The Joy Of Engineering

Computer Science/ Computer Engineering
Final Report

Michael Yan, Eric Leung, Binna Han
December 2011

Abstract

In this project, we have embedded programming by creating a new firmware for the HP 20b calculator. Since the HP 20B calculator offers a number of ways to get into the embedded firmware of the calculator, our project proposes to explore the calculator, hack into the embedded firmware and potentially use the base calculator as a launching platform for custom operational versions. In other words, we'd like to take the HP-20b and reprogram it to do other things, including, potentially, emulating a computer system. During our lab sessions, we were able to break into its embedded code and made it act like a computer rather than a traditional calculator. Once we were able to code the software to do something that it should not do, we were able to reverse our hacking acts by making our calculators behave like calculators again.

Introduction

This project is an exploration into embedded systems, where we have written code integrated with an electronic system in order to implement our own functionality. In this case we have written firmware for an HP 20b calculator using the C programming language. In four labs we have coded the calculator from the basics of displaying text on the screen to a Reverse Polish Notation calculator.

Section 2 is a user guide to our calculator. Section 3 describes the social implications of our project and how it would affect the world. Section 4 describes the hardware platform of our project and how basic functions interact with the hardware. Section 5 is a summarization of our software architecture, i.e. the four labs. Section 6 goes into the details of our implementations of the four labs. Section 7 describes what we have learned from this project and Section 8 is the criticisms of this course and what improvements could have been made.

2 User Guide

For the first lab, the user does not have to provide any input. We have created a program that makes the calculator display the word Badboyz, scrolling from right to left. For example, the first three iterations will display:

					B	A	D	B	O	Y	Z

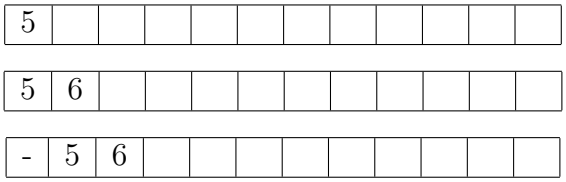
				B	A	D	B	O	Y	Z	

			B	A	D	B	O	Y	Z		

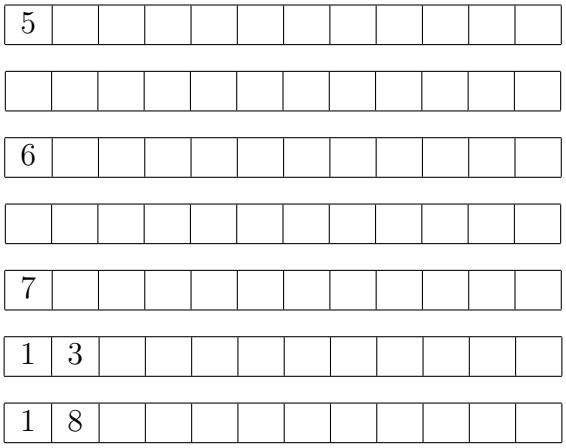
For the second lab, the user can press any key on the calculator. The calculator will display SEE on the first three spaces on the left, and the key that the user entered on the fifth blank. For example, if the user presses '5', '6', the display will be:

S	E	E		5							

For the third lab, the user can press number keys, operation keys or the negative sign key. The calculator will display numbers starting on the second black on the left when the user is inputting numbers, and will clear the screen once an operation key is pressed. The calculator will display a negative sign on the first blank if the negative sign key is pressed, and will clear it if the negative sign key is pressed again. However, the calculator will not do calculations. For example, if the user presses '5', '6', and the negative sign, the display will be:



If the user presses an operation sign, such as a plus sign, the screen will be cleared, until the user presses numbers again. For the fourth lab, the calculator behaves as an RPN calculator. The user has to input a number, and then press number to have the number stored in the calculator. The, the user has to input another number, before doing any operations. The calculator will take the number on the screen, the last pressed number, and the operation key to do calculations. The result will be displayed on the screen. The other input methods for the user and the displays are the same as the third lab, but the calculator is able to perform calculations now. For example, if the user inputs, '5', 'enter', '6', 'enter', '7', '+', '+', the display will be:



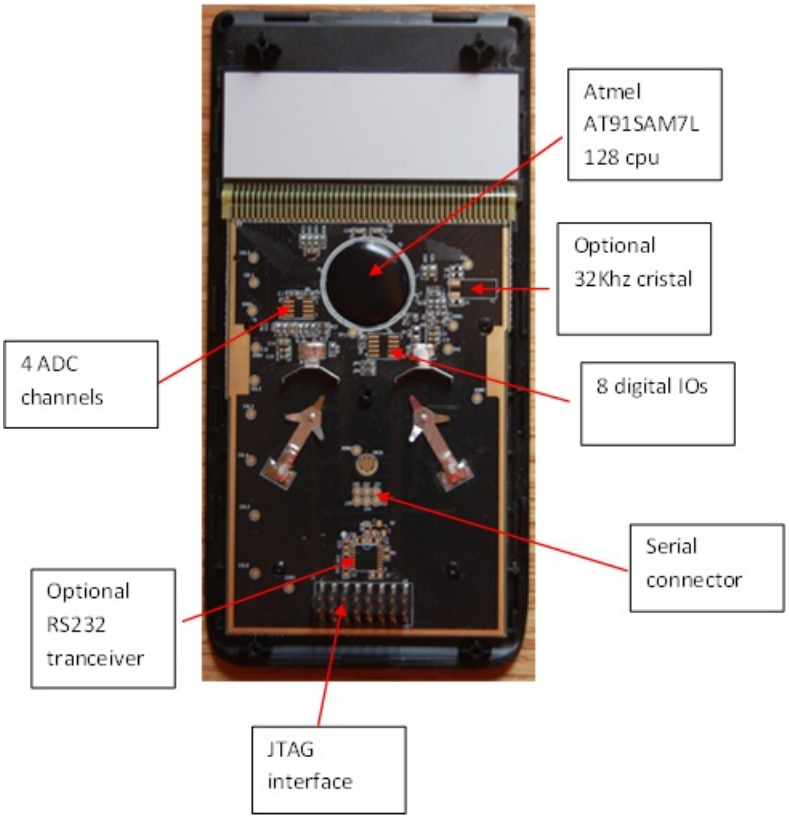
Section 3

The social implications of our calculator are far and widespread. It will allow people to perform quick and error-free calculations that will help in many situations such as trade.

