Quiz Lockout, Scoreboard, and Timer System Using Microcontollers

A Design Project Report
Presented to the Engineering Division of the Graduate School
of Cornell University
in Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical and Computer Engineering

by
Richard J. D. West '05
2006 Master of Engineering Candidate
Electrical and Computer Engineering

Project Advisor: Bruce R. Land

Degree Date: May, 2006

Abstract

The goal of this Master of Engineering design project was to build a control and scoring system for high school quiz bowls. The system consists of a moderator unit and player units which ensures that only one of eight players may buzz-in to answer a question. A scoreboard and timer unit design is discussed which is flexible enough for any style of competition. Since the printed circuit boards are being fabricated as this report goes to print, no field tests of the complete system have been conducted. However, individual components have been tested, and additional work on the project will continue after this report is printed. Please visit `http://instruct1.cit.cornell.edu/courses/eceprojectsland/STUDENTPROJ/2005to2006/rw88/` for complete and uptodate documentation, source code, schematics, and layouts in full color.

Dedication

This design project is dedicated to the members of the Colchester High School Scholars' Bowl team past, present, and future. May we continue to compete strong and have fun doing it. Mr. Devino and Mr. Desrosiers, thank you for a great four years of competition. Hopefully this system will last longer than the old one did.

– Richard J. D. West, Colchester High School Class of 2002

Report Approved by:
Project Advisor: ________________________________ **Date:**__________________

Executive Summary

High school quiz bowls are common around the nation. In the state of Vermont, the quiz bowl equivalent is the Vermont-NEA Scholars' Bowl. While the heart of any Scholars' Bowl match is the students, the technological heart is the control/lockout system. The lockout system ensures that only one student is able to buzz-in to answer a question but there exists an equal opportunity for all the students. Several commercial lockout systems exist, but they are fairly expensive.

The goal of this Master of Engineering design project is to design a complete lockout system and scoreboard/timer system suitable for use in a Scholars' Bowl match. This system will be donated to the Colchester High School Scholars' Bowl team for use in their practices and matches. While the design aspects of the project are complete, delays fabricating the printed circuit boards mean that the project will not be complete by the time this report goes to print. Field tests of the complete system have not be conducted, but individual components have been successfully tested. Full testing and finishing touches will continue after this report is printed. Please visit `http://instruct1.cit.cornell.edu/courses/eceprojectsland/STUDENTPROJ/2005to2006/rw88/` for complete and uptodate documentation, source code, schematics, and layouts in full color.

Contents

List of Figures

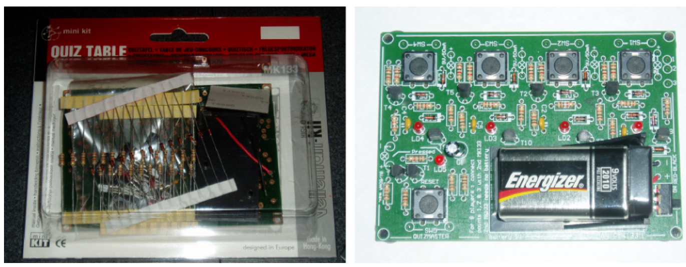
List of Tables

List of Source Code Listings

1 Introduction

High school quiz bowls are common around the nation. In the state of Vermont, the quiz bowl equivalent is the Vermont-NEA Scholars' Bowl. Scholars' Bowl has grown over its twenty-two years of competition to include thirty-three teams from across Vermont and parts of New Hampshire [1]. While some of these teams are traditional powerhouses, every year presents a completely new competition with a completely new set of strong teams.

While the heart of any Scholars' Bowl match are the students, the technological heart is the lockout system. The lockout system ensures that only one student is able to buzz-in to answer a questions but there exists an equal opportunity for all the students. Several commercial lockout systems exist, but they are fairly expensive. An inexpensive alternative is the minikit shown in figure 1. This minikit is an analogue lockout system for four players and a moderator gathered around a table. Of course, this inexpensive alternative is not suitable for competition on the scale of Scholars' Bowl.



(a) The minikit. (b) The finished product.

Figure 1: A commercially available lockout system kit from qkits.com.

The goal of this Master of Engineering design project is to design a lockout system capable of meeting the demands of a Scholars' Bowl match. The lockout system consists of the moderator and player units which is presented in section 2. In addition to the moderator and player units, a scoreboard and timer unit design is presented in section 3. While neither of these units have been completed due to fabrication delays, many aspects of the hardware and software have successfully been tested as discussed in section 4.

2 Moderator and Player Units

In a Scholars' Bowl match, there are two teams of four players each [1]. Each player has a buzzer with which they may communicate their desire to answer a question to the moderator's unit. The moderator's unit must ensure that only the first player to buzz-in is granted the right to answer the question. This right is indicated by illuminating the player's buzzer, locking-out the other players, and sounding an audible tone. Once the player has been recognized by the moderator, the player has five seconds in which to answer the question [1]. When the five seconds has elapsed, another audible tone is produced. The moderator then resets the lockout and proceeds to allow other players to buzz-in or onto another question.

2.1 High-level Design

The high-level structure of the moderator and player units can be seen in figure 2. Each player has a button (buzzer) which is connected to a player unit shared by two players. There are four player units in total which are connected to the moderator unit. Typically, each player would have their own unit, but an eight unit arrangement takes more time to setup and has more cords to trip over. By grouping two players together, the number of cords is halved with little impact on the players themselves.

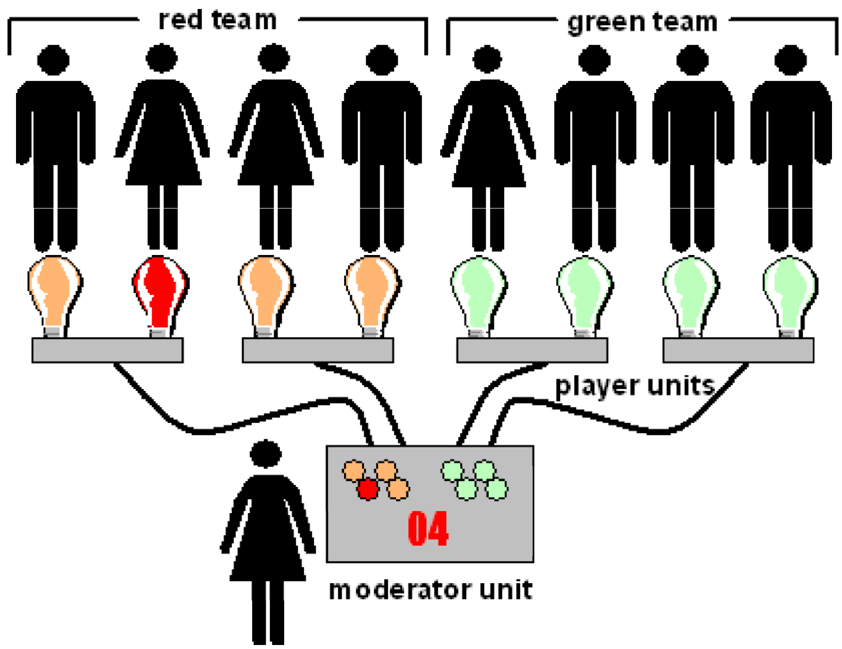


Figure 2: High level view of the moderator and player units.

Both the players and the moderator must be able to ascertain the state-of-play at all times during a match. Therefore, the moderator and player units must efficiently convey as much information as possible. Information such as the amount of time remaining to answer a question, the state of the lockout, and who has buzzed-in are especially important. As a result, both the moderator unit and the player units illuminate to indicate someone has buzzed in. The time remaining to answer a question is displayed both on the moderator unit and on a peripheral board for the benefit of the players.

2.2 Software Design

Atmel's ATmegaXX4 series of microcontrollers[1] were originally going to be used as the heart of the moderator's unit. The ATmegaXX4 series has configurable external interrupts on every I/O pin making it ideal [2]. Enabling external interrupts on the pins connected to the players' buzzer would ensure only one player can be granted the right to answer the question. Unfortunately, the ATmegaXX4 series of microcontrollers had some production problems which delayed distribution [3]. As a result of this delay, the planned ATmegaXX4 had to be replaced by an ATmega32[2].

Since the ATmega32 does not have external interrupt support on every pin [4], the pins connected to the players' buzzer have to be polled in a tight loop. The polling loop must be as tight as possible to minimize the probability that two players can buzz-in during the same iteration through the loop. During each iteration,

[1] The ATmegaXX4 series consists of the ATmega164, ATmega324, and the ATmega644. At the time of writing this report, only the ATmega644 is in production [3].

[2] The pin configurations for the ATmega16, ATmega32, ATmega64, ATmega164, ATmega324, and ATmega644 are identical except for the "alternative functions" available on each pin [3].

the polling loop must check the players' buzzers only if the players are not locked-out, but the moderator's buttons must be checked regardless of the state of the lockout.

Whenever a button/buzzer is detected as pushed, appropriate action is taken. If the unit is not locked-out and a player buzzes-in, the unit locks-out and illuminates the player's light. Since the pins for the lights alternate with the pins for the buzzers, a light can be illuminated by simply shifting the value of PINx left by one and assigning the resulting value to PORTx (where x is either A or B). In order to attract the moderator's attention, a tone is sounded corresponding to which team the player is on. Since the player must wait to be recognized by the moderator before answering the question, the moderator's timer is cleared to prevent a mistaken timeout. The pseudocode for this action is presented in listing 1.

```
if not locked-out
  if a player buzzed-in
    lockout unit
    illuminate light
    make sound
    clear moderator's timer
```

Listing 1: Psuedocode for handling players buzzing-in.

The moderator may reset the lockout or the timer at any time. The timer can be set to either five or ten seconds. Five seconds is the traditional amount of time to answer a question in Scholars' Bowl, but a recent addition to the rules allows for an additional five seconds (ten total) for mathematics questions requiring calculations [1]. Whenever the timer expires, the unit locks-out and sounds an audible tone. To unlock the unit, the reset button must be pressed, but the timer can be zeroed without locking-out the unit with the zero seconds button. The pseudocode code for these actions is presented in listing 2.

```
if moderator has pressed a button
  if reset
    unlock unit
    clear players' lights
    clear moderator's timer
  else if 0 seconds
    clear moderator's timer
  else if 5 seconds
    set moderator's timer for 5 seconds
    enable timeout audio
  else if 10 seconds
    set moderator's timer for 10 seconds
    enable timeout audio
```

Listing 2: Psuedocode for handling moderator's buttons.

The moderator's timer is interrupt driven to ensure its accuracy is at least as accurate as the 16MHz crystal oscillator. The crystal oscillator drives a 16-bit hardware timer (timer1) with a prescalar of sixty-four. With this prescalar, each clock cycle takes one two-hundred-and-fifty-thousandths of a second or four microseconds. A four microsecond clock cycle allows for many convenient interrupt periods using timer1's Clear Timer on Compare Match (CTC) Mode and setting the Output Compare Register (OCR1A) [4]. Equation 2 shows how to compute the OCR1A value for a desired interrupt period. This interrupt period needed to be small enough to allow for accurate timing but long enough to maintain a tight polling loop.

After some testing, an interrupt period of fifty milliseconds was chosen as a good balance.

$$f_{OCR1A} = \frac{f_{clk}}{2 * Pre * (1 + OCR1A)} \quad (1)$$

$$T_{OCR1A} = \frac{1}{f_{OCR1A}} \quad (2)$$

All audio is produced using timer2's waveform generator. Since timer2 can handle waveform generation entirely in hardware, there is no additional interrupts required to control the waveform. This has the benefit that the software only needs to enable and disable timer2 to toggle the audio on and off. The frequency of the audio can be set using equation 2.

2.3 Hardware Design

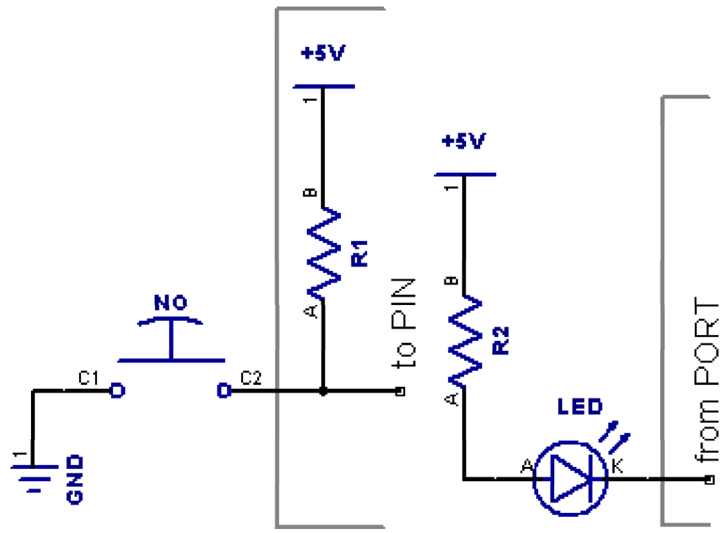
The key to the hardware design for the moderator and player units is simplicity. For the moderator unit, there are four main functions the hardware must serve: connecting to the player units, interacting with the moderator, displaying the timer, and sounding audible tones. Since the tones are produced using a simple piesoelectric buzzer, the only special connection required to the microcontroller is to one pin. The timer needs to be two digits long since both five and ten seconds are valid amounts of time to answer questions. Outputting both digits directly to a seven-segment display would require fourteen pins. The pin count for the display can be reduced to only eight if two 4511 integrated circuits were used to drive the seven-segment displays.

Interactions with both the moderator and the players require the largest number of pins. Each of the eight players has a button input and a light output for a total of sixteen pins. The moderator only requires four buttons: one to reset the lockout, one to zero the timer, one to set the timer for five seconds, and one to set the timer for ten seconds. All of the lights and buttons are active-low devices (see figure 3). The buttons would require pull-up resistors if the microcontroller does not have software-controlled internal pull-up resistors. Resistors are required for the LEDs to limit the current to below their maximum operating current.

The player units are connected to the moderator unit via RJ11-6 crossover cables. Each cable provides power and ground to the player units as well as two lines for the buttons and two lines for the LEDs. The LEDs are mounted on the player units whereas the buttons are connected to player unit via a small miniboard. This miniboard is placed inside a piece of PVC piping that serves as a player's buzzer and contains a small capacitor to reduce mechanical switch bounce.

3 Scoreboard and Timer Unit with Remote Control

The thrill of competition comes from competing against your opponents and the clock. As the final seconds of the match tick away, you look up at the scoreboard and reflect on the large deficit you just overcame. You now hope your slim lead will not disappear with the next question. The battle continues until the final buzzer sounds, and that is why you compete.



(a) Active-low pushbutton. (b) Active-low LED.

Figure 3: Hardware setups for (a) an active-low pushbutton and (b) an active-low LED.

3.1 High-level Design

Scores in Scholars' Bowl matches typically range between just under zero up to about four hundred points. As a result, each team's score needs to be represented by three digits. Each digit needs to be visible over a wide variety of viewing angles and distances. The viewing distance increases as the digit size increases, so the digits have to be fairly large to be seen over a reasonable distance. The hundreds digit doubles as the negative since should a team's score drops below zero during the course of a match.

A match consists of several rounds of varying lengths. The first round lasts ten minutes, and the second round lasts nine minutes. Between the first and second rounds there is a quick rapid-fire round which last sixty seconds for the receiving team and forty-five seconds for the opposing team [1]. Practice matches, however, have an arbitrary time limit, so it is necessary to allow the round timer to be set to any arbitrary time. Once this time expires, an audio tone sounds to signal the end of the round.

3.2 Software Design

Compared with the software design for the moderator and player units, the software design for the scoreboard and timer unit is more complex. This complexity stems from the scale of the unit with its ten digits. It is necessary to encode these ten digits efficiently using tens-complement, packed binary coded decimal (BCD; see 3.2.1). Even using packed BCD, multiple microcontrollers are needed to produce all ten digits. These microcontrollers need to communicate between each other (see 3.2.3). Further communication is needed between the scorekeeper (see 3.2.2).

3.2.1 Binary Coded Decimal (BCD) Arithmetic

Binary Coded Decimal (BCD) is an alternative means to encode decimal values within a computer. Instead of expressing decimal values as a sum of powers of two, BCD expresses each decimal digit separately as a sum of powers of two. Table 1 shows how to equivalently express decimal digits in binary, unpacked BCD, and packed BCD. Packed BCD is more space efficient than unpacked BCD since two decimal digits are represented per byte in packed BCD [5].

decimal	binary	unpacked BCD	packed BCD
0	0x00	0x0000	0x00
1	0x01	0x0001	0x01
2	0x02	0x0002	0x02
⇣	⇣	⇣	⇣
9	0x09	0x0009	0x09
10	0x0A	0x0100	0x10
11	0x0B	0x0101	0x11
⇣	⇣	⇣	⇣
15	0x0F	0x0105	0x15
16	0x10	0x0106	0x16
17	0x11	0x0107	0x17
⇣	⇣	⇣	⇣
97	0x61	0x0907	0x97
98	0x62	0x0908	0x98
99	0x63	0x0909	0x99

Table 1: Equivalent means to encode decimal values using binary, unpacked BCD, and packed BCD. All the non-decimal values are expressed in hexadecimal for convenience. Adapted from [5].

While packed BCD is space efficient, it is not necessarily computationally efficient since arithmetic has to be performed per nibble as opposed to per byte. However, performing arithmetic per nibble has the advantage that normalizing an invalid BCD digit also forces a carry. A BCD digit is invalid if its value is between ten and fifteen as a result of binary arithmetic. If the arithmetic operation is addition, a BCD digit is normalized by adding six. If the arithmetic operation is subtraction, a BCD digit is normalized by subtracting six [6]. This process is summarized in Listing 3.

```
peform binary arithmetic on score
foreach bcd_digit in score
  if bcd_digit is invalid
    adjust bcd_digit by +/- 6 to normalize and force carry;
```

Listing 3: Psuedocode for BCD arithmetic. Adapted from [5, 6].

Through repeated subtractions, it is possible for a team's score to become negative. Negative numbers can be expressed in several ways using packed BCD just as in binary. In binary, negative numbers can be expressed in sign-magnitude encoding or twos-complement encoding. Likewise, in packed BCD, negative numbers can be expressed in sign-magnitude encoding or tens-complement encoding (see table 2). In sign-magnitude encoding, the upper nibble is used as a sign-digit to encode if the number is positive (zero) or negative (nine). This zero/nine sign encoding dramatically reduces the range of packed BCD values as well as introduces the problem of a double zero–one positive, the other negative [5].

