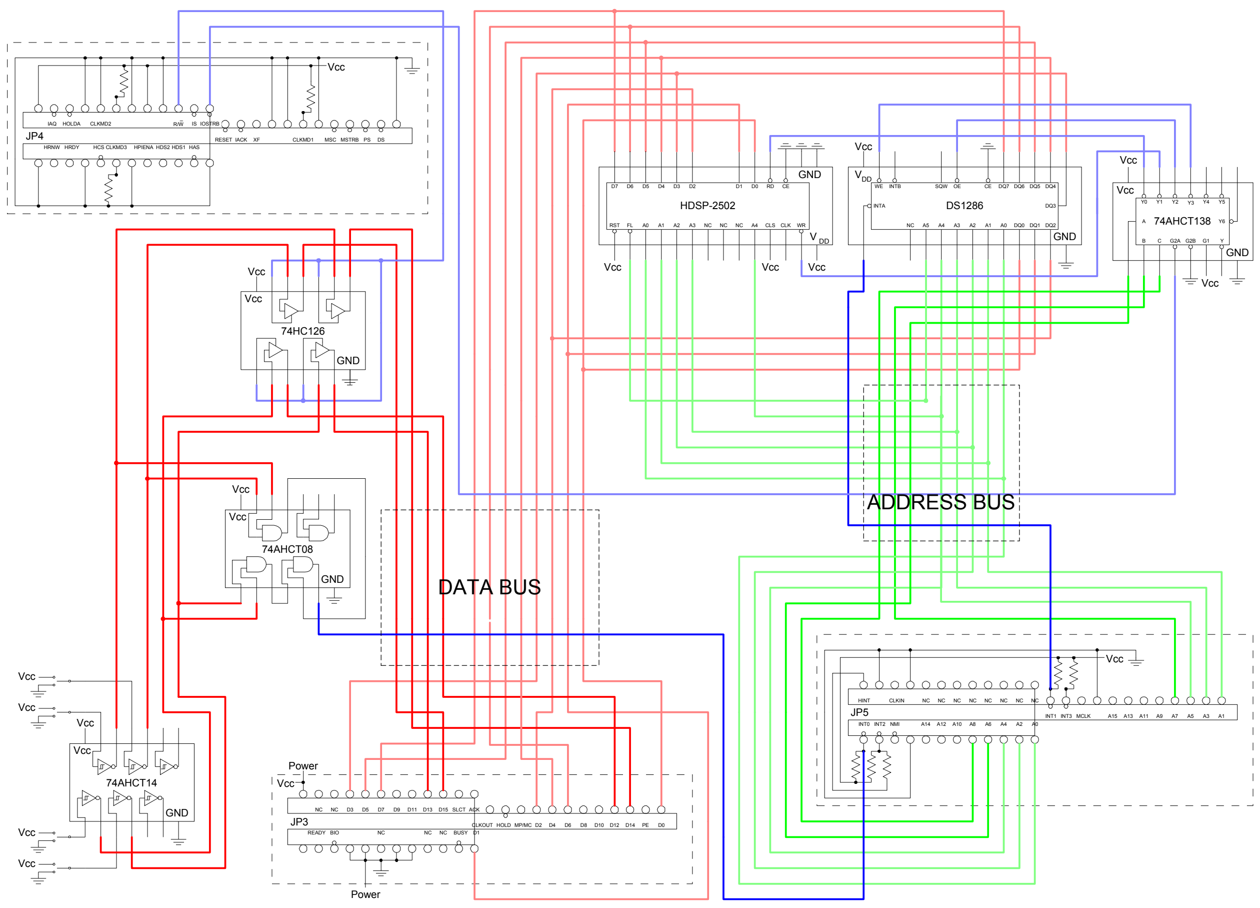


Figure 3: Pin Diagram



Figures 4,5,6,7: I/O Address Map with Data Descriptions

Figure 4: LED address

DECIMAL	OPEREATION MODE			COMPONENT ADDRESSING						HEX	DESCRIPTION	
	A8	A7	A6	A5	A4	A3	A2	A1	A0			
0	0	0	0	0	X	X	0	0	0	000	Character 1	Read from the LED's
1	0	0	0	0	X	X	0	0	1	001	Character 2	Flash RAM
2	0	0	0	0	X	X	0	1	0	002	Character 3	(Blinking of Characters Control)
3	0	0	0	0	X	X	0	1	1	003	Character 4	
4	0	0	0	0	X	X	1	0	0	004	Character 5	
5	0	0	0	0	X	X	1	0	1	005	Character 6	
6	0	0	0	0	X	X	1	1	0	006	Character 7	
7	0	0	0	0	X	X	1	1	1	007	Character 8	
32	0	0	0	1	0	0	X	X	X	020	Read from the LED's UDC Address Register (UDC RAM Pointer)	
40	0	0	0	1	0	1	0	0	0	028	Row 1	Read from the LED's
41	0	0	0	1	0	1	0	0	1	029	Row 2	UDC RAM
42	0	0	0	1	0	1	0	1	0	02A	Row 3	(Custom Charater Memory)
43	0	0	0	1	0	1	0	1	1	02B	Row 4	
44	0	0	0	1	0	1	1	0	0	02C	Row 5	
45	0	0	0	1	0	1	1	0	1	02D	Row 6	
46	0	0	0	1	0	1	1	1	0	02E	Row 7	
48	0	0	0	1	1	0	X	X	X	030	Read from the LED's Control Word Register (Commands)	
56	0	0	0	1	1	1	0	0	0	038	Character 1	Read from the LED's
57	0	0	0	1	1	1	0	0	1	039	Character 2	Character RAM
58	0	0	0	1	1	1	0	1	0	03A	Character 3	(Display Memory)
59	0	0	0	1	1	1	0	1	1	03B	Character 4	
60	0	0	0	1	1	1	1	0	0	03C	Character 5	
61	0	0	0	1	1	1	1	0	1	03D	Character 6	
62	0	0	0	1	1	1	1	1	0	03E	Character 7	
63	0	0	0	1	1	1	1	1	1	03F	Character 8	
64	0	0	1	0	X	X	0	0	0	040	Character 1	Write to the LED's
65	0	0	1	0	X	X	0	0	1	041	Character 2	Flash RAM
66	0	0	1	0	X	X	0	1	0	042	Character 3	(Blinking of Characters Control)
67	0	0	1	0	X	X	0	1	1	043	Character 4	
68	0	0	1	0	X	X	1	0	0	044	Character 5	
69	0	0	1	0	X	X	1	0	1	045	Character 6	
70	0	0	1	0	X	X	1	1	0	046	Character 7	
71	0	0	1	0	X	X	1	1	1	047	Character 8	
96	0	0	1	1	0	0	X	X	X	060	Write to the LED's UDC Address Register (UDC RAM Pointer)	
104	0	0	1	1	0	1	0	0	0	068	Row 1	Write to the LED's
105	0	0	1	1	0	1	0	0	1	069	Row 2	UDC RAM
106	0	0	1	1	0	1	0	1	0	06A	Row 3	(Custom Charater Memory)
107	0	0	1	1	0	1	0	1	1	06B	Row 4	
108	0	0	1	1	0	1	1	0	0	06C	Row 5	
109	0	0	1	1	0	1	1	0	1	06D	Row 6	
110	0	0	1	1	0	1	1	1	0	06E	Row 7	
112	0	0	1	1	1	0	X	X	X	070	Write to the LED's Control Word Register (Commands)	
120	0	0	1	1	1	1	0	0	0	078	Character 1	Write to the LED's
121	0	0	1	1	1	1	0	0	1	079	Character 2	Character RAM
122	0	0	1	1	1	1	0	1	0	07A	Character 3	(Display Memory)
123	0	0	1	1	1	1	0	1	1	07B	Character 4	
124	0	0	1	1	1	1	1	0	0	07C	Character 5	
125	0	0	1	1	1	1	1	0	1	07D	Character 6	
126	0	0	1	1	1	1	1	1	0	07E	Character 7	
127	0	0	1	1	1	1	1	1	1	07F	Character 8	