Portable devices such as this calculator will also allow developing countries to improve their education system. This calculator can be brought to classrooms all over the world and learning will not be restricted by geography. Although calculator may seem detrimental to math studies, it can be beneficial when the intent of usage is proper. It will save valuable time for productive learning, especially at a higher level where the basics has been mastered, and more emphasis is placed on concept understanding. Moreover, since this calculator is just a precursor for more advanced embedded systems, it can be used to create a customized program that could be used for his own use.

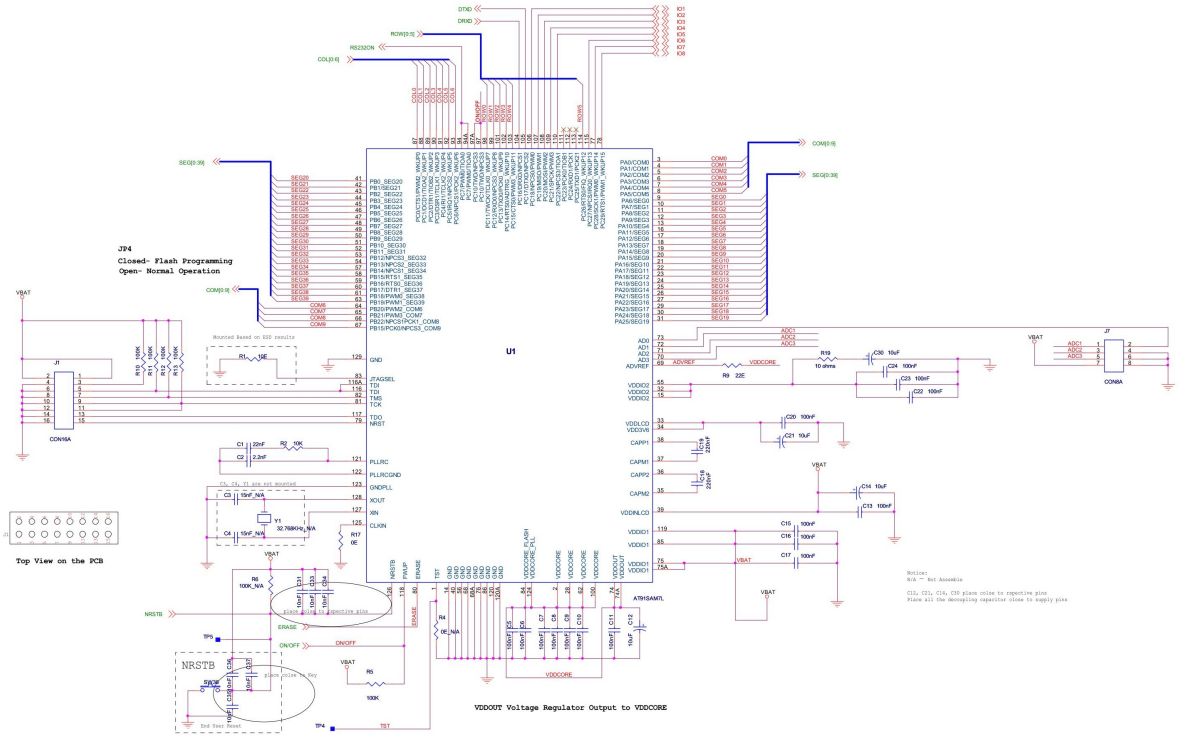
4 The Platform

HP 20B calculator is made of series of Atmels AT91AM series of a single standard processor chip, which are built around the ARM processor core, surrounded by memory, a system controller, and LCD controller. There are series of microcontrollers that are designed for lower power usage. The HP20B calculator has a build in 128KB of flash rom in the 30Mhz ARM cpu that powers it. Figure 1 shows a block diagram of the AT91SAM71 chip. It is able to run on low battery because the system controller controls the power supply and the clock through software, allowing it to save energy by not powering on the unnecessary peripherals.[1]



Joint Test Action Group (JTAG) is an interface that is generally used for IC debug ports,

CPU programming, and application development. The JTAG port is connected to the AT91SAM7L128 processor, which allows the communication with the processor. HP20b calculators circuit board has a slot to link the connector cable to bring out the JTAG signals and the JTAG adapters are connected to the USB ports. During our lab workshop sessions, we used a software called OpenOCD on Linux.[2] The OpenOCD software allowed us to communicate to the AT91SAM7L128 CPU through USB and JTAG.

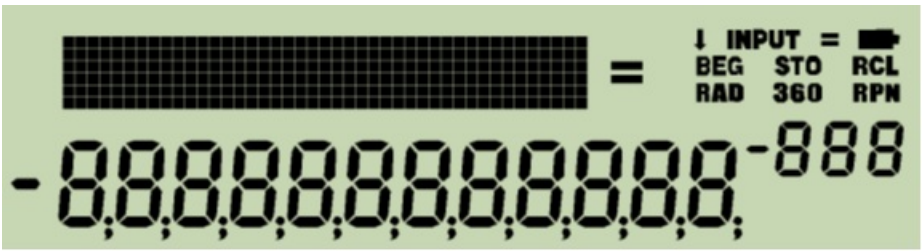


LCD display

A Liquid crystal display (LCD) is a flat panel display that uses the light modulating properties of liquid crystal. These crystals are actually liquid chemicals that align perfectly when subjected to electrical fields; when they're properly aligned, they allow light to pass through them. LCDs use this property by using electrical currents to align the crystals and allow varying levels of light to pass through and create the desired images.[3] HP 20B calculators LCD is displayed when complex AC waveforms are generated by the LCD controller. The liquid crystals are sandwiched between two pieces of polarized glass. The fluorescent light source, known as the backlight, emanates light that passes through the first substrate.[4] The electrical currents then cause the crystals to align, allow varying levels of light to pass through to the second substrate. The end result is what you see onscreen.

