ESE 123 Digital Clock
Microcontroller and Code Description
Stony Brook University

Anthony Tricarichi

April 30, 2010

1 Preface

The purpose of this document is to give you a good understanding of the assembly code and the hardware used in our project this year. Assembly can be very intimidating but it is very important to not let it get the better of you. At its heart, you are still giving a piece of hardware a description of what you want it to do. The most challenging part of assembly is not the language but rather understanding the hardware and understanding how to take advantage of the resources you have. Throughout this document I will be doing my best to simplify the description of the hardware and to make this part of the project a little less daunting.

1.1 Notes

I will be referencing my blog throughout this article. You may access it at Http://www.microcontrollerplace.com. If you have further questions, you may email me at anthony.tricarichi@sunysb.edu

2 Computing for dummies

In case you haven't been aware already, this year in ESE 123 we will be using a programmable microcontroller to control elements in our circuit. This is a new thing for us and isn't going to be easy. I hope this document can guide you and make everything a little bit easier. By the end of this we hope you have a basic understanding of advanced circuit control elements.

2.1 Microcontrollers in a nutshell

If you're used to traditional elements in a circuit, a microcontroller can be a very weird thing. We usually are not used to elements that can change their behaviors depending on how you set it up. A resistor is always a resistor, a capacitor is a capacitor and an op amp is an op amp, no matter what! These components all have fixed functions that don't change. A microcontroller on the other hand is an element that changes. A microcontroller can be a PWM (Pulse Width Modulation) controller, digital clock, timer, comparator, mixer, digital signal processor, motor controller, led controller, toothbrush, almost anything. With a microcontoller you are only limited by your imagination. The reason we can change the function of a microcontroller is the fact we can reprogram it with different software depending on application. This is what makes a microcontroller special; the fusion of hardware and software makes a very versatile platform. So how does it work?

2.2 Nuts and Bolts

A microcontroller is simply a computer on a chip, very similar to what you have on your desk at home, but it has been squeezed down to a DIP-28 package. The major difference between your home computer and a microcontroller is speed, size and cost. A typical microcontroller costs around 2$ has a clock speed of about 8Mhz, and has only a few KB of RAM and ROM (if you don't know what this means, hold tight ill explain in a little bit!). This might be puny compared

to what you use at home, but the power of a microcontroller comes from how its used. A microcontroller is a special purpose device, it is usually prograammed to do only one thing. Your desktop computer on the other hand, can do many things. It has an operating system which controls the flow of programs and allows you to do many things at once. This ability to change to fit need is what demands resources. Our microcontroller is only going to be controlling a digital clock. At the heart of it all it has to do is count. For this role, it is very efficient and cost effective (You wouldn't want your new desktop dedicated to only counting numbers (except for research physicists)). Now that we have a basic understanding of what a microcontroller is, lets find out how it works.

2.3 Computation Theory

What is a computer? At its most basic definition a computer is anything that can do numerical calculations and as we learn further we find that the most basic building block of computer arithmetic is the XOR gate and the half adder (Http://mcuplace.com/mcu/blog4.php/2008/08/21/truth-tables-never-lie). However our microcontroller isn't that simple; it is comprised of many logic gates that form discrete components inside the chip's die (the piece of silicon in a chip that does all the work). What makes our microcontroller more of a computer than say a half adder is the fact that it is software programmable and for that we need three basic components: a CPU, RAM and ROM.

2.3.1 The CPU

The Central Processing Unit is the most essential part of any computer. This is the component that actually carries out the work. A CPU takes two things in: an instruction and data. The instruction tells the CPU what to do with the data and where to put it when its done. Thus all a processor does is take in data, manipulate it based on some instruction and outputs it wherever the user wants it to be. Depending on the processor, a CPU will either take one instruction per clock cycle or take several clock cycles to complete one instruction. In either case, a (simple) processor's performance is measures in MIPS(Millions of instructions per second). Instructions are usually very simple and in most cases they consist of tasks such as:

- Basic Arithmetic (Add Subtract)
- Logic function (AND OR XOR etc..)
- Boolean (Is true Is false)

Despite the simplicity of most operators, the fact a CPU can do these very fast is what makes your computer so powerful (Again, speed is calculated in MILLIONS of instructions a second)

2.4 The RAM

Random Access Memory is memory on a chip which is very fast and which the processor has direct access to. Because this is the memory that the processor has direct access to, we hold the data we end up processing in the RAM. RAM is fast, but not permanent, so when you shut off the microcontroller this RAM

gets erased, so it is only temporary(This memory needs a constant source of charge to operate). RAM is divided up by memory addresses, and holds a value equal to the bus size of the microcontroller (8-bits for this class). You can think of memory as post office boxes in which each one has a number and holds one value. How the RAM is organized is the main thing that makes one microcontroller different from another. So we will be spending quite some time on this in later sections. As we mentioned above, each memory location has an address and a value. When you give the CPU an instruction that points to a memory address, the CPU can read and/or modify that value and write back to that address. In short all the RAM does is hold the values you are directly working with in your program for the CPU to process them. However, this is not so simple as we will see later. There is a lot to be said about RAM, but because it can be so specific to architecture there is little to say about it generally.

2.5 The ROM

Read Only Memory is the permanent storage on your microcontroller, when you upload your program, this is where it resides. Also on most microcontrollers you have EEPROM (Electronically Erasable Programmable ROM) in which you can store values for later use. This memory is NON-Volatile and will not be erased when the microcontroller shuts off.
Now that we have the basics down, lets break down the hardware we are using

3 Hardware: The ATtiny 48

For this project we are using the 8-bit ATtiny 48 μC(Microcontroller). 8-bit refers to the primary data bus width and memory address width, this means we can store in a single register(memory address) a single 8-bit number (0-256). This seems limiting but part of the experience is learning how to work with your system and to be creative with what you have.

3.1 Specs

- 28 Pin DIP package
  Max 16MHz clock for 16MIPS
- 32 GPR(General Purpose Registers)
- 4K Program memory
- 64 Bytes EEPROM
- 256 Bytes SRAM
- Peripherals
  * Internal and External Interrupts
    One 16-bit and one 8-bit timer
  * 6-channel 10-bit Analog to digital module

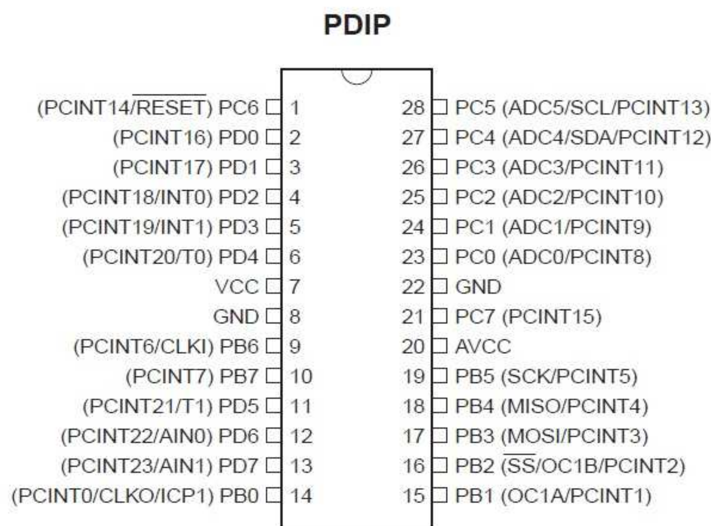
* SPI Serial Interface
* On-chip Comparator
* Programmable Watchdog timer

- Power consumption at 1Mhz, 1.8V 240μA

So what can we do with all those toys? Ultimately you will determine what features you use from your application. For our digital clock we will be taking advantage of one of the timer modules. However, for the most part we are just going to be using the pins for I/O and controlling elements in our circuit.

3.2 The Architecture

It is very important to note how everything works inside this particular microcontroller. The main thing that makes an AVR(This brand of microcontroller) different from a PIC(Another brand) and a X86 processor(whats in your computer) is how everything in the processor interface together. So lets start with the physical chip.



So here is the pin out for our ATtiny microcontroller. It might look a little daunting but its actually the best pin out you'll ever see. The important thing to know about a microcontroller is that it is programmable, so the majority of the pins are Digital Input/Outputs, which means we can program them to be either an Input(take data in) or an Output(Output data). When a pin is set to Input data, you read that port to see if its a 1(VCC Voltage/High Voltage) or a 0(Ground Voltage). When you set a port/pin to output, you can set the voltage on the pin to a 1 or a 0. With this in mind, when we look at the pin out; we see three things:

1. VCC and VEE[1] (High voltage and ground)

[1](Terminology comes from BJT Transistors where you apply HIGH voltage to the COLLECTOR and LOW voltage to the EMMITTER
Also VDD and VSS are the same thing where we use terminology from MOSFET transistors where the DRAIN is high voltage and the SOURCE is low voltage

2. GPIO (General Purpose Input/Outputs)

3. Preferable Properties associated to the GPIO pins

So for this chip we have:

Pin 7	VCC
Pin 8	VEE/Ground
Pin 22	VEE/Ground
Pin 20	AVCC [2]
Pins 14:19,9:10	Port B
Pins 23:28,1& 21	Port C
Pins 3:6, 11:13	Port D

So we have our power pins and 3 8-bit bi-directional ports. This totals to 28 pins which is the number of pins of the device. (isn't that nice? no NC pins!) What each pin does is determined by you, the programmer! The only pins you really need to worry about is the power pins. Those are always static and not programmable.