Figure 5: LED data

DESCRIPTION									
		D7	D6	D5	D4	D3	D2	D1	D0
Character 1	Read from the LED's	X	X	X	X	X	X	X	Flash
Character 2	Flash RAM								
Character 3	(Blinking of Characters Control)								
Character 4									
Character 5									
Character 6									
Character 7									
Character 8									
Read from the LED's UDC Address Register (UDC RAM Pointer)		X	X	X	X	UDC number (0 to 15)			
Row 1	Read from the LED's	X	X	X	Dot Data				
Row 2	UDC RAM								
Row 3	(Custom Charater Memory)								
Row 4									
Row 5									
Row 6									
Row 7									
Read from the LED's Control Word Register (Commands)		Clear	Start Test	Test flag	Blinking	Flash	Brightness		
Character 1	Read from the LED's	UDC/ASCII	UDC don't care or Character #			UDC # or Character #			
Character 2	Character RAM								
Character 3	(Display Memory)								
Character 4									
Character 5									
Character 6									
Character 7									
Character 8									
Character 1	Write to the LED's	X	X	X	X	X	X	X	Flash
Character 2	Flash RAM								
Character 3	(Blinking of Characters Control)								
Character 4									
Character 5									
Character 6									
Character 7									
Character 8									
Write to the LED's UDC Address Register (UDC RAM Pointer)		X	X	X	X	UDC number (0 to 15)			
Row 1	Write to the LED's	X	X	X	Dot Data				
Row 2	UDC RAM								
Row 3	(Custom Charater Memory)								
Row 4									
Row 5									
Row 6									
Row 7									
Write to the LED's Control Word Register (Commands)		Clear	Start Test	Test flag	Blinking	Flash	Brightness		
Character 1	Write to the LED's	UDC/ASCII	UDC don't care or Character #			UDC # or Character #			
Character 2	Character RAM								
Character 3	(Display Memory)								
Character 4									
Character 5									
Character 6									
Character 7									
Character 8									

Figure 6: RTC and miscellaneous address

128	0	1	0	0	0	0	0	0	0	080	0.01 Seconds	Read from the RTC's
129	0	1	0	0	0	0	0	0	1	081	Seconds	Clock, Calender,
130	0	1	0	0	0	0	0	1	0	082	Minutes	Time of Day Alarm
131	0	1	0	0	0	0	0	1	1	083	Minute Alarm	
132	0	1	0	0	0	0	1	0	0	084	Hours	
133	0	1	0	0	0	0	1	0	1	085	Hour Alarm	
134	0	1	0	0	0	0	1	1	0	086	Days	
135	0	1	0	0	0	0	1	1	1	087	Day Alarm	
136	0	1	0	0	0	1	0	0	0	088	Dates	
137	0	1	0	0	0	1	0	0	1	089	Months	
138	0	1	0	0	0	1	0	1	0	08A	Years (Two Digits)	
139	0	1	0	0	0	1	0	1	1	08B	Read from the RTC's Command Register	
140	0	1	0	0	0	1	1	0	0	08C	0.01 Seconds	Read from the RTC's
141	0	1	0	0	0	1	1	0	1	08D	Seconds	Watchdog Alarm
142	0	1	0	0	0	1	1	1	X	08E	Read from the RTC's	
144	0	1	0	0	1	X	X	X	X	090	General Use Memory	
192	0	1	1	0	0	0	0	0	0	0C0	0.01 Seconds	Write to the RTC's
193	0	1	1	0	0	0	0	0	1	0C1	Seconds	Clock, Calender,
194	0	1	1	0	0	0	0	1	0	0C2	Minutes	Time of Day Alarm
195	0	1	1	0	0	0	0	1	1	0C3	Minute Alarm	
196	0	1	1	0	0	0	1	0	0	0C4	Hours	
197	0	1	1	0	0	0	1	0	1	0C5	Hour Alarm	
198	0	1	1	0	0	0	1	1	0	0C6	Days	
199	0	1	1	0	0	0	1	1	1	0C7	Day Alarm	
200	0	1	1	0	0	1	0	0	0	0C8	Dates	
201	0	1	1	0	0	1	0	0	1	0C9	Months	
202	0	1	1	0	0	1	0	1	0	0CA	Years (Two Digits)	
203	0	1	1	0	0	1	0	1	1	0CB	Write to the RTC's Command Register	
204	0	1	1	0	0	1	1	0	0	0CC	0.01 Seconds	Write to the RTC's
205	0	1	1	0	0	1	1	0	1	0CD	Seconds	Watchdog Alarm
206	0	1	1	0	0	1	1	1	X	0CE	Write to the RTC's	
208	0	1	1	0	1	X	X	X	X	0D0	General Use Memory	
≥256	1	X	X	X	X	X	X	X	X	100	No Action	

Figure 7: RTC and miscellanious data

0.01 Seconds	Read from the RTC's	Tenths of seconds				Hundreths of seconds			
Seconds	Clock, Calender,	Forced 0	Tens of seconds			Ones of seconds			
Minutes	Time of Day Alarm	Forced 0	Tens of minutes			Ones of minutes			
Minute Alarm		Mask bit	Tens of minutes			Ones of minutes			
Hours		Forced 0	12hr/24hr	PM/20 hr	10 hr	Ones of hours			
Hour Alarm		Mask bit	12hr/24hr	PM/20 hr	10 hr	Ones of hours			
Days		Forced 0					Ones of days		
Day Alarm		Mask bit	Forced 0				Ones of days		
Dates		Forced 0		Tens of date		Ones of date			
Months		OSC Mask	SQW Mask	Forced 0	10 month	Ones of months			
Years (Two Digits)		Tens of years				Ones of years			
Read from the RTC's Command Register		Transfer	Int pin SW	Bhi/Blo	Pulse/level	WDA Mask	ToDA Mask	WDA flag	ToDA flag
0.01 Seconds	Read from the RTC's	Tenths of seconds				Hundredths of seconds			
Seconds	Watchdog Alarm	Tens of seconds				Ones of seconds			
Read from the RTC's General Use Memory									
0.01 Seconds	Write to the RTC's	Tenths of seconds				Hundreths of seconds			
Seconds	Clock, Calender,	Forced 0	Tens of seconds			Ones of seconds			
Minutes	Time of Day Alarm	Forced 0	Tens of minutes			Ones of minutes			
Minute Alarm		Mask bit	Tens of minutes			Ones of minutes			
Hours		Forced 0	12hr/24hr	PM/20 hr	10 hr	Ones of hours			
Hour Alarm		Mask bit	12hr/24hr	PM/20 hr	10 hr	Ones of hours			
Days		Forced 0					Ones of days		
Day Alarm		Mask bit	Forced 0				Ones of days		
Dates		Forced 0		Tens of date		Ones of date			
Months		OSC Mask	SQW Mask	Forced 0	10 month	Ones of months			
Years (Two Digits)		Tens of years				Ones of years			
Write to the RTC's Command Register		Transfer	Int pin SW	Bhi/Blo	Pulse/level	WDA Mask	ToDA Mask	WDA flag	ToDA flag
0.01 Seconds	Write to the RTC's	Tenths of seconds				Hundredths of seconds			
Seconds	Watchdog Alarm	Tens of seconds				Ones of seconds			
Write to the RTC's General Use Memory									
No Action									