The picture below shows a LCD display of 400 pixels screen containing a 6*43 matrix display,

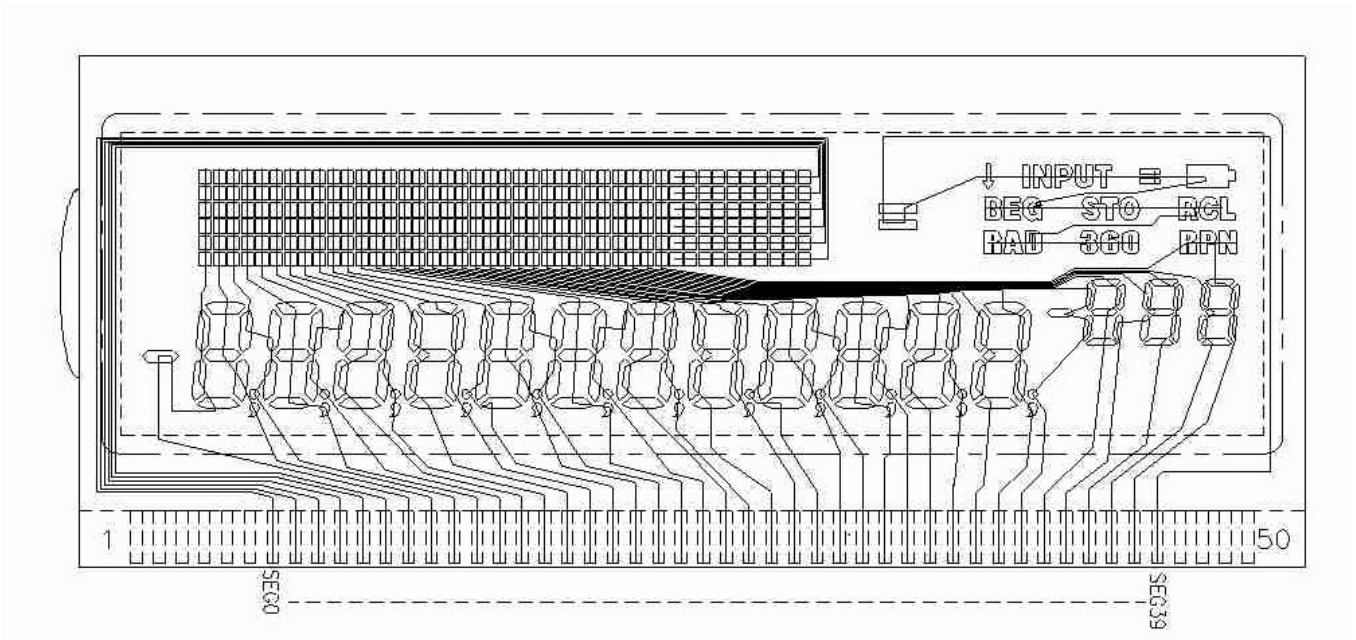
12+3 digits 7 segment display and 11 indicators; this is what LCD looks like if all the pixels are turned ON. In appearance to software, LCD looks like a series of memory locations whose bits control individual LCD segments.[5]



LCDs are available in wide range of sizes and are very energy efficient. Another benefit of LCDs is that it requires low electrical power and thus enables it to be used in battery-powered electric equipment.

As introduced in lab, the lcd_init() function turns on the power supply of the LCD so that the calculator can actually print to it. The lcd_put_char7() function takes in two parameters, ch and col. The ch parameter is the ASCII character that is to be printed out onto the screen while col is the column that the character is to be printed. The lcd_print7() function takes in a char * and simply iterates through all 15 columns calling lcd_put_char7() on each character of the array.

The figure below shows the schematics of the LCD.



Keyboard

HP20Bs keyboard functions in a similar way as the computer keyboards that we took apart in class. In this section we will explore HP20Bs keyboard technology including its microprocessor SAM7L chip, and its circuitry.

HP20Bs keyboard has a standard matrix type with miniature wires that run through its rows and columns and these wire circuits are able to be shorted together by the keys (Figure I show the keyboards layout). HP20Bs keyboard has its own SAM7L chip and circuitry that carries information to and from that processor.[6] A large part of this circuitry makes up the key matrix, which is a grid of circuits underneath the keys; each circuit is broken at a point below each key. The SAM7L chip and pins allows the calculator to translate which buttons are being pressed by reading the state of each pin.[7] (Figure 3 shows the schematics of the HP 20Bs keyboard).

For example, when a button is pressed, a circuit is completed, allowing a short current to flow through. When a button is pressed down long enough, the processor comprehends the signal as if you were pressing the key repeatedly. Furthermore, since HP20Bs keyboard is mechanical in nature, mechanical key switch allows metal to metal contact. When a button is pressed, it pushes down the metal beneath the calculator, and presses against the flat surface of the key matrix and completes a circuit. When released, it forces the button to go back to its original resting position. The processor deciphers which key is being pressed by finding a circuit that is closed, and compares it to a character map, which expresses the position of each key in the matrix and what each keystroke represents.

		"rows"					
		PC11	**PC12**	**PC13**	**PC14**	**PC15**	**PC26**
"columns"	**PC0**	N	I/YR	PV	PMT	FG	Amort
	PC1	CshFl	IRR	NPV	Bond	%	RCL
	PC2	INPUT	(	)	+/−	←	
	PC3	▲	7	8	9	÷	
	PC4	▼	4	5	6	×	
	PC5	shift	1	2	3	−	
	PC6		0	.	=	+	

Figure 1: The HP 20b's keyboard layout. When pressed, each key shorts two pins: one for its column, one for its row.