3.2.1 The Peripherals

So unfortunately it isn't that simple, although every I/O is an I/O, certain pins have alternate functions which we set when we are writing the code, in the pin out it lists next to each GPIO what special functions that pin also does. Unless you plan on using certain peripheral functions of the microcontroller you can just regard the pins as an GPIO and ignore the peripheral features (don't forget to turn them off in code!) When we need a peripheral its good to know which pins you will end up using and plan ahead in your application. This is the hardest part of using microcontrollers: determining where your resources go.

3.3 The Guts

Just like every computer has a CPU, RAM and ROM so does our ATtiny, so lets start discussing how our chip ticks. One thing to note, even though there are many microcontrollers in the ATtiny series of microcontrollers, they all use the same base architecture, so what you learn here applies to many other AVRs (the series of microcontrollers that the ATtiny is in) and even some other chips.

3.3.1 Our little AVR CPU

There is often very little to note about the CPU, its usually pretty simple, for the most part all you really care about is, how fast it is and what it can do. In our case, the core of this chip does 1MIPS (million Instructions Per Second) per MHZ clock. That simply means that on every clock cycle, (transition from High to Low) we perform one basic operation. It is very important to understand what this means. It takes our 2$ microcontroller $1\mu S$ (millionth of a second) to process a piece of data, this is pretty insane. Our code, which is only a few hundred instructions long will cycle a few thousand times a second if we don't have any stops or delays in it. So despite our microcontroller not being as fast as desktop PC, it is still pretty dang fast! (what does that say about the efficiency of your operating system and computer ;))Because we know how fast the microcontroller operates and how long an instruction takes we have a timescale of what our

microcontroller is doing with respect to time, and how to relate what our code does to the real world. So we've described how fast our CPU is, but what can it do? One of the most important things to note, is this is a RISC processor (Reduced Instruction Set Computer). All that means, is that every instruction we give the processor is going to be a simple function (Add Subtract, shift, etc.) RISC design focuses on fast processing of simple instructions vs CISC which has very complex instructions(say divide two floating point numbers) but generally are processed slower. Another thing to note is the instruction length, this tells us how many instructions we have and how complicated they are (larger the instruction width the more complex the instruction) We have a 16-bit instruction length and 123 instructions. Majority of those 123 operations we can complete in one clock period. Despite our instructions being simple, we can still do quite a lot with them. This is what makes our microcontroller powerful, very fast simple instructions.

3.3.2 The ROM

There isn't much to say here, our microcontroller uses flash memory to store the programs we write onto the device. For our chip, we have a capacity of 4KB and we can program the device about 10,000 times before we start getting in trouble. We also have 64 Bytes of EEPROM which we can use to permanently store data we would like to record.

3.3.3 The RAM

Alright, its the CPU that does all the work, but the RAM defines the chip. What varies most from chip to chip is how the RAM is arranged and how you access it. Lets square some basic concepts away, In most microcontrollers we have two types of RAM, RAM you can access and can change and RAM you cannot. The RAM you can change are our General Purpose Registers, this is where we hold code specific values that we need to operate on (Our data). These act as our variables in our program and allow us to manipulate data during runtime. We use these general purpose registers to hold values such as time(hours,minutes,seconds), various numbers, temperature, etc. Any value we wish to manipulate and keep track of we manipulate in RAM and in our general purpose register. Each one of our GPRs has an address and value associated with it. From now on we will be using hexadecimal[3] numbers to represent values. If you are not familiar with hexadecimal I recommend you check out my blog post on alternate number systems.

Address (In Hex)	Value (In Hex)	Variable name
0x00	0x0A	Hours
0x01	0x05	Minutes
0x02	0x14	Seconds

Here we have 3 8-bit memory addresses, 0,1 and 2. In each address we hold a value which correlates to Hours, Minutes and Seconds. When we analyze this data we see in those three registers we hold time time; 10:05, and 20 seconds. A

[3]Hexadecimal is a base-16 number system, we denote hex numbers by the prefix 0x or the suffix h
Ex. 0x01, 01h

very important thing is to understand how data in your registers doesn't mean much until you go through the effort of analyzing it. Usually this step in writing your code comes as interpreting whats stored at the address and converting it into the signals to drive a display. But when data is stored in the form, it makes it very easy for us to manipulate it. When we want to manipulate the seconds (say increment by one) we goto address 0x02 and manipulate that data held at that address. This is the RAM you will be interfacing the most with since this is where your values are held
Now, there is RAM you don't have complete control over. We call these Special Function Registers. We know how the microcontroller has many peripherals that can do many things. We have timers, we have analog to digital converters and many other modules. To control these modules the microcontroller sets aside certain RAM address for the control bits of these modules. A control bit is merely a 1 or 0 that defines an operating condition for the module. Some control bits are simple such as turning the module on and off or complicated like running an A/D routine. In either case, all you ever do with these bits is set them for what you want the module to do. This is typically done on startup in a startup routine, in which you go to the Special Function Registers of the modules that you would like to use and you configure them for how you would like to use them. A typical startup routine would look like:

1. Set all ports digital
2. Set all ports output
3. Enable timer one
4. goto main routine (Setup is done)

In this routine all you are doing is going to the special function register of interest and configuring it with the right word (8-bit argument). So lets look at how this is done in our microcontroller. We want to set port c to outputs. The first thing we do is look at the data sheet. For our microcontroller the data sheet is located at: (http://www.atmel.com/dyn/resources/prod_documents/doc8008.pdf)
For our ATtiny, the DDRC register controls PORTC I/O state, when a logic 1 is written to DDRC, the pin is output. When a 0 is written the pin is input. To set PORTC as output we do the following:

1. Load 0xFF to scratch register
2. Move scratch register to DDRC

After that PORTC is now configured for output. This might seem a little confusing, but it will all make more sense in context of the software. All you need to know now is that Special Function Registers are registers that are DEFINED (in hardware) registers that control operation features of the microcontroller.

4 The Software

As I stated earlier, what makes a microcontroller special is the ability to use software to control hardware. We went over the basics of the hardware so lets

begin discussing the software. It is the software which makes a microcontroller useful, without software our chip will do nothing. The software is what we write which defines our application. We write our software around our hardware to perform a specific function. Because software can be re-written, the possibilities are endless for a single microcontroller. No one microcontroller is bound to one application and changing software can breath new life into hardware(Might want you to reconsider throwing out that old cellphone, you might be able to turn it into an RF spectrum analyzer) Lets go over some general design philosophy before we jump into the ESE 123 Digital Clock code.

4.1 Some clarification

It can be very helpful to define some things before we actual jump into our code. Its always good to outline your design goals as well as how you plan on going about them. Lets begin with why we chose to use assembler language over a higher level language like c.

4.1.1 Assembler Language

Assembly is like no other language you will ever program with, despite that it is the base of all others you will eventually program with. Assembly is the language that your hardware can DIRECTLY interpret. All other language use a complier (You code gets translated to assembly) so what you write isn't actually what the processor does. Assembly gets assembled, (hah) in which the tags you used to write the code gets converted into op-code, the op-code gets strung together to form a string of hexadecimal numbers and this gets written into the program memory, word for word. All the assembler did was take the pretty English tags you had for your instructions and register names and converted them into numbers. If you really wanted to you could write your code as opcode and directly uploaded it to the microcontroller (This is the basis for device hacking). So there is one major advantage and one major disadvantage. The disadvantage is that assembly requires a vastly different mindset to program with it. The advantage is the fact assembly requires a completely different mindset to program with it. Assembly isn't easy, but its incredibly powerful. The power comes in the control you have over your code and the speed at which it is executed. With assembly you know exactly what the processor is doing at all times and it gives you a peek into the mechanics of your system. With a language like c, you have no clue what the processor is doing at any given time or even where your registers are. With that said, assembly gives you an incredibly intuitive picture of a very complex piece of hardware. This is why I prefer it over higher level languages like c, the control it offers you and the speed it offers.