APPENDIX C

MISCELLANEOUS DIAGRAMS

Figure 8: Routing a Strobe

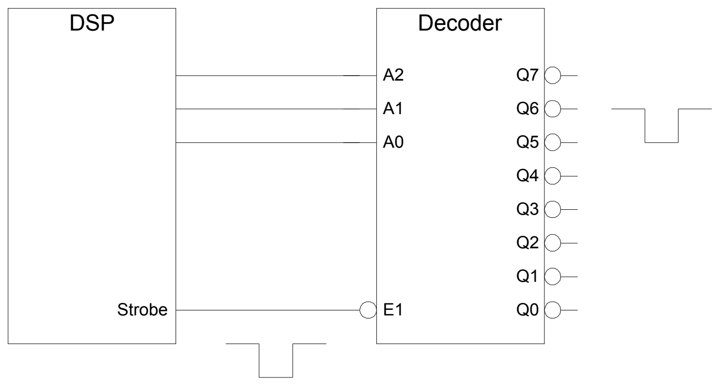


Figure 9: Read and Write Cycles of LED and RTC

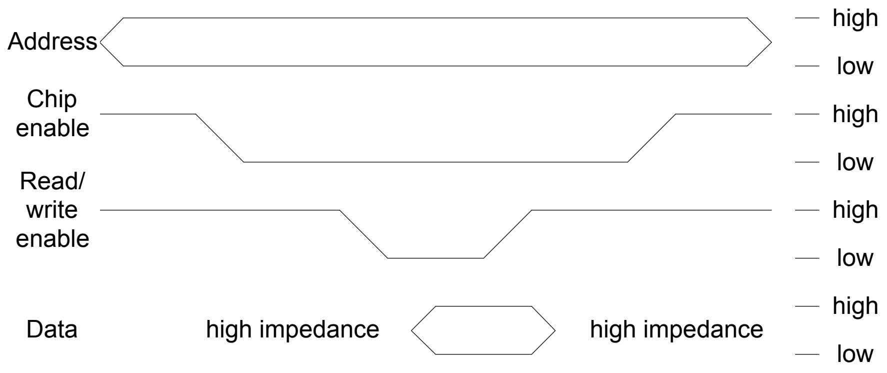


Figure 10: Read and Write Cycles of DSP without Wait

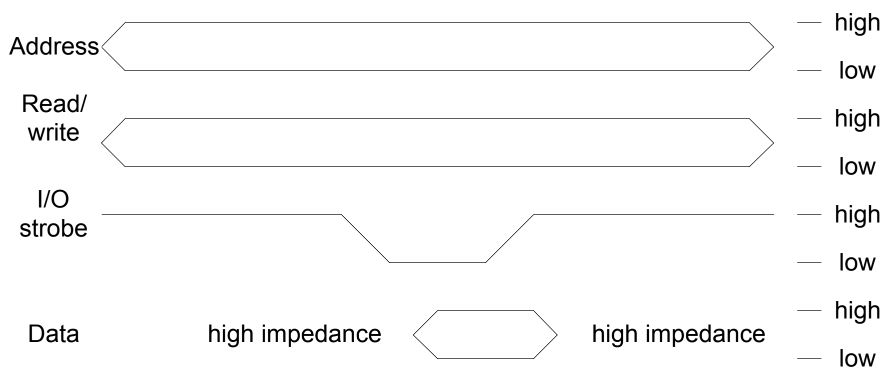


Figure 11: Read and Write Cycles of DSP with Wait

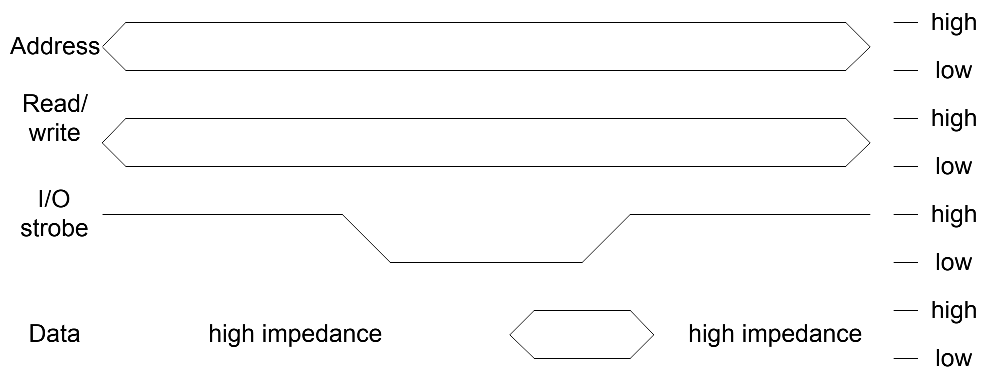
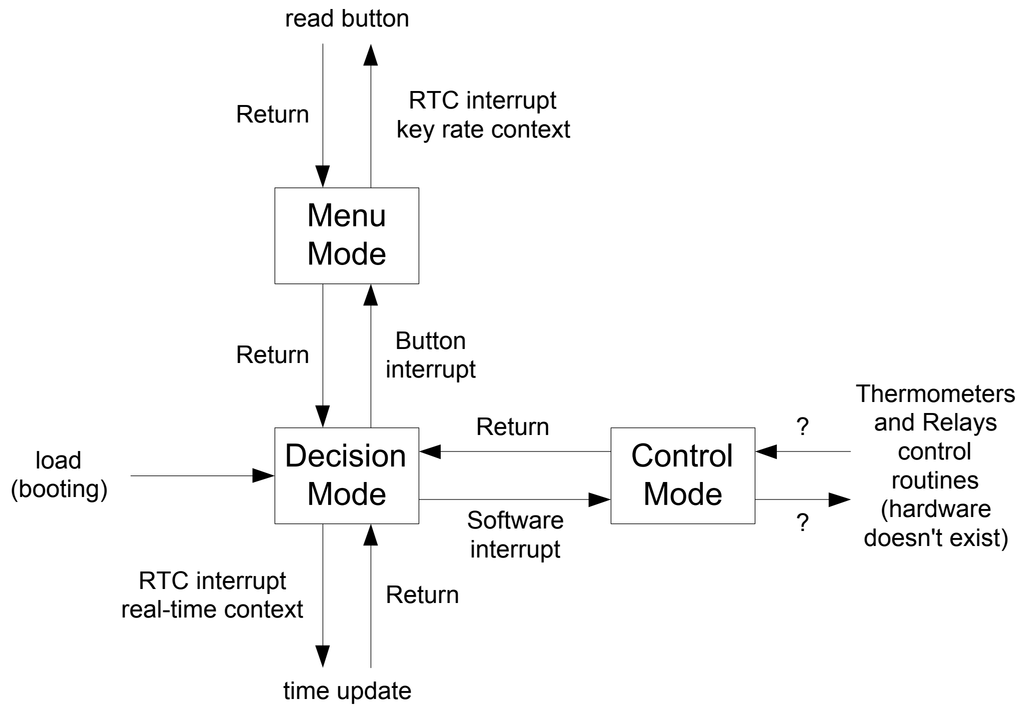
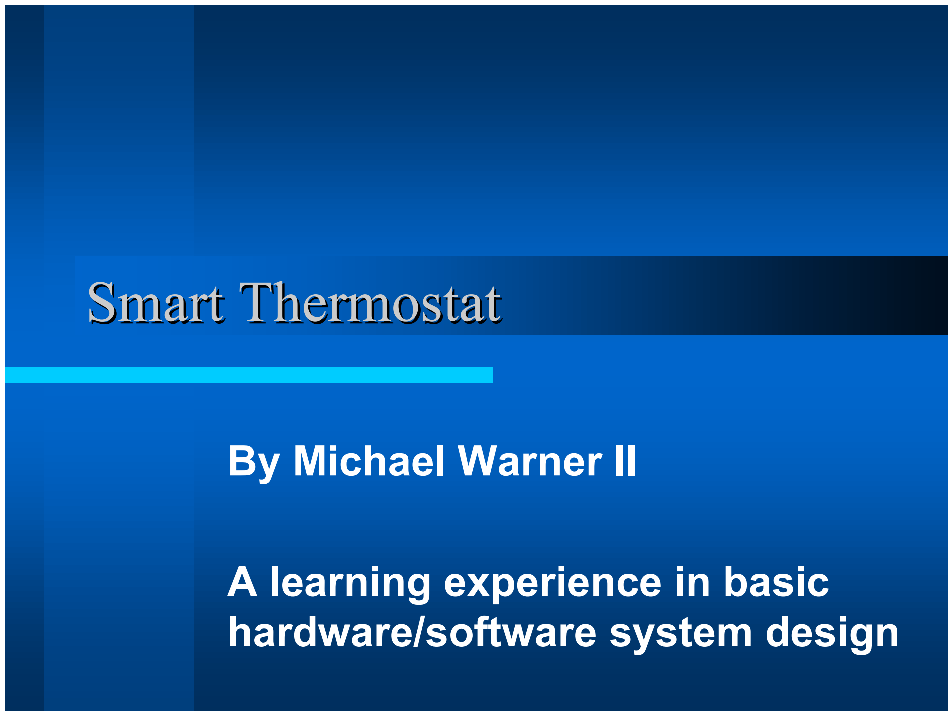


