ENGI E1112 Departmental Project Report: Computer Science/Computer Engineering

Jonathan Fletcher, Thomas Segarra, Marco Nedungadi, Amir Budhai

December, 2012

Abstract

Our goal was to examine possible solutions to common challenges faced by computer scientists/engineers through a series of labs in which we gradually built functional firmware for an HP20b calculator. The software was written in C, utilizing a library of functions created by the professor.

First, we wrote an algorithm that allowed the LCD to display positive and negative integers containing several digits, by breaking an integer into its component digits and displaying each one separately. We then coded a function to detect when a key was pressed, and identify which key; this required an understanding of the electrical mechanism by which a calculator keyboard's buttons work. Building upon this, we wrote a function that would store numbers in the calculator's memory and perform four basic operations (addition, subtraction, multiplication and division) on them.

The project compelled us to improve our communication skills, as those of us with programming experience had only been accustomed to programming alone. We learned the value of commenting clearly and frequently in our code, and gained an appreciation for the amount of time computer scientists spend debugging (not just software, but also hardware; we once spent hours befuddled by four consecutive dead batteries).

1 Introduction

The HP 20b calculator, meant especially for use in finance, business and real estate, is quite well-suited for hacking, because it is so well-documented by the company. It is unusual for the hardware and software of a calculator to be released so freely, but HP apparently encourages experimentation of this sort. This, coupled with Professor Edwards' diligence in arming the calculator to be programmed by

novices (providing, for instance, a library of useful functions, the details of many of which are beyond our comprehension), enabled us to write code in C that could be run by the calculator's powerful processor (see Section 3.1).

Our final product can perform four basic mathematical operations on positive and negative whole numbers, entered in Reverse Polish Notation (RPN; see Section 2.1). Though the HP20b calculator is usually capable of many advanced statistical functions and various entry modes, such things were above our current capacity to create. RPN may seem a counterintuitive method, but it is the simplest to implement; the details of the implementation are explored in Section 5.

2 User Guide

2.1 The RPN System: An Overview

This calculator utilizes RPN, or Reverse Polish Notation, a system developed by Charles Leonard Hamblin in the 1950s. Generally speaking, this means that the numbers are entered prior to the operations one wishes to perform on them. Each number is stored in a series of memory registers called the "stack," explained below.

2.2 Entering a Number

To input an integer, one must press the keys corresponding to the digits of the integer, starting from the leftmost place value. The backspace key (an arrow pointing left) eliminates the last digit pressed. To make the integer negative (or make a negative integer positive), press the +/- key.

2.3 The Stack & Performing Operations

The memory stack is a method for storing numbers to later be operated on. It can be visualized as a large vertical stack of boxes that can each hold one number. As a number is entered, it is in a state where it can be edited. It is not in the stack, because the stack stores numbers once they are finished being entered. It is instead stored in a temporary box. When finished entering a number, the INPUT button is pressed. This stores the number to the lowest unfilled box. The first number to be inputted will be on the bottom, then the next on top of it, and so forth. There are 16 boxes in the stack. After INPUT is pressed, the temporary box is reset, meaning that the number in it is not erased, but when the user begins typing, a new number will overwrite it. When an operation (+, -, *, /) key is pressed, the number in the top-most filled box and the number in the temporary box are operated on. For example, if the operation is subtraction, the calculator will perform (number in top-most filled box) minus (number in temporary box). It is always in this order. If all non-temporary boxes are empty, nothing happens. The calculator will

overwrite the temporary box and store the result of the operation in it. The box is then reset, as stated above. To store this value for use in a later operation, INPUT must be pressed to store it in the stack. For further clarification, see the examples that follow.

2.4 Examples

1. To subtract 2 from 5 (the operation 5 - 2), press the keys in the following order:

- 5 (first number)
- INPUT (stores 5 in memory)
- 2 (second number)
- - (performs subtraction)
- The screen will display a result of 3.

2. To perform (-2 * 3) + (21 / 7):

- 2, +/-, INPUT, 3, *, INPUT, 2, 1, INPUT, 7, /, +
- This will yield a result of -3.