Both of these problems are addressed by tens-complement encoding which recenters the range of packed BCD values around a single zero. Tens-complement arithmetic is completely analogous to twos-complement arithmetic [5, 6]. In twos-complement arithmetic, negating a number is performed by subtracting it from the largest unsigned binary value and adding one to the result. Since the largest unsigned binary value represents negative one when signed, the negating process can be expressed as subtract the value from negative one and

(a) Unsigned BCD

packed BCD	decimal
00	0
01	1
02	2
↓	↓
09	9
10	10
11	11
12	12
↓	↓
97	97
98	98
99	99

(b) Sign-magitude BCD

packed BCD	decimal
00	0
01	1
02	2
↓	↓
07	7
08	8
09	9
90	-0
91	-1
92	-2
↓	↓
97	-7
98	-8
99	-9

(c) Tens-complement BCD

packed BCD	decimal
00	0
01	1
02	2
↓	↓
47	47
48	48
49	49
50	-50
51	-49
52	-48
↓	↓
97	-3
98	-2
99	-1

(d) Packed BCD ranges

		Unsigned	Sign-magnitude	Tens-complement
2-digit	Min	0	-9	-50
	Max	99	9	49
3-digit	Min	0	-99	-500
	Max	999	99	499
4-digit	Min	0	-999	-5000
	Max	9999	999	4999

Table 2: Decimal equivalence of unsigned, sign-magnitude, and tens-complement packed BCD and their ranges. Adapted from [5].

add one Thanks to a convenient property of binary, two-complement simplifies to a single logic operation and a single arithmetic operation. Similarly, in tens-complement arithmetic, negating a number is performed by subtracting it from the largest unsigned BCD value (a nine in every digit position) and adding one to the result. Unfortunately, there is no quick manner in which to compute the tens-complement of a BCD value. Listing 4 shows the pseudocode for twos-complement and tens-complement.

```
twos-complement:
  subtract from largest unsigned binary value and 1
  i.e. y = ((0xF..F - x) + 1);
  or equivalently, invert bits and add 1
  i.e. y = ((~x) + 1);
tens-complement:
  subtract from largest unsigned BCD value and add 1
  for 2-digits, 0x99; for 3-digits, 0x999; etc.
  i.e. y = ((0x9..9 - x) + 1);
```

Listing 4: Pseudocode for Twos- and Tens-complement. Tens-complement pseudocode adapted from [6]

3.2.2 RC5 Infrared Remote Control Protocol

To avoid running a large bundle of cable to the scoreboard and timer unit, control for the unit is by remote control using Philips' RC5 infrared remote control protocol. The RC5 protocol uses a bi-phase Manchester code. In a Manchester code, a logical bit is split into two phases where the two phases are complements of eachother. These complemented phases produce a falling edge for a logic zero and a rising edge for a logical one. When transmitting a high phase, the RC5 protocol modulates the signal at 36kHz to distinguish it from background infrared noise [7, 8, 9].

An RC5 command frame is fourteen bits long as shown in figure 4. The first two bits are start bits which are always logical one. The third bit is a control bit which toggles each time a button is pressed. After the control bit comes five system bits and six command bits. The address bits are used to distinguish commands intended for different devices. The command bits contain one of sixty-four commands which vary by device [7, 8, 9].

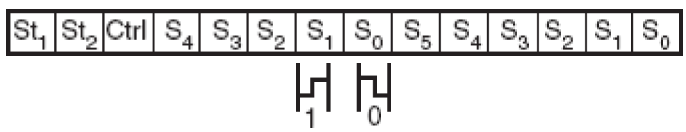


Figure 4: The RC5 infrared protocol frame. From [7, 8].

The software for both transmitting and receiving commands using an RC5 frame is derived from Atmel AVR Application Note 415 [8] and 410 [7], respectively. As in [7], the project requires a 38 kHz modulated signal to be decoded by the demodulator. The demodulated signal is read by the master ATmega32 which forwards commands to two slave ATtiny26(L)s.

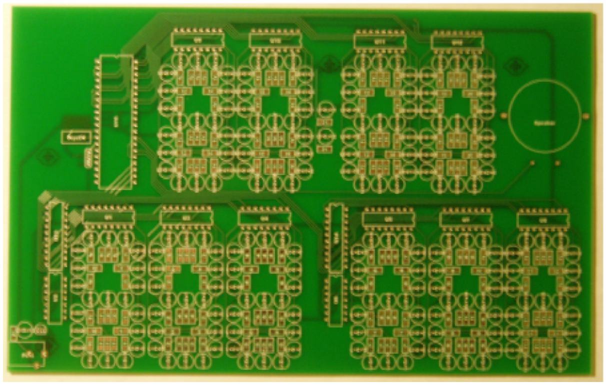
3.2.3 Inter-Microcontroller Communication

Communication between the ATmega32 and the two ATtiny26(L)s is very simple. The ATtiny26(L)s poll the current command from the ATmega32. If the new command is different from the previous command, a new command has been received and is executed. If not, a new command has not been received. This polling scheme has the advantage that the inter-microcontroller communication is simply one-way without any need for an acknowledgment. A disadvantage of this polling scheme is that a command that needs to be executed more than once must be issued with an intervening null command. This null command can only be issued by the ATmega32 if it receives two distinct commands from the remote control. Therefore, the scorekeeper must press the same button multiple times for multiple responses.

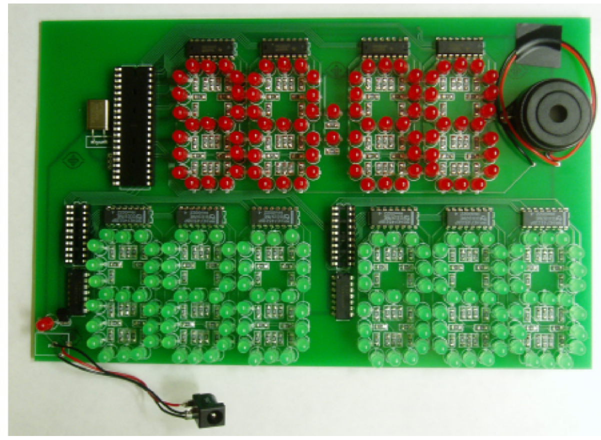
3.3 Hardware Design

The first incarnation of the scoreboard and timer unit (figure 5) has several problems. The greatest of these problems is power consumption. Each of the two-hundred-and-thirteen LEDs requires twenty milliamps of current for a total current requirement of 4.26 amps. Neither the power supply nor the regulator were rated to supply this much current. To supply the necessary current, a stand-alone, self-regulated power supply was purchased which can source upto six amps at five volts.

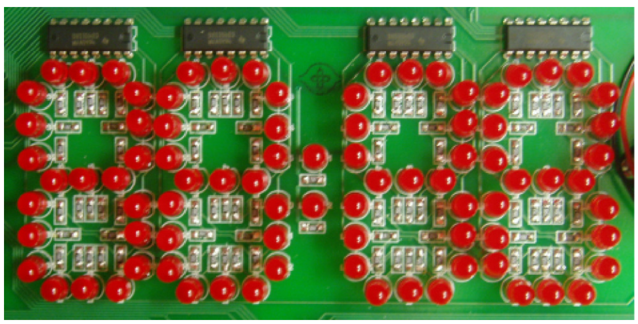
In addition to the supply problems, long power and ground traces caused the supply voltages to drift towards eachother due to resistive losses. These losses were significant enough that ICs connected to the far end of the trace would receive only a brownout-level voltage. By rewiring and thickening the supply traces,



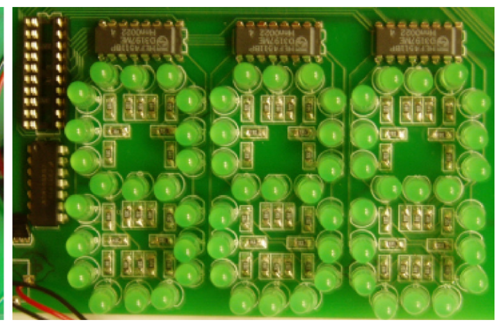
(a) The pcb.



(b) The finished product.



(c) A closeup of the timer.



(d) A closeup of a team score.

Figure 5: Photos of the first version of the scoreboard and timer unit.

the resistivity and the length of the traces should decrease. Currently, the revised scoreboard and timer unit is being fabricated, but the completed printed circuit board (PCB) will not arrive back from the fab until after this report goes to printing.

While much of the hardware for the scoreboard and timer unit is pretty self-explanatory, generating a negative sign for the scores requires some additional hardware. Each digit of the score is driven using a 4511. The 4511 has an input (_BI_) which blanks the display when driven low by an inverted control signal from the ATtiny26L. When the control signal (neg) is high, the 4511 blanks the display, and the control signal drives the g-segment of the hundreds digit. The logic for both these operations can be found in equations 3 and 4.

$$\_BI\_ = \overline{neg \oplus neg} \tag{3}$$

$$g_{out} = g_{in} \oplus neg \tag{4}$$

$$\overline{g_{out}} = \overline{g_{in} \oplus neg}$$

$$g_{out} = \overline{\overline{g_{in} \oplus neg}} \tag{5}$$

This logic can easily be implemented using a 4001 Quad-NOR integrated circuit. Since NOR is an inverting logic operation, equation 4 has to be modified slightly. The modified logic is shown in equation 5 and requires two NOR gate. The first NOR gate performs the inverted logic, and the second NOR gate is wired as an inverter to produced the desired g_{out} from equation 4. In total, only three of the four NOR gates on the 4001 are actually used.

4 Testing

Since the printed circuit boards are still being fabricated, there has been no field testing of the complete system. However, individual components have been tested using several ATSTK500 development boards as seen in figure 6. Working with the ATSTK500 development boards allowed for software debugging and tuning. Most aspects of the software were able to be tested in this manner. The notable exception was the remote control code due to the poor signal quality of a 38kHz through a standard breadboard.

Figure 6: The author surrounded by equipment for testing purposes. Photo courtesy of Bruce Land.

Several aspects of the hardware design had to be constructed on a breadboard for testing purposes if they could not be tested using the ATSTK500. The ATSTK500 has eight LEDs and eight normally-open pushbuttons which can substitute for any LED or pushbutton throughout the design, but the ATSTK500 does not have a piezo speaker or a seven-segment display. A piezo speakers and seven-segment displays had to be wired on a breadboard in order to test both the audio and the functionality of the 4511 BCD to seven-segment decoder.

5 Conclusion

While the project is currently incomplete as this report goes to printing, it will be completed before graduation 29 May 2006[3]. The completed project will be fully packaged and field tested prior to being donated to Colchester High School's Scholars' Bowl team. The Scholars' Bowl season is over for the year, but this project should be beneficial for the team for both competition and practices. Since practices are typically attended by more than eight people (alumni and faculty love to join in), expanding the moderator unit to allow for additional moderator and/or player units to be daisychained would allow for more players to have a real buzzer when attending practices.

6 Acknowledgements

The author would like to acknowledge the following people and companies for their support with this project:

- Bruce Land for his support as advisor, boss, and friend.
- Advance Circuits for providing low cost, good quality PCB fabrication to students without massive limitations to their designs. Please visit the Advance Circuits website at http://www.4pcb.com.
- Stanislav Ruev of Novarm for his technical support and for generously donating a full version of DipTrace Professional PCB-Design Tool. Please visit the DipTrace website at http://www.diptrace.com.

[3]Please visit `http://instruct1.cit.cornell.edu/courses/eceprojectsland/STUDENTPROJ/2005to2006/rw88/` for complete and uptodate documentation, source code, schematics, and layouts in full color.

References

[1] Vermont-NEA Scholars' Bowl. http://members.aol.com/kcommo/vtsbowl/index.htm. Maintained by K. Commo. Last updated 30 April 2006. Accessed 1 May 2006.

[2] Atmel Corporation. Atmel Mega164/324/644 Advance Information. 2953A-AVR-06/05.

[3] Atmel Corporation. http://www.atmel.com/. Maintained by Atmel Corporation. Last updated 26 April 2006. Accessed 1 May 2006.

[4] Atmel Corporation. Atmel Mega32(L) datasheet. 2503H-AVR-03/05.

[5] DIY Calculator: BCD Instructions. Available from http://www.diycalculator.com/docs/dload-bcd-instructions.pdf. Accessed 1 May 2005.

[6] Jones on BCD Arithmetic. http://www.cs.uiowa.edu/ jones/bcd/bcd.html. Maintained by D. W. Jones. Last updated 2002. Accessed 1 May 2006.

[7] Atmel Corporation. AVR410: RC5 IR Remote Control Receiver. Rev. 1473B-AVR-05/02.

[8] Atmel Corporation. AVR415: RC5 IR Remote Control Transmitter. Rev. 2534A-AVR-05/03.

[9] SB-Projects: IR remote control: Philips RC-5. http://www.xs4all.nl/ sbp/knowledge/ir/rc5.htm. Maintained by S. Bergmans. Last updated 14 October 2005. Accessed 1 May 2006.

[10] Atmel Corporation. Atmel Tiny26(L) datasheet. Rev. 1477G-AVR-03/05.

[11] Atmel Corportaion. Atmel Tiny28(L) datasheet. Rev. 1062G-AVR-01/06.

A Complete Source Code

Complete source code is available for download from `http://instruct1.cit.cornell.edu/courses/eceprojectsland/STUDENTPROJ/2005to2006/rw88/code/` and is written using CodeVisionAVR. All of the source code is published under the GNU General Public License. Please consult the above URL for any changes to the source code since its publication in early May, 2006.

A.1 ATmega32 Source for the Moderator & Player Units

This source code is located at `http://instruct1.cit.cornell.edu/courses/eceprojectsland/STUDENTPROJ/2005to2006/rw88/code/mod_m32.c`.

```
/* Richard West '05
 * 2006 Master of Engineering Candidate
 * Electrical and Computer Engineering
 *
 * Master of Engineering Design Project
 *
 * COPYRIGHT & LICENSE:
 * Copyright (C) 2006 Richard West
 *
 * From http://www.gnu.org/copyleft/gpl.html:
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * as published by the Free Software Foundation; either version 2
 * of the License, or (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA
 * 02110-1301, USA.
 *
 * DESCRIPTION:
 *
 */

#include <mega32.h>

// audio definitions
#define NONE    0x00
#define TIME    0x01
#define TEAMA   0x02
#define TEAMB   0x03
#define PWM_OFF 0x00 // turn off pwm
#define PWM_ON  0x1C // CTC with toggle on match
#define F_TIME  0x18 // 0x18 = 5000 Hz
#define F_TEAMA 0x31 // 0x31 = 2500 Hz
#define F_TEAMB 0x7C // 0x7C = 1000 Hz

// time definitions
#define STOP 0x00
#define RUN  0x01

signed char i, j;
unsigned char wait, time, time_status;
unsigned char lockout;
unsigned char audio;

interrupt [TIM1_COMPA] tim1_compa_isr(void) {
  if (0 < wait) wait--;
```

```
  else if (RUN == time_status) {
    // reset wait for another second and update time
    wait = 20;
    time--;
    PORTC = time;
  }
}

void main(void) {
  // initialize timer1
  TIMSK  = 0x10;  // enable timer1 compareA interrupt
  TCCR1B = 0x0B;  // clear on match A, set prescalar to 64
  OCR1A  = 12500; // interrupt every 1/20 seconds

  // initialize timer2 as HW PWM
  TCCR2 = PWM_OFF;
  OCR2  = F_TEAMA;

  // initialize I/O
  DDRA  = 0x55; // buzzers on odds, lights on evens
  PORTA = 0xFF; // set pull-ups for buzzers and turn off lights
  DDRB  = 0x55; // buzzers on odds, lights on evens
  PORTB = 0xFF; // set pull-ups for buzzers and turn off lights
  DDRC  = 0xFF; // all outputs for timer
  PORTC = 0x00; // set time display to 00
  DDRD  = 0x80; // moderator buttons on 3..0, audio on 7
  PORTD = 0x0F; // pull-ups for buttons

  // initialize variables
  wait        = 0x00;
  time        = 0x00;
  time_status = STOP;
  lockout     = 0x00;
  audio       = NONE;

  // enable interrupts
  #asm("sei");

  // start-up test
  // cycle through team A
  TCCR2 = PWM_ON;
  OCR2  = F_TEAMA;
  j     = 0xBF;
  PORTA = j;
  for (i = 0; i < 4; i++) {
    // wait 0.2 second
    wait  = 4;
    while(0 != wait);
    TCCR2 = PWM_OFF;
    j   >>= 2;
    PORTA = j;
  }
  PORTA = 0xFF;

  // cycle through team B
  TCCR2 = PWM_ON;
  OCR2  = F_TEAMB;
  j     = 0xBF;
  PORTB = j;
  for (i = 0; i < 4; i++) {
    // wait 0.2 second
    wait = 4;
    while(0 != wait);
    TCCR2 = PWM_OFF;
    j   >>= 2;
    PORTB = j;
  }
  PORTB = 0xFF;
```

```
    // test time expired audio for 0.2 seconds
    TCCR2 = PWM_ON;
    OCR2  = F_TIME;
    wait  = 4;
    while(0 != wait);

    TCCR2 = PWM_OFF;
    // done start-up test

    while(1) {
      // check timer
      if ((0x00 == time) && (RUN == time_status)) {
        time_status = STOP;
        lockout     = 0x01;
        audio       = TIME;
      }

      // if not locked out, check the buzzers
      if (!lockout) {
        if (0xFF != PINA) {
          // turn on light, lock out,
          // and enable the audio
          PORTA       = PINA >> 1;
          PORTC       = 0x00;
          time        = 0x00;
          time_status = STOP;
          lockout     = 0x01;
          audio       = TEAMA;
        }
        else if (0xFF != PINB) {
          // turn on light, lock out,
          // and enable the audio
          PORTB       = PINB >> 1;
          PORTC       = 0x00;
          time        = 0x00;
          time_status = STOP;
          lockout     = 0x01;
          audio       = TEAMB;
        }
      }

      // check moderator's buttons
      if (0xFF != PIND) {
        if (0 == PIND.0) {
          // reset lockout system
          PORTA       = 0xFF;
          PORTB       = 0xFF;
          PORTC       = 0x00;
          lockout     = 0x00;
          time        = 0x00;
          time_status = STOP;
          audio       = NONE;
        }
        else if (0 == PIND.1) {
          // zero timer
          PORTC       = 0x00;
          wait        = 0x00;
          time        = 0x00;
          time_status = STOP;
          audio       = NONE;
        }
        else if (0 == PIND.2) {
          // set timer for five seconds
          PORTC       = 0x05;
          wait        = 0x00;
          time        = 5;
          time_status = RUN;
```

```
      }
      else if (0 == PIND.3) {
        // set timer for ten seconds
        PORTC       = 0x10;
        wait        = 0x00;
        time        = 10;
        time_status = RUN;
      }
    }

    // enable audio if needed
    if (NONE != audio) {
      if (TIME == audio) {
        TCCR2 = PWM_ON;
        OCR2  = F_TIME;
      }
      else if (TEAMA == audio) {
        TCCR2 = PWM_ON;
        OCR2  = F_TEAMA;
      }
      else if (TEAMB == audio) {
        TCCR2 = PWM_ON;
        OCR2  = F_TEAMB;
      }
      audio = NONE;

      // wait 0.2 seconds and turn audio off
      wait  = 4;
      while(0 != wait);
      TCCR2 = PWM_OFF;
    }
  }
}
```