Figure 12: Hierarchy of Proposed Software



APPENDIX D

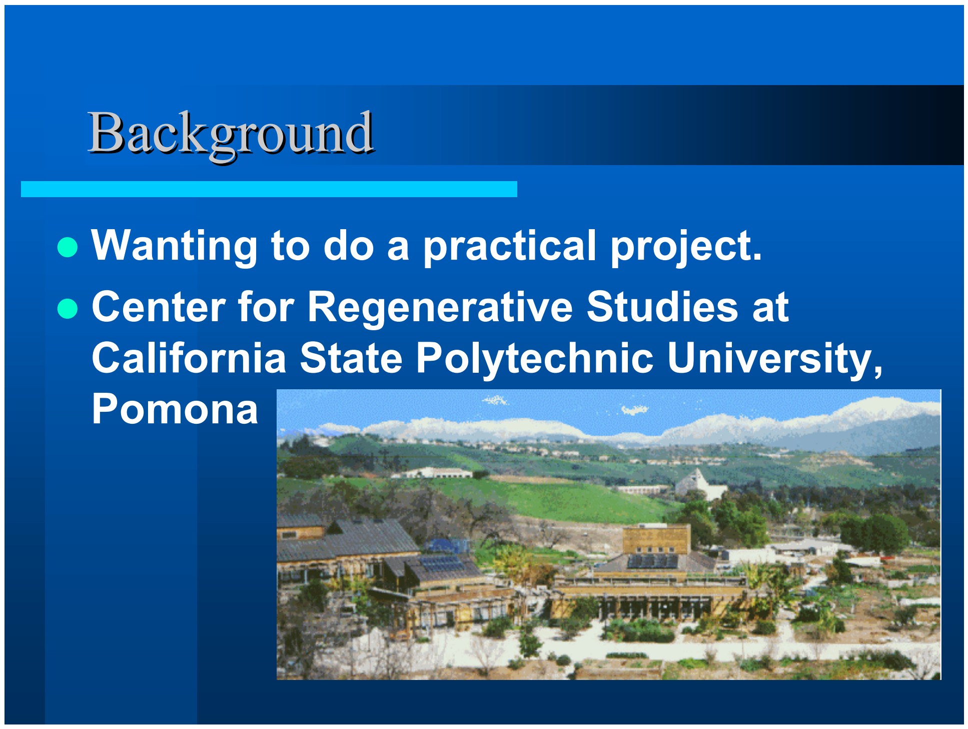
PROJECT PRESENTATION



Smart Thermostat

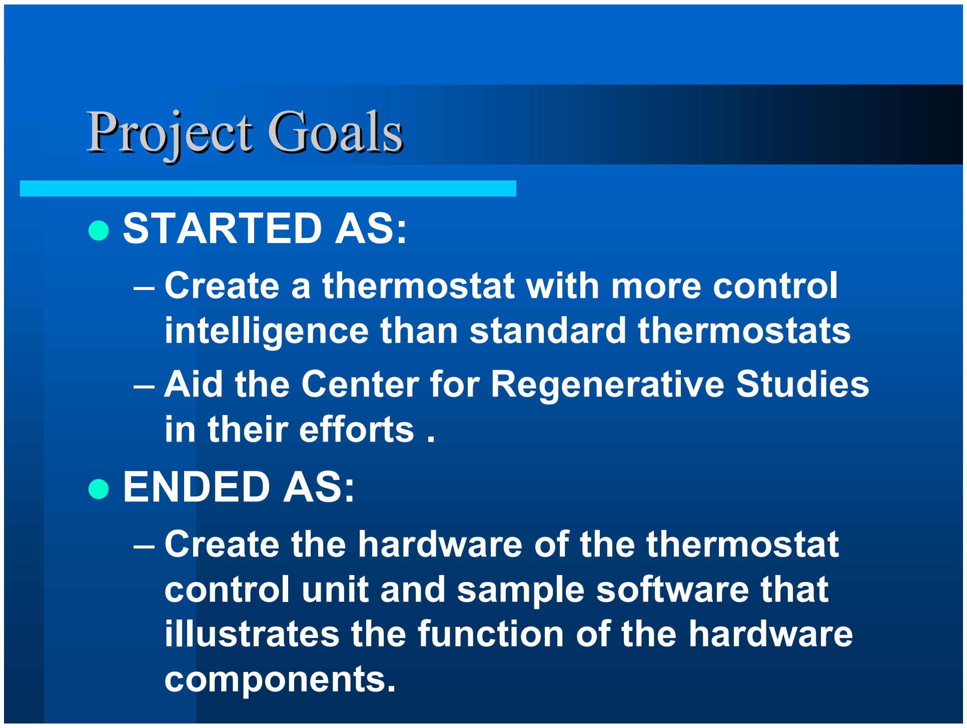
By Michael Warner II

A learning experience in basic hardware/software system design



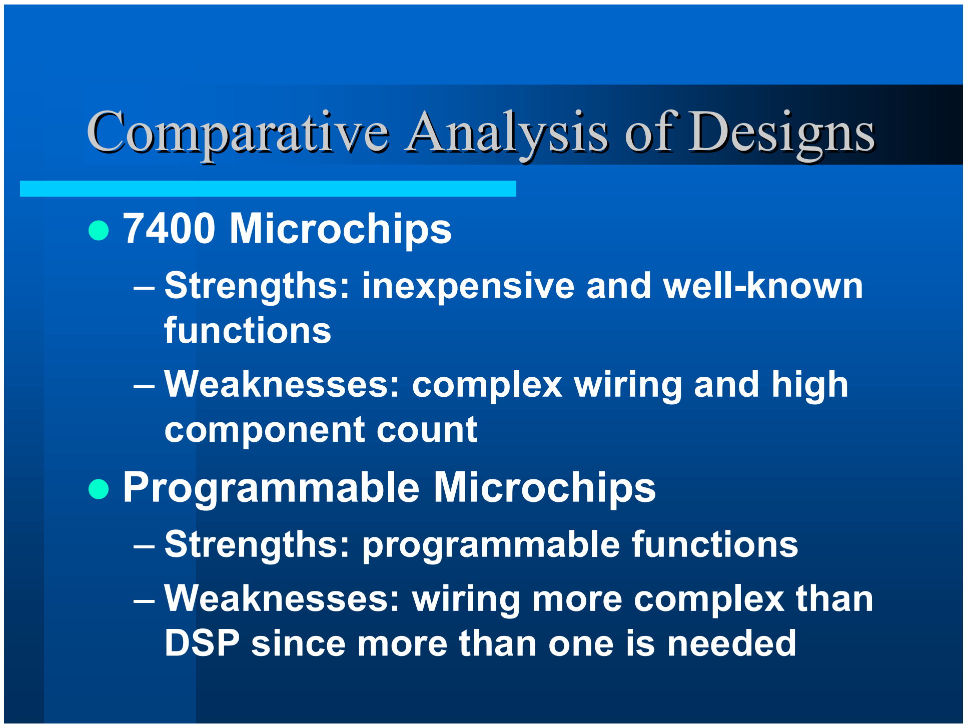
Background

- Wanting to do a practical project.
- Center for Regenerative Studies at California State Polytechnic University, Pomona



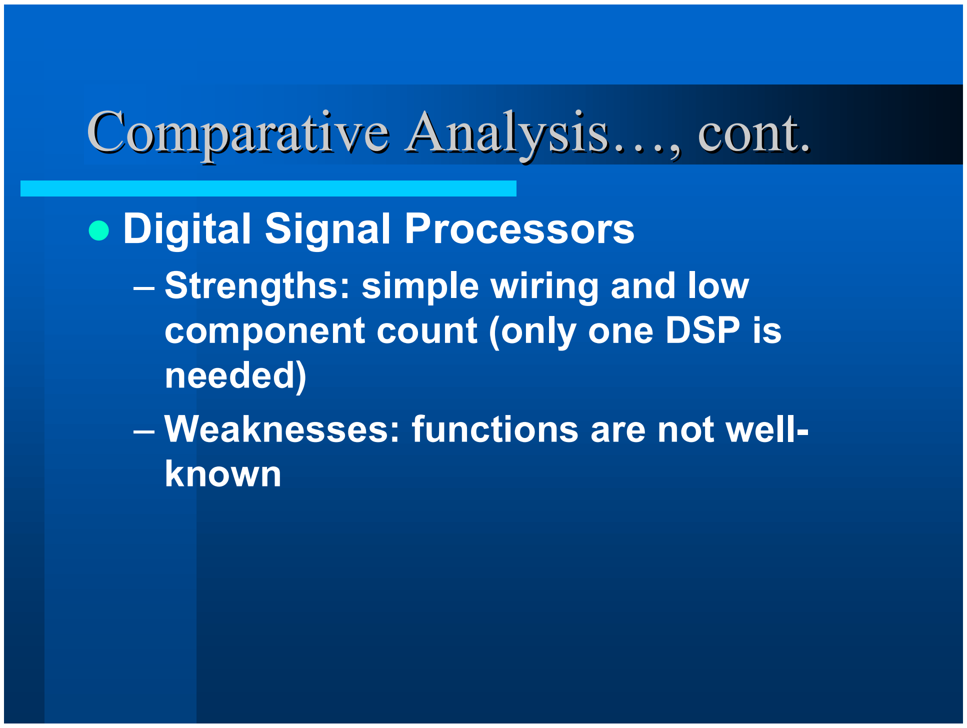
Project Goals

- STARTED AS:
  - Create a thermostat with more control intelligence than standard thermostats