Note that in each case, the number of times "INPUT" was pressed is equal to the number of times an operation was pressed. This means that every value saved to the stack was used in an operation. If "INPUT" is pressed *n* more times, there are *n* values still stored in the stack. If more operations are pressed than "INPUT," nothing will occur for the cases in which the stack is empty.

3 The Platform

3.1 The Processor

The processor in the HP20b is called the Atmel SAM7L; it is an ARM-based microcontroller. It is designed, among other things, to minimize consumption of power and support various peripherals (such as the LCD and the built-in "watchdog" timer), making it an ideal choice for a calculator of this size. It operates at 36 MHz, and has 128 kilobytes of flash memory. One of the processor's components responsible for its great efficiency is the "supply controller" (see Figure 2), which is a system within the processor's system controller, enabling the calculator to function in varying "power modes." This ensures that the calculator does not power on more elements than it must, or use more power than it needs.

Figure 1: The HP 20b, before and after development began

3.2 The LCD Display

The LCD has twelve spaces for displaying digits and sometimes a negative sign. It also has a second row above the first, which we did not make use of. A diagram of the signals sent to the display, from the HP20b SDK, is provided in Figure 3. The functions used to control the LCD via software were provided by the professor. Of these, the function most frequently used was lcd_put_char7, which takes one character and one integer argument, representing the character to be displayed and the position (from 0 to 11) in which to display it on the LCD, respectively. Another LCD-related function we used was lcd_init, which essentially turns on the display and prepares it to receive instructions.

3.3 The Keyboard

Our dissection of a computer keyboard yielded a rudimentary understanding of the means by which a computer senses and identifies key presses. Generally, there are three plastic sheets stacked on top of each other. The top and bottom sheets have circuits printed on them; the middle sheet prevents them from touching, except through holes corresponding to each individual key. Pressing a key causes the two circuits to connect through the hole. Since the circuits are arranged as a matrix, the position in the matrix at which the contact is made serves as a unique identifier,