A.2 ATtiny28(L) Assembly for the Remote Control Unit

This source code is located at `http://instruct1.cit.cornell.edu/courses/eceprojectsland/STUDENTPROJ/2005to2006/rw88/code/remote.asm` and is adapted from Atmel's application note AVR415: RC5 IR Remote Control Transmitter.

```
.include <tn28def.inc>
;********************************************************************************************
;*
;* Constants
;*
;********************************************************************************************

.equ pulses       = 256 - 32 ; 256 - pulses per bit half
.equ numberofbits = 30       ; 2 * number of bits to transfer + 1

.equ invPA2ovf      = (1<<3) | 2
.equ activePA2ovf   = (2<<3) | 2
.equ inactivePA2ovf = (3<<3) | 2

.equ noMOD       = 0          ; MODCR value for no output
.equ F38KHz_D25 = (2<<3) | 3 ; MODCR value for 38KHz output, 25% dutycycle
.equ F38KHz_D33 = (3<<3) | 2 ; MODCR value for 38KHz output, 33% dutycycle
.equ F38KHz_D50 = (5<<3) | 1 ; MODCR value for 38KHz output, 50% dutycycle
.equ F38KHz_D67 = (3<<3) | 4 ; MODCR value for 38KHz output, 67% dutycycle
.equ F38KHz_D75 = (2<<3) | 5 ; MODCR value for 38KHz output, 75% dutycycle

.equ FAULT = 0xFF


;********************************************************************************************
;*
;* Register definitions
```

```
;*
;*********************************************************************************************

.def select  = R0 ; Register to hold MSB of transmission
.def command = R1 ; Register to hold LSB of transmission
.def zero    = R2 ; Register preset to zero

; -- This line will generate a warning that R27 is already
; -- defined as XH. The warning can be ignored.
.def allhigh = R27 ; Register preset to 0xFF

.def temp        = R16 ; Temporary register
.def toggle      = R17 ; Register to contain toggle bit value
.def toggle_mask = R18 ; Register to contain toggle mask
.def row_scan    = R19 ; Scan value Row
.def row_saved   = R20 ; Saved value Row
.def col_scan    = R21 ; Scan value Col
.def col_saved   = R22 ; Saved value Col
.def ptr         = R23 ; Pointer value used with the lookup table
.def old_ptr     = R24 ; Last lookup table pointer value (needed to calculate toggle bit)
.def keys        = R25 ; Value used to count number of pressed keys

; -- This line will generate a warning that R26 is already
; -- defined as XL. The warning can be ignored.
.def bitnumb = R26 ; Register which contains the number of bits to be transfered


;*********************************************************************************************
;*
;* Interrupt Vectors
;*
;*********************************************************************************************

;.cseg
.org 0x0000
  rjmp reset ; Reset vector

.org LLINTaddr ; Low level Interrupt Vector Address
  rjmp send

.org OVF0addr
  rjmp bitfinished


;***** RC5 Lookup Table;
;*
;* Format of data should be in binary
;* 11XSSSSSCCCCCC11
;* Here the command word is shown as 0xC003 | (SYSCODE << 8) | (command << 2)
;*
;***********************************************************************************

.equ SCORE = 21 ; The system code for Scoreboard

lookup:
; Column 0
.dw 0xC003 | (SCORE << 8) | (2 << 2) ; Start timer
.dw 0xC003 | (SCORE << 8) | (3 << 2) ; Stop timer
.dw 0xC003 | (SCORE << 8) | (4 << 2) ; Add 10 minutes
.dw 0xC003 | (SCORE << 8) | (5 << 2) ; Sub 10 minutes
.dw 0xC003 | (SCORE << 8) | (6 << 2) ; Add 1 minute
.dw 0xC003 | (SCORE << 8) | (7 << 2) ; Sub 1 minute
.dw 0xC003 | (SCORE << 8) | (1 << 2) ; Reset timer
.dw 0xC003 | (SCORE << 8) | (8 << 2) ; Add 10 seconds

; Column 1
.dw 0xC003 | (SCORE << 8) | (9 << 2)  ; Sub 10 seconds
.dw 0xC003 | (SCORE << 8) | (10 << 2) ; Add 1 second
```

```
.dw 0xC003 | (SCORE << 8) | (11 << 2) ; Sub 1 second
.dw 0xC003 | (SCORE << 8) | (12 << 2) ; Save 0
.dw 0xC003 | (SCORE << 8) | (13 << 2) ; Load 0
.dw 0xC003 | (SCORE << 8) | (14 << 2) ; Save 1
.dw 0xC003 | (SCORE << 8) | (15 << 2) ; Load 1
.dw 0xC003 | (SCORE << 8) | (63 << 2) ; Test unit

; Column 2
.dw 0xC003 | (SCORE << 8) | (20 << 2) ; Add 5 to A
.dw 0xC003 | (SCORE << 8) | (21 << 2) ; Sub 5 from A
.dw 0xC003 | (SCORE << 8) | (16 << 2) ; Zero A
.dw 0xC003 | (SCORE << 8) | (17 << 2) ; Add 1 to A
.dw 0xC003 | (SCORE << 8) | (18 << 2) ; Sub 1 from A
.dw 0xC003 | (SCORE << 8) | (36 << 2) ; Add 5 to B
.dw 0xC003 | (SCORE << 8) | (37 << 2) ; Sub 5 from B
.dw 0xC003 | (SCORE << 8) | (32 << 2) ; Zero B

; Column 3
.dw 0xC003 | (SCORE << 8) | (33 << 2) ; Add 1 to B
.dw 0xC003 | (SCORE << 8) | (34 << 2) ; Sub 1 from B
.dw 0xC003 | (SCORE << 8) | (0 << 2)  ; Ext 1
.dw 0xC003 | (SCORE << 8) | (0 << 2)  ; Ext 2
.dw 0xC003 | (SCORE << 8) | (0 << 2)  ; Ext 3
.dw 0xC003 | (SCORE << 8) | (0 << 2)  ; Ext 4
.dw 0xC003 | (SCORE << 8) | (0 << 2)  ; Ext 5
.dw 0xC003 | (SCORE << 8) | (0 << 2)  ; Ext 6


;*** Reset handler ***************************************************
;*
;* Executed on reset
;*
;*********************************************************************

reset:

;*** Must set up Stack to be able to run in emulator..
; ldi r16,0x5F+3*2 ; set up a three level deep stack
; out 0x3d,r16

  clr zero    ; Initialize "zero" register
  ser allhigh ; Initialize "allhigh" register
  clr bitnumb ; Initialize bitcounter register

  out DDRD,allhigh ; Set Port D as output

  out PORTA,allhigh ; No IR output, all other PORTA pins pulled high
  sbi PACR,PA2HC    ; Enable high current driver

  sbi ICR,TOIE0 ; Enable Timer Overflow

  ldi temp,F38KHz_D50 ; Set up hardware modulator
  out MODCR,temp

  ldi toggle_mask, 0b00100000 ; Bit 5 is the toggle bit


;*** Main loop *******************************************************
;*
;* Code executed after each interrupt
;*
;*********************************************************************

main_loop:
  cli            ; Disable interrrupts
  tst  bitnumb   ; Is transmission finished
  breq Pwdn_mode ; Transmission not complete
  ldi  temp,(1<<PLUPB)|(1<<SE) ; Enable IDLE mode
```

```

  rjmp pwdn_enable ; Enter IDLE mode

Pwdn_mode:
  sbi ICR,LLIE ; Enable low level interrupt when transmission complete
  ldi temp,(1<<PLUPB)|(1<<SE)|(1<<SM) ; Enable PowerDown mode

pwdn_enable:
  out MCUCS,temp

  sei   ; Enable interrupts
  sleep ; Enter powerdown

  rjmp main_loop ; Loop to top of main loop each interrupt


;*** Low level interrupt handler ************************************
;*
;* Executed on low level interrupt (key pressed, no transmission)
;*
;********************************************************************


send:
  cbi ICR,LLIE ; Disable low level interrupt during transmission

;*** find_command ***************************************************
;*
;* Scans keyboard and stores the correct word to transfer in the
;* R1:R0 register pair.
;*
;* Registers used : temp, row_scan, col_scan
;* Flags used : C
;*
;* Format:
;*                R1                         R0
;*
;*        St St T0 S4 S3 S2 S1 S0 - C5 C4 C3 C2 C1 C0 x1 x0
;*        ----- -- --------------   ----------------- -----
;*          |   |  |System code     |Command          |Unused
;*          |   |
;*          |   +- Toggle Bit
;*         +----- Start Bits
;*
;********************************************************************
  ldi keys,193   ; Set keys to 193, add (8*8-1) | 0xFF = 0 for valid input
  ldi col_scan,1 ; Initialize scan of first Column

cont_col_scan:
  out DDRD,col_scan ; Set One Column to output, the rest tristated.
  out PORTD,zero    ; Write "0" to the output, tristate all other lines
  nop
  nop
  in  row_scan,PINB ; Read response from input pins

  cpi  row_scan,0xff ; Any key pressed?
  brne key_pressed   ; If yes then branch to count routine

  subi keys,-8 ; if no keys pressed, add 8 to keys

ret_key_pressed:

  out PORTD,allhigh ; Pull all input pins high
  out DDRD,allhigh

  lsl  col_scan      ; Check next line on the keyboard
  brcc cont_col_scan ; If bit is not shifted through, continue scan

```

```
  ldi  ptr,fault ; Initialize to error value.
  tst  keys      ; One, and only one zero should have been found in row scan.
  brne ch_end    ; If number of ones found != 63 then return with faulty ptr

fl2:
  inc  ptr
  lsr  row_saved
  brcs fl2 ; until: ptr contains binary value of "row"

fl3:
  sbrc col_saved,0 ; test if lsb is one -> current column contains the pressed button
  rjmp fcont       ; If correct column, value calculated
  subi ptr, -8     ; If not correct column, add 8 to pointer value
  lsr  col_saved   ; Test next column
  rjmp fl3

fcont:
  ldi ZL,low(lookup)*2
  ldi ZH,high(lookup)*2
  lsl ptr    ; Adjust pointer value to compensate for byte/word wide addressing
  add ZL,ptr ; Add pointer value to lookup table base address
  adc ZH,zero

  lpm            ; Load low byte in transmission (last byte to transfer)
  mov command,r0 ; Move low byte to correct storage position
  inc ZL         ; Select next byte in lookup table
  lpm            ; Load high byte in transmission (first byte to transfer)

  cp   old_ptr,ptr        ; is it a new command?
  breq same_ptr           ; Do not invert togglebit if same command
  eor  toggle,toggle_mask ; Invert toggle bit

same_ptr:
  bst toggle,5 ; Copy Toggle bit to T-flag
  bld select,5 ; Insert Toggle bit into syscode byte from T-flag

;***************************************************************************
;*
;* Code to start a transmit sequence
;* Transmits 14 bits, bit 1 in input command must be 1 to ensure
;* glitch free operation
;*
;***************************************************************************
  ldi temp,inactivePA2ovf ; Ensure no output at start of transmission
  out TCCR0,temp          ; Inserts a dummy inactive period of 288*38KHz
                          ; cycles at start of each transmission

  ldi bitnumb,numberofbits ; Initialize bitcounter

ch_end:
  out DDRD,allhigh ; Set keyboard in "detect" mode
  out PORTD,zero   ; Restore passive scan pattern on keyboard

  mov old_ptr,ptr ; Save ptr value,

  ret ; Return from interrupt

;***************************************************************************
;*
;* Code to store away keypad data and find number of pressed keys in this column
;*
;***************************************************************************

key_pressed:
  mov col_saved,col_scan ; Store Column value
  mov row_saved,row_scan ; Store Row value

cont_row_scan:
```

```
  sbrc row_scan,0      ; Bit 0 = 1
  inc  keys            ; Increase for each logical one found
  lsr  row_scan        ; Rotate row value one plase to the left
  brne cont_row_scan   ; Continue until all one's in row is gone
  rjmp ret_key_pressed

;*** transmit ******************************************************
;*
;* Sends the complete syscode and command stored in the register pair
;* select:command.
;*
;*******************************************************************

bitfinished:
  dec  bitnumb  ; Decrease bitcounter
  breq finished ; If all bits have been transmitted, end transmission

  ldi temp,pulses ; Reload timer
  out TCNT0,temp

  sbrc bitnumb,0 ; Is this the second half of this bit transfer?
  rjmp firsthalf

  ldi  temp,invPA2ovf ; If second half, Load Invert PA2 On Next Compare
  rjmp intfin

finished:
  sbic TCCR0,CS02   ; Was last interrupt longwait?
  rjmp end_longwait ; Signal end of transmit

  ldi bitnumb,1 ; Load bitcounter to ensure correct operation

  ldi temp,207   ; Preload counter to give 50176 cycles delay
  out TCNT0,temp ; The sending of the next byte will give an extra delay
                 ; of 3456 cycles, giving a minimum of 53632 cycles between
                 ; transmissions
  ldi temp,5 ; Set prescaler to CK/1024

  rjmp intfin

end_longwait:
  out TCCR0,zero ; Disable timer after command and longwait
                 ; finished
  in   temp,PINB   ; Check keybord for to ensure correct operation of toggle bit
  cpi  temp,0xFF
  brne NotSetFault ; If keys pressed, proceed

  ldi old_ptr,fault ; If no keys pressed, load pointer with error value to ensure correct
                    ; operation of the toggle bit

NotSetFault:
  ret ; Return from interrupt, transmission complete

firsthalf:
  lsl command ; Move output bit to carry
  rol select

  brcc outlow ; Next bit is a low value

  ldi  temp,inactivePA2ovf ; Set next interval to output signal
  rjmp intfin

outlow:
  ldi temp,activePA2ovf ; Set next interval to output no signal

intfin:          ; Return from interrupt, not finished transmission
  out TCCR0,temp ; Set moulator options/timer prescaler
  ret
```

A.3 ATmega32 Source for the Scoreboard & Timer Unit

This source code is located at `http://instruct1.cit.cornell.edu/courses/eceprojectsland/STUDENTPROJ/2005to2006/rw88/code/sb_m32.c`. A portion of this source code is adapted from Atmel's application note AVR410: RC5 IR Remote Control Receiver.

```
/* Richard West '05
 * 2006 Master of Engineering Candidate
 * Electrical and Computer Engineering
 *
 * Master of Engineering Design Project
 *
 * COPYRIGHT & LICENSE:
 * Copyright (C) 2006 Richard West
 *
 * From http://www.gnu.org/copyleft/gpl.html:
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * as published by the Free Software Foundation; either version 2
 * of the License, or (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA
 * 02110-1301, USA.
 *
 * DESCRIPTION:
 * Source code for the ATmega32 used to control the timer and two
 * ATtiny26(L) ICs as part of the larger scoreboard and timer unit.
 * The time is maintained in packed BCD format in the following
 * order: 10 minutes digit in the upper nibble of PORTA, 1 minute
 * digit in the lower nibble of PORTA, 10 seconds digit in the
 * upper nibble of PORTB, and 1 second digit in the lower nibble of
 * PORTB.  Each digit serves as the input to a 4511 BCD to Seven
 * Segment IC (see schematics).
 *
 * PORTC is used to control the rest of the scoreboard and timer
 * unit by issuing commands to the two ATtiny26(L) ICs (see schematic).
 * The control for the piezo siren is also operated by a control signal
 * on PORTC.  All user control input is received by the 38kHz infrared
 * demodulator on PIND.0.
 *
 * See the scoreboard and timer schematics and the ATmega32 source
 * for more information.
 */

#include <mega32.h>

// remote control address
#define IR_ADDRESS   0b10101

// remote control commands
#define NO_COMMAND   0b000000
// timer commands
#define RESET_TIME   0b000001
#define START_TIME   0b000010
#define PAUSE_TIME   0b000011
#define ADD10_MIN    0b000100
#define SUB10_MIN    0b000101
#define ADD01_MIN    0b000110
#define SUB01_MIN    0b000111
#define ADD10_SEC    0b001000
```

```
#define SUB10_SEC    0b001001
#define ADD01_SEC    0b001010
#define SUB01_SEC    0b001011
#define SAVE_TIME0   0b001100
#define RECALL_TIME0 0b001101
#define SAVE_TIME1   0b001110
#define RECALL_TIME1 0b001111
// team A commands
#define A_RESET      0b010000
#define A_ADD1       0b010001
#define A_SUB1       0b010010
#define A_ADD5       0b010100
#define A_SUB5       0b010101
// team B commands
#define B_RESET      0b100000
#define B_ADD1       0b100001
#define B_SUB1       0b100010
#define B_ADD5       0b100100
#define B_SUB5       0b100101
// test unit
#define TEST_UNIT    0b111111

// command definitions
#define RESET    0b000
#define ADD1     0b001
#define SUB1     0b010
#define ADD5     0b011
#define SUB5     0b100
#define TEST_NEG 0b101
#define TEST_POS 0b110
#define NO_CMD   0b111

// timer_status definitions
#define PAUSE 0x00
#define START 0x01

#pragma regalloc-
// Time field definitions
typedef struct {
  unsigned char sec; // Port B
  unsigned char min; // Port A
} time_struct;

union {
  unsigned int full;
  time_struct parts;
} time;

unsigned char wait;
unsigned char timer_status, dec_timer;
#pragma regalloc+

// IR Demodulator Definitions
// note: the 38 kHz demodulator is inverting
#define LOW  1
#define HIGH 0

/*#pragma regalloc-
// RC5 field definitions
typedef struct {
  unsigned int command : 6;
  unsigned int address : 5;
  unsigned int control : 1;
  unsigned int start   : 2;
  unsigned int junk    : 2;
} frame_struct;

union {
```

```
  unsigned int full;
  frame_struct bits;
} frame;

unsigned char cmd_wait;
unsigned char cmd_bit_done, cmd_bit_value, cmd_bit_count;
unsigned char cmd_frame_done;
#pragma regalloc+*/

#pragma regalloc-
// Command bitfield definitions
typedef struct {
  unsigned char cmdA : 3; // 2..0
  unsigned char cmdB : 3; // 5..3
  unsigned char buzz : 1; // 6
  unsigned char junk : 1; // 7 (not used)
} cmd_struct;

union {
  unsigned char full;
  cmd_struct bits;
} cmd;

unsigned char cmd_received, new_cmd;
#pragma regalloc+

// eeprom entries for saved times
// saved_time[0] = 09:00
// saved_time[1] = 00:45
eeprom int saved_time[2] = {0x0900, 0x0045};

//#define DEBUG
#ifdef DEBUG
unsigned char i;
#endif

// Timer1 Compare Interrupt Service Routine
interrupt [TIM1_COMPA] tim1_compa_isr() {
  if (0 < wait) wait--;
  else if (timer_status == START) {
    // reset wait for another second
    wait      = 20;
    dec_timer = 0x01;
  }

  /*if (cmd_wait > 0) cmd_wait--;
  else {
    // read next bit from PIND.0

    // state machine here
  }*/
}

void main(void) {
  // initialize timer1
  TIMSK  = 0x10;  // enable timer1 compareA interrupt
  TCCR1B = 0x0B;  // clear on match A, set prescalar to 64
  OCR1A  = 12500; // interrupt every 1/20 seconds

  // initialize I/O
  DDRA  = 0xFF; // all output
  DDRB  = 0xFF; // all output
  DDRC  = 0xFF; // all output (C.7 not used)
  DDRD  = 0x00; // D.0 is input (D.6-1 not used)
  PORTA = 0x00;
  PORTB = 0x00;
  PORTC = 0xFF;