  - Aid the Center for Regenerative Studies in their efforts .
- ENDED AS:
  - Create the hardware of the thermostat control unit and sample software that illustrates the function of the hardware components.



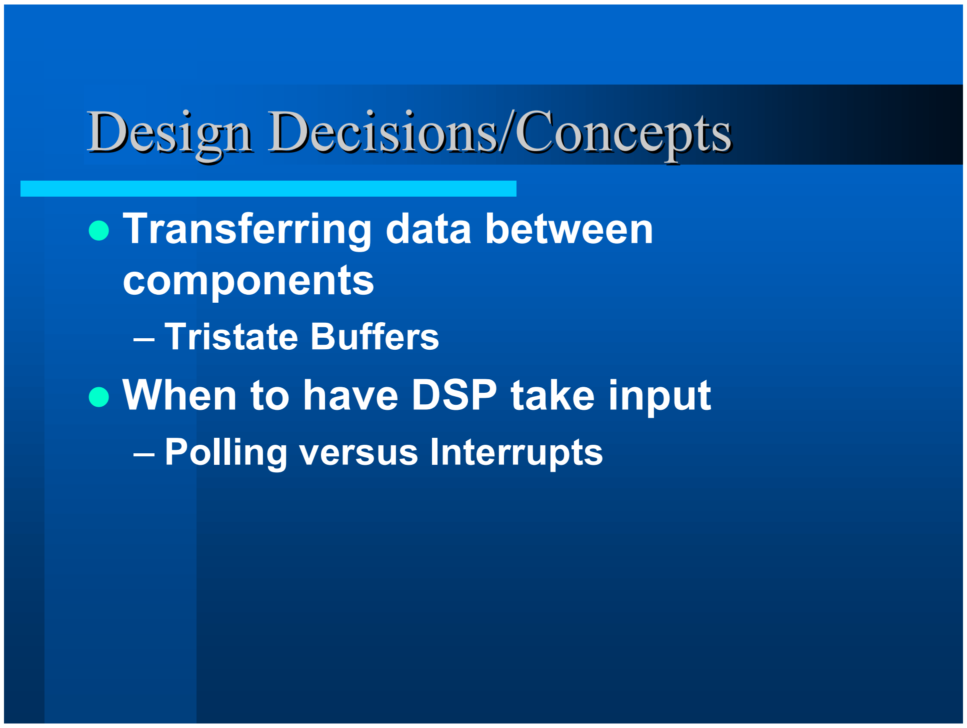
Comparative Analysis of Designs

- 7400 Microchips
  - Strengths: inexpensive and well-known functions
  - Weaknesses: complex wiring and high component count
- Programmable Microchips
  - Strengths: programmable functions
  - Weaknesses: wiring more complex than DSP since more than one is needed