4.1.2 Using C

Now, even though we are using assembler, it is important to know when to use C and higher level languages. As you begin to design for more sophisticated applications it will be necessary to learn and implement C. The biggest advantage of using C is that the complier keeps track of your data registers and you can

easily implement higher-order data types like floating point, string and whatnot. With assembler, you are limited to each register containing 256 states(You may string registers together and implement your own data types however). For complicated applications such as sophisticated devices which demand controlling many sensors, performing many calculations and driving complicated displays, C sometimes is just easier and faster(To implement). So when you need to get a complicated design out the door quickly, c is your friend.

4.1.3 Algorithm Design

No matter what language you decide to use, what you end up doing is the same. You design an algorithm to break down what you would like to do into small pieces that all come together to make your finished product. An algorithm is a step by step description of what you would like to do. All software begins as a simple algorithm, which increases in complexity until you have the finished code written in whatever language you choose.

4.1.4 Designing an Algorithm

We know that an algorithm is description of what we would like to do, but defining that isn't always easy. The most important thing when figuring out what you want to do is to define your goals, i.e. what would you like to accomplish? Once you define your goal, you have to realize how you will accomplish that goal and you can only do that once to realize the tools you have to work with. So simply put, algorithm design is about defining goals and then figuring out how to do them with the tools you have access to. So lets look at how we will go about defining what our clock does.

5 Our Algorithm

We know very well what our goal is, we would like to build a clock, and we know what are tools are, they are the ATtiny48 microcontroller. We must understand the basic description of a clock before we can understand how to utilize our tools. A clock is just merely a device that responds to some periodic event. So for a clock to work, we need some periodic (and accurate) event we can keep track of. Now, our microcontroller has an internal clock, which oscillates at 8Mhz, we could use this but that means we would have to keep track of 8 million instructions before we would get any useful increment of time (The second). We happen to be also building a USB power supply, which runs off of 60hz AC. That means we would only have to keep track of 60 ticks before we got the second we need, this is much more reasonable than 8 million (and much more accurate). Now that we have the basic component of a clock we have to think about what we want to do with this periodic signal we have. Well were building an alarm clock so we should want to display this time some how. We have LED on our board so on every second we want to turn them on and off based on our signal, we can easily do this by setting the appropriate value on the output port to light an LED. So we have a periodic signal, a way to display it and all we have left is the alarm.

5.1 The Alarm

This is the first section of the algorithm where we have branching. This is different than the previous sections where everything was procedural. That means we got the time and then we displayed it, this procedure happens in that order all the time until our device breaks. Here there are several independent sections of code that deal with the alarm and perform different things for it. We have to configure the alarm to what time we want it to go off at and we want the alarm to go off at the specific value. The event that triggers the branching is a button press as well as when the stored time we have for the alarm matches the current time. We can express these actions with an IF clause. Lets start with the most intuitive example, a button press.

$$\begin{aligned} IF \quad ButtonPress \rightarrow ButtonRoutine \\ Else \rightarrow Continue \end{aligned} \tag{1}$$

So depending on which action happens (the button is pressed or it isn't) determines where in the code we go next. So IF we have a button press (the user wants to set the alarm) we go to a routine which deals with setting the alarm. IF the button is NOT pressed we just continue on with the code.

6 The Pseudo Code

The Pseudo Code is an English description of how we will go about implementing out algorithm in terms we can eventually code in. Everything in the pseudo code has to be implementable in our software. So the Pseudo Code for our algorithm is.

1. Wait for the first tick (One second)

2. Display the time

3. Check the alarm (Is it set?)
   - YES: Alarm goes off

     NO: Alarm remains silent

4. Did the user press the button?
   - YES: Go to the routine which sets the alarm

     NO: Continue with the code

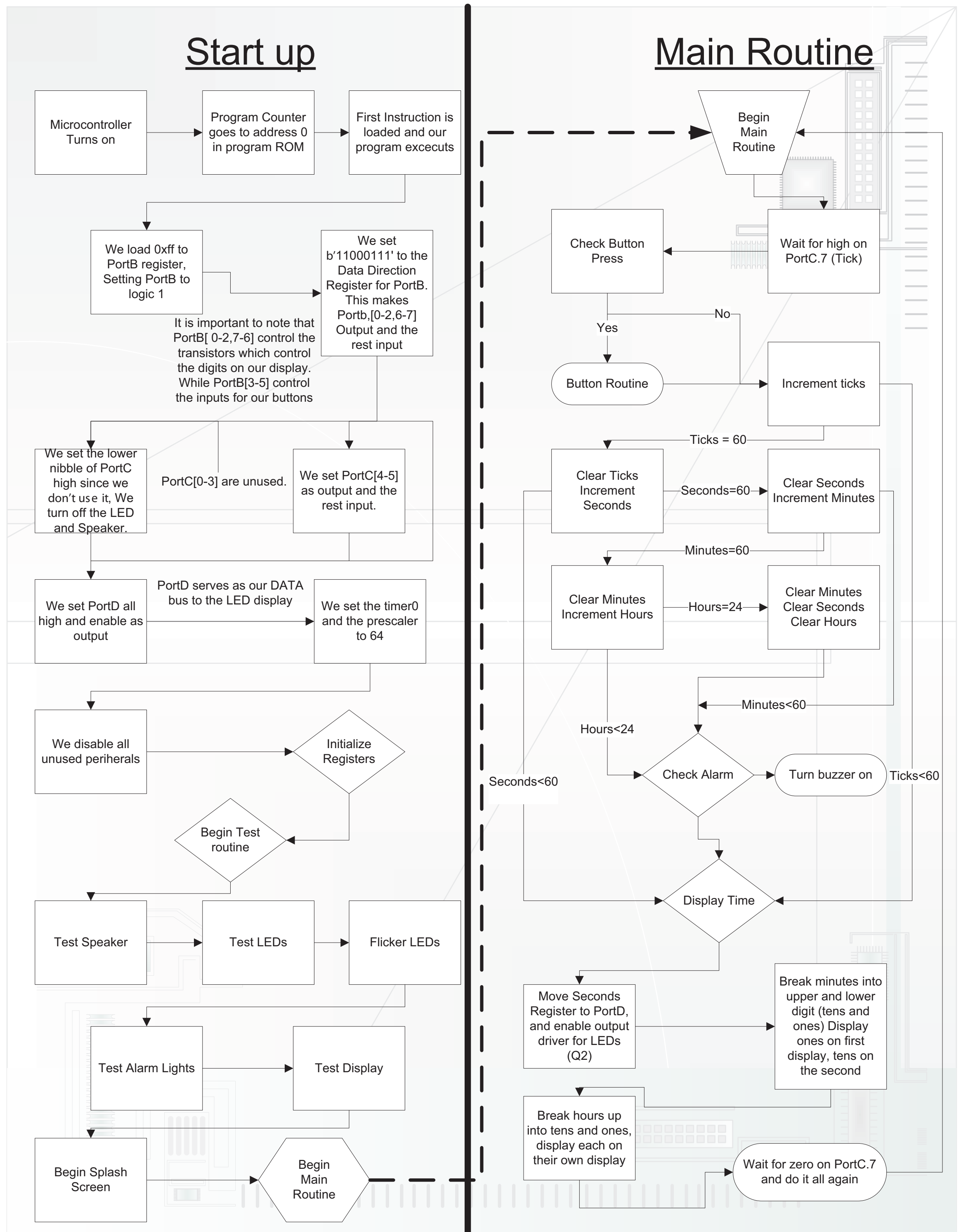
5. Go back to the beginning.

Looks simple enough, so now lets just put that in a text file and assemble it! Not so fast....

7 One GIANT leap

We are working with microcontrollers and we are working with machine code, the descriptions of the algorithm we are going to be working with are very basic. We recall that our CPU can merely do basic functions like add subtract

and other logic orientated functions, thus we have to break our code down to steps of addition and basic logic. This might sound daunting but as long as we take small steps and define clearly what we are doing, writing the assembly to complete our goal is almost trivial. So lets begin with a chart.