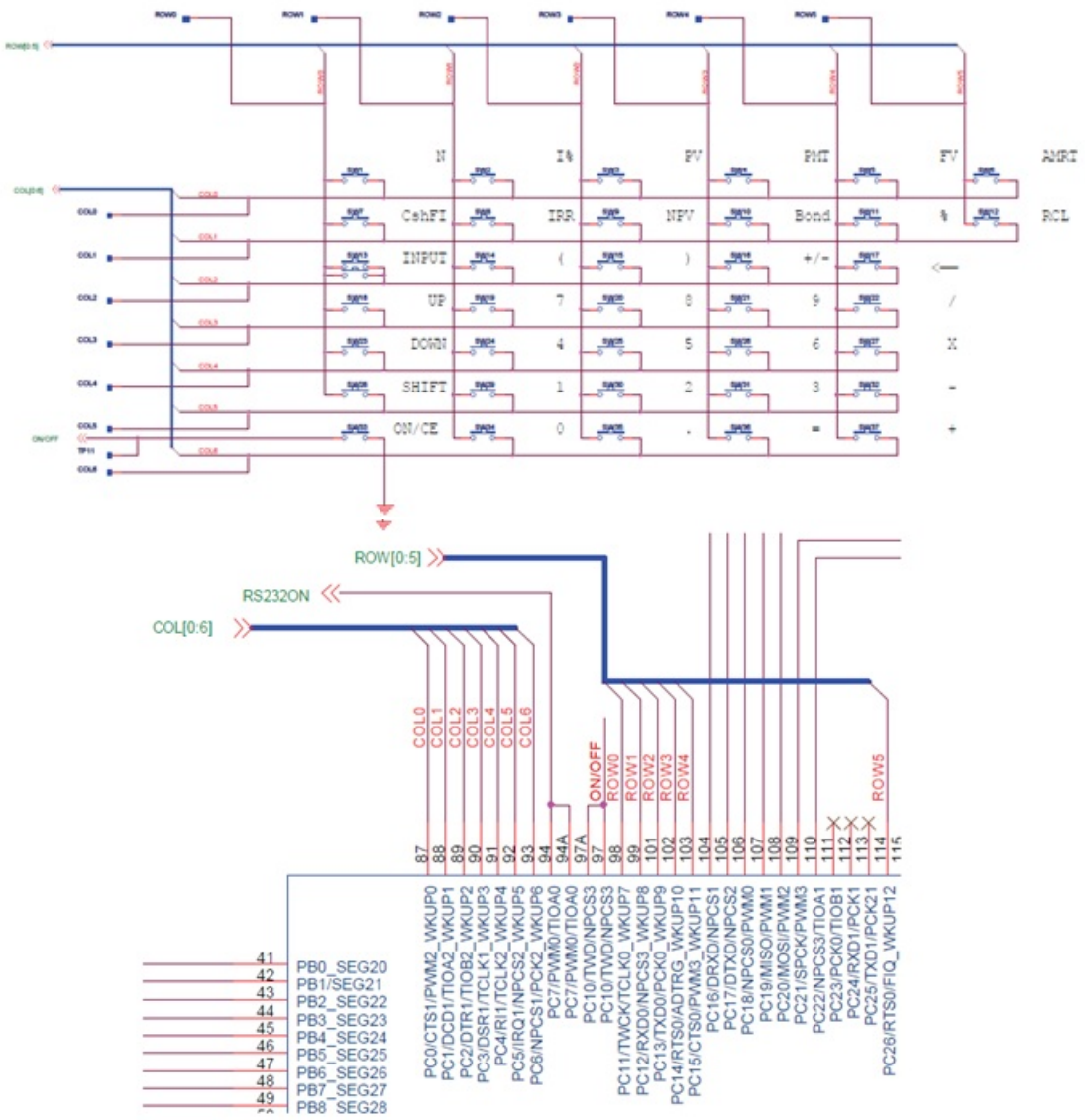


Figure 3: Schematics for the HP 20b keyboard. The top is the keyboard matrix itself; the bottom shows how the matrix is connected to the SAM7L chip. Note that the ON/CE key is separate (not part of the matrix) and that "columns" of keys are actually horizontal.

Section 5

The software architecture of our calculator is based on four labs. In Lab 1 we wrote a program to display a scrolling message from right to left that looped when it reached the end. This lab introduced the basic LCD functions that allowed us to put text onto the screen. The second lab was reading input from the keyboard and printing it out onto the screen, but was only able to display one key. The third lab combined the first two labs into continuously displaying multiple user inputs on the screen as well as storing that input into an underlying data structure. The fourth lab used the previous three labs to make a

Reverse Polish Notation calculator. It read input from the keyboard, displayed it on the screen and stored the input in a data structure as well as performed operations on these data structures. We essentially rebuilt the calculator from the ground up from display to input to functionality with each lab built off the previous ones.

6 Software Details

We have written four labs which have modified the calculator in multiple ways throughout the semester: a Scrolling Display, Scanning the Keyboard, Entering and Displaying Numbers, and an RPN calculator. The sub-sections here will explain in detail our implementations for each lab.

6.1 Lab 1: A Scrolling Display

In this lab, the calculator displays a string of words on the screen, scrolling from right to left. Figure 6.1 below contains the code for this lab.

First, we have an array of characters called name[], which contains the word we want to scroll, Badboyz. We also create a pointer for this string, which points at the first character in the string in the beginning.

Then we create another array of characters called printScreen[], which is the material we want to print on the screen, and only contains blanks in the beginning.

Next we declare a few variables: integer n is a tool we use for the for-loop, integer lengthOf-Screen is twelve, which stands for the number of spaces on the calculator's monitor, integer counter keeps track of where the string Badboyz is at on the monitor.

The while(1) is the main loop in this lab that makes the scrolling continues forever.

We make the calculator print printScreen[], which is all blanks in the beginning. The following for-loop moves one character in printScreen[] to the left.

The next while-loop slows the program down, so that the display does not scroll too fast.

The next if-statement determines what printScreen[] should be for the next display. If the pointer is pointing at '\0', which is the end of the array Badboyz, the last element on printScreen should then be a blank, rather than '\0'. If the pointer is pointing at any other characters, the last element on printScreen[] should be the next character on Badboyz, and the pointer moves to the next character in Badboyz.

If counter reaches the end of the screen, we reset the pointer of Badboyz to the first character, and we reset the counter to zero as well, so that the process can work all over again.