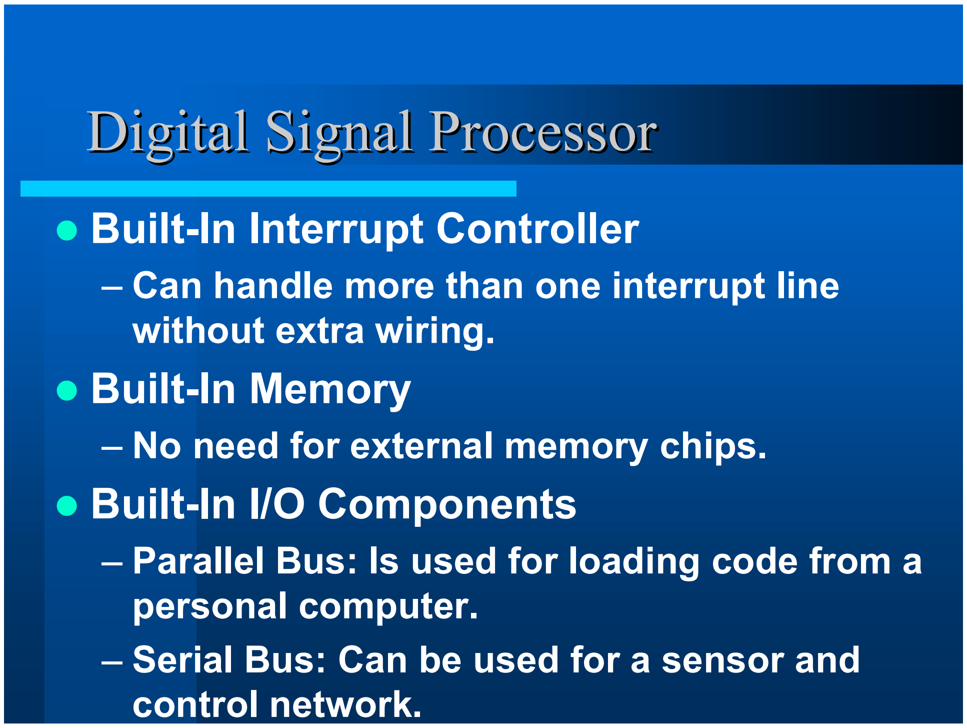
Comparative Analysis…, cont.

- Digital Signal Processors
  - Strengths: simple wiring and low component count (only one DSP is needed)
  - Weaknesses: functions are not well-known



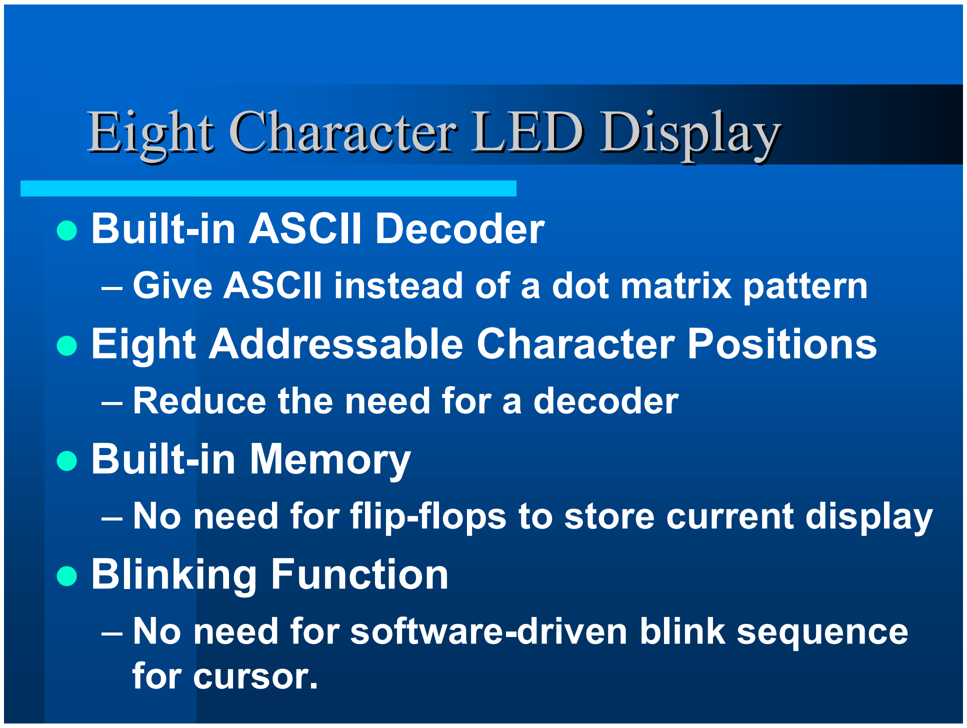
Design Decisions/Concepts

- Transferring data between components
  - Tristate Buffers
- When to have DSP take input
  - Polling versus Interrupts



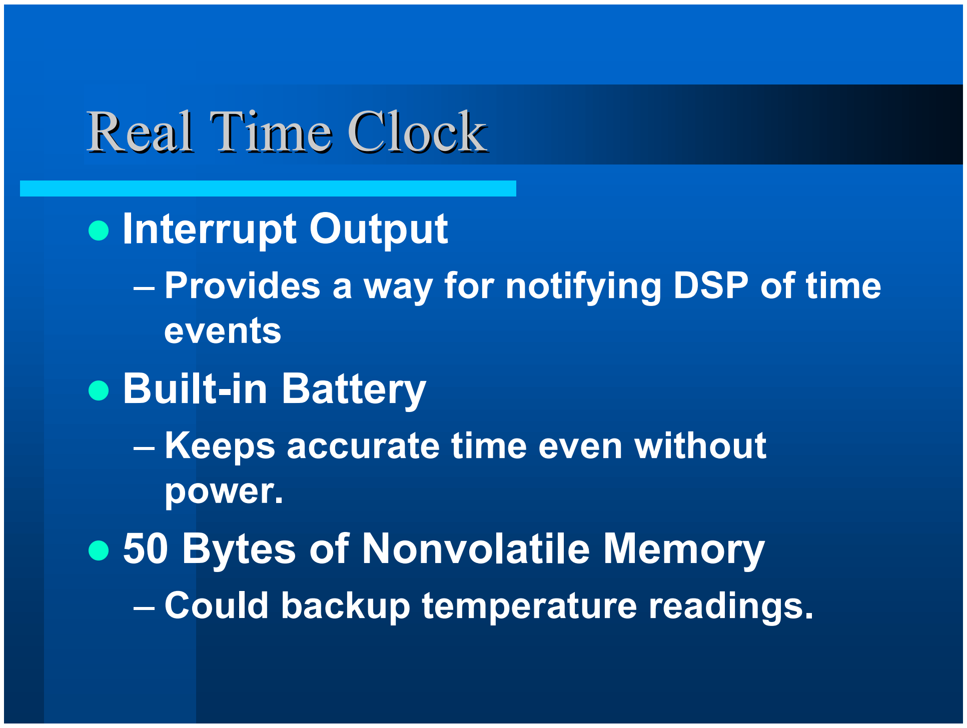
Digital Signal Processor

- Built-In Interrupt Controller
  - Can handle more than one interrupt line without extra wiring.
- Built-In Memory
  - No need for external memory chips.
- Built-In I/O Components
  - Parallel Bus: Is used for loading code from a personal computer.
  - Serial Bus: Can be used for a sensor and control network.