```

```
    // initialize variables
    time.full      = 0x0000;
    wait           = 0x00;
    timer_status   = PAUSE;
    dec_timer      = 0x00;
    /*frame.full     = 0x0000;
    cmd_wait       = 0x00;
    cmd_bit_done   = 0x00;
    cmd_bit_value  = 0x00;
    cmd_bit_count  = 0x00;
    cmd_frame_done = 0x00;
    cmd.full       = 0x00;*/
    cmd_received   = 0x01;
    new_cmd        = TEST_UNIT;

    // enable interrupts
    #asm("sei");

    while(1) {
      /*// check if Manchester-coded bit detected
      if (cmd_bit_done) {
        // reset flag
        cmd_bit_done = 0x00;

        // update RC5 frame
        frame.full <<= 1;
        if (cmd_bit_value) frame.full |= 0x01;

        // update bit count
        cmd_bit_count++;

        if (cmd_bit_count == 14) {
          // RC5 frame is now complete
          cmd_frame_done = 0x01;
          cmd_bit_count  = 0x00;
        }
      }

      // check RC5 frame if completed
      if (cmd_frame_done) {
        // reset flag
        cmd_frame_done = 0x00;

        // parse frame to see if it is legitimate
        if ((frame.bits.start == 0b11) && (frame.bits.control == 0b0)) {
          // only accept command if intended for this device
          if (frame.bits.address == IR_ADDRESS) {
            // extract and signal new command
            new_cmd      = frame.bits.command;
            cmd_received = 0x01;
          }

          // clear RC5 frame
          frame.full = 0x0000;
        }
      }*/

      // process command if received
      if (cmd_received) {
        // reset flag
        cmd_received = 0x00;

        // execute command
        switch (new_cmd) {
          case (NO_COMMAND):
          cmd.bits.cmdA = NO_CMD;
          cmd.bits.cmdB = NO_CMD;
          break;
```

```
        case (TEST_UNIT):
        if (timer_status == PAUSE) {
          // test timer and scoreboard
          time.full     = 0x8888;   // time = 88:88
          cmd.bits.cmdA = TEST_POS; // scoreA = 888
          cmd.bits.cmdB = TEST_POS; // scoreB = 888
          cmd.bits.buzz = 0b0;
          PORTA         = time.parts.min;
          PORTB         = time.parts.sec;
          PORTC         = cmd.full;

          // wait 0.2 seconds
          wait = 4;
          while(0 < wait);

          cmd.bits.cmdA = TEST_NEG; // scoreA = -88
          cmd.bits.cmdB = TEST_NEG; // scoreB = -88
          cmd.bits.buzz = 0b0;
          PORTC         = cmd.full;

          // wait 0.2 seconds
          wait = 4;
          while(0 < wait);

          time.full     = 0x0000;   // time = 00:00
          cmd.bits.cmdA = RESET;    // scoreA = 000
          cmd.bits.cmdB = RESET;    // scoreB = 000
          cmd.bits.buzz = 0b1;      // siren ON
          PORTA         = time.parts.min;
          PORTB         = time.parts.sec;
          PORTC         = cmd.full;

          // wait 0.1 seconds
          wait = 2;
          while(0 < wait);

          cmd.bits.cmdA = NO_CMD;
          cmd.bits.cmdB = NO_CMD;
          cmd.bits.buzz = 0b0;      // siren OFF
          PORTC         = cmd.full;
          // done test of timer and scoreboard
          #ifdef DEBUG
          for (i = 0; i < 15; i++) {
            // wait 0.1 seconds
            wait = 2;
            while(0 < wait);

            cmd.bits.cmdA = ADD1;
            cmd.bits.cmdB = ADD1;
            PORTC         = cmd.full;

            // wait 0.4 seconds
            wait = 8;
            while(0 < wait);

            cmd.bits.cmdA = NO_CMD;
            cmd.bits.cmdB = NO_CMD;
            PORTC         = cmd.full;
          }
          for (i = 0; i < 20; i++) {
            // wait 0.1 seconds
            wait = 2;
            while(0 < wait);

            cmd.bits.cmdA = SUB1;
            cmd.bits.cmdB = SUB1;
            PORTC         = cmd.full;
```

```
          // wait 0.4 seconds
          wait = 8;
          while(0 < wait);

          cmd.bits.cmdA = NO_CMD;
          cmd.bits.cmdB = NO_CMD;
          PORTC         = cmd.full;
        }
        for (i = 0; i < 5; i++) {
          // wait 0.1 seconds
          wait = 2;
          while(0 < wait);

          cmd.bits.cmdA = ADD5;
          cmd.bits.cmdB = ADD5;
          PORTC         = cmd.full;

          // wait 0.4 seconds
          wait = 8;
          while(0 < wait);

          cmd.bits.cmdA = NO_CMD;
          cmd.bits.cmdB = NO_CMD;
          PORTC         = cmd.full;
        }
        for (i = 0; i < 4; i++) {
          // wait 0.1 seconds
          wait = 2;
          while(0 < wait);

          cmd.bits.cmdA = SUB5;
          cmd.bits.cmdB = SUB5;
          PORTC         = cmd.full;

          // wait 0.4 seconds
          wait = 8;
          while(0 < wait);

          cmd.bits.cmdA = NO_CMD;
          cmd.bits.cmdB = NO_CMD;
          PORTC         = cmd.full;
        }
        #endif
      }
      break;

      case (RESET_TIME):
      if (timer_status == PAUSE) {
        time.full     = 0x0000; // time = 00:00
        cmd.bits.buzz = 0b0;    // mute piezo siren
      }
      break;

      case (START_TIME):
      if (timer_status == PAUSE) {
        timer_status = START;
      }
      break;

      case (PAUSE_TIME):
      if (timer_status == START) {
        timer_status = PAUSE;
      }
      break;

      case (ADD10_MIN):
      if (timer_status == PAUSE) {
```

```
            // increment timer by 10 minutes and wrap
            time.parts.min += 0x10;
            if ((time.parts.min & 0xF0) > 0x90) {
              time.parts.min &= 0x0F;
            }
          }
          break;

          case (SUB10_MIN):
          if (timer_status == PAUSE) {
            // decrement timer by 10 minutes and wrap
            time.parts.min -= 0x10;
            if ((time.parts.min & 0xF0) > 0x90) {
              time.parts.min &= 0x0F;
              time.parts.min |= 0x90;
            }
          }
          break;

          case (ADD01_MIN):
          if (timer_status == PAUSE) {
            // increment timer by 1 minute and wrap
            time.parts.min += 0x01;
            if ((time.parts.min & 0x0F) > 0x09) {
              time.parts.min &= 0xF0;
            }
          }
          break;

          case (SUB01_MIN):
          if (timer_status == PAUSE) {
            // decrement timer by 1 minute and wrap
            time.parts.min -= 0x01;
            if ((time.parts.min & 0x0F) > 0x09) {
              time.parts.min &= 0xF0;
              time.parts.min |= 0x09;
            }
          }
          break;

          case (ADD10_SEC):
          if (timer_status == PAUSE) {
            // increment timer by 10 seconds and wrap
            time.parts.sec += 0x10;
            if ((time.parts.sec & 0xF0) > 0x90) {
              time.parts.sec &= 0x0F;
            }
          }
          break;

          case (SUB10_SEC):
          if (timer_status == PAUSE) {
            // decrement timer by 10 seconds and wrap
            time.parts.sec -= 0x10;
            if ((time.parts.sec & 0xF0) > 0x90) {
              time.parts.sec &= 0x0F;
              time.parts.sec |= 0x90;
            }
          }
          break;

          case (ADD01_SEC):
          if (timer_status == PAUSE) {
            // increment timer by 1 second and wrap
            time.parts.sec += 0x01;
            if ((time.parts.sec & 0x0F) > 0x09) {
              time.parts.sec &= 0xF0;
            }
```

```
        }
        break;

        case (SUB01_SEC):
        if (timer_status == PAUSE) {
          // decrement timer by 1 second and wrap
          time.parts.sec -= 0x01;
          if ((time.parts.sec & 0x0F) > 0x09) {
            time.parts.sec &= 0xF0;
            time.parts.sec |= 0x09;
          }
        }
        break;

        case (SAVE_TIME0):
        if (timer_status == PAUSE) {
          // save current time into EEPROM
          saved_time[0] = time.full;
        }
        break;

        case (RECALL_TIME0):
        if (timer_status == PAUSE) {
          // restore time from EEPROM
          time.full = saved_time[0];
        }
        break;

        case (SAVE_TIME1):
        if (timer_status == PAUSE) {
          // save current time into EEPROM
          saved_time[1] = time.full;
        }
        break;

        case (RECALL_TIME1):
        if (timer_status == PAUSE) {
          // restore time from EEPROM
          time.full = saved_time[1];
        }
        break;

        case (A_RESET):
        cmd.bits.cmdA = RESET;
        break;

        case (A_ADD1):
        cmd.bits.cmdA = ADD1;
        break;

        case (A_SUB1):
        cmd.bits.cmdA = SUB1;
        break;

        case (A_ADD5):
        cmd.bits.cmdA = ADD5;
        break;

        case (A_SUB5):
        cmd.bits.cmdA = SUB5;
        break;

        case (B_RESET):
        cmd.bits.cmdB = RESET;
        break;

        case (B_ADD1):
        cmd.bits.cmdB = ADD1;
```

```
          break;

          case (B_SUB1):
          cmd.bits.cmdB = SUB1;
          break;

          case (B_ADD5):
          cmd.bits.cmdB = ADD5;
          break;

          case (B_SUB5):
          cmd.bits.cmdB = SUB5;
          break;

          default:
          cmd.bits.cmdA = NO_CMD;
          cmd.bits.cmdB = NO_CMD;
          cmd.bits.buzz = 0b0;
          break;
        }
      }
      // else signal that no command was received
      else {
        cmd.bits.cmdA = NO_CMD;
        cmd.bits.cmdB = NO_CMD;
      }

      // decrement timer every second
      if (dec_timer) {
        // reset flag
        dec_timer = 0x00;

        // perform binary subtraction
        time.full -= 0x0001;

        // normalize result into packed BCD
        // note: resetting seconds occurs before normalizing
        if ((time.parts.min & 0xF0) > 0x90) {
          time.full -= 0x6000;
        }
        if ((time.parts.min & 0x0F) > 0x09) {
          time.full -= 0x0600;
        }
        if (time.parts.sec == 0xFF) {
          time.parts.sec = 0x59;
        }
        if ((time.parts.sec & 0xF0) > 0x90) {
          time.full -= 0x0060;
        }
        if ((time.parts.sec & 0x0F) > 0x09) {
          time.full -= 0x0006;
        }

        // timer has finished
        if (time.full = 0x0000) {
          cmd.bits.buzz = 0b1;   // sound siren
          timer_status  = PAUSE; // stop timer
        }
      }

      // update display and commands
      PORTA = time.parts.min;
      PORTB = time.parts.sec;
      PORTC = cmd.full;
    }
  }
```

A.4 ATtiny26(L) Source for the Scoreboard & Timer Unit

This source code is located at http://instruct1.cit.cornell.edu/courses/eceprojectsland/STUDENTPROJ/2005to2006/rw88/code/mod_t26.c.

```
/* Richard West '05
 * 2006 Master of Engineering Candidate
 * Electrical and Computer Engineering
 *
 * Master of Engineering Design Project
 *
 * COPYRIGHT & LICENSE:
 * Copyright (C) 2006 Richard West
 *
 * From http://www.gnu.org/copyleft/gpl.html:
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * as published by the Free Software Foundation; either version 2
 * of the License, or (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA
 * 02110-1301, USA.
 *
 * DESCRIPTION:
 * Source code for the ATtiny26(L) used to control an individual
 * team's score as part of the larger scoreboard and timer unit.
 * The score is maintained in packed BCD format in the following
 * order: 100s digit in the lower nibble of PORTA, 10s digit in
 * the upper nibble of PORTB, and 1s digit in the lower nibble of
 * PORTB.  Each digit serves as the input to a 4511 BCD to Seven
 * Segment IC (see schematics).  While an accurate score is
 * maintained, the output is limited between -99 and 999 since
 * those are the limits for the three digit displays.  The negative
 * sign is controlled by a single bit (PORTA.4) which is an input
 * to a 4001 Quad 2-Input NOR IC.
 *
 * There is no direct user input to this ATtiny26(L).  There are
 * only three bits of input (upper three bits of PINA) which
 * originate from a "master" ATmega32.  See the scoreboard and
 * timer schematics and the ATmega32 source for more information.
 *
 * FUSE BITS:
 * CLSEL3..0 = 0100 -> 8 MHz calibrated internal RC clock
 * RSTDISBL = 0 -> Disable reset and use B.7 as an I/O pin
 * (note: parallel programming required if reset is diabled)
 */

#include <tiny26.h>

// command definitions
#define RESET    0b000
#define ADD1     0b001
#define SUB1     0b010
#define ADD5     0b011
#define SUB5     0b100
#define TEST_NEG 0b101
#define TEST_POS 0b110
#define NO_CMD   0b111

unsigned char old_cmd, new_cmd;
```

```
unsigned char do_add, do_sub;
unsigned int score, temp_score;
unsigned int increment;

void main(void) {
  // initalize I/O
  DDRA  = 0x1F; // A.7-5 inputs, A.4-0 outputs
  DDRB  = 0xFF; // all outputs
  PORTA = 0x00;
  PORTB = 0x00;

  // initialize variscoreles
  old_cmd    = NO_CMD;
  do_add     = 0x00;
  do_sub     = 0x00;
  score      = 0x0000;
  temp_score = 0x0000;
  increment  = 0x0000;

  while (1) {
    // get command from upper 3 bits of PINA
    new_cmd = ((unsigned)(PINA >> 5));

    // process command if changed
    if (new_cmd != old_cmd) {
      // record command
      old_cmd = new_cmd;

      // execute command
      switch (new_cmd) {
        case NO_CMD:
        break;

        case RESET:
        score     = 0x0000; // score = 000
        break;

        case ADD1:
        increment = 0x0001;
        do_add    = 0x01;
        break;

        case SUB1:
        increment = 0x0001;
        do_sub    = 0x01;
        break;

        case ADD5:
        increment = 0x0005;
        do_add    = 0x01;
        break;

        case SUB5:
        increment = 0x0005;
        do_sub    = 0x01;
        break;

        case TEST_NEG:
        score     = 0x9912; // score = -88
        break;

        case TEST_POS:
        score     = 0x0888; // score = 888
        break;

        default:
        break;
      }
```

```
  }

  // perform packed BCD addition if required
  if (do_add) {
    // reset flag
    do_add = 0;

    // perform binary addition of increment
    score += increment;

    // normalize result into packed BCD
    if ((score & 0x000F) > 0x0009) {
      score += 0x0006;
    }
    if ((score & 0x00F0) > 0x0090) {
      score += 0x0060;
    }
    if ((score & 0x0F00) > 0x0900) {
      score += 0x0600;
    }
    if ((score & 0xF000) > 0x9000) {
      score += 0x6000;
    }
  }

  // perform packed BCD subtraction if required
  if (do_sub) {
    // reset flag
    do_sub = 0;

    // perform binary subtraction of increment
    score -= increment;

    // normalize result into packed BCD
    if ((score & 0x000F) > 0x0009) {
      score -= 0x0006;
    }
    if ((score & 0x00F0) > 0x0090) {
      score -= 0x0060;
    }
    if ((score & 0x0F00) > 0x0900) {
      score -= 0x0600;
    }
    if ((score & 0xF000) > 0x9000) {
      score -= 0x6000;
    }
  }

  // update displayed score
  if (score > 0x4999) {
    // maintain score but keep display within limits
    if (score < 0x9901) temp_score = 0x9901; // temp_score = -99
    else temp_score = score;

    // compute 10s complement
    temp_score = ((0x9999 - temp_score) + 1);

    // assert negative sign flag
    PORTA |= 0x10;

    // note: 100s digit will always be 0 if the score is negative,
    // but the 100s digit is never displayed.  Therefore, updating
    // the 100s digit is unnecessary.

    // update 10s and 1s digits
    PORTB = (unsigned char)(temp_score & 0x00FF);
    if ((PORTB & 0x0F) > 0x09) {
      PORTB += 0x06;
```

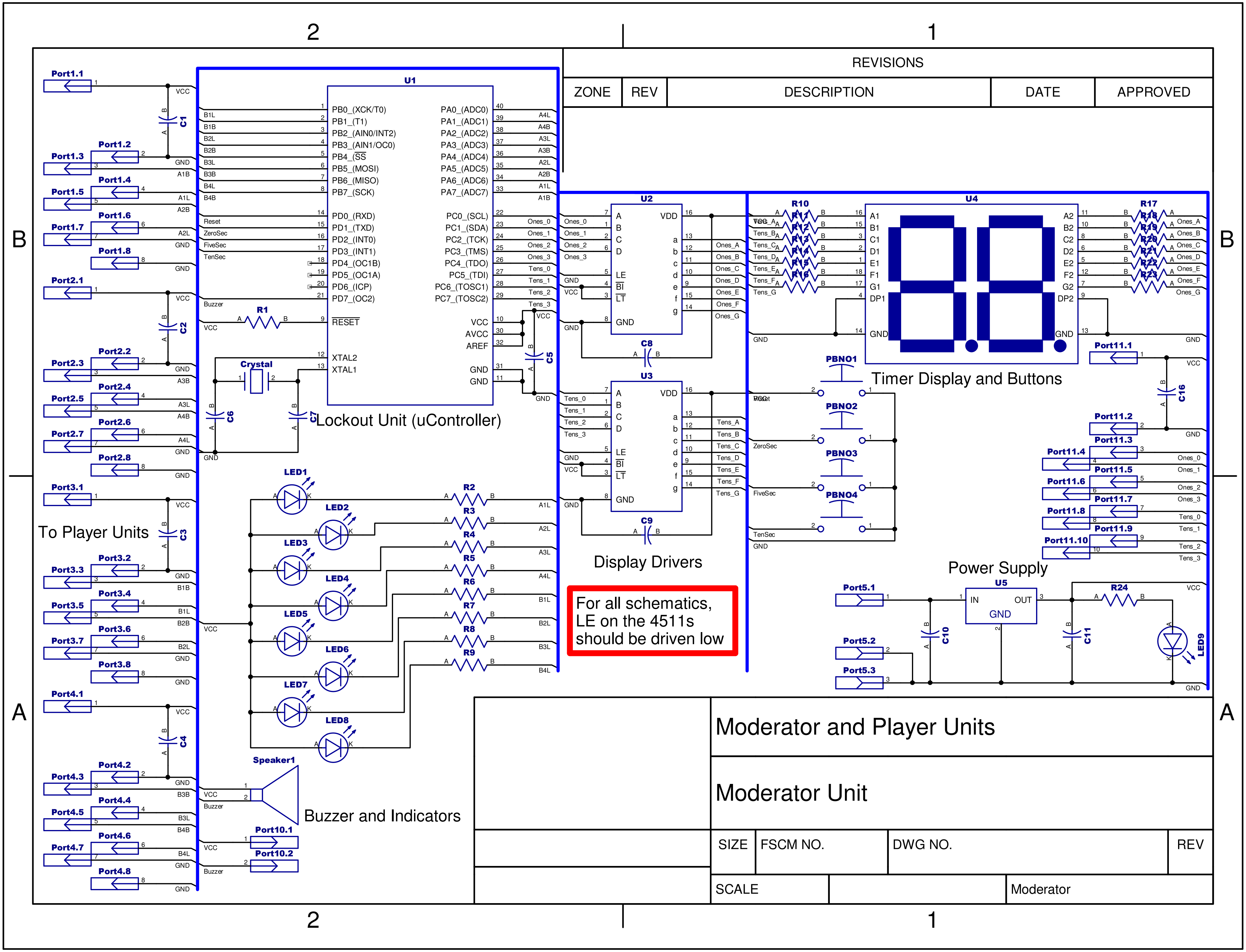
```
      }
    }
    else {
      // maintain score but keep display within limits
      if (score > 0x0999) temp_score = 0x0999; // temp_score = 999
      else temp_score = score;

      // update all digits
      PORTA = (unsigned char)(temp_score >> 8);
      PORTB = (unsigned char)(temp_score & 0x00FF);
    }
  }
}
```

B Schematics

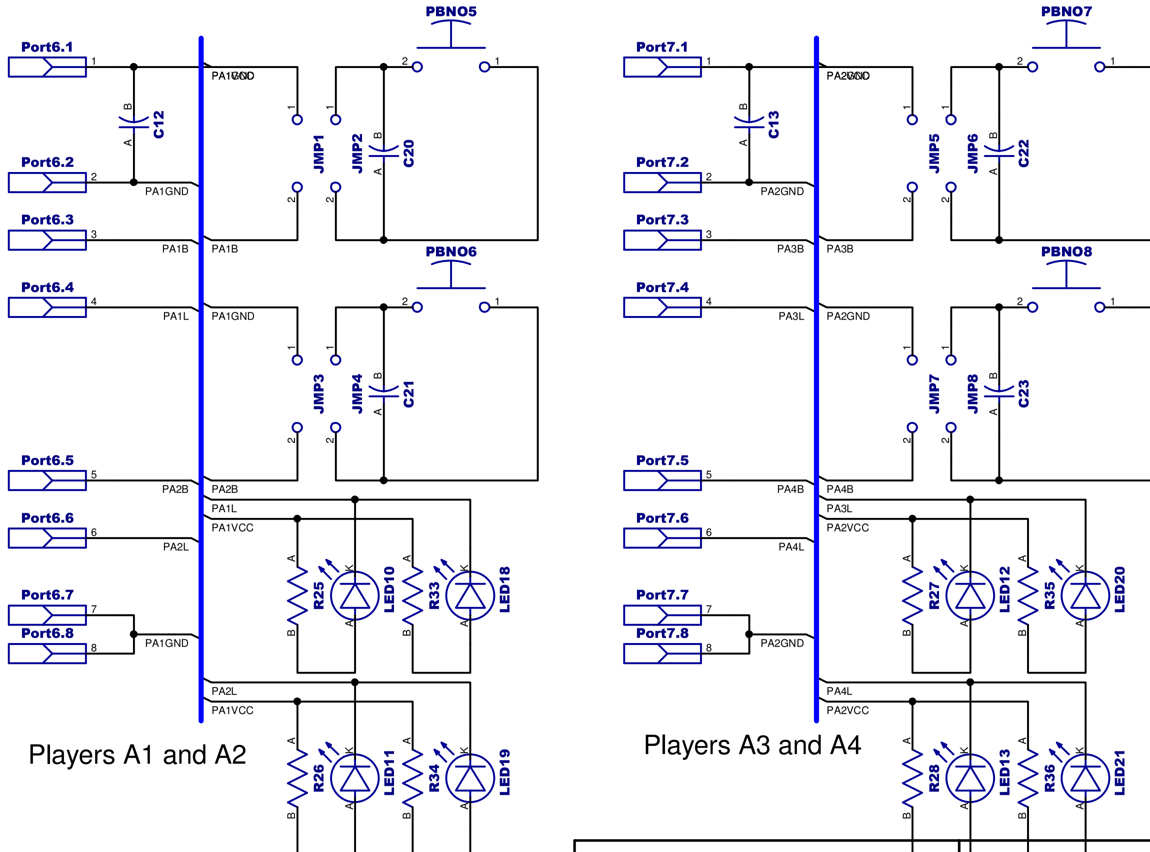
The schematics are attached. They are available from http://instruct1.cit.cornell.edu/courses/eceprojectsland/STUDENTPROJ/2005to2006/rw88/schematics/ in both PDF and DCH format for DipTrace. DipTrace is available from http://www.diptrace.com.