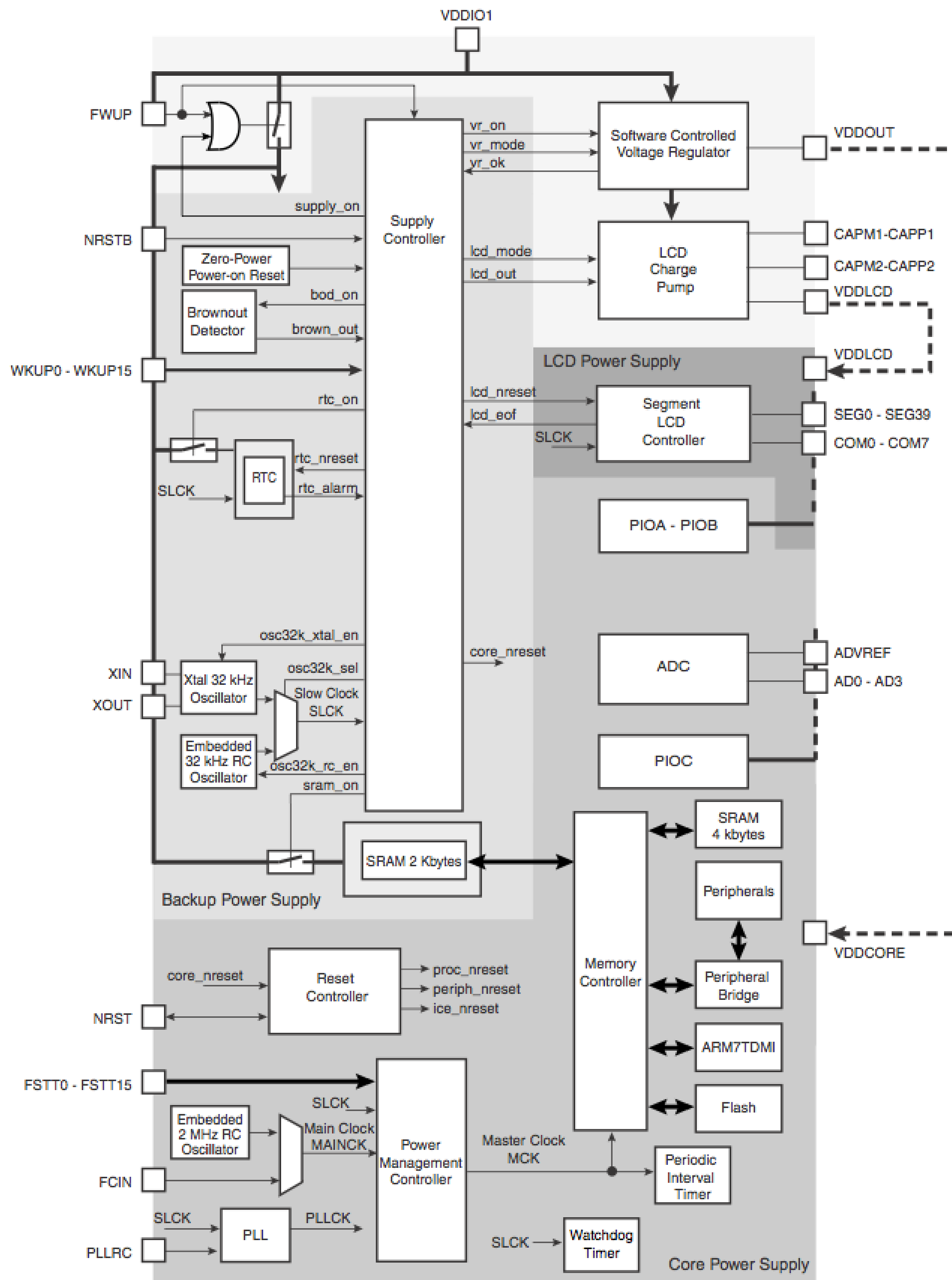

Figure 2: Diagram of the SAM7L's system controller, responsible for much of the processor's function