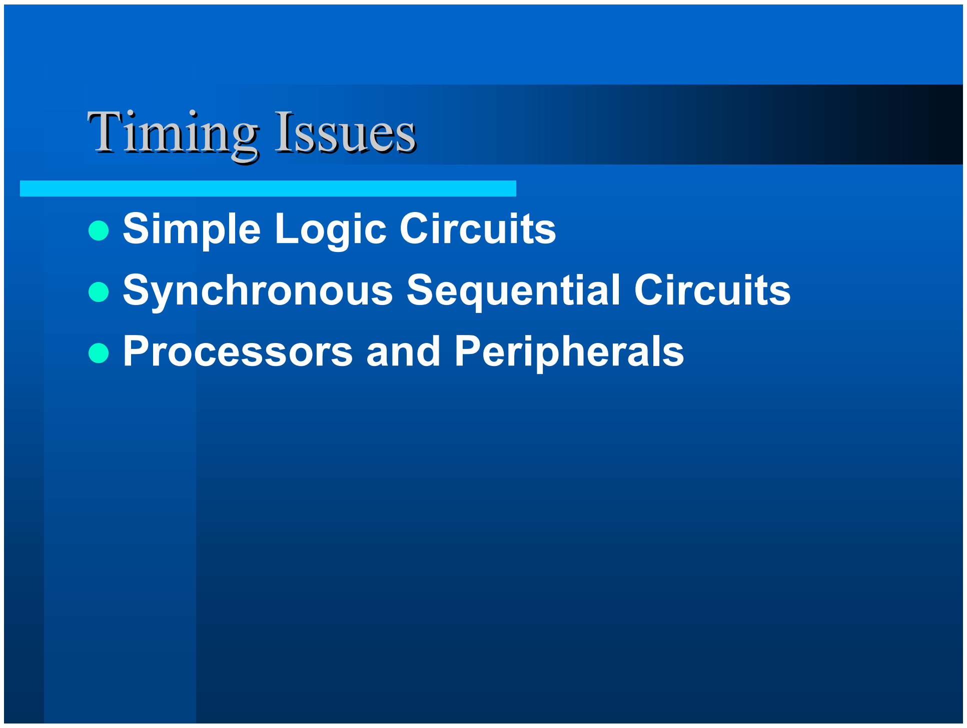
Eight Character LED Display

- Built-in ASCII Decoder
  - Give ASCII instead of a dot matrix pattern
- Eight Addressable Character Positions
  - Reduce the need for a decoder
- Built-in Memory
  - No need for flip-flops to store current display
- Blinking Function
  - No need for software-driven blink sequence for cursor.



Real Time Clock

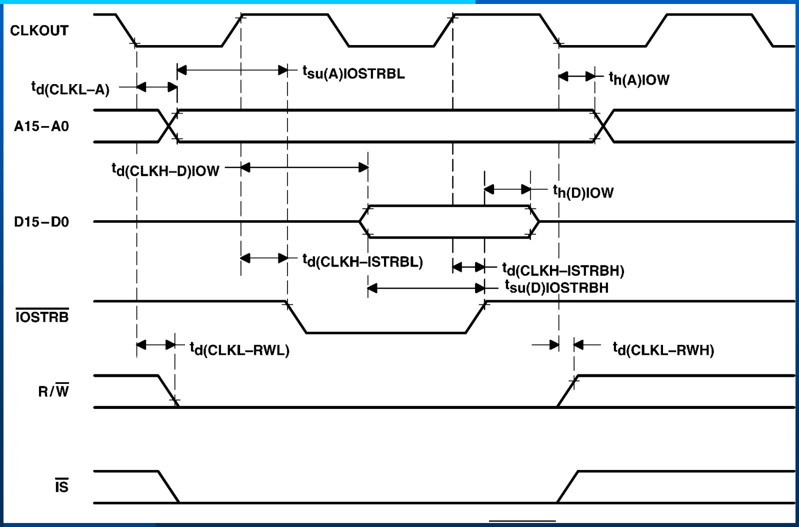
- Interrupt Output
  - Provides a way for notifying DSP of time events
- Built-in Battery
  - Keeps accurate time even without power.
- 50 Bytes of Nonvolatile Memory
  - Could backup temperature readings.



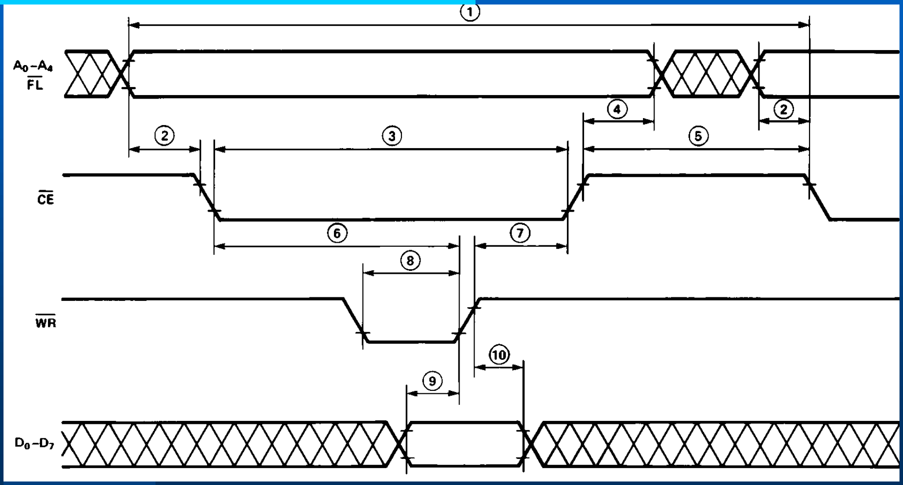
Timing Issues

- Simple Logic Circuits
- Synchronous Sequential Circuits
- Processors and Peripherals