At the very end, we increment counter by one, so it keeps track of the number of iterations.

```c
//main.c for Lab 1

#include "AT91SAM7L128.h"
#include "lcd.h"

int main()
{
   //initializes the screen
   lcd_init();

   //string to be printed
   char name[] = "Badboyz";

   //string for a blank screen
   char printScreen[] = "         ";

   //pointer for "Badboyz"
   char *pname = &name[0];

   int n;
   int lengthOfScreen = 12;
   int counter = 1;
   while (1)
   {
      //print blank in the beginning
      lcd_print7(printScreen);

      //scrolls one character to the left
      for (n = 0; n < lengthOfScreen; n++)
      {
         printScreen[n] = printScreen[n+1];
      }
      //slows down the iterations
      int x = 1;
      while (x < 50000)
      {
         x = x + 1;
      }
      //if it reaches the end of "Badboyz"
      if (*pname == '\0')
      {
         printScreen[11] = " ";
      }
      else //put char at "Badboyz" pointer to printScreen[] to be printed
      {
```

```
46          printScreen[11] = *pname;
47          pname = pname + 1;
48       }
49       //reaches the end of the screen, restart process again
50       if (counter == lengthOfScreen)
51       {
52          pname = &name[0];
53          counter = 0;
54       }
55       counter=counter+1;
56    }
57   return 0;
58 }
```

Figure 6.1: Our Solution to Lab 1: A Scrolling Message

6.2 Lab 2: Scanning the Keyboard

In this lab, we are asked to modify the keyboard_key method so that the calculator will display the key that is pressed by the user. Figure 6.2a contains the code in keyboard.c, and 6.2b contains the code for main.c. We are provided with a few methods: keyboard_init(), which resets every column to high voltage, keyboard_column_low(int column), which resets the particular column in the parameter to a low voltage, and keyboard_row_read(int row), which returns which row is being pressed as an integer.

We create a two-dimensional array const char keysPressed[columns][rows] which contains the corresponding characters on the keyboard of the calculator.

The modified keyboard_key() method will record which key is pressed by the user by determining where the intersection of the low voltage is, and returns an integer from the two-dimensional array. One important note is that our two-dimensional array is of type char, whereas we are returning an integer representation of the characters is the keyboard_key() method. The value returned by keyboard_key() is in fact the ASCII (American Standard Code for Information Interchange) value of the character. We will solve this problem in the main.c class.

In our keyboard_key() method, we first initializes the keyboard, which sets every column to high voltage. We then initializes two variables c for column, and row for rows. We then enter a double for-loop. The first for-loop iterates through the columns, and set each column to low. The for-loop within it iterates through the rows. If a button is pressed, that particular row and column will be shorted. With this information, we pass the shorted row and column into our two-dimensional array, which will return a character. We return an integer representation of this character if a key is pressed, and returns negative one if no keys are being pressed.

In our main.c, figure 6.2b, we have a for-loop that runs forever. We first initializes the

keyboard, so each column has a high voltage. We then declare two variables, char 'display' and int 'display2'. Char display will change the integer value of keyboard_key() into a character representation, which is the element in the two-dimensional array. Int 'display2' contains the integer value returned from keyboard_key(). We then have an if-statement, which will decide what to print out on the screen. If 'display2' does not equal to negative one (recall that in keyboard.c, it returns negative one if no keys are being pressed), it indicates that a key is being pressed, and the screen will display SEE on the first three spaces on the left, and 'display' at the fifth space, where 'display' is the character representation of the key pressed by the user. If 'display2' equals negative one, it indicates that nothing is being pressed, and the screen will display NO KEY on the monitor.

```
// keyboard.c for Lab 2

#include "AT91SAM7L128.h"
#include "lcd.h"
#include<stdlib.h>
#define KEYBOARD_COLUMNS 0x7f
#define KEYBOARD_ROWS 0x400fc00
#define columns 7
#define rows 6

const unsigned char keyboard_row_index[] = {11,12,13,14,15,26};

void keyboard_init()
{
  // Initialize the keyboard: Columns are outputs, rows are inputs
  AT91C_BASE_PMC->PMC_PCER = (uint32) 1 << AT91C_ID_PIOC; // Turn on PIOC clock
  AT91C_BASE_PIOC->PIO_PER = KEYBOARD_ROWS | KEYBOARD_COLUMNS; // Enable control
  AT91C_BASE_PIOC->PIO_PPUDR = KEYBOARD_COLUMNS; // Disable pullups on columns
  AT91C_BASE_PIOC->PIO_OER = KEYBOARD_COLUMNS; // Make columns outputs
  AT91C_BASE_PIOC->PIO_PPUER = KEYBOARD_ROWS; // Enable pullups on rows
  AT91C_BASE_PIOC->PIO_ODR = KEYBOARD_ROWS; // Make rows inputs

  AT91C_BASE_PIOC->PIO_SODR = KEYBOARD_COLUMNS; // Drive all columns high

}

void keyboard_column_high(int column)
{
  AT91C_BASE_PIOC->PIO_SODR = 1 << column;
}

void keyboard_column_low(int column)
{
  AT91C_BASE_PIOC->PIO_CODR = 1 << column;
```

```
}

int keyboard_row_read(int row)
{
  return (AT91C_BASE_PIOC->PIO_PDSR) & (1 << keyboard_row_index[row]);
}

// array of all keys
const char keysPressed[columns][rows] =
{
  {'N', 'I', 'P', 'M', 'F', 'A'},
  {'C', 'R', 'V', 'B', '%', 'L'},
  {'\r', '(', ')', '~', '\b', 0},
  {'\v', '7', '8', '9', '/', 0},
  {'\n', '4', '5', '6', '*', 0},
  {'S', '1', '2', '3', '-', 0},
  { 0,  '0', '.', '=', '+', 0}
};

int keyboard_key()
{
   // initlaize keyboard, reset every column to high
   keyboard_init();
   int x; // goes through rows
   int i; // goes through columns
   for (x = 0 ; x < columns; x++)
   {
      keyboard_column_low(x);

         for (i = 0 ; i < rows ; i++)
      {
            if (!keyboard_row_read(i))
         {
            return keysPressed[x][i];
         }
      }
   }
   return -1; // no keys are pressed
}
```