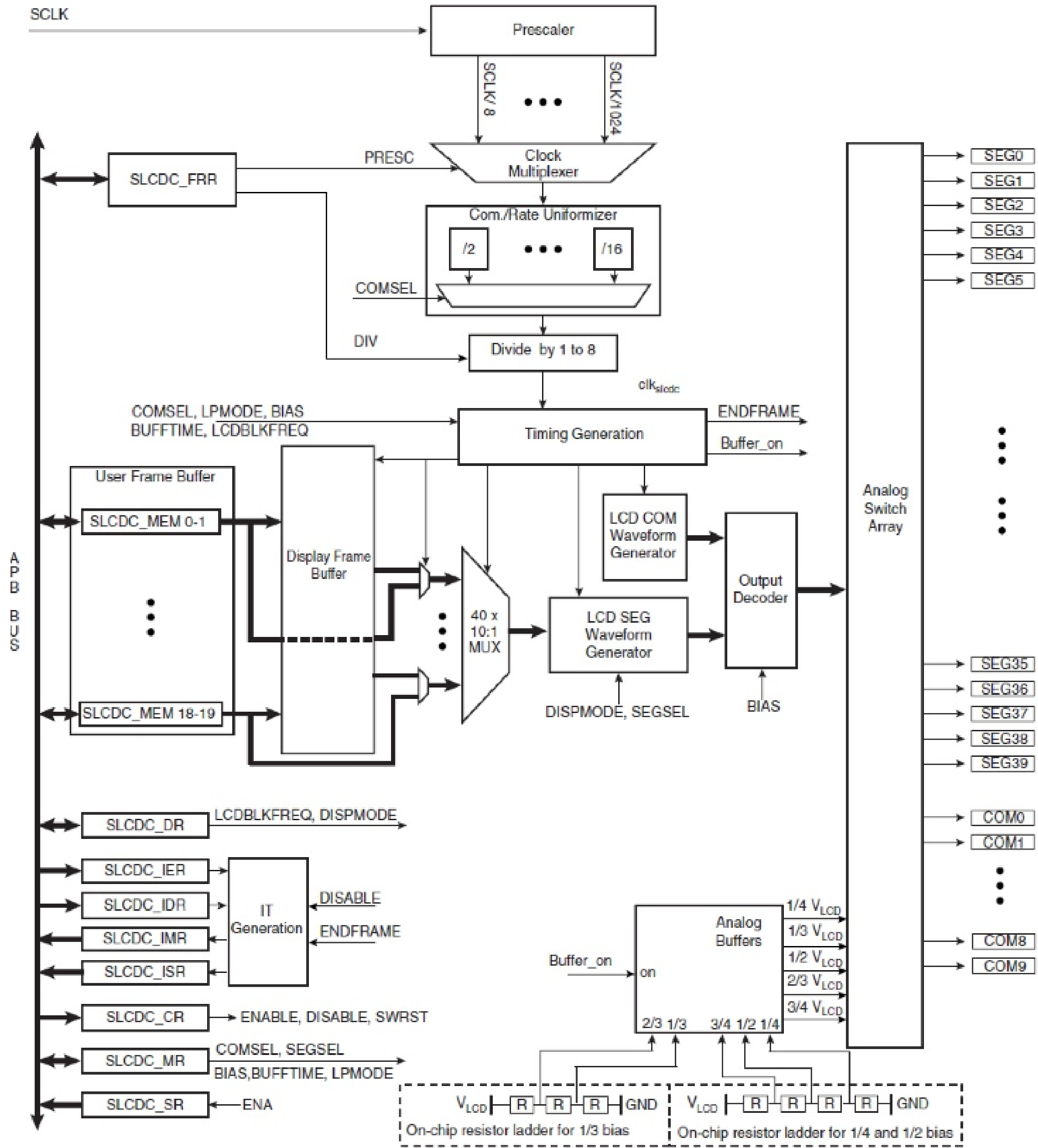

Figure 3: LCD Display Signal Diagram

sending an electrical signal to a unique combination of two pins on a processor, which sends that information to the computer.

The HP 20b's keyboard operates in a similar fashion: it is organized as a matrix of seven rows and six columns, with a logic 1 voltage across each row and column. These voltages can be measured and altered through software. Pressing a button creates a short circuit between two points, the location of which can be detected by an iterative algorithm, detailed in Section 5.

4 Software Architecture

The main() function first prepares the LCD and keyboard to exchange information with the processor, makes a "0" appear on the screen at startup, and then continually calls the keyboard_get_entry() function for an endless amount of time. keyboard_get_entry() waits, doing nothing until a key has been pressed and subsequently released, and then calls the keyboard_key() function, which scans the keyboard and returns the identity of the key that has been pressed. If it was a number key, keyboard_get_entry() places it on the display, appending an existing number if one has been entered; if it was the backspace key, it deletes the last digit of the number, if possible (i.e., the number is not 0); if it was the "INPUT" key, the entered number is placed on the stack as described in Section 2.3; if it was an operation key, that operation is performed as specified in Section2.3. keyboard_get_entry() then calls the display() function, which makes the result appear on the screen.

5 Software Details

5.1 Lab 1: Displaying an Integer

For this lab, we were asked to write a function that accepts a single integer argument, positive or negative, and displays it on the calculator's LCD.

Our code, shown in Figure 4, accepts one integer argument *a*. It begins by utilizing the professor's lcd_put_char7 function, explained in Section 3.2, to clear the screen; it uses a *for* loop to step through each of the LCD's eleven segments and display a blank space. An *if* statement checks whether the integer to be displayed is a 0. If it is, then it simply displays a zero and stops; otherwise, the *else* clause performs the digit-separation algorithm.

The algorithm begins by using an *if* statement to check whether *a* is negative. If it is, the boolean variable *isNeg* (otherwise assumed to be false) is set to true so that a negative sign can be added later, and *a* is set to negative *a*, so that the remainder of the algorithm can operate on the absolute value of the integer to be displayed.