Start up
Main Routine
Microcontroller Turns on
Program Counter goes to address 0 in program ROM
First Instruction is loaded and our program excecuts
We load 0xff to PortB register, Setting PortB to logic 1
We set b'11000111' to the Data Direction Register for PortB. This makes Portb,[0-2,6-7] Output and the rest input
It is important to note that PortB[0-2,7-6] control the transistors which control the digits on our display. While PortB[3-5] control the inputs for our buttons
We set the lower nibble of PortC high since we don't use it, We turn off the LED and Speaker.
PortC[0-3] are unused.
We set PortC[4-5] as output and the rest input.
We set PortD all high and enable as output
PortD serves as our DATA bus to the LED display
We set the timer0 and the prescaler to 64
We disable all unused periherals
Initialize Registers
Begin Test routine
Test Speaker
Test LEDs
Flicker LEDs
Test Alarm Lights
Test Display
Begin Splash Screen
Begin Main Routine
Begin Main Routine
Check Button Press
Wait for high on PortC.7 (Tick)
No
Yes
Button Routine
Increment ticks
Ticks = 60
Clear Ticks Increment Seconds
Seconds=60
Clear Seconds Increment Minutes
Minutes=60
Clear Minutes Increment Hours
Hours=24
Clear Minutes Clear Seconds Clear Hours
Minutes<60
Hours<24
Seconds<60
Check Alarm
Turn buzzer on
Ticks<60
Display Time
Move Seconds Register to PortD, and enable output driver for LEDs (Q2)
Break minutes into upper and lower digit (tens and ones) Display ones on first display, tens on the second
Break hours up into tens and ones, display each on their own display
Wait for zero on PortC.7 and do it all again

7.1 One Step Back

Here is the ESE123 digital clock code in flow chart form. From now on in this document we will be discussing the code that applies specifically to the Spring 2010 ESE 123 Digital Clock design. So right now a lot has been dropped on you as a reader, you have this really complicated flow-chart which defines the inner working of the code. This chart is not meant to frighten you but to merely guide you along in the upcoming sections. This flowchart is to help organize the code in an easy to read manner. I suggest you glance over and try to make sense of what the flow chart has on it. I promise by the end of this section the flowchart will be a very useful reference while going over this code.

7.2 Our Design

With the flowchart out of the way, the next section deals with how we manage our resources on our hardware. As explained earlier, we have a fixed number of I/O resources to accomplish what we would like to accomplish, so it is very useful to break down how you will divide up your resources before you actually write any code.

7.3 Resource Management

With our general algorithm we outlined what we would like to do, for this class we are using an ATtiny 48 microcontroller. We have two 8-bit ports (B and D) and one 7-bit port; Port C. So lets make a list of what we're hoping to use.

1. Some Display for Hours, Minutes Seconds
2. Some Buzzer for our alarm
3. Some Buttons to control our alarm
4. Some Periodic Source

7.3.1 Our Display

For this class we are using 7 segment displays. A 7 segment display is merely an array of 7 LEDs which share a common anode or cathode. This means that one display needs 8 pins (our 7 LEDs and that common anode/cathode) Each display can only display one digit at a time, and each increment of time (Seconds,Minutes,Hour) needs two digits. That means we need 6 displays which is 48 pins!(6*8) We only have 23 pins available to use! So how do we deal with this? So we are pretty slow creatures, our response time is about 20ms, our microcontroller operates at 8MIPS which means on average 160,000 instructions pass by in that same amount of time. So we can take advantage of that. Now lets say we turn a display on for a short, but noticeable amount of time, say a few hundred μS We then turn it off, and the turn on the next display for a few hundred μS. We do this for each display and we keep going from one display to the next, looping from last to first. If we were to look at this array of displays, it would seem as if all were on at once. Now the biggest advantage of this is that our pin usage is only one 7-bit port (for the 7 segments) and one bit (per display) to select that particular display (and turn it on). We go from needing

48 pins to only 13 pins. This is the art of multiplexing, you use the fact that the human response is slow to save hardware resources. We can use this technique of multiplexing for almost anything and it will be a very valuable one in your future.
For this project we use 4 multiplexed 7 segment displays for hours and minutes and a multiplexed array of LEDs for seconds. this in total takes the entire PortD register and PortB[0-1] and PortB[5-7][4].

7.4 The Buttons

We simply need three buttons for this project; we need a button to set the alarm, increment and decrement. These use up PortB[2-4]. When a button is pressed the according pin on PortB is set to 0, otherwise it is internally set to 1. One of our buttons has a LED light on it, we dedicate PortC.4 to control that LED.

7.5 The Buzzer

We dedicate PortC.5 for the buzzer. The buzzer we are using emits a sound whenever it is given power, thus when a logic 1 is on PortC.5, the buzzer sounds.

7.6 The Clock Source

So what is a clock without a periodic source? simple; a counter. We could have used many things for our clock sources. We could of used our internal oscillator of the microcontroller (Inaccurate), We could of used an external oscillator (too fast), or we could have used an external time source (too complicated). This year we decided to use a tap from our power supply of the 60hz AC we rectified. Surprisingly, our power lines are very well regulated to keep that 60hz steady, so as a clock source it is fairly reliable (not the best though). So for this project we dedicated PortC.7 to keep track of this 60hz AC signal and to act as our tick and the main drive of our code.

[4] Notation Time!
Portx.n means Port X, bit N so PortC.5 means Port C, 5th bit. PortC[0-7] means PortC bits 0 THROUGH 7

7.7 All together now!

So we've outlines our resources, lets list them:

7 Segment Data line	PortD[0-7]
Seconds LED Array	PortB.0
Display 1	PortB.1
Display 2	PortB.2
Display 3	PortB.6
Display 4	PortB.7
Alarm Button	PortB.3
Increment Button	PortB.4
Select Button	PortB.5
Alarm LED	PortC.4
Buzzer	PortC.5
Main Tap(Periodic Signal)	PortC.7
	PortC[0-3]

This is how our microcontrollers resources are divided amongst the hardware.

7.7.1 The challenge

As you read this article from now on, notice that the lower 4 bits of PortC are free and open for your own use. I challenge you by the end of this document to have an idea for what you would like to use them for if you were writing this code (death clock robot anyone?).

8 The Real Deal

We've outlined the hardware,the algorithm and the procedure in the code, so now lets move on to the actual assembly code. This year we've broken the code into 4 separate files. We have a setup file which deals with the setup routine (left side of our flowchart). We have the main clock code which is described in the right part of the flowchart. We also have a file of common subroutines we call as well as a file that handles the logic for button presses. The code will be heavily commented so the purpose of this part of the document is merely to go over the syntax of the assembly and how we integrate our design into actual code. The code will be referenced by section and attached as an appendix.

8.1 Startup

This is the file that our assembler first assembles. It includes all of the setup parameters that we define the microcontroller to use to function as a digital clock. In this routine we set the specific I/O Ports to function as we outlined above. We also configure the timer for use in our code as a clock source. So lets dig into our setup.asm file. The first instruction we encounter is not an instruction at all but a preprocessor directive This directive is ".include".

8.1.1 The Preprocessor

So part of our assembler (the program that converts our assembly to machine hex) is the preprocessor. This preprocessor takes commands (which are inde-

pendent from our assembly) and modifies our code based on that directive. The sole purpose of our preprocessor is to make our code look pretty. This allows our code to be readable for things other than our CPU. It also makes our code more flexible by separating important routines into separate files and using them in different code. The preprocessor is your friend, so its best you learn how to use it! Our first line in our code is a preprocessor directive, .include"x". All this does is say "go to file x and dump its contents where you see this command," so .include"tn48def.inc" goes to file tn48def.inc and dumps it into our assembly code. The first two files we include are tn48def.inc and header.h.

tn48def.inc: This file contains definitions (will be explained next) defined by Atmel for their ATtiny microcontroller. The definitions are mostly the actual memory locations for their special function registers(The memory location associated with the 32 general purpose registers r[0-32])

header.h: This is a file we created to take care of all the definitions we made to simplify our clock's operation.

8.1.2 header.h

Now there are two more important preprocessor directives we must go over: .equ and .def. In reality they do a very similar thing, they take a "tag" (a string of characters that means something to us(lets say a word)) and replace every instance of that "tag" with something else. The difference between the two directives is that something else, .def is a more general replacement and defines a string as anything while .equ is a literal equality that replaces that string with a literal numeral expression. If we look at our first definition, we see that we r0 (Our first general purpose register) is defined as tens. So every times we use the tag "tens" we actually mean r0. So the preprocessor will go through our code and replace every instance of tens with r0. Our first .equ directive (equality) is that $buzz_on$ is equal to 0x0. Thus, every instance of $buzz_on$ will be replaced by 0. All the definitions and equalities are explained in the header file as comments, but most of them are pretty self explanatory. Now lets head back to setup.asm.

8.2 The .org directive

Our next line in our setup file is this directive .org. This is the origin directive and it tells our preprocessor where in memory to store our code. Ill take this as a good place to get back into the hardware part of things while we go over how our program resides in memory.
Lets start with a general view of things: whenever we make an assembler code we have 3 things associated with any instruction.

1. The line number
2. The location in PROGRAM memory
3. The actual instruction

So in a nutshell, the .org directive just tells our assembler where to START that program memory location count. Simply, .org 0x00 tells our assembler to start at zero. So we get something like this:

Line Number	Program Memory Location	Instruction
N00	0x00	Get A
N01	0x01	Get B
N02	0x02	Add A and B
N03	0x03	Store C

Why do we have to do this? If we look at how our program memory is laid out, we see something. For our ATtiny, our program doest go 0x0,0x1,0x2,0x3, etc... it goes 0x0,0x14,0x15,0x16,0x17. Because of that we have use the .org directive to tell the preprocessor where in program memory where to store our program. The reason the program memory isn't sequential and we don't do 0x0,0x1,0x2,0x3 is because we have special functions associated with the blocks 0x1-0x13. When our microcontroller first turns on, we jump to address 0, this is called our Reset Vector. Addresses 0x1-0x13 we reserve for our interrupt vectors (These are where our microcontroller jumps to when an interrupt is triggered(We will go over that later)). The program address 0x14 is the start of our sequential program memory and where we begin storing our routines and instructions.