Figure 6.2a: Our Solution in keyboard.c to Lab 2: Scanning the Keyboard

```c
// main.c for Lab 2

#include "AT91SAM7L128.h"
#include "lcd.h"
#include "keyboard.h"

int main()
{

  lcd_init();
  keyboard_init();

  for (;;)
  {
   keyboard_init();
   char display = keyboard_key(); // returned char from keyboard_key
   int display2 = keyboard_key(); // returned int from keyboard_key

   if (display2 != -1) //if some key is pressed
   {
      // display
      lcd_print7("SEE");
      lcd_put_char7(' ', 3);
      lcd_put_char7(display, 4); //display key-pressed at 5th blank
      lcd_put_char7(' ', 5);
   }
   else
   {
      lcd_print7("NO KEY"); //display when nothing is pressed
   }


  }

  return 0;
}
```

Figure 6.2b: Our Solution in main.c to Lab 2: Scanning the Keyboard

6.3 Lab 3: Entering and Displaying Numbers

In this lab, we are asked to enhance the method modified in lab 2, and make the calculator display integers continuously when the user presses number keys, and clear the screen when an operation is being pressed. Figure 6.3a contains the code in our keyboard.c class, and Figure 6.3b contains the code in our main.c class.

In our keyboard.c class, we now have the functioning keyboard_key() method from the last lab. We used professor Edward's code for the keyboard_key() method because we feel that his program is better written and will less likely contain error.

We have a new method in this lab, keyboard_get_entry(struct entry *result), which does actions according to the value read in from the keyboard_key method(). In this method, we first declare a variable of type integer, ent, which is the value of keyboard_key(). We have an if-statement afterwards. If ent does not equal to negative one (recall negative one is returned by keyboard_key() if no keys are being pressed), then we move into a list of actions.

If ent equals one hundred and twenty-six, which is the ASCII representation of a negative sign, then we set the number in the struct as NEGATIVE_SIGN, which is one hundred, and the operation in the blank as a blank.

If ent is between the ASCII value of forty-eight and fifty-seven, which represents one and nine respectively as characters, the number is struct is set to be ent minus forty-eight, which will yield the correct number. And again the operation in the struct is set as a blank.

To make our code a little shorter, rather than making more if-statements for the operation keys, we create a switch statement, which will do actions according to the value of ent.

Case forty-three is the case for the plus sign, since plus has the ASCII value of forty-three. We set the operation of the struct as +, and the number in the struct as negative one, indicating that the key pressed is not a number. The break statement makes the program skip the other cases and goes to the end of the switch statement. The other cases work exactly the same, returning the correct operation sign according to their ASCII value. We have a line lcd_init() in the case of an equal sign because we want to be sure that the screen is clear and fresh after the user has entered an equal sign.

If the ent, the number returned by keyboard_key(), is negative one, it indicates that no keys are pressed, and we set the number in the struct as NO_KEY_BEING_PRESSED, which is negative two.

Now that we have these helper methods in the keyboard.c class, we can make adjustments in our main.c class. In our main method, we have declared a few variables. We first create a string of characters printScreen[], which contains all blanks in the beginning (this technique is the same as in lab1). Int pos is the position of where the next character should be printed on the screen. Int pressed is a variable that determines whether a key is being pressed, and

will be used to stop the calculator from printing the same character over and over again. This point will be further illustrated below. Int lengthOfScreen contains the number of blank spaces on the screen, which is eleven, excluding the first blank which is saved for the negative sign.

Then we move into a forever for-loop, which will determine which actions to take and what values to print on the screen. First thing we do in the for-loop is to get the number and the operation in the struct from the keyboard_get_entry(struct entry *result) from the keyboard.c class.

The following if-statement prevents the calculator from printing the pressed key over and over again. If entry.number equals NO_KEY_BEING_PRESSED, which is negative two, then the value of pressed becomes one.

The program will enter the following if-statement if a key is pressed, which means that pressed equals one, and entry.number does not equal NO_KEY_BEING_PRESSED. Now we know that something is being pressed by the user, we take that value and determine what key is pressed, and does appropriate action.

The following if-statement checks if the negative sign is being pressed. If the negative sign indeed is being pressed, and the first element in printScreen[] (printScreen[0]) is not a blank, then the first element in printScreen[] becomes a negative sign. If the first element already has a negative sign, then remove that negative sign. We print the value in printScreen[] onto the monitor afterwards, which concludes our check for negative sign.

The next if-statement checks if an operation key is being pressed. If entry.number equals negative one, which means that an operation key is being pressed, the screen is initialized, the the whole printScreen[] array becomes all blanks. The position is reset to one, so that when the user enters the next number, it will be placed at the second space on the left.

Finally, if the user entered a number between zero and nine, we place that number at the position of printScreen[], which will be one in the beginning. We print printScreen[] on the screen afterwards, and increment pos by one, so that if the user enters another number, the number will be added to the array printScreen[], rather than overwriting the previous numbers entered.

After all the checks, we set the value of pressed to zero, so that when it returns to the top of the forever for-loop again, the program will not automatically print the same number over and over again.

Even though this integer pressed seemed trivial, we had taken a very long time to figure this out. We have tried to make changes to the program, but it was still printing one number on the whole screen. So we stopped and started to discuss on why the calculator was behaving in such a way. We found out that even though the user pressed and released the key in one second, the one second would still mean a long time to the calculator, thus it would assume the user was holding on the button, and thus printed out the same number all over