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

#define SEGMENTS 11

void display(int a)
{
    int index = 11; // start with the rightmost LCD segment
    int isNeg = 0; //  initially assume the number is not negative

    for (i = 0 ; i <= SEGMENTS ; i++) {
        lcd_put_char7('␣', i);          // clear the screen
    }

    if (a == 0)
        lcd_put_char7('0', index);
    else {
      if (a < 0) {
            isNeg = 1; // record that the number is negative, then
            a = -a;    // use only its magnitude
      }
      while (a != 0) { // while there are digits left to display
        int digit = a % 10;                     // find last digit,
        lcd_put_char7(digit + '0', index);      // display it, then
        a = (a - digit)/10;                     // remove it from 'a'
        index--;
      }
      if (isNeg == 1) lcd_put_char7('-', index);
    }
}
```

Figure 4: Lab 1, Displaying an Integer

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

// Initialize the keyboard and set all columns high
// with pullups on the rows
extern void keyboard_init(void);

// Set the given column high
extern void keyboard_column_high(int column);

// Set the given column low
extern void keyboard_column_low(int column);

// Return true if the row is high, false otherwise
extern int keyboard_row_read(int row);

char key[7][6] =        {'N', 'I', 'P', 'P', 'F', 'A'},
                        {'C', 'I', 'N', 'B', '%', 'R'},
                        {'I', '(', ')', '~', '<',  0 },
                        {'^', '7', '8', '9', '/',  0 },
                        {'v', '4', '5', '6', '*',  0 },
                        {'S', '1', '2', '3', '-',  0 },
                        { 0 , '0', '.', '=', '+',  0 } };

char keyboard_key(){
        keyboard_init();
        for (i = 0 ; i < 7 ; i++){
                keyboard_column_low(i);
                for (j = 0 ; j < 6 ; j++){
                        if(!keyboard_row_read(j)){
                                return key[i][j];
                                // do not check while key is down
                                while(!keyboard_row_read(j)) {}
                        }
```

Figure 5: Lab 2, Scanning the Keyboard

```
                }
                keyboard_column_high(i);
        }

        return 0; // when no key is down
}

int main()
{
        // disable the watchdog timer
        *AT91C_WDTC_WDMR = AT91C_WDTC_WDDIS;

        lcd_init();
        keyboard_init();

        char c;
        while(1){
                c = keyboard_key();
                if(c == 0){                         // if no key is down,
                        lcd_put_char7(' ', 11); // display nothing
                } else {
                        lcd_put_char7(c, 11);
                }
        }
        return 0;
}
```

Figure 6: Lab 2, Scanning the Keyboard (cont.)

A *while* loop runs, displaying each digit until all the digits have been displayed. It uses the modulo operator to find the remainder when the integer is divided by ten, thus extracting the integer's last digit. This digit is then converted to an ASCII character by adding to it the ASCII value of '0', and displayed in the rightmost segment of the LCD (denoted by the variable *index*). It subtracts this digit from *a* and then divides the result by ten, effectively chopping off the last (already displayed) digit of *a*. It then decrements *index* so that the next digit can be displayed to the left of the previous one, and starts the loop over again, using the number's remaining digits.

When all the digits have been displayed, *a* becomes 0, so the *while* loop terminates. Finally, if the number was ascertained to be negative, a negative sign is placed in front of the number using lcd_put_char7.

5.2 Lab 2: Scanning the Keyboard

The goal in Lab 2 was to produce a function that could indicate which key was down, or that no key was down. The code itself can be found in Figure 5 and Figure 6.

First, a two-dimensional character array called *key* is created, containing a character corresponding to the value of each key; the position of a character in this matrix corresponds to the position of its key on the keyboard, for convenience. The spaces in the matrix that do not correspond to actual keys are given values of 0, making them equivalent to no key being pressed. When evaluating this approach, please note its striking similarity to Professor Edwards' key matrix, which we had not seen when we created ours.