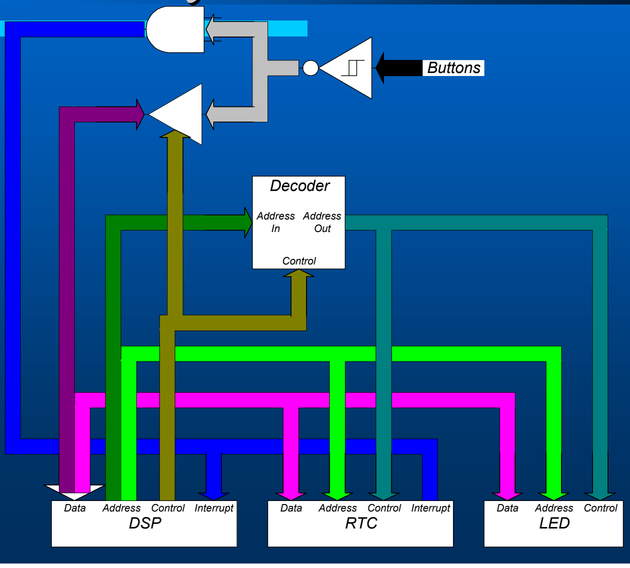
Timing Issues, cont.: DSP



Timing Issues, cont.: LED & RTC

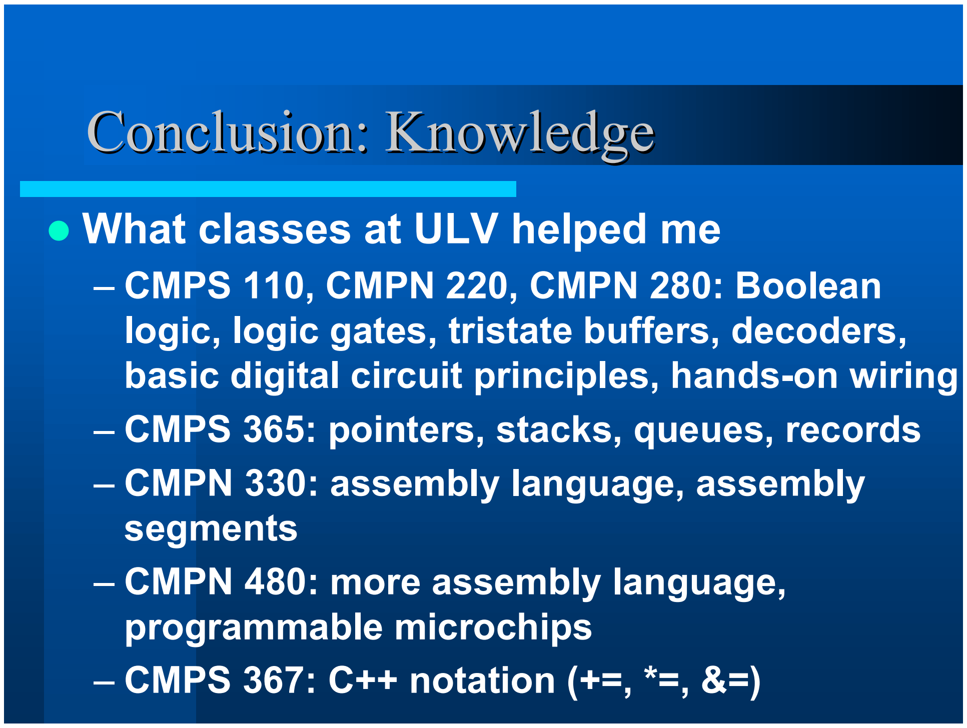


Hardware Layout



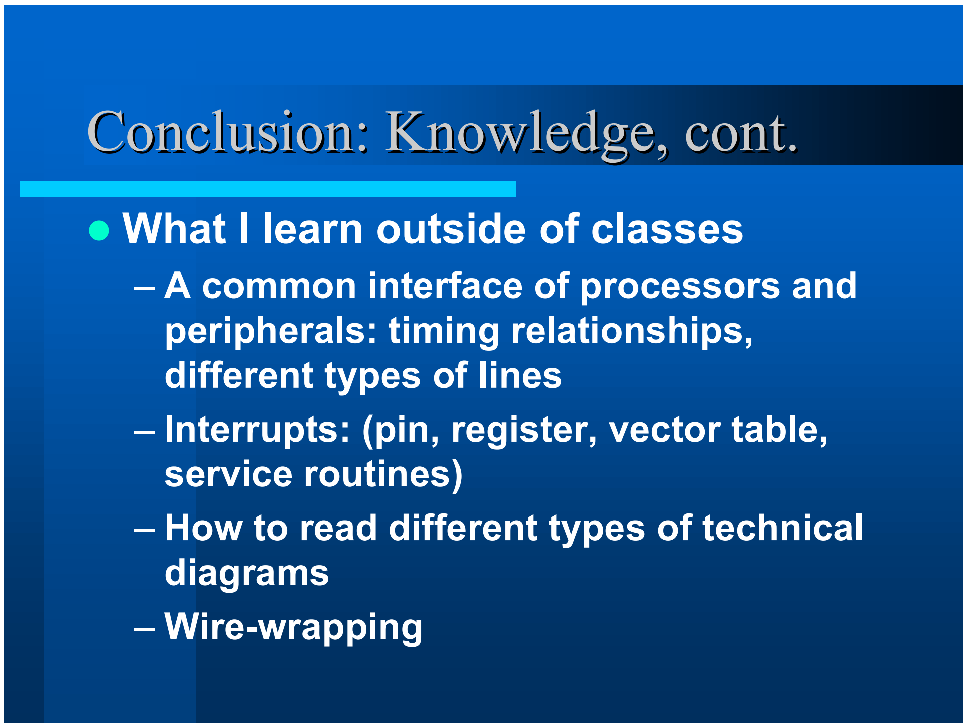
Diagnostic Software Layout

- Interrupt Vector Table
- Variables
- Initialize LED and RTC
- Infinite Loop
- Button Interrupt Service Routine
- RTC Interrupt Service Routine



Conclusion: Knowledge

- What classes at ULV helped me
  - CMPS 110, CMPN 220, CMPN 280: Boolean logic, logic gates, tristate buffers, decoders, basic digital circuit principles, hands-on wiring
  - CMPS 365: pointers, stacks, queues, records
  - CMPN 330: assembly language, assembly segments
  - CMPN 480: more assembly language, programmable microchips
  - CMPS 367: C++ notation (+=, *=, &=)



Conclusion: Knowledge, cont.

- What I learn outside of classes
  - A common interface of processors and peripherals: timing relationships, different types of lines
  - Interrupts: (pin, register, vector table, service routines)
  - How to read different types of technical diagrams
  - Wire-wrapping

WORKS CITED

Adcock, Tomothy A. "What Is Fuzzy Logic? An Overview of the Latest Control Methodology." Implementation of Fuzzy Logic, Selected Applications. San Jose, California: Adobe Acrobat, 1992.

Burrows, Richard. Interviewed September 16, 1998. Pasadena, CA.

Dallas Semiconductor. Application Note 104: Minimalist Temperature Control Demo. San Jose, California: Adobe Acrobat, 1999.

---. Ds1286 Watchdog Timekeeper. San Jose, California: Adobe Acrobat, 1998.

Deitel, H. M. and P. J. Deitel. C How to Program. Second ed. Englewood Cliffs, NJ: Prentice Hall, 1994.

Forney, James S. Dos 5 Demystified. Blue Ridge Summit, PA: TAB Books, 1991.

Hantronix. "Commands for Character Modules." Character Modules. San Jose, California: Adobe Acrobat, 1998. 45.

---. "Processor Interfacing." About Lcds (Tech Talk). San Jose, California: Adobe Acrobat, 1998. 46.

Hennessy, John L. and David A. Patterson. Computer Organization and Design: The Hardware/Software Interface. San Francisco, CA: Morgan Kaufmann Publishers, Inc., 1994.

Hewlett Packard. Eight Character 5 Mm and 7 Mm Smart Alphanumeric Displays: Technical Data. San Jose, California: Adobe Acrobat, 1998.

Intel. 8086 16-Bit Hmos Microprocessor: 8086/8086-2/8086-1. San Jose, California: Adobe Acrobat, 1990.

Kernighan, Brian W. and Dennia M. Ritchie. The C Programming Language. Prentice Hall Software Series. Second ed. Englewood Cliffs, NJ: PTR Prentice Hall, 1988.

Korthof, William. Interviewed Spring 1998. Pomona, CA.

Mano, M. Morris. Computer System Architecture. Third ed. Englewood Cliffs, NJ: Prentice-Hall, 1993.

---. Digital Design. Second ed. Englewood Cliffsw, NJ: Prentice-Hall, 1991.

Mazidi, Muhammad Ali. The 80x86 Ibm Pc & Compatible Computers, Volumes I & Ii: Assembly Language, Design and Interfacing. Ed. Janice Gillispie Mazidi. Engle Cliffs, NJ 07632: Prentice Hall, 1995.

Motorola Semiconductor. Mc146818 Technical Data. San Jose, California: Adobe Acrobat, 1988.

Rizzoni, Giorgio. Principles and Applications of Electrical Engineering. Second ed. Chicago, IL: Irwin, 1996.

Texas Instruments. Sn54ahct138, Sn74ahct138: 3-Line to 8-Line Decoders/Demultiplexers. Adobe Acrobat: San Jose, CA, 1998.

---. Sn54ahct14, Sn74ahct14 Hex Schmitt-Trigger Inverters. Adobe Acrobat: San Jose, CA, 1998.

---. "Tms320c54x Assembly Language Tools User's Guide.". 1997 ed. Owensville, Missouri: Cusom Printinng Company, 1998.

---. "Tms320c54x Dskplus User's Guide: Dsp Starter Kit.". 1996 ed. Owensville, Missouri: Custom Printinng Company, 1997.

---. Tms320c54x Dsp Reference Set: Algebraic Instruction Set. 1997 ed. Vol. 3. 4 vols. Owensville, Missouri: Cusom Printinng Company, 1997.

---. Tms320c54x Dsp Reference Set: Cpu and Peripherals. 1997 ed. Vol. 1. 4 vols. Owensville, Missouri: Cusom Printinng Company, 1997.

---. Tms320c54x Dsp Reference Set: Mnemonic Instruction Set. 1997 ed. Vol. 2. 4 vols. San Jose, California: Adobe Acrobat, 1997.

---. Tms320c54x Optimizing C Compiler User’S Guide. San Jose, California: Adobe Acrobat, 1999.