the screen. This idea came from the while-loop we made in the first lab to slow the scrolling down. We needed some variables in the program to prevent the calculator from printing one value for many times. Finally, we came up with an algorithm to stop the program, which was to add in this variable pressed with changing values to prevent the program from repeatedly entering the if-statement.

```
1  //keyboard.c for Lab 3
2
3  #include "AT91SAM7L128.h"
4  #include "keyboard.h"
5
6  #define NUM_COLUMNS 7
7  #define NUM_ROWS 6
8  #define KEYBOARD_COLUMNS 0x7f
9  #define KEYBOARD_ROWS 0x400fc00
10 #define NO_KEY_BEING_PRESSED -2
11 #define NEGATIVE_SIGN 100
12
13 const unsigned char keyboard_row_index[] = {11,12,13,14,15,26};
14
15 /* Character codes returned by keyboard_key */
16
17 const char keyboard_keys[NUM_COLUMNS][NUM_ROWS] = {
18   {'N', 'I', 'P', 'M', 'F', 'A'},
19   {'C', 'R', 'V', 'B', '%', 'L'},
20   {'\r', '(', ')', '~', '\b', 0},
21   {'\v', '7', '8', '9', '/', 0},
22   {'\n', '4', '5', '6', '*', 0},
23   {'S', '1', '2', '3', '-', 0},
24   { 0,  '0', '.', '=', '+', 0}};
25
26 void keyboard_init()
27 {
28   // Initialize the keyboard: Columns are outputs, rows are inputs
29   AT91C_BASE_PMC->PMC_PCER = (uint32) 1 << AT91C_ID_PIOC; // Turn on PIOC clock
30   AT91C_BASE_PIOC->PIO_PER = KEYBOARD_ROWS | KEYBOARD_COLUMNS; // Enable control
31   AT91C_BASE_PIOC->PIO_PPUDR = KEYBOARD_COLUMNS; // Disable pullups on columns
32   AT91C_BASE_PIOC->PIO_OER = KEYBOARD_COLUMNS; // Make columns outputs
33   AT91C_BASE_PIOC->PIO_PPUER = KEYBOARD_ROWS; // Enable pullups on rows
34   AT91C_BASE_PIOC->PIO_ODR = KEYBOARD_ROWS;  // Make rows inputs
35   AT91C_BASE_PIOC->PIO_SODR = KEYBOARD_COLUMNS; // Drive all columns high
36
37 }
38
39 void keyboard_column_high(int column)
```

```
{
  AT91C_BASE_PIOC->PIO_SODR = 1 << column;
}

void keyboard_column_low(int column)
{
  AT91C_BASE_PIOC->PIO_CODR = 1 << column;
}

int keyboard_row_read(int row)
{
  return (AT91C_BASE_PIOC->PIO_PDSR) & (1 << keyboard_row_index[row]);
}

//keyboard_key function for reading what the user has pressed
// from Professor Edward's program
int keyboard_key()
{
  int row, col;
  for (col = 0 ; col < NUM_COLUMNS ; col++)
  {
     keyboard_column_low(col);
     for (row = 0 ; row < NUM_ROWS ; row++)
     {
        if (!keyboard_row_read(row))
        {
           keyboard_column_high(col);
           return keyboard_keys[col][row];
        }
     }
     keyboard_column_high(col);
  }
  return -1;
}

void keyboard_get_entry(struct entry *result)
{
   int ent = keyboard_key();
   // if something is pressed
   if (ent != -1)
   {
      //All the numbers are in ASCII
      ent = keyboard_key();

      if (ent == 126) //if the user inputs a negative sign
```

```
85      {
86         result->number = NEGATIVE_SIGN;
87         result->operation = ' ';
88      }
89      if (ent >= 48 && ent <= 57) // if the user presses 1-9 on the calculator
90      {
91         result->number = ent - 48; //convert form ASCII
92         result->operation = ' ';
93      }
94      switch(ent) //If the user inputs an operation.
95      {
96         case 43: // "+" sign
97            result -> operation = '+';
98            result -> number = -1;
99            break;
100        case 45: // "-" sign
101           result -> operation = '-';
102           result -> number = -1;
103           break;
104        case 42: // "*" sign
105           result -> operation = '*';
106           result -> number = -1;
107           break;
108        case 47: // "/" sign
109           result -> operation = '/';
110           result -> number = -1;
111           break;
112        case 61: // "=" sign
113           result -> operation = '=';
114           result -> number = -1;
115           lcd_init();
116           break;
117     }
118  }
119  else //If no key is being pressed.
120  {
121     result -> number = NO_KEY_BEING_PRESSED;
122  }
123 }
```

Figure 6.3a Our Solution in keyboard.c to Lab 3: Entering and Displaying Numbers

```c
// main.c for Lab 3

#include "AT91SAM7L128.h"
#include "lcd.h"
#include "keyboard.h"

int main()
{

  lcd_init();
  keyboard_init();

  for (;;)
  {
   keyboard_init();
   char display = keyboard_key(); // returned char from keyboard_key
   int display2 = keyboard_key(); // returned int from keyboard_key

   if (display2 != -1) //if some key is pressed
   {
      // display
      lcd_print7("SEE");
      lcd_put_char7(' ', 3);
      lcd_put_char7(display, 4); //display key-pressed at 5th blank
      lcd_put_char7(' ', 5);
   }
   else
   {
      lcd_print7("NO KEY"); //display when nothing is pressed
   }
  }
  return 0;
}
```

Figure 6.3b: Our Solution in main.c to Lab 3: Entering and Displaying Numbers

Section 6.4

In this lab we created a Reverse Polish Notation calculator that extends off the previous three labs. We used a stack to store the numbers that the user entered because RPN notation only operates on the numbers most recently entered, which a stack is perfect for. The members of this programs was the stack, which was simply an array that held ten integers and a stack pointer, which was an integer that held the position the next element would be added. The push and pop functions are written into the main method and they respectively add and remove elements from the stack while also changing the stack pointer accordingly. The

program read the input from the user using the keyboard_get_entry function from lab 3 and performed operations depending on the input. There are three cases:

If the input was a number followed by INPUT, the program pushed the entry into the stack.

If the input was a number followed by an operation, the program pushed that number onto the stack and popped the top two numbers off the stack. Then the program performed the arithmetic operations on those two numbers, pushed the result back onto the stack and displayed the result on the screen. A switch case statement was used to select the correct operation.

If the number was just an operation, it was necessary to decrement the stack pointer before popping off the top two numbers so that the arithmetic operation wasn't performed on garbage values.