B.1 Moderator and Player Units

This schematic is available from http://instruct1.cit.cornell.edu/courses/eceprojectsland/STUDENTPROJ/2005to2006/rw88/schematics/moderator.pdf (PDF format) or http://instruct1.cit.cornell.edu/courses/eceprojectsland/STUDENTPROJ/2005to2006/rw88/schematics/moderator.dch (DipTrace DCH format).

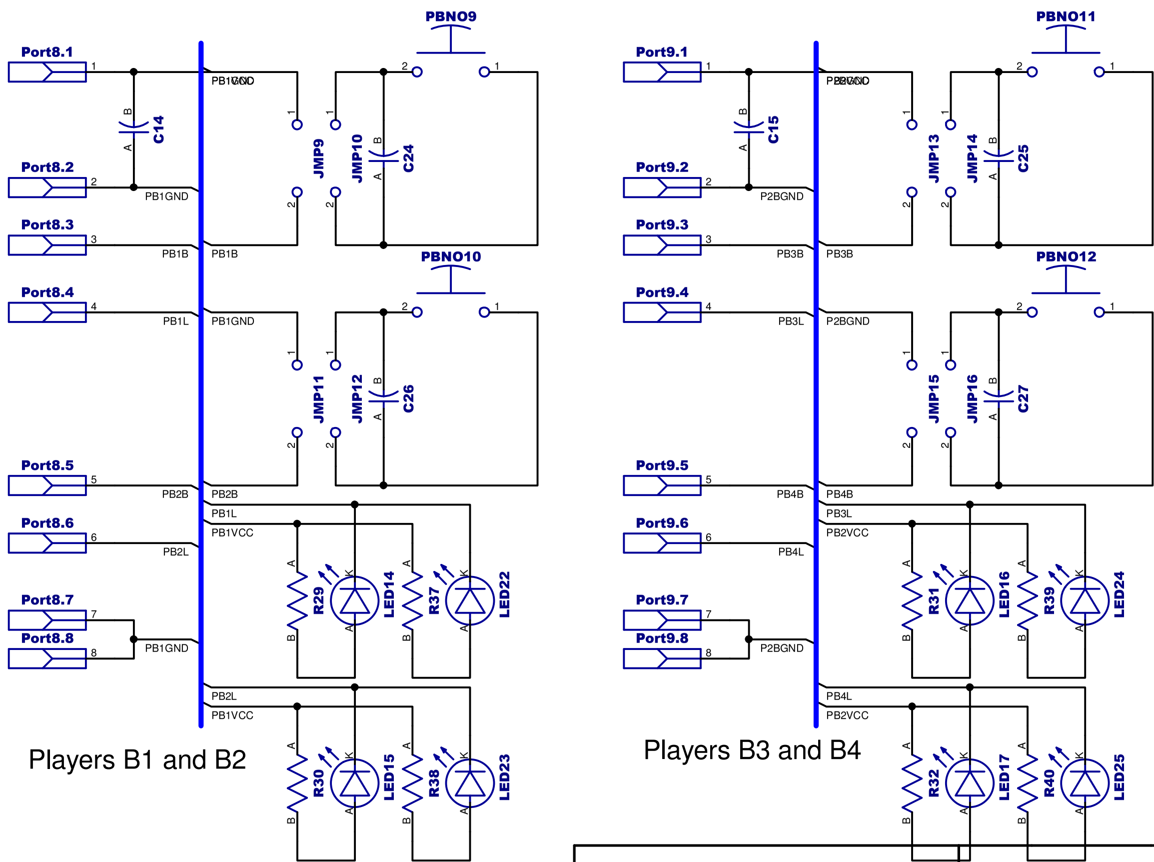

REVISIONS
ZONE
REV
DESCRIPTION
DATE
APPROVED
U1
PB0_(XCK/T0)
PB1_(T1)
PB2_(AIN0/INT2)
PB3_(AIN1/OC0)
PB4_(SS)
PB5_(MOSI)
PB6_(MISO)
PB7_(SCK)
PD0_(RXD)
PD1_(TXD)
PD2_(INT0)
PD3_(INT1)
PD4_(OC1B)
PD5_(OC1A)
PD6_(ICP)
PD7_(OC2)
RESET
XTAL2
XTAL1
PA0_(ADC0)
PA1_(ADC1)
PA2_(ADC2)
PA3_(ADC3)
PA4_(ADC4)
PA5_(ADC5)
PA6_(ADC6)
PA7_(ADC7)
PC0_(SCL)
PC1_(SDA)
PC2_(TCK)
PC3_(TMS)
PC4_(TDO)
PC5_(TDI)
PC6_(TOSC1)
PC7_(TOSC2)
VCC
AVCC
AREF
GND
Lockout Unit (uController)
Crystal
R1
C1
C2
C3
C4
C5
C6
C7
To Player Units
Port1.1
Port1.2
Port1.3
Port1.4
Port1.5
Port1.6
Port1.7
Port1.8
Port2.1
Port2.2
Port2.3
Port2.4
Port2.5
Port2.6
Port2.7
Port2.8
Port3.1
Port3.2
Port3.3
Port3.4
Port3.5
Port3.6
Port3.7
Port3.8
Port4.1
Port4.2
Port4.3
Port4.4
Port4.5
Port4.6
Port4.7
Port4.8
U2
U3
C8
C9
Display Drivers
For all schematics, LE on the 4511s should be driven low
U4
R10
R17
Timer Display and Buttons
PBNO1
PBNO2
PBNO3
PBNO4
Port11.1
Port11.2
Port11.3
Port11.4
Port11.5
Port11.6
Port11.7
Port11.8
Port11.9
Port11.10
C16
Power Supply
U5
IN
OUT
GND
C10
C11
R24
LED9
Port5.1
Port5.2
Port5.3
LED1
LED2
LED3
LED4
LED5
LED6
LED7
LED8
R2
R3
R4
R5
R6
R7
R8
R9
Speaker1
Port10.1
Port10.2
Buzzer and Indicators
Moderator and Player Units
Moderator Unit
SIZE
FSCM NO.
DWG NO.
REV
SCALE
Moderator

REVISIONS				
ZONE	REV	DESCRIPTION	DATE	APPROVED



Moderator and Player Units				
Team A Player Units				
SIZE	FSCM NO.	DWG NO.		REV
SCALE		Team A Players		

ZONE	REV	DESCRIPTION	DATE	APPROVED

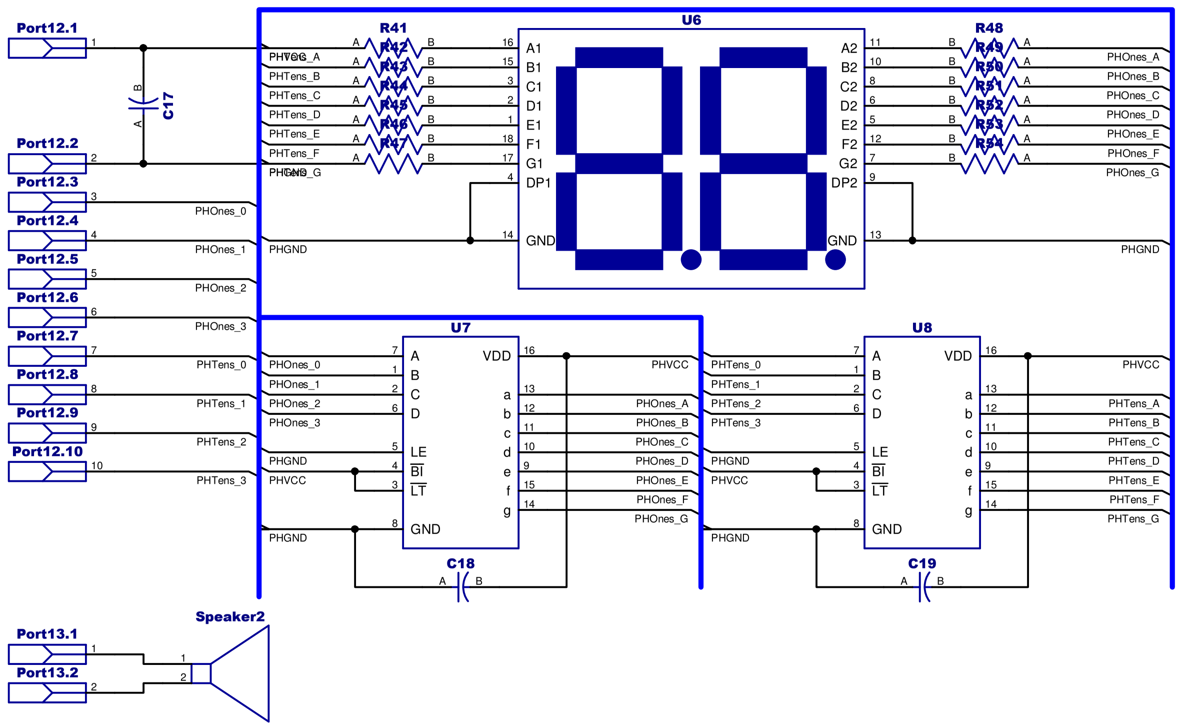


Moderator and Player Units

Team B Player Units

SIZE	FSCM NO.	DWG NO.	REV
SCALE		Team B Players	

REVISIONS				
ZONE	REV	DESCRIPTION	DATE	APPROVED

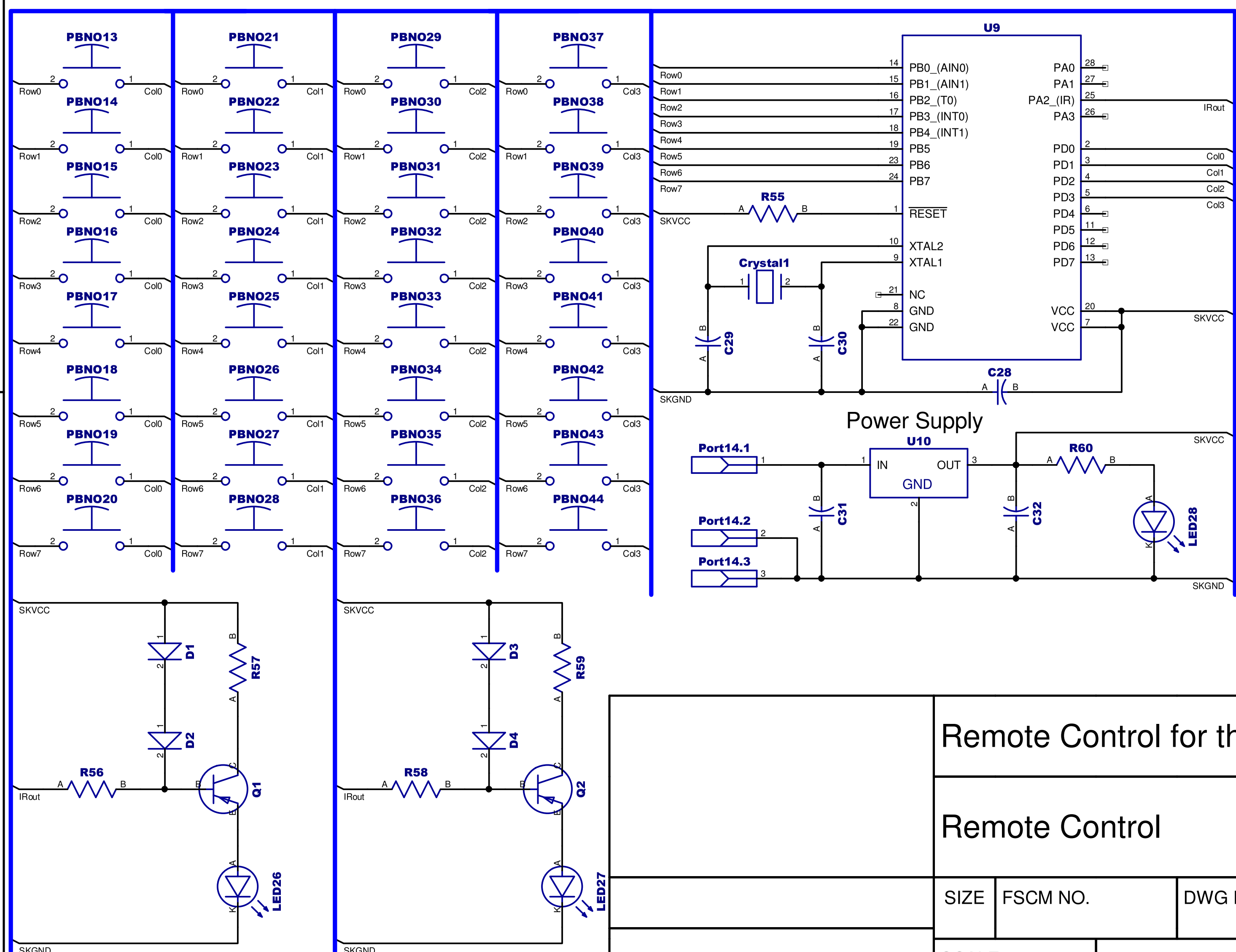


Moderator and Player Units			
Peripheral Display and Buzzer			
SIZE	FSCM NO.	DWG NO.	REV
SCALE		Peripherals	

B.2 Remote Control for Scoreboard and Timer Unit

This schematic is available from http://instruct1.cit.cornell.edu/courses/eceprojectsland/STUDENTPROJ/2005to2006/rw88/schematics/remote.pdf (PDF format) or http://instruct1.cit.cornell.edu/courses/eceprojectsland/STUDENTPROJ/2005to2006/rw88/schematics/moderator.dch (DipTrace DCH format). The remote control schematic is part of the moderator and player units schematic since they were fabricated on the same PCB.

REVISIONS				
ZONE	REV	DESCRIPTION	DATE	APPROVED



Remote Control for the Scoreboard and Timer Unit

Remote Control

SIZE	FSCM NO.	DWG NO.	REV
SCALE		Scorekeeper	

B.3 Scoreboard and Timer Unit

This schematic is available from `http://instruct1.cit.cornell.edu/courses/eceprojectsland/STUDENTPROJ/2005to2006/rw88/schematics/schematic.pdf` (PDF format) or `http://instruct1.cit.cornell.edu/courses/eceprojectsland/STUDENTPROJ/2005to2006/rw88/schematics/schematic.dch` (DipTrace DCH format).

B.3.1 Revisions

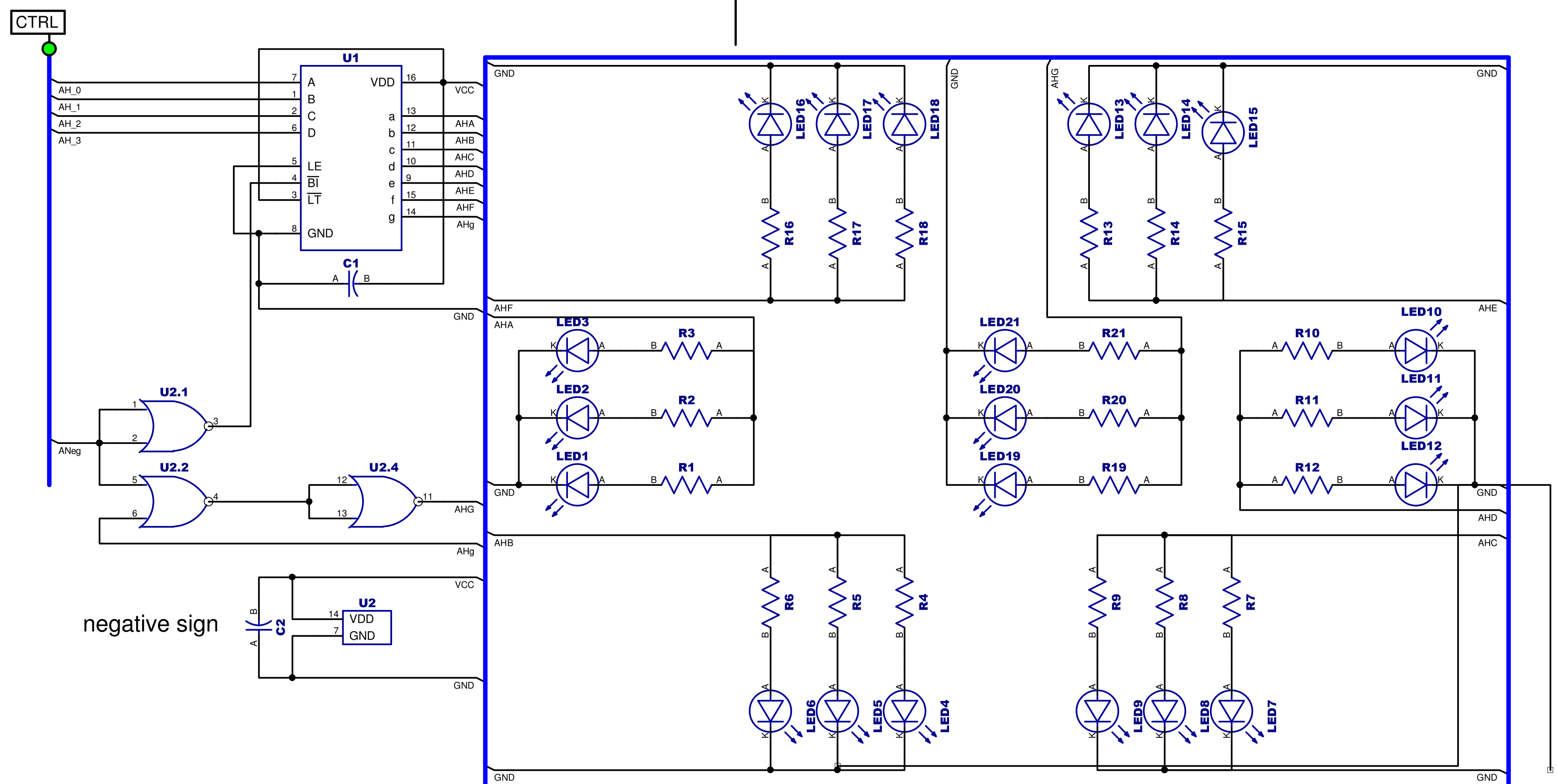
Due to a disagreement between the schematic and the datasheet for the 4511, the latch enable pin was mistakenly wired to VCC and not GND. Latch enable (LE) needs to be wired to GND for the displays to update as the digits change. Different manufacturers' datasheets describe the latch enable pin as either active high or active low, but the functionality of the pin is identical for all 4511s. This error was corrected for the Moderator and Player Units before that board was fabricated.