The function keyboard_key(), which contains the bulk of this lab, takes no arguments and returns a character identifying the key that is down. It begins by calling keyboard_init(), a function provided by the professor which prepares the keyboard to sense key presses by setting all the voltages in the key matrix high. A *for* loop steps through each column, setting it low. Another *for* loop inside the first one steps through each row, testing each to see if any has a low voltage. If one does, then the key in the row and column of the present iteration must be down, and its character value (from the aforementioned array) is returned. A do-nothing *while* loop stops the algorithm from scanning while a key is down, so that one key press is not interpreted as many. If no key in the present column is down, that column is returned to high voltage, and the loop moves on to the next column. If all the columns were tested without finding a key down, the function returns the value 0, indicating that no key is being pressed.

The main() function in Figure 6 shows the way we tested our keyboard_key()

function. An infinite *while* loop calls keyboard_key(); if no key is down, the LCD displays a blank space. Otherwise, the LCD displays the identity of the key being pressed, which was returned by keyboard_key().

5.3 Lab 3: Entering and Displaying Numbers

The objective of Lab 3 was to write a function that would allow the user of the calculator to enter and edit numbers. We did this by means of a single function, keyboard_get_entry(), which is called from main() inside an infinite *while* loop. It takes no arguments and returns no values; this function executes all of the necessary commands itself.

```
void keyboard_get_entry() {
        int c;
        int num = 0;
        while(1){
                while(keyboard_key()!= -1); // wait if no key is down
                while(keyboard_key() == -1); // wait while key is down
                c = keyboard_key();
                if (c >= '0' && c <= '9') {
                        num = num * 10 + (c - '0');
                } else if(c == '\b'){ // backspace
                        num = num/10;
                } else if(c == '~'){ // +/-
                        num = -num;
                } else if(((c == '\r') || (c == '/')) || (((c == '*')
|| (c == '-')) || (c == '+'))){
                        return;
                }
                display(num);
        }
}

int main()
{
        // disable the watchdog timer
        *AT91C_WDTC_WDMR = AT91C_WDTC_WDDIS;

        lcd_init();
        keyboard_init();
```

```
        display(0);
        while(1) {
                keyboard_get_entry();
        }
        return 0;
}
```

The variable *num* stores the number to be displayed. When the function is first called, this is 0, as 0 is the default value displayed on most calculators. The function then enters an infinite *while* loop. Two do-nothing *while* loops cause the function to wait if no key is down, and wait while a key is down, so that a key press is only processed when the key is released. This, like our Lab 2 code, prevents unintended multiple key-presses. After this, it calls keyboard_key() to identify which key was pressed. If the key has an ASCII value between that of 0 and that of 9, inclusive, then a digit was pressed; the ASCII value is converted to its integer equivalent, then added to the existing number times ten. This increments the place value of every integer (if there are any) already on the screen, adding the new digit on the right, in an intuitive fashion. If the backspace key is pressed, *num* is divided by ten; since integer division in C rounds down, this serves to eliminate the last digit entered. If the +/- key is pressed, *num* is simply multiplied by negative 1. If the "INPUT" key or any operation is pressed, the while loop is terminated. keyboard_get_entry() is therefore called again (since it is, itself, inside an infinite while loop in main()), which allows entry of a new number to begin. No matter which key was pressed, at the end of one iteration of the *while* loop in keyboard_get_entry(), the modified value of *num* is displayed on the LCD.

This section contains one element we should have implemented differently: the handling of the +/- key. It is incompatible with the way digits are entered. If one has entered "-13," and wishes to make it "-135," intuition would dictate that the next step is to press 5. However, pressing 5 at this point would, in fact, give a result of -125; -13 would be multiplied by 10, giving -130, and then the 5 would be added. To achieve a proper result, one must make 13 positive, add the 5, then make it negative again. This quirk would surely irritate any reasonable consumer. A superior implementation would be the one we used in Lab 1, whereby the number's negative status is stored as a boolean variable apart from the number itself.

5.4 Lab 4: An RPN Calculator

In lab 4, we created the code for operating an RPN calculator. The results of our implementation differ slightly from those described during class, but use the same fundamental principles and will yield results just as accurately.