Program Memory Address	Function/Contents
0x00	Reset Vector
0x01-0x013	Interrupt Vectors
0x14	First Instruction
0x15	Second Instruction

This might be a little counter intuitive, but its how our hardware works, and no matter what platform you're working with you will have to know where the interrupt and reset vectors are so that you know how to plan and organize your code.

8.3 Relative Jumps

Because of how our memory is laid out, the first actual instruction we come across is the rjmp instruction. This is not a preprocessor directive, this is an actual instruction our assembler converts to op-code that our microcontroller directly interrupts. If we look at our data-sheet we see that for rjmp the op-code is 1100 - xxxx - xxxx - xxxx with the x's denoting the memory address of the jump. So for our first instruction we tell the microcontroller to go to the memory address that is defined by the initialize tag. Lets get a visual of what exactly is going on.

Line Number	Program Address	Instruction	Opcode
microcontroller turns on, PC is set to reset vector of 0x00			
N00	0x00	rjmp initialize	0xC014
initialize is a tag which is set to address of 0x14, microcontroller jumps to that address			
N01	0x14	first instruction executes	

8.4 Relative Jumps, Branches, Calls, Program Counters, and the Stack

So far we've introduced one instruction so far, the relative jump command. This command along with a few others are very different from all our other commands. rjmp along with branch and call all control program flow. They determine where a program will go and allow us to make our code conditional (Remember the IF clause?). Now we can further break these control elements into two sets, ones which modify the stack and ones which don't.

8.4.1 Branches and Jumps

As we mentioned earlier, all a jump does is tell the microcontroller to go to a specific program memory address. A microcontroller (or any hardware for that matter) keeps track of where it is in program memory through a program counter. All this program counter does is keep a reference to where it is in program memory. When you turn on the microcontroller this is reset to 0 and it increments itself after every instruction. When you use a jump, you are adding your jump size to the PC which in turn tells the microcontroller to jump to that piece in program memory. A branch is merely a conditional jump. When some condition is met, jump to whatever tag we choose. For example, in the "done" subroutine in setup.asm we have a branch if negative command (brne). If the negative integer flag is set, the command will jump. In the case of the "done" subroutine it will jump to the subroutine "loopy". Looking at our instruction set, we have many types of branches to play with.

8.4.2 Calls, a different kind of control

A call is a whole different beast of a control element. When you make a call on a subroutine you jump to that routine and stay there until you tell the microcontroller to return. If you are familiar with functions in a language like C, this makes much more sense. However if you are not, this is utterly confusing. Remembering that all the jump command does is modify the PC, a call does the same thing, but it also does something special: it stores the old PC in a special register called the "stack," and it begins a new on one on TOP of the old one. So, if you think of it like a cake, at the base is your older values and at the topmost is your current working PC. Stacks are said to have a LIFO(Last In, First Out) control structure. The point of storing the old PC is so that you can return to it, this is the major difference between call and jump, a call you can return to where you called a subroutine, while a jump you are just going to that subroutine. You will eventually learn when to use one over the other, but a jump is generally used when you want to control the flow of a program and a call is used when you want to use a subroutine many times as a function. (say doing an analog to digital conversion; you would call a routine that would do that and store the result in a register).

8.4.3 A pretty diagram

To better illustrate the difference between jumps and calls I will break out the tabular environment once more.

Jumps:

Line Number	Instruction	NEXT Program counter value
N00	Load A	0x01
N01	Load B	0x02
N02	Add A,B	0x03
N03	rjmp 0x00	0x00
We go back to line 0 and loop forever		

Calls:

Line Number	Instruction	Next program counter value		Stack		
				L1	L2	L3
N00	Load A	0x01		0x0		
N01	Load B	0x02		0x1		
N02	CALL ADD	0x20		0x2		
ADD is on line 32 and located in memory location 0x20. the program now goes to that memory location						
N32	ADD A,B	0x21		0x2	0x20	
N33	RETURN	0x03		0x2	0x21	
The return command brings us back to our starting position as the second level of the stack is popped off and we begin where the last PCL was set.						

So as you can see, jumps calls and branches all serve a similar purpose (to control program flow) but they do it in two different ways. As you program more, you will realize when it is appropriate to use a jump and when it is appropriate to call a subroutine. Now, lets move on to instructions that do the grunt work of our code.

9 The Grunt Work

If you've survived this far, the end isn't too far from now! The next sections cover instructions that actually move and process data. There are the instructions that move your data from one register to another and do the ones that do actual processing in our code. Many of these instructions are very self explanatory, so I am going to go over how they work rather than what they do.

10 Our First Subroutine

I think it's worth it now to explain how our code is laid out.

// - Denotes a comment, anything after this is disregarded by the assembler

A "subroutine" is just a section of code that we group to do a particular function, outside of us giving it a name, it has no merit to the assembler or the microcontroller. A subroutine is a fabricated device to organize our code, unlike in languages like c,fortan and java where it implies something different. We denote the begging of a subroutine with a tag. Our first tag is "initialize:"

tagname: - Denotes a tag which the assembler uses to note program memory location.

When we use a function like call or goto, we can use that tagname to refer to that section in our code.
Now, as mentioned above, these tags are merely for your convenience. When the preprocessor processes your assembly code and prepares it to be assembled, the tags get replaced with the program memory location.

Line number	Program Memory location	Command	Disassembly
N00		Initialize:	0x00
N01	0x00	Some Command	Some disassembly
N02	0x01 rjmp Initialize	rjmp 0x00	0xC014

Now that we got that out of the way lets actually look at the first few lines of the initialize routine.

10.1 Types of instructions

We went over instructions that control program flow, but there are two other types that are worth mentioning, the instructions that actually move data in our code and the instructions that are used with the program flow instructions to give our code thought, conditional instructions.

10.1.1 The Movers

These instructions are the ones in which we use to move data from register to register, load data into a register and to control flow of our data. These are instructions like, move, load, output, etc. They all do the above and generally they are either general purpose or meant for dealing with only specific registers. (as you will learn below ”out”, is an instruction that only works with I/O ports.) These are generally very simple, they usually accept two arguments, namely, destination and source.

10.1.2 The Conditional Instructions

Instructions like call and jump are useless if we cannot have them happen based on a condition. (IF A goto B IF NOT stay C) For our microcontroller we have several conditional statements, but they all work the same way. A conditional statement will evaluate an argument. (Lets use the compare instruction as an example) This instruction will compare the two registers, and it will set the equal flag based on the result, once this flag is set we use a branch statement to execute our jump.

Instruction	What happens
cpi ticks,60	compare IMMEDIATE (60) with ”ticks” register
brne wait_ out	Branch if NOT EQUAL. (This executes if the above returns false, otherwise it will be skipped)
some instruction	This is executed if the above if cpi is false, otherwise the branch is executed and we jump to another part of the code.

As you can see, the CPI instruction controls our program flow through a branch because it control the flag which sets the branch. Phew! This type of programming is something that you will just have to get used to, it isn’t too hard, but its a completely different way of controlling program flow than in c or java.

However, all the structures that exist in c and java (For, While, if, switch) are all implementable in assembly. (duh!) They just take a different form.

10.2 Setting up the ports

One of the first things we do when we write our code, is outline what ports we will be using. We do this because the first piece of code we generally write configures the ports and the peripherals on the microcontroller. Here is our first section of code:

```
initialize:
//setup portb
ldi scr,0xff ;turn pnp transistors off, pullups on
out portb,scr
ldi scr,0b11000111
out ddrb,scr
```

The purpose of this section of code is to setup PortB. We recall earlier that PortB handles driving the displays and controlling the buttons. Thus, we need it to operate as both an input and output port. When we read the data sheet we find that every port has a Data Direction Register associated with it. By writing a logical 1 to the bit that correlated to each pin (Ie bit 0 refers to PortB.0) we enable it as output. A logical 0 enables it as input. The first instruction we come across is ldi.

ldi - Load Immediate. It accepts two arguments, an address and a literal value(number).

Our first instruction ldi scr,0xff loads the value of 0xff to the register scr. scr is a scratch register we defined to be r20. If that was easy enough to comprehend, the next one is even easier!.
The next instruction is to move from register to register.

out - out, accepts two arguments, first is where to move TO and second is where to move FROM. This instruction is for I/O space ONLY. (meaning you use it with ports)

So out portb,scr simply means move from scr to portb (thus portb now hold the value of 0xff).
So wait... why don't we just load PortB with 0xff? Generally it is bad practice to load a value immediately into a special function register, it causes instabilities and might glitch your code. It is always best to refer to the data sheet to make sure you understand the differences between each type of move instruction. We will be encountering more types as we move along but we wont have time to go over them all.