At the end of the loop the stack pointer is incremented because it always points to the space right after the loop and we decremented it to access members when doing the operations.

```
1  // main.c for Lab 4
2
3  #include "AT91SAM7L128.h"
4  #include "lcd.h"
5  #include "keyboard.h"
6
7  int stack[10]; //Our stack is an array of 10 integers
8  int stackPointer;
9
10 int main()
11 {
12     stackPointer = 0;
13     struct entry entry;
14     // Disable the watchdog timer
15     *AT91C_WDTC_WDMR = AT91C_WDTC_WDDIS;
16
17     lcd_init();
18     keyboard_init();
19
20     for (;;) {
21
22         int num1;
23         int num2;
24         int result;
25
26         keyboard_get_entry(&entry);
27
28         if (entry.operation == '\r') /*If the user enters a number
```

```
                                add it to the stack*/
    {
        stack[stackPointer++] = entry.number; // Advance the stack pointer
        lcd_init();
    }
    if (entry.number != INT_MAX)          // If a number is entered
    {
        stack[stackPointer] = entry.number;
    }

    if (entry.operation == '+' ||         // User has entered an operation
        entry.operation == '-' ||
        entry.operation == '*' ||
        entry.operation == '/' )
    {
        if (entry.number == INT_MAX)      // Only an operation is entered
        {
            stackPointer--;
        }

        num1 = stack[stackPointer--]; // Pop the first number and decrement
                                      // the stack pointer
        num2 = stack[stackPointer]; // Pop the next number

        switch(entry.operation)
        {
            case '+':

                result = num1 + num2;
                stack[stackPointer] = result; //Push the result onto the stack
                lcd_print_int(result);
                break;

            case '-':

                result = num2 - num1;
                stack[stackPointer] = result;
                lcd_print_int(result);
                break;

            case '*':

                result = num1 * num2;
                stack[stackPointer] = result;
                lcd_print_int(result);
```

```
                    break;

                case '/': //Division does not work due to faults we cannot control

                    result = num1 / num2;
                    stack[stackPointer] = result;
                    lcd_print_int(result);
                    break;
            }
            stackPointer++;
        }
    }
    return 0;
}
```

Figure 6.4: Our Solution in main.c to Lab 4: Reverse Polish Notation Calculator

7 Lessons Learned

Throughout this semester, we have been working with the C programming language. We learned not only about the syntax of C programming language, but more importantly, to formulate algorithms before approaching problems with an engineers mindset. In the first two labs, we started programming right after we have received the assignment, and luckily, it did not take us too long to finish the assignment. However, this method did not work for us in the third and fourth lab. We started programming right after we received the assignment again, but we could not figure out the solution, and it seemed that we were running in circles. Rather than programming like a blind man with a Rubiks cube, we took out a piece of paper and started to write out our algorithm for the problem. We wrote down the different variables, and the outcome of each steps of the program. In about five minutes of thinking, discussing and writing out the iterations, we were able to find a crucial error in our program, and immediately made adjustments to our program. The most important lesson we learned from this experience is that before approaching the problem, we should always think out the algorithm and write it out on a piece of paper. This is a valuable skill to have, and can be applied not only to computer science, but other subjects as well.

8 Criticism of the Course

In the beginning of this semester, we were highly encouraged by Professor Vallancourt to take a gateway subsection that we have never been exposed to in order to broaden our aspects of different fields of engineering. Accordingly, some of us who came into this course were newly exposed to computer science and had a difficult time in understanding the shotgun introduction to C programming language. On the other hand, the other half of our members who have been programming for years and years were able to work at the fast paced environment and were able to grasp the materials very quickly. Regardless of our background knowledge of computer science, we all truly enjoyed being in this computer programming section of our

gateway course because we were all able to transition out of our science-oriented high school way of thinking to an engineering point of view. Fundamental concepts of computer science and hacking skills was learned and reviewed in an engineering context. We especially enjoyed the time when we broke apart a broken computer keyboard to learn about the schematics of the plastic keyboard processor chips and how the matrix circuits grid worked underneath the keys. By doing so, we were easily able to implement the knowledge to our calculators keyboards.

Although this course definitely had more pros than cons, there were several things that could be improved in this course. First the rooms were a bit uncomfortable because of its tight space and sometimes we ended up standing or kneeling because there were not enough chairs for everyone. Secondly, since all of the students in this course had a gateway lecture starting at 10:00am and continued onto gateway section right afterwards until 4:00pm, which usually ended up extending until around 4:30pm, we basically had to spend our whole Friday in a lab. Although we all love computer science and didnt mind staying later to up the labs, we sometimes wished that we could get out early to enjoy our Friday. In future, it would be beneficial for students if they were allowed access to the computers labs to work on the calculator project during the weekdays evenings. This would be beneficial especially for those students who have other extracurricular activities on Friday evenings and would give students ample time to finish up the required labs.

In addition, because of the varying backgrounds of everyones experience in computer programming, we think it would be a great idea if there could be some extra lecture sessions for those who need extra help in computer language and the basics of programming. This would allow the newbies to catch up faster and understand the lab materials better. Because each one of us had different background knowledge in computer science, its difficult to describe the difficulty level of this course. We could say it was a split because the individual labs definitely came easier for some and difficult for some others. But overall, this course was very enjoyable and manageable for everyone.

During the start of our lab, we usually always had a clear general idea of how the pieces should fit together in the end, but the problem was writing down the code that would support our big picture. When we had trouble finding the bug and got stuck on figuring out the code, it was very helpful to discuss it with the group and go over the code step by step. Writing out the plan of action before starting to code was very helpful as well. Nonetheless the code reviews were the most helpful material. By looking at the code reviews, we were able to compare our code with other groups results to improve upon our codes. Since every group came up with different coding solutions, it was interesting to compare our results with other groups results and find a simpler way making it work.

Notes

[1]Hp-20b repurposing project. Online http://www.wiki4hp.com/doku. php?id=20b:repurposing_project.

[2]Stephen Edwards. Repurposing an HP Calculator Lab 1: Hello World, Fall 2011.

[3]D.J.R. Cristaldi, S. Pennisi, F. Pulvirenti, "Liquid Crystal Display Drivers - Techniques and Circuits", Springer, Mar., 2009.

[4]Developer kit for HP 20b financial calculator, October 2009.

[5]Developer kit.

[6]Stephen Edwards. Repurposing an HP Calculator Lab 2: Listening to the Keyboard, Fall 2011.

[7]Developer kit.