Due to the large power requirements, a separate self-regulated power supply had to be purchased to supply five volts at six amps. To support this extra current load, the power and ground traces had to be widened and rewired. This rewiring could not be accomplished entirely within the confines of the PCB, so some external wiring of components is necessary to complete the power and ground supplies.

Another minor revision made to the printed circuit board was to increase the hole size for the piezo siren's mounting holes. In the originally fabricated printed circuit board, the external diameter of the mounting holes was the correct size, but the interior diameter was not. Also, the control for the piezo siren has been changed from active low to active high to simplify the code.

As a cosmetic detail, the through-holes for the IR demodulator have been moved to allow the demodulator to lay flat against the printed circuit board without bending over the ATMega32.

REVISIONS				
ZONE	REV	DESCRIPTION	DATE	APPROVED



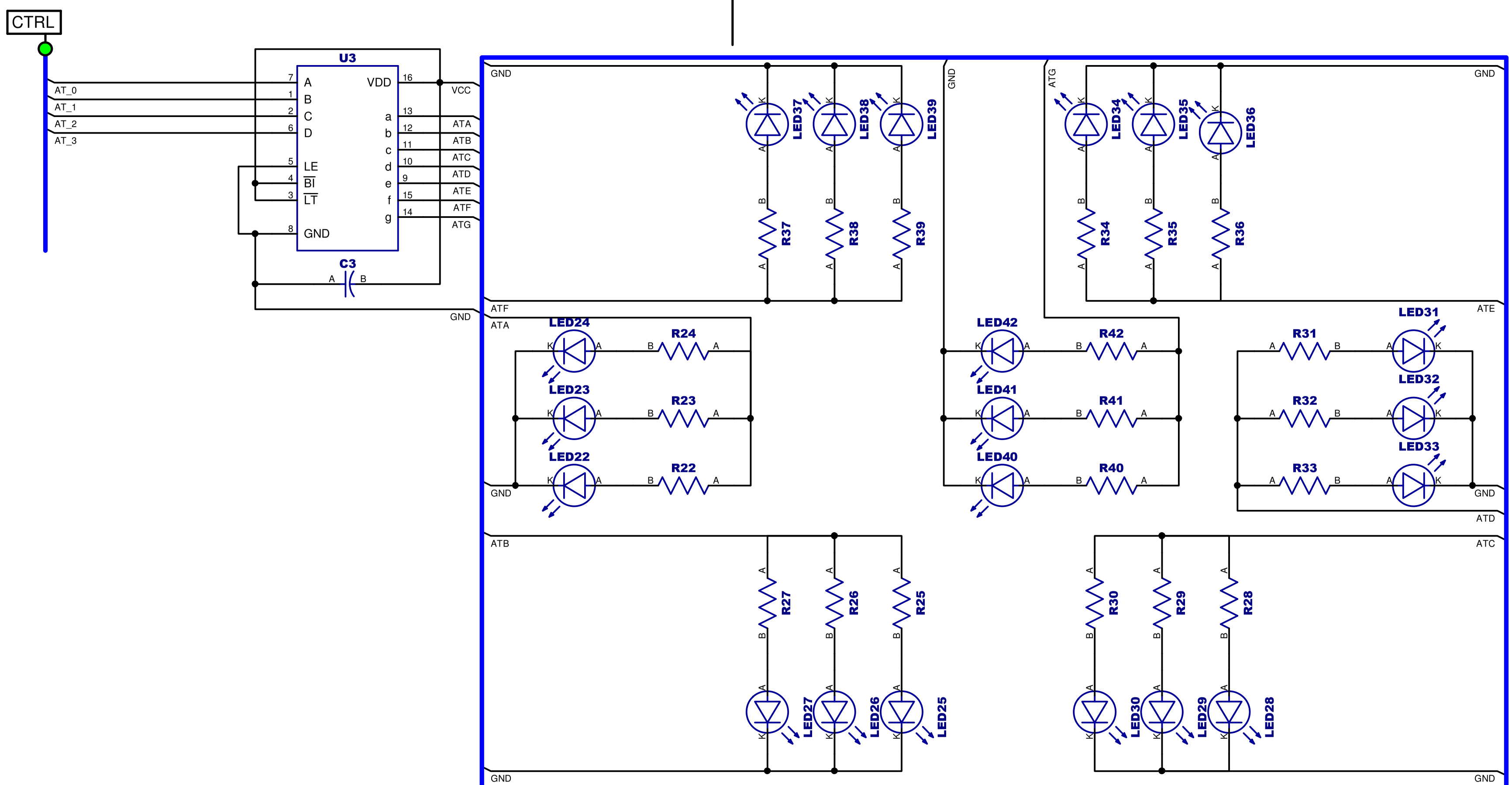
For all the schematics, LE on the 4511s should be driven low

Scoreboard & Timer

Team A - Hundreds Digit

SIZE	FSCM NO.	DWG NO.	REV
SCALE		A Hundreds	

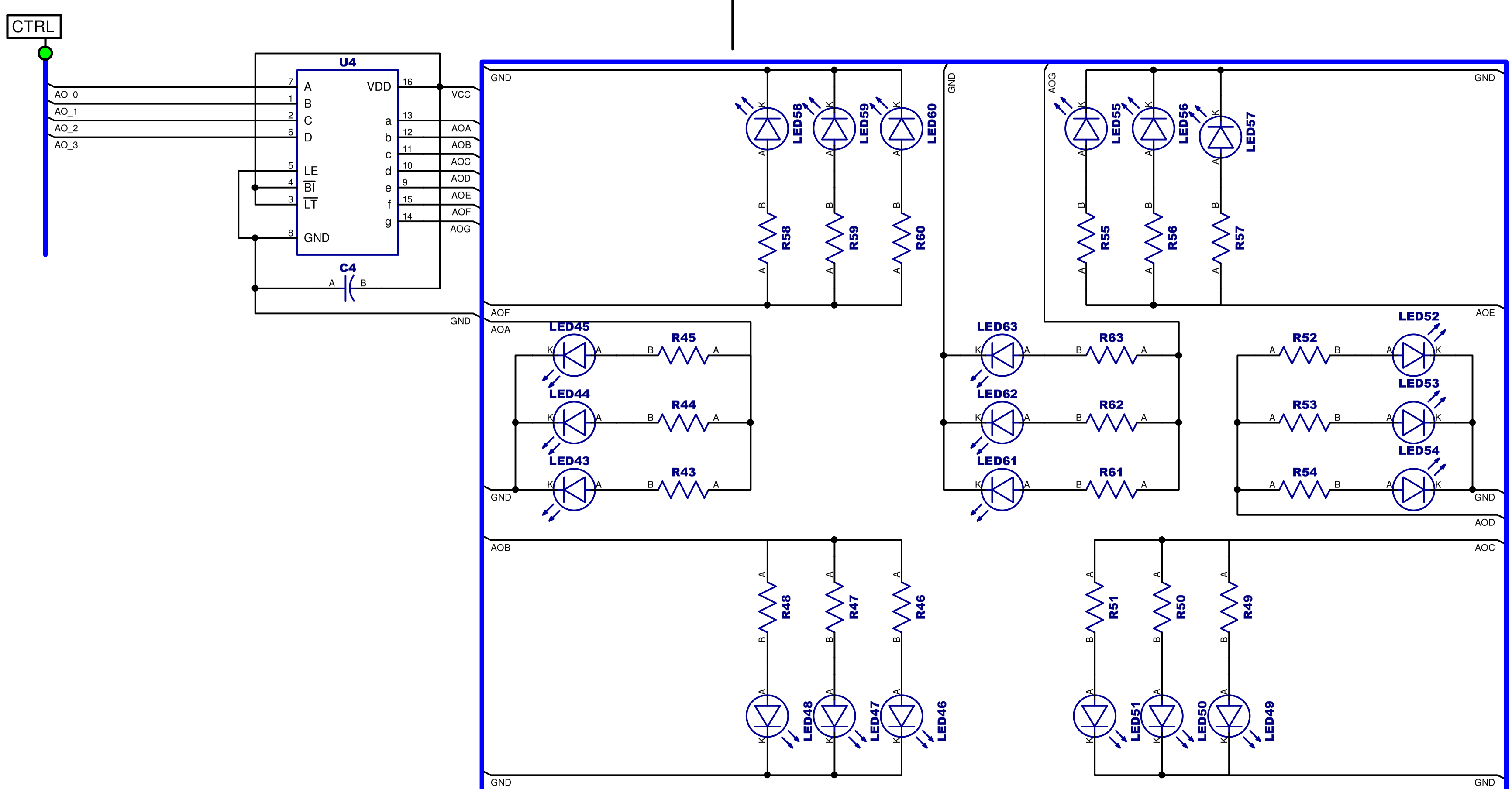
REVISIONS				
ZONE	REV	DESCRIPTION	DATE	APPROVED



Scoreboard & Timer

Team A - Tens Digit

SIZE	FSCM NO.	DWG NO.	REV
SCALE		A Tens	



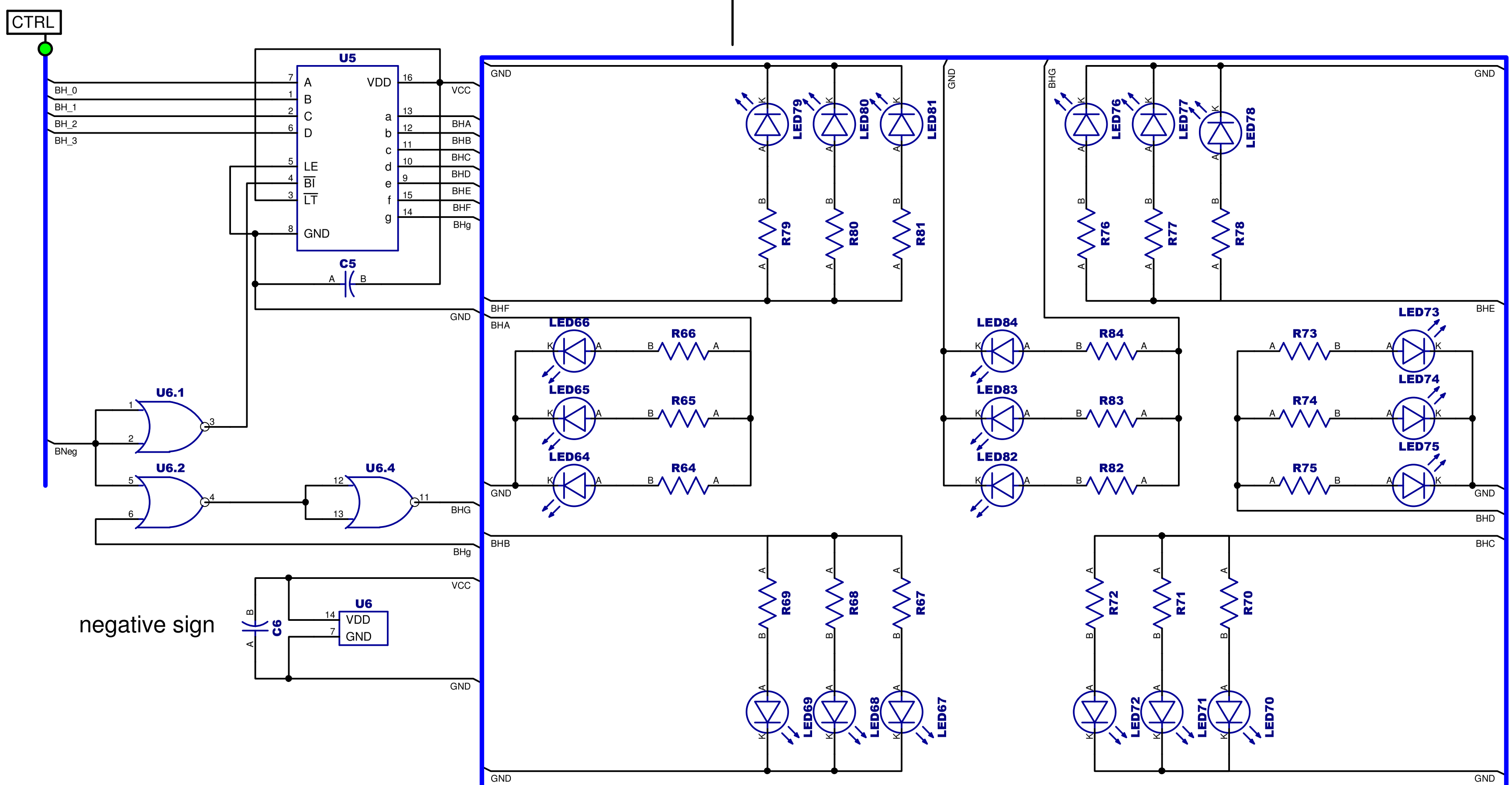
REVISIONS				
ZONE	REV	DESCRIPTION	DATE	APPROVED

Scoreboard & Timer

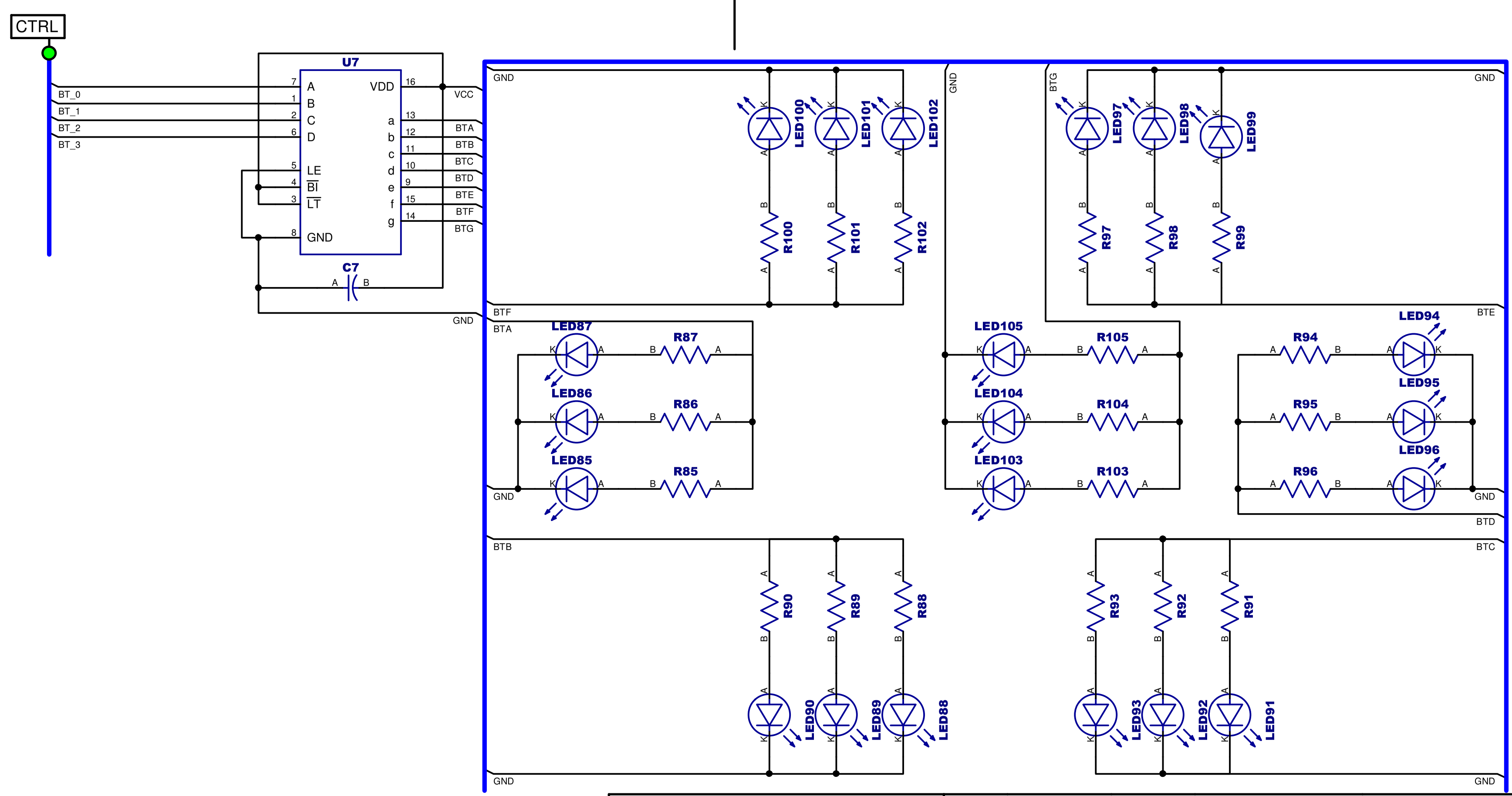
Team A - Ones Digit

SIZE	FSCM NO.	DWG NO.	REV
SCALE		A Ones	

REVISIONS				
ZONE	REV	DESCRIPTION	DATE	APPROVED



Scoreboard & Timer				
Team B - Hundreds Digit				
SIZE	FSCM NO.	DWG NO.		REV
SCALE		B Hundreds		



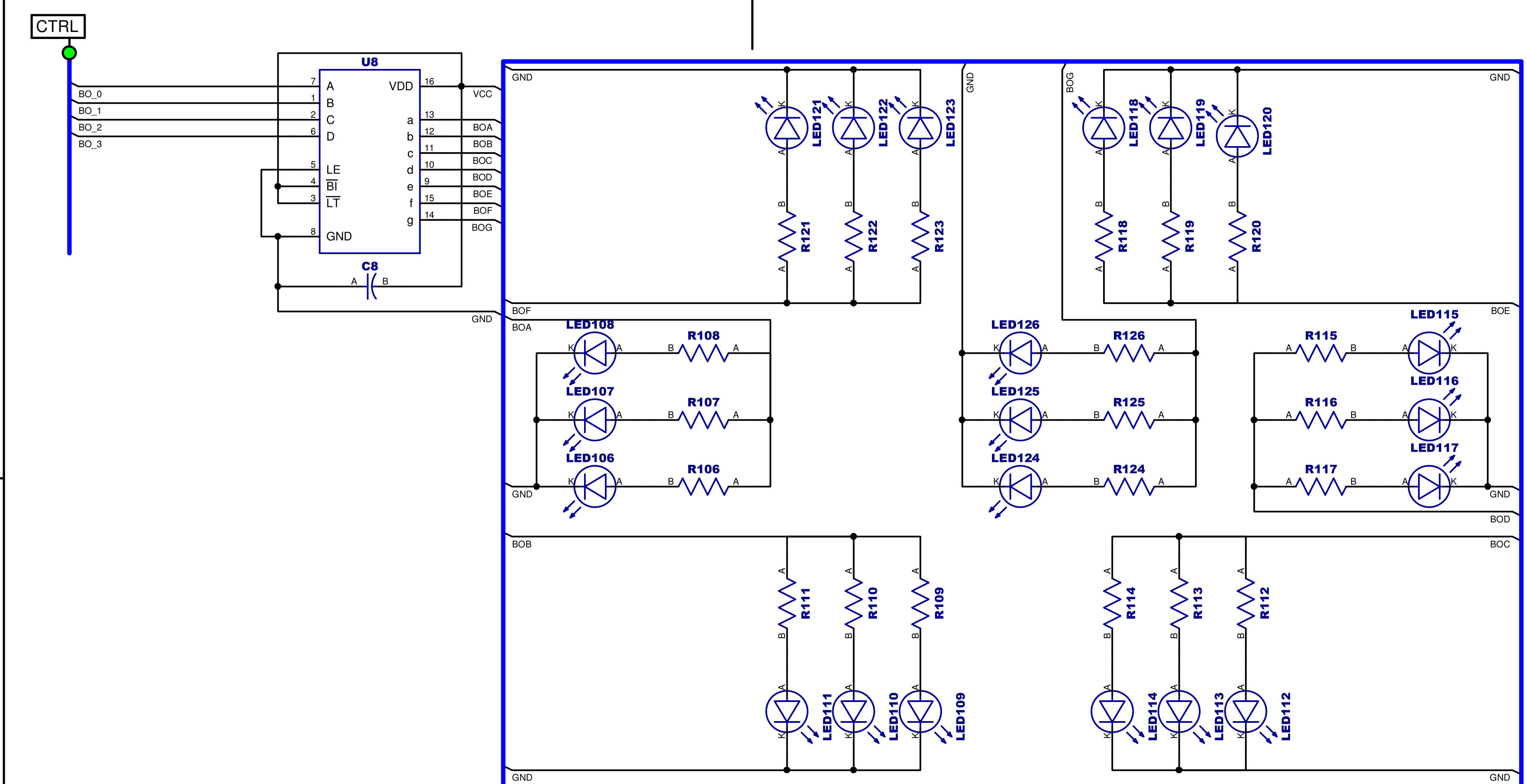
REVISIONS				
ZONE	REV	DESCRIPTION	DATE	APPROVED