10.2.1 Understanding it all

As you can see, the assembly is really easy. So why is it so hard? The hardest part of programming for a microcontroller is understanding what a 1 means and what a 0 is. Its great we sent the value of 0xff to PortB, but why did we do that? First thing to note, when you SEND a value to a port, it only applies to the ones which are enabled as output. Sending a value to a input doesn't do anything

(might cause a glitch). We must only concern ourselves with the output ports we set aside on PortB. The purpose of the outputs on PortB is to drive the PNP transistors. What we know about the PNP transistors is that a logic 0 will turn them on, but a logic 1 will disable them. When they are enabled, the display turns on and the value that is on PortD is displayed. When we write logic 1's to PortB(thats what 0xff is) we disable all the displays, so no matter what is on PortD, nothing is displayed. This is what we want when we turn on the clock. So if you understood that, there is no reason to go over any other code in this section, because we are doing the same thing. We might just be clearing and setting bits instead of whole registers. To demonstrate this, the next two lines in the code, merely enable PortB [0-2] and [6-7] as output and the rest as input, just as we described earlier.

10.3 Instruction and special function register description Time

In this section we will briefly be going over the instructions of the setup routine as we come across them. This section is mostly for reference and might be a little dry.

SER - Set Register; this command will set all bits of a register to logic 1.

CLR - Clear Register; this command will set all bits of a register to logic 0.

10.4 The Timer

Like we said earlier, its important to note that our microcontroller has hardware peripherals that we can use and take advantage of. One particular peripheral we will be using is the timer. To understand how the timer works we must understand the special function register that is in charge of controlling our timer.

10.4.1 The TCCR0A Register

Control registers are pretty simple, they store the control bits for a module. Setting them will enable or disable some feature. For our purposes this register controls the Timer0 hardware timer. So let's look at the control bits.

Bit Number	Bit Name	Description
7	RES	Reserved, not for use
6	RES	Reserved, not for use
5	RES	Reserved, not for use
4	RES	Reserved, not for use
3	CTC0	Clear Timer on Match Mode Enable/Disable
2	CS02	Clock Select Bit 2
1	C201	Clock Select Bit 1
0	CS00	Clock Select Bit 0

As we see, theres only two things we have to configure. The first thing is whether we want the timer to clear itself when the compare module makes a match (no we don't, we want a normal timer). The second thing we need to configure is the clock source (which also turns on the timer). A more detailed explanation of these registers are in the data sheet, but it is pretty straight-forward. Now

the CTC0 bit is obvious, we want that to be 0 so it is disabled, however the CS0 bits aren't as trivial.
The data sheet provides us with the following table:

Bit 2	Bit 1	Bit 0	Mode
0	0	0	No Prescaler / Timer is off
0	0	1	No Prescaler, Timer input is internal clock
0	1	0	Internal Clock/8 Prescaler
0	1	1	Internal Clock/64 Prescaler
1	0	0	Internal Clock/256 Prescaler
1	0	1	Internal Clock/1024 Prescaler
1	1	0	External Clock on T0 pin, falling edge
1	1	1	External Clock on T0 pin, rising edge

This being a timer, we need a source for our "ticks" these control bits CS0[0-2] all have to do with selecting our clock source and our prescaler value. A prescaler is merely a hardware implemented clock division thus with a clock/8 prescaler it takes 8 clocks for one tick. As you can see, everything is pretty self explanatory, the best thing to do is to play around with the settings and see how they change things.
Now that we went over that we will continue going over some instructions.

STS - Store Direct to SRAM; We use this command when we want to write to a special function register in our RAM.

10.5 More I/O crap

The atmel series of chips seems to have tons of different instructions that deal with moving data from register to register and from register to port. We use a few and i'm just going to briefly outline them.

STS - Store Direct to SRAM; We use this command when we want to write to a special function register in our RAM.

Like we had the OUT instruction earlier for outputting to a port, we have an IN instruction for taking data IN from a port.

IN - IN; We use this command when we want to take the contents of a port and output it to a GRP (IN scr,portd).

We also have an instruction for moving from GRP to GRP

MOV - Move; This instruction moves to one address FROM another. i.e. MOV GPR1,GPR2 moves the contents of GPR2 to GPR1

10.6 Other instructions

There are tons of other instructions, its not worth going over them all, because in all data sheets there is a list of instructions and what they do, with a little common sense and trial and error, you shouldn't have any issues.(Under the current revision of the ATtiny48 data sheet, the instruction set summary is on page 280, section 24)I am going to finish this section with a brief list of some common instructions.

Some of the more important instructions are the ones that deal with bitwise operations.

SBI/CBI - Set Bit Immediate, Clear Bit Immediate; These instructions will set or clear a desired bit, sbi portc,5 will set the 5th bit of portc to logic 1, cbi portc,5 will clear it to logic 0.

SBIS/SBIC - Skip if Bit SET, Skip if Bit CLEAR(I/O space only); These are very powerful control instructions that allow you to check the status of a port and control flow based on that status. When the condition evaluates true, the instruction will skip the next line of code (Increment the program counter with two instead of one). This is very useful if you put a jump right after this instruction. That would allow you to follow one program path if true, and another if false (and by switching the logic you can switch the program flow)

To give a quick example of this lets look at the following table. In this scenario, PortB.0 is set as a button, a button press denotes a logical 0.

Instruction	What it does
Some code	The code that preceded our demo
sbic portb,0	This command will skip the next line if portb.0 is a logcal 0 (and the button is pressed)
rjmp someOtherPlace	When the button isn't pressed we go somewhere else
some instruction	Otherwise, we execute code signifigant to a button press

Likewise, the inverse of that logic is true.

INC/DEC - Increment, Decrement; These instruction add one (Increment) or subtract one(Decrement)

10.7 A Quick Synopsis

To conclude this section I'd like to go over what we did in the setup file. The first thing we did was setup the ports of the microcontroller for our design. We knew what each pin had control over and because of that we knew what to set each port register as and each data direction register as. The next thing we did was to set up the timer module. We did this by setting the appropriate bits in the special function register that dealt with the timer. The rest of the setup routine is the splash screen that we see on startup, for the most part it isn't necessary but it gives you a good idea how to do things like setting up tables (a type of data structure) and doing loops. The next part of the code is the main driver for our clock

11 The Main Program

The main program is the part of the code that is the brains of the clock, it controls the ticking and makes sure that you can see the results. We can break the main program into several parts. The clock driver, the button driver and the display driver.

11.1 A note

For the scope of this article, as a guide for beginners into the working of the mirocontroller, we will be ignoring the button driver. It doesn't affect the workings of our clock as a clock, it merely adds user interface to our design. Although this is important, the magnitude at which the user interface has been implemented is simply over-whelming. I could spend another 10 pages alone going over the button code but, I feel it isn't worth it. However i will be first going over conceptually how the buttons work.

11.1.1 States

If you look at the code, you notice these awkward things called "states." The way our button procedure works is that each button press/combination will set the state the microcontroller runs in. By default, we are always in the first state (Normal operation) until the buttons are pressed. Once a button is pressed, depending on the combination of presses, we will enter a new state (Setting the alarm, changing the alarm, etc.). This isn't conceptually hard, but for our final design we have 21 states! This complicated the code, but for a design that doesn't use interrupts, its the best approach.

11.1.2 A Brief note about interrupts

The best use for interrupts is user interface. Traditionally you either have interrupts controlling the mechanics of the code and the main program running the user interface. If we used such a paradigm, we would have the ticks trigger an interrupt (A port change interrupt) we would then run the clock driver and the display driver then return to the main code and poll the buttons all the time until another tick. The advantage of this is that the time is absolute, we ALWAYS will get our tick since our code will INTERUPT itself to handle a tick. The disadvantage is that we'll still have awkward code for polling the buttons. The other way of using interrupts in user interface is to keep the main code as the driver and have a button press trigger and interrupt This is the exact opposite of before but holds different consequences The consequences of using interrupts is that it simplifiers your user input code (You don't have to juggle two operating schemes at the same time, they work independently) and your main routine becomes more efficient. Now with that said, we will be ONLY be discussing the clock code in state one, or normal operation and we will be IGNORING button presses.

11.2 The Clock Driver

The first thing we do is we wait for our "tick." As explained before, the tick is the physical thing we use for our clock source. In this case it is the positive side of our 60HZ AC signal. When this event happens (every 60'th of a second) we exit our loop tick_tock and begin executing the part of the code that keeps track of time.