Scoreboard & Timer

Team B - Tens Digit

SIZE	FSCM NO.	DWG NO.	REV
SCALE		B Tens	

REVISIONS				
ZONE	REV	DESCRIPTION	DATE	APPROVED

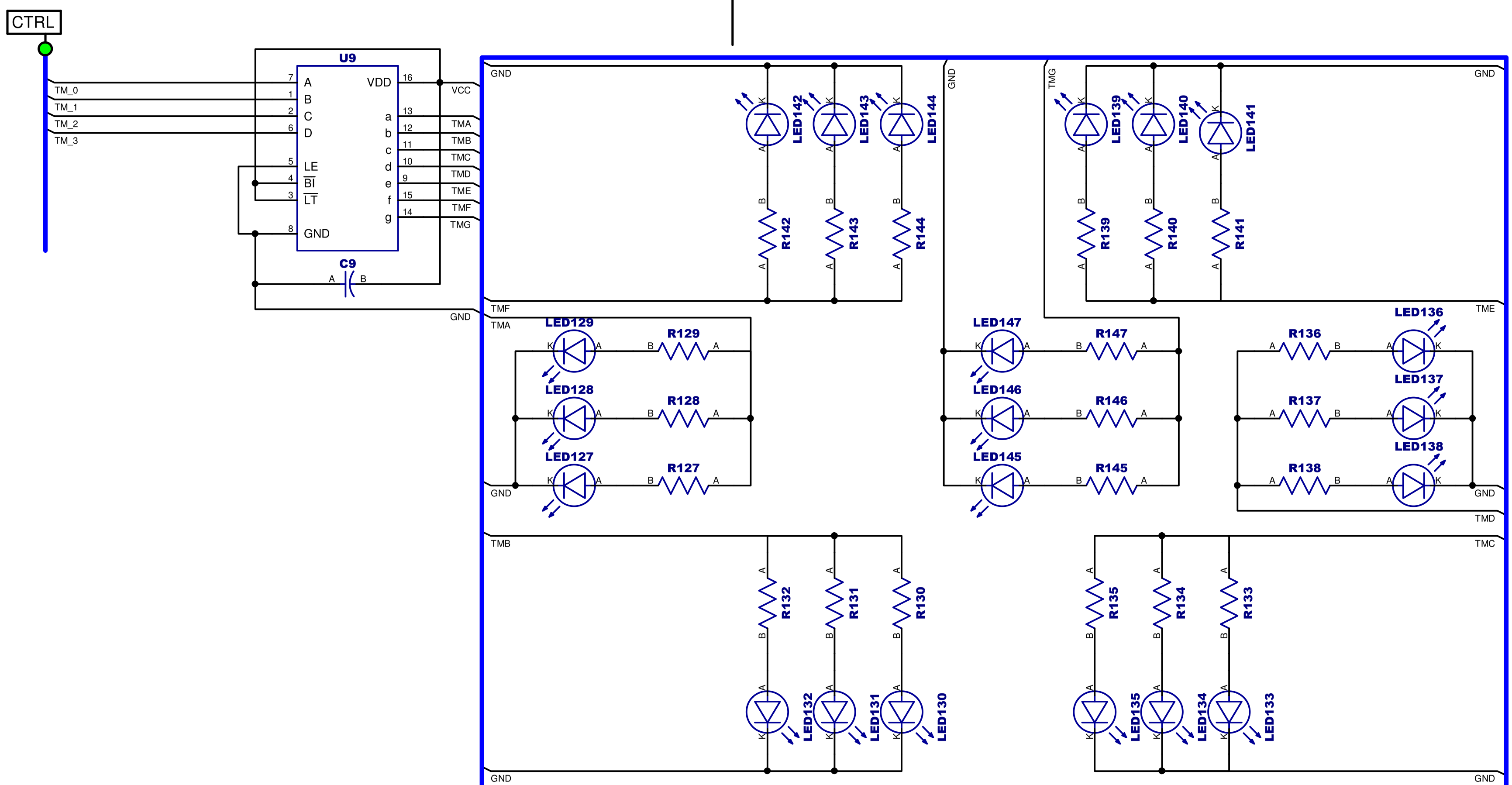


Scoreboard & Timer

Team B - Ones Digit

SIZE	FSCM NO.	DWG NO.	REV
SCALE		B Ones	

REVISIONS				
ZONE	REV	DESCRIPTION	DATE	APPROVED

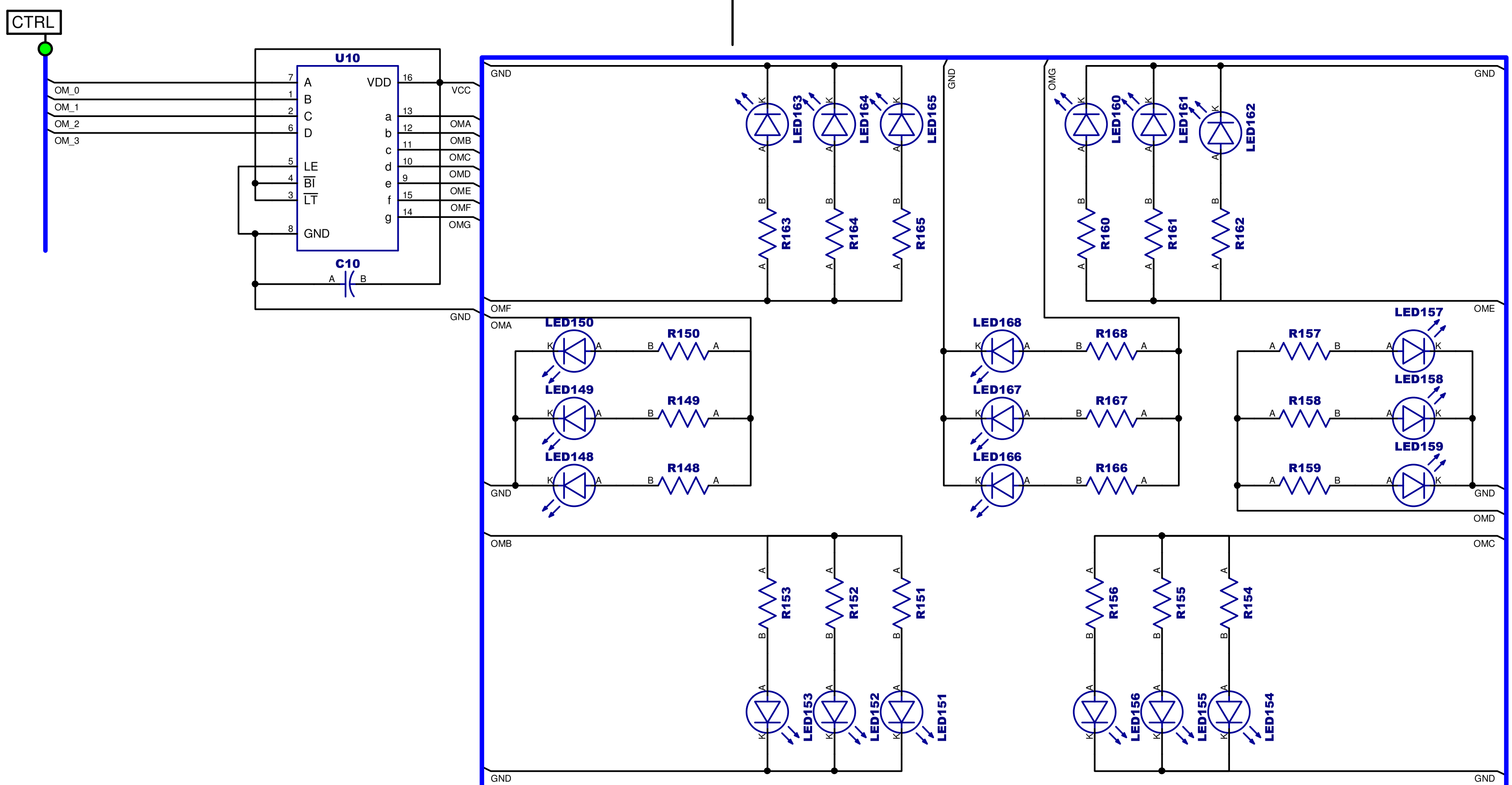


Scoreboard & Timer

Ten Minutes Digit

SIZE	FSCM NO.	DWG NO.	REV
SCALE		TenMinute	

REVISIONS				
ZONE	REV	DESCRIPTION	DATE	APPROVED

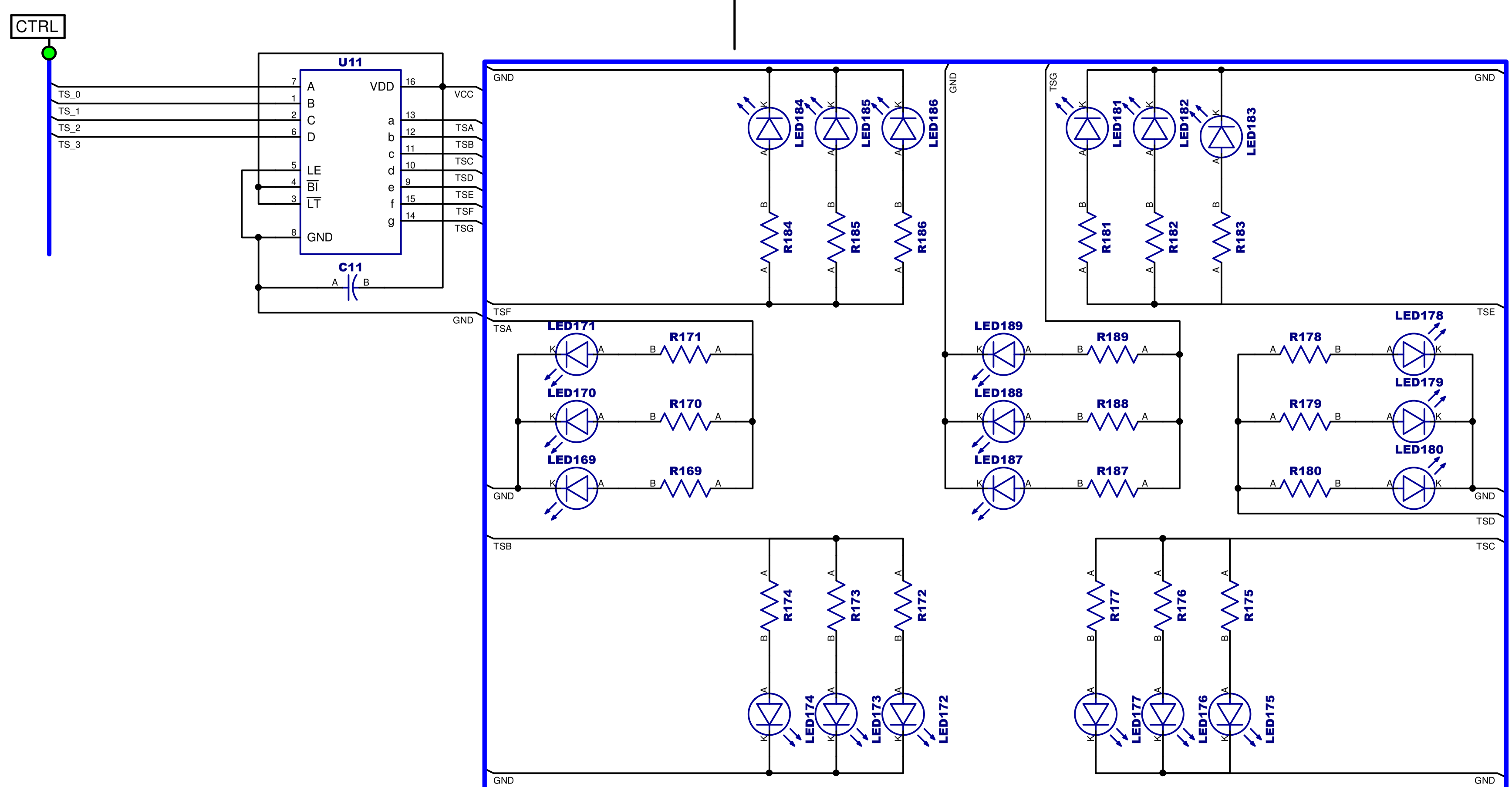


Scoreboard & Timer

One Minutes Digit

SIZE	FSCM NO.	DWG NO.	REV
SCALE		OneMinute	

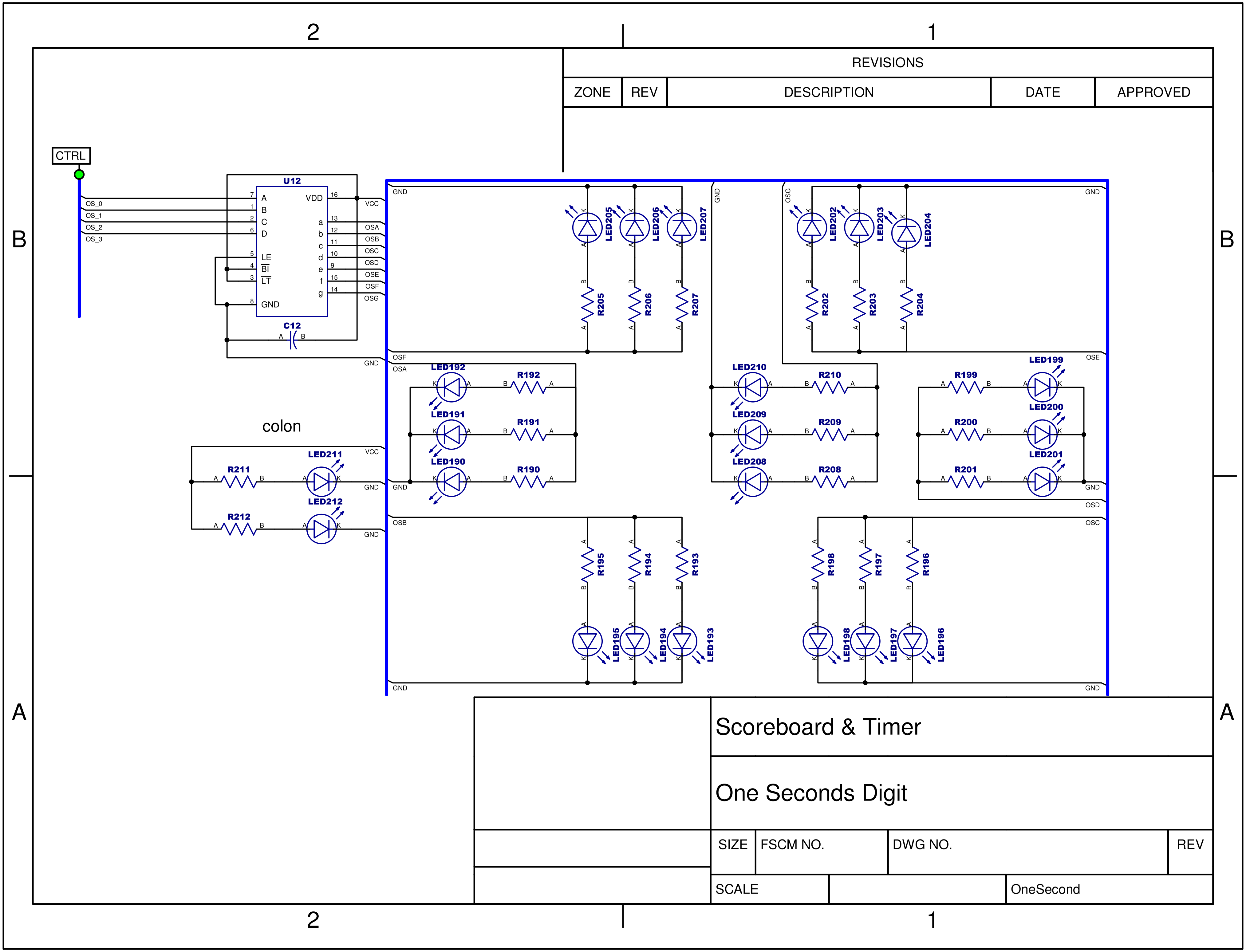
REVISIONS				
ZONE	REV	DESCRIPTION	DATE	APPROVED

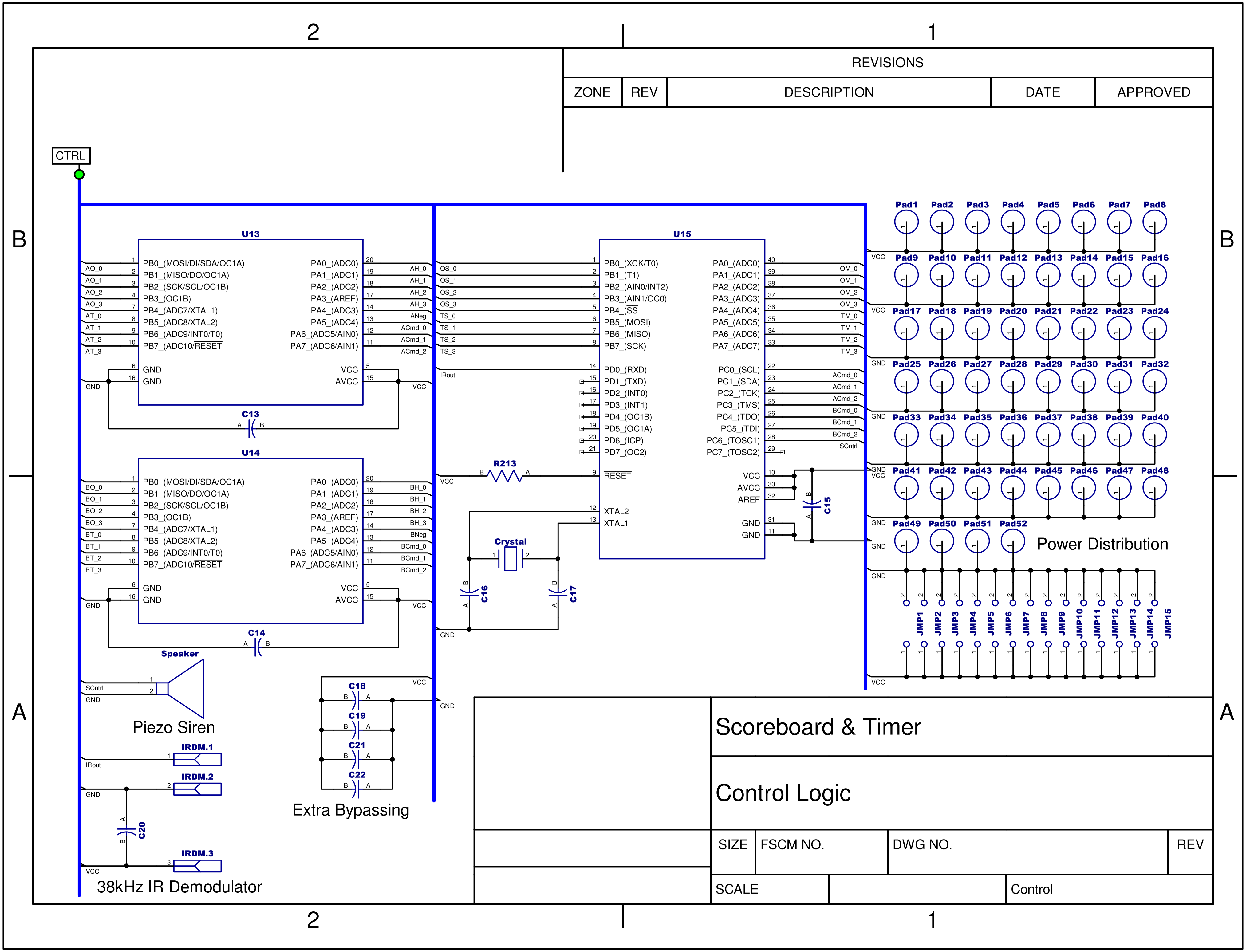


Scoreboard & Timer

Ten Seconds Digit

SIZE	FSCM NO.	DWG NO.	REV
SCALE		TenSecond	


REVISIONS
ZONE
REV
DESCRIPTION
DATE
APPROVED
CTRL
OS_0
OS_1
OS_2
OS_3
U12
VDD
GND
C12
VCC
OSA
OSB
OSC
OSD
OSE
OSF
OSG
colon
LED211
LED212
R211
R212
LED190
LED191
LED192
LED193
LED194
LED195
LED196
LED197
LED198
LED199
LED200
LED201
LED202
LED203
LED204
LED205
LED206
LED207
LED208
LED209
LED210
R190
R191
R192
R193
R194
R195
R196
R197
R198
R199
R200
R201
R202
R203
R204
R205
R206
R207
R208
R209
R210
Scoreboard & Timer
One Seconds Digit
SIZE
FSCM NO.
DWG NO.
REV
SCALE
OneSecond


REVISIONS
ZONE
REV
DESCRIPTION
DATE
APPROVED
CTRL
U13
U14
U15
C13
C14
C15
C16
C17
C18
C19
C20
C21
C22
R213
Crystal
Speaker
Piezo Siren
IRDM.1
IRDM.2
IRDM.3
38kHz IR Demodulator
Extra Bypassing
Power Distribution
Scoreboard & Timer
Control Logic
SIZE
FSCM NO.
DWG NO.
REV
SCALE
Control

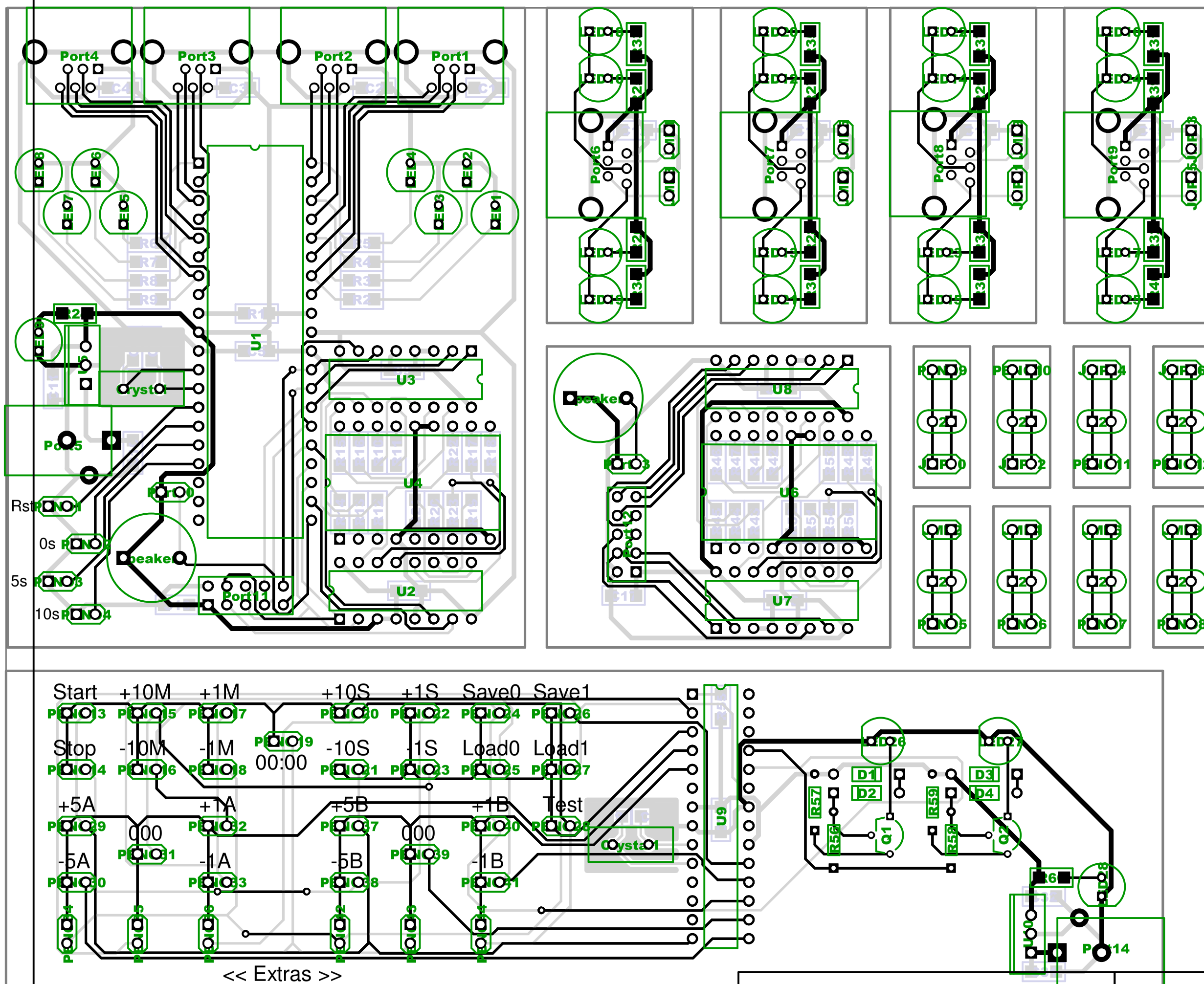
C Printed Circuit Board (PCB) Layouts

These PCB layouts are available from `http://instruct1.cit.cornell.edu/courses/eceprojectsland/STUDENTPROJ/2005to2006/rw88/layouts/` in both PDF format and DIP format for DipTrace. DipTrace is available from http://www.diptrace.com.

C.1 Moderator and Player Units and the Remote Control

This PCB layout is available from `http://instruct1.cit.cornell.edu/courses/eceprojectsland/STUDENTPROJ/2005to2006/rw88/layouts/moderator.pdf` (PDF format), `http://instruct1.cit.cornell.edu/courses/eceprojectsland/STUDENTPROJ/2005to2006/rw88/layouts/moderator.dip` (DipTrace DIP format), or `http://instruct1.cit.cornell.edu/courses/eceprojectsland/STUDENTPROJ/2005to2006/rw88/layouts/moderator.zip` (zipped Gerber files). The remote control occupies the lower half of the PCB.

REVISIONS				
ZONE	REV	DESCRIPTION	DATE	APPROVED

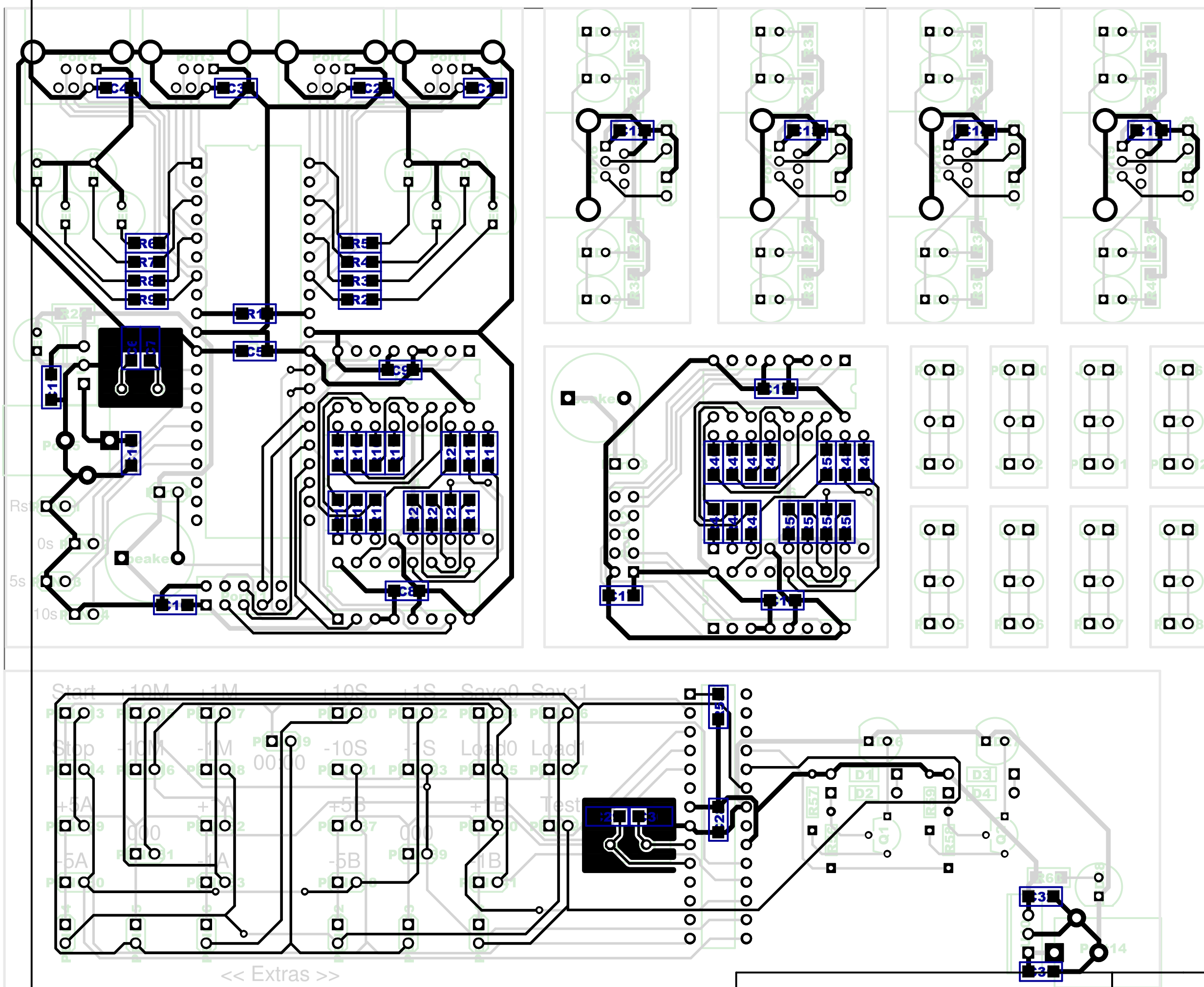


Moderator and Player Units

and Remote Control

SIZE	FSCM NO.	DWG NO.	REV
SCALE		Sheet 1	

REVISIONS				
ZONE	REV	DESCRIPTION	DATE	APPROVED



Moderator and Player Units

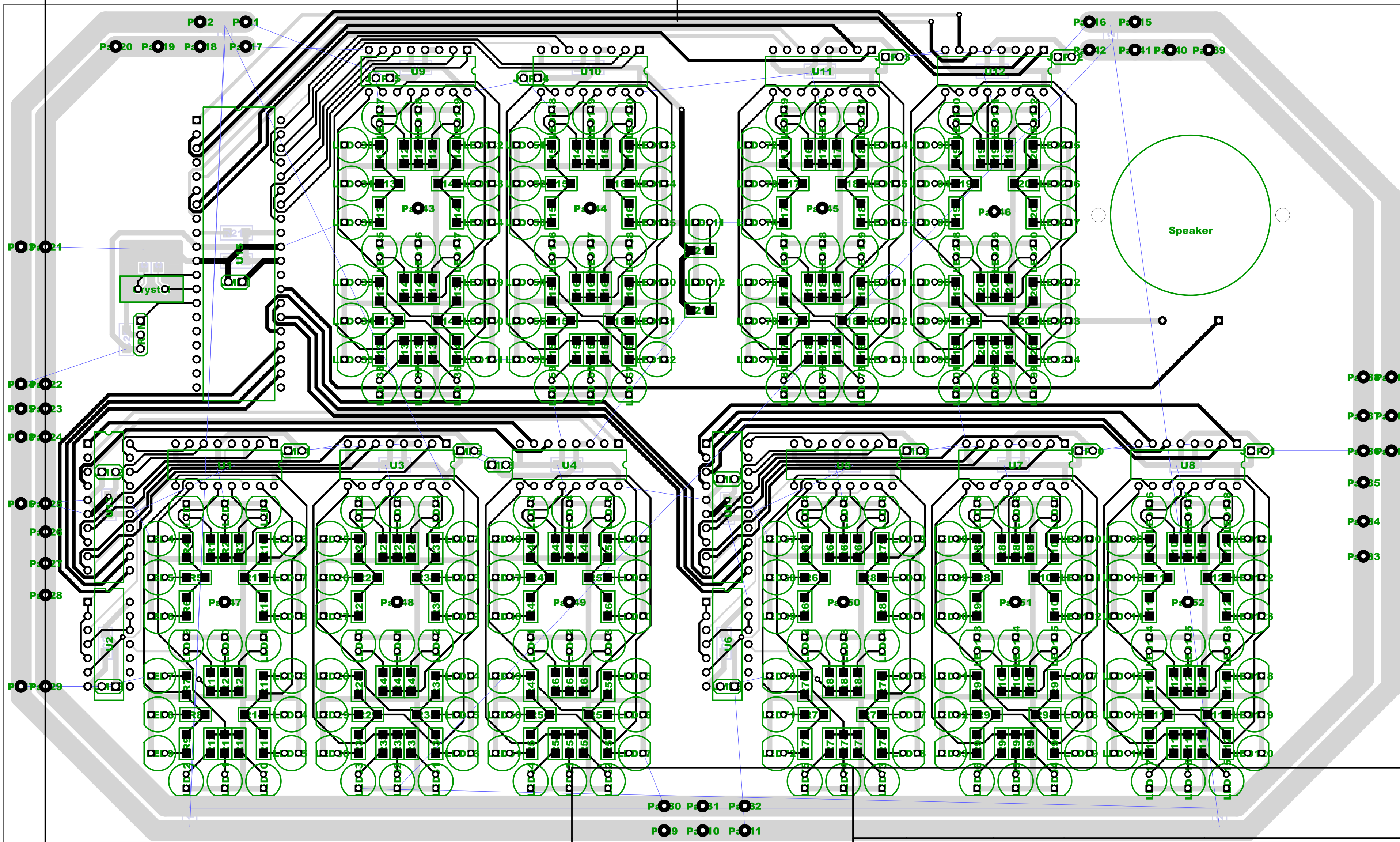
and Remote Control

SIZE	FSCM NO.	DWG NO.	REV
SCALE		Sheet 1	

C.2 Scoreboard and Timer Unit

This PCB layout is available from `http://instruct1.cit.cornell.edu/courses/eceprojectsland/STUDENTPROJ/2005to2006/rw88/layouts/scoreboard.pdf` (PDF format), `http://instruct1.cit.cornell.edu/courses/eceprojectsland/STUDENTPROJ/2005to2006/rw88/layouts/scoreboard.dip` (DipTrace DIP format), or `http://instruct1.cit.cornell.edu/courses/eceprojectsland/STUDENTPROJ/2005to2006/rw88/layouts/scoreboard.zip` (zipped Gerber files).

REVISIONS				
ZONE	REV	DESCRIPTION	DATE	APPROVED

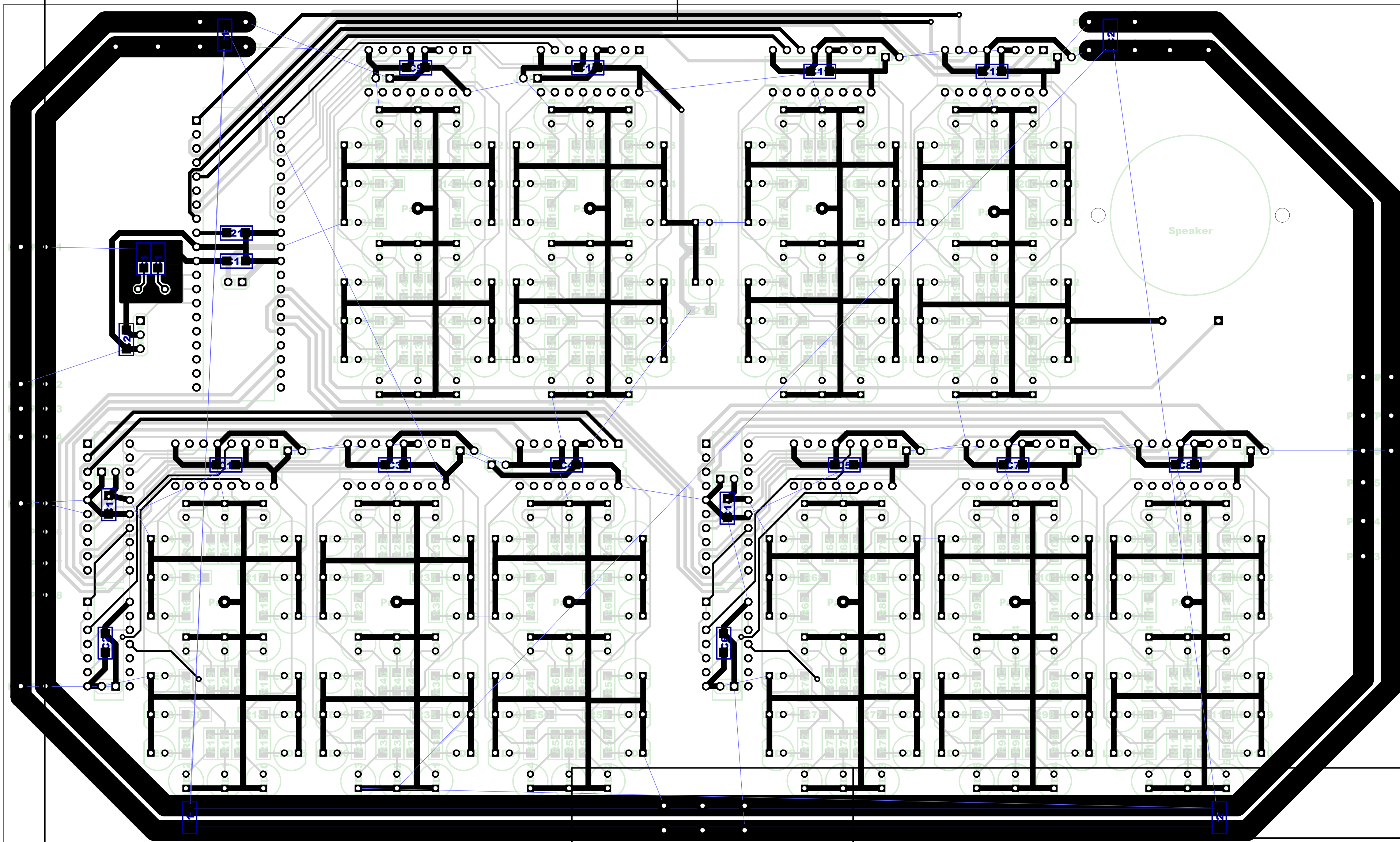


Scoreboard & Timer

SIZE	FSCM NO.	DWG NO.	REV

SCALE		Sheet 1

REVISIONS				
ZONE	REV	DESCRIPTION	DATE	APPROVED



Scoreboard & Timer

SIZE	FSCM NO.	DWG NO.	REV
SCALE		Sheet 1	