11.2.1 Time

This part is very simple. When we get a "tick" we increment the ticks register. Then we compare it to 60, 60Hz AC, 60 ticks to a second. Once we have 60 ticks, we increment seconds and clear ticks. We do this up until we have 24 hours in which we clear everything. There is your clock, explained in two sentences. If you were to use just those few lines of code, you would have a perfect clock. All the other code is merely stuff that supports those twenty or so lines, but our clock, is nothing more than adding one and comparing to some values significant to our society. Fun. With that said, we need to realize two things. First, no matter what, once we get the new time we need to display it. Second, when a new minute or hour rolls overs, we need to check the alarm (We can implement an alarm that activates on the 60th of a second, but who would want that?). And that is precisely what we do.

11.2.2 The Alarm

The alarm procedure is very simple. All we do is compare the current time to one which is stored in the microcontroller's memory (which we choose when setting the alarm). If there is a match, the clock will annoy the hell out of us until we push the alarm button again. If there isn't a match, we continue to the display time routine.

11.3 The Display Driver

Once we have our time, we need to display it. For our display, we have two types of routines we do, one for seconds, and one for minutes/hours.

11.3.1 Seconds

For our clock, we have 6 LEDs which "count" in binary up to 60 (max up to 64). When a LED is lit, it represents a one in our binary clock. To do this is almost trivial, we have to realize a few things first. We are using PNP transistors to control both the display and the LED, so everything is backwards. The LEDs are controlled by PortB.0, so we first have to turn on the display, then send the inverted seconds register to PortD (our data bus) and voila! The LEDs will display our seconds, in binary. So we call a delay(of roughly 2100 clocks) and then turn off the display. Next!

11.3.2 Multiplexed displays

The type of display procedure we are using is called multiplexed displays. As mentioned earlier, we turn one display on for a short period of time, then move on to the next. This happens once every tick, so our displays get updated every 60th of a second (thats better than most movies). The delay routine we call, determines the brightness of the display, the longer it is, the longer our display is on, the brighter it is. Likewise, the inverse is true. So if you find your display is too bright or dim, just edit the delay routine in the subroutines file to change it as you like!

11.3.3 Minutes, Hours, Segments Oh My!

As we mentioned earlier, we are using the 7-segment displays to display our minutes and hours, and for both we use the same exact routine (except the bit on PortB for turning on a display is changed). Because of this I will only discuss the hours. The way our hours are stored in our microcontroller is as a number, so when its 12 o' clock, we store 12 in the hours register. If you haven't noticed, each 7-segment display is 1 digit, and each display doesn't have an input for numbers. We have to DECODE the number held into our register as information that our 7-segment display can read. That is precisely what the get_segs routine does. We put the number we want into the scratch register, call the function, and in get_segs_h and get_segs_l registers we have both the upper and lower digits (tens in the h or high register and the ones are stored in the l or low register). We load the hight register into the higher digit segment which represents the tens digit,then we call a delay (to make the display visible for a period of time). We then load the lower digit in the lower digit segment, and delay. We do the same thing for the minutes, except that we load the mins register into the scratch register, and we change the display by changing PortB remembering that PortB1.2 was used for the tens and ones of the hours and 6.7 was used for the tens and ones of the minutes.

11.3.4 The End?

After we update the display we are done! The clock does everything a clock should, we get the time and display it. The rest of this routine deals with killing time (hah) and waiting for our 60Hz AC signal to go low. Now, if you ever wanted to add additional function to the clock, you can very well add routines in between, before or after any of the ones we've written. Our entire main routine takes very little time despite it does so much. Most of the time the code spends its time in the TickTock routine and the WaitLow routine. We end up with a fairly simple main routine, but our main program is only simpe because we have the help of our subroutines.

12 Subroutines

The subroutines file holds all the most called and general routines. For all practical purposes, its just code, its well commented and all of them are pretty general routines. There are two routines I would like to briefly go over, get_segs and delay.

12.1 Delay

Delay routines are simple and essential. Often we need to translate the time frame the microcontroller works on (nanoseconds) into a time frame that works with the human response time (milliseconds). or we might just need to kill time for other reasons. We use delays to turn on our display so that they're on long enough to see, but not on too long that they eat in to our resources (that 1/60th of a second). A delay routine is just simply a routine that will loop (doing nothing) for an integer number times. The most convenient is 256 (one filled register) but we can tailor a delay for almost any timebase.

12.1.1 A Necessary Evil

The whole nature of a delay routine is to kill processor cycles, when processor time is essential delay routines are frowned upon. Its much better to use code which branches and recombines to kill time (thus doing useful work) than a delay routine. In our code they are acceptable since we barely tax the CPU.

12.2 Get Segments

As we mentioned above, we must convert integer number stored in registers into something we can pipe onto the data bus and display on our segment displays. The get_segments routine does that. The routine takes the integer in the register, divides it by 10 until it can't. The number of times we divided by 10 is the tens, and the remainder is the ones. That part isn't spectacular but converting it into something our displays can use requires a little bit of trickery.

12.2.1 Tables

Tables are a way of organizingpieces of correlated data. In our case we create a lookup table. Depending on the offset(determined by the magnitude of the digit) we will load a value that relates to that offset. For our case, we setup a lookup table for converting the digits into the signals for the 7-segment display. The table works by setting aside a block of memory and the digit points to the memory location of the properly converted word.
So lets make a table at 0x00 and we will load it with the vales to display that digit on a 7 segment display.

Memory Location	Contents	Digit that is displayed
0x00	b'00001001'	0
0x01	b'11101011'	1
0x02	b'10000101'	2
0x03	b'11000001'	3
0x04	b'01100011'	4
0x05	b'01010001'	5
0x06	b'00010001'	6
0x07	b'11001011'	7
0x08	b'00000001'	8
0x09	b'01000001'	9
0x10	b'11111111'	OFF

So lets equate the tag tableStart with memory location 0x00. When we separate our hours/minutes into digits, we offset the memory location set in tableStart by that amount. So, when the ones digit of minutes is set to 5, we POINT to the location at tableStart+0x05. We see that the memory location of 0x05 is encoded with the data to display a 5 on our 7-segment display. This works for all digits 0-9. In essence, this is indirect referencing (pointers in C). It's a little bit confusing but extremely powerful. This is the easiest way to do a conversion like this, any other algorithms would just waste clock cycles. This is a way to take advantage of our RAM to save clock cycles. We use an abundant resource to save one which might be a little more valuable. You can also do large lookup tables in ROM (at the cost of speed).

13 Closing Comments

Is that all? I'm afraid I've barely even begun to scratch the surface! There are so many topics that I've barely even begun to touch. Microcontrollers are vast and versatile platforms that only limit you by your imagination. Hopefully after this document you have at least enough resources to feel a little more comfortable with these devices and can start working with them on a more intuitive level. A very important thing to realize is that all platforms are the same. Whether you work with a PIC microcontroller, an ARM processor. an AVR microcontroller or an INTEL x86 processor, the game is the same. The only thing that changes is the memory layout and the instruction set. All of computing however, is based on the same design philosophies I've briefly gone over in this writeup. Its very important to never become intimidated by a piece of hardware because in the end its all just sand. With that said, that doesn't mean learning an architecture comes without effort. You must be willing to sit down and DIG through data sheets to learn everything about a device. You WILL come across code that WILL not work despite depsite how logically sound you think your code is. You will then spend three days looking through a 700 page data sheet only to find in some footnote on page 546 that you can't use these two instructions one after another because of a small fault in the silicon. When you program with assembly, you are not writing software, you are telling the hardware what to do. C and other programming languages are software, assembly is not. Someone who programs C doesn't need to care about the program memory structure of their hardware nor do they need to know what vector the timer overflow points to. Understanding your hardware is key to programming with low level languages and devices, and in the end it will make you a better programmer with higher level languages when you know how your system works at a fundamental level. With all that said, practice makes perfect! You cant learn assembly or microcontrollers from reading books or writeup. You would learn 100 times more by sitting down with a programmer and some code than by reading this document. This is provided merely to help you jump start yourself into the world of embedded system design.

13.1 Thanks for all the fish!

I'd also like to thank some people for helping me throughout writing this document.
First and foremost, I'd like to thank Professor David Westerfeld for taking the time and initiative for providing such a rewarding experience to those who take ESE123 at Stony Brook University. As well for writing all the code and allowing me to work with him through the semester.
I'd also like to thank my friend Spencer Thomas for proofreading this behemoth of a document.
I'd like to thank Andrew Thomas and Ezafat Khan for proofreading and providing their insight.

Finally i would like to leave you the reader with some links to websites I've found useful myself while learning about microcontrollers and electronics.

14 References

14.1 General Reference:

- My blog - http://www.mcuplace.com
  All About Circuits - http://www.allaboutcircuits.com
- Microchip(Pic Microcontrollers) - http:www.microchip.com
- ATMEL - http://www.atmel.com/
- Wikipedia - http://www.wikipedia.org

14.2 Places to buy electronic components:

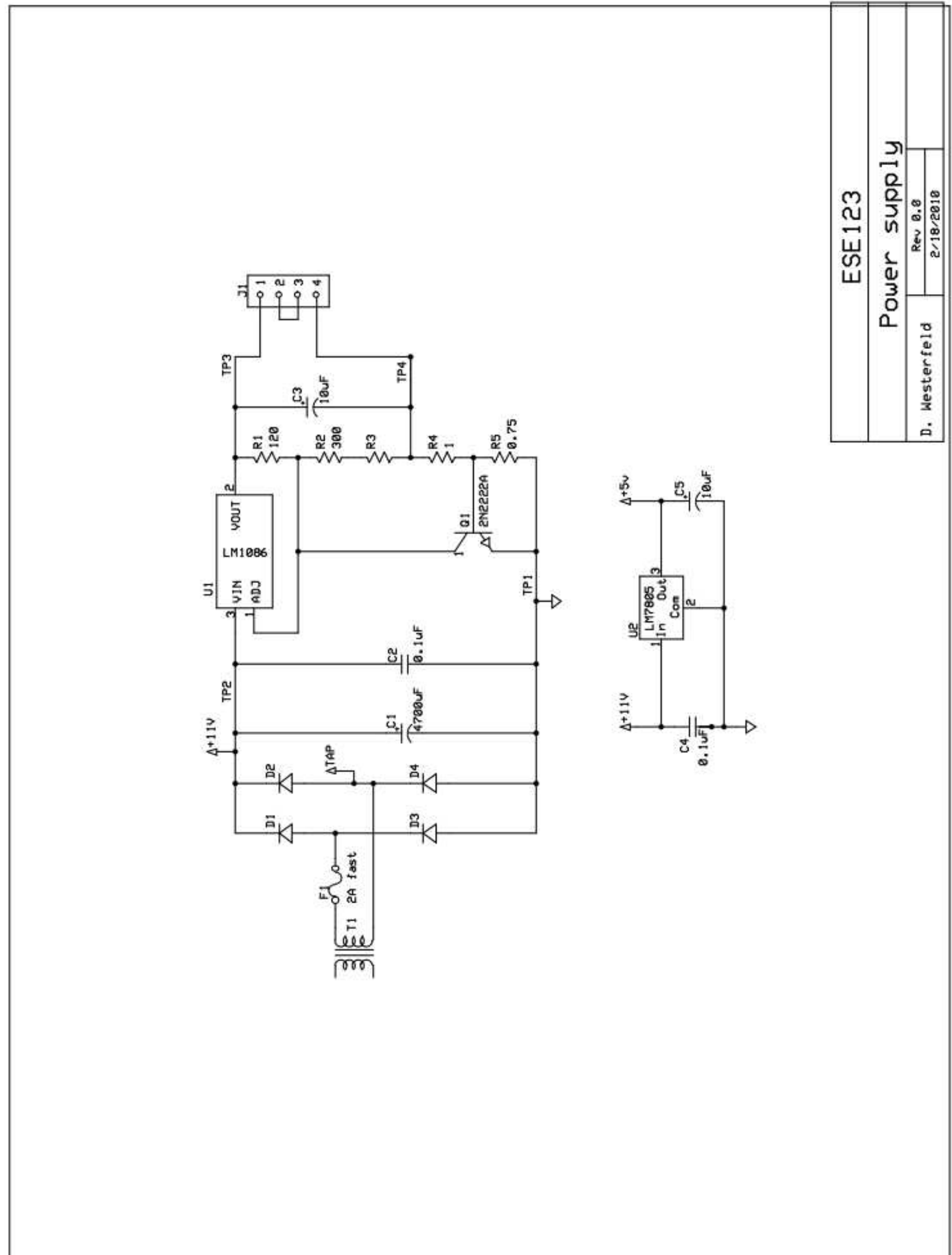
- Mouser electronics(Discrete Components) - http://www.mouser.com/
  Digi-key (Discrete Components) - http://www.digikey.com/
- Spark Fun Electronics (Projects, Cool stuff) - http://www.sparkfun.com
- All Electronics (Surplus) - http://www.allelectronics.com/
- Circuit Specialists - http://www.circuitspecialists.com/
- Goldmine Electronics (Cheap surplus) - http://www.goldmine-elec.com/
- EBay (Cheap/rare components) - http://www.ebay.com

14.3 Places to Sample Electronic components(need .edu email):

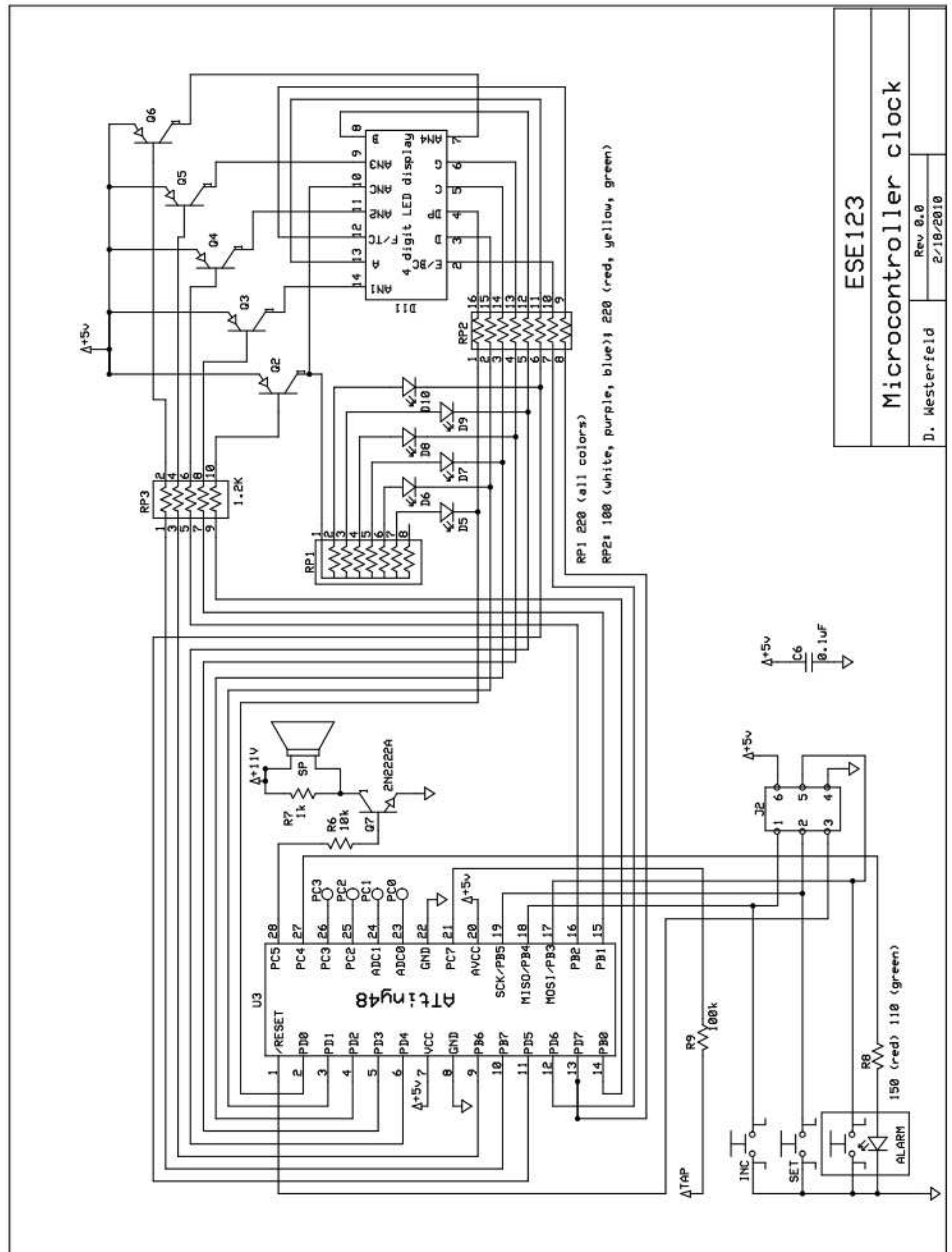
- Microchip - http:www.microchip.com
  Texas instruments - http://www.ti.com

15 Appendix

15.1 Schematics Page 1



15.2 Schematics Page 2



15.3 Code

Due to not wanting to add 20 extra pages to this document, the code can be gotten online from two places. http://mcuplace.com/ese123 . I will also be posting a blog article on this writeup, you will beable to download it there aswell. If you cannot find a copy, email me at anthony.tricarichi@sunysb.edu and I will be glad to send a copy!