One problem we encountered was that for an unknown reason, the calculator has trouble dividing numbers. Thus, we had to implement our own function for division. It works by subtracting the numerator from the denominator and counting (incrementing a variable) each time this must occur until another instance of subtraction would "pass" 0. The function detects whether or not the quotient will be negative by testing the product of the operands. Both numerator and denominator are treated as positive, and the negative is "added back in" at the end of the calculation. Because of limitations in our general code in dealing with decimals, all division rounds down.

```
int divide(num,den)
{
        int quotient = 0;
        int neg = 0;
        if (num * den < 0) neg = 1;
        num = abs(num);
        den = abs(den);
        while (num >= den) {
                num = num - den;
                quotient++;
        }
        if (neg == 1) {
                quotient = quotient * -1;
        }
        return quotient;
}
```

The stack we use for storing numbers is an array of length 16. The length we chose somewhat arbitrary; we found it unlikely that someone needing to store that many numbers would rely on such a simple calculator, and anyone able to remember that many numbers most likely does not need a simple calculator. We decided that 16 values provided ample room for storing numbers as needed without potentially taxing the hardware unnecessarily; we do not know the potential implications of attempting to store a very large quantity of numbers.

In order to indicate which level of the stack to read or write, we use an integer

variable called *ptr*, short for the word "pointer." It is not a pointer in the technical sense, but in visualizing the stack, it "points" to the next element in the stack after the most recently filled one, or the "lowest" empty part of the stack. Because the first element of an array in C is labeled as the 0th element, *ptr* conveniently also tells the calculator how many values are stored in the stack. For example, if the lowest empty element of the stack is 0, *ptr* = 0, meaning that it points to the 0th element and that there are 0 values saved. If there is one stored value, the lowest empty level of the stack will be 1, and *ptr* = 1. *ptr* points to the first element and there is one value saved. That is why we chose ptr to indicate the next empty level, rather than the last used level of the stack.

The remainder of our code for Lab 4 consists of a modified version of the keyboard_get_entry() function from Lab 3. Note that the new code for appending a number on the screen uses two lines instead of one, and solves the problem that the previous lab had with negative numbers. What follows is the first portion of the updated function.

```
void keyboard_get_entry()
{
        int c;
        int digits = 0; // number of digits
        int num = 0; // number displayed
        int reset = 0;
        int stack[16];
        int ptr = 0; // # of values are saved in stack
        while(1){
                while(keyboard_key()!= -1); // wait if no key is down
                while(keyboard_key() == -1); // wait while key is down
                c = keyboard_key();
                if (c >= '0' && c <= '9') {
                        if (reset == 1){
                                num = 0;
                                digits = 0;
                                reset = 0;
                        }
                        if (digits < 9) {  // avoid overflow
                                if (num >= 0) num = num * 10 + (c - '0');
                                if (num < 0) num = num * 10 - (c - '0');
                                digits++;
```

```
                        if (num == 0){
                                digits = 0; // do not count extra 0s
                        }
                }
        } else if(c == '\b') {
                num = num/10;
                if (digits > 0) digits--;
        } else if(c == '~') {
                num = -num;
```

If the key pressed is the "INPUT" key or any of the operations, the stack is involved. If "INPUT" is pressed, *num* is saved to the stack where *ptr* is pointing, and then *ptr* is increased by 1. If an operation is pressed, *ptr* is decreased by 1, causing it to point at the most recently filled spot in the stack. Then *stack[ptr]* and *num* are operated on, in that order (i.e., if the operation is subtraction, it will perform *stack[ptr]* minus *num*). The result of the calculation is assigned to *num*. If *ptr* is 0 when an operation is pressed, it means that there is nothing stored in the stack and the calculator does no calculations.

Next, the boolean variable *reset* is set to 1. This means that typing a new number will overwrite *num*. Thus, if the user wants to save the value of *num* after a calculation, he or she must press "INPUT". However, since *num* is not overwritten until a digit is pressed, one can perform an operation on the number on the screen by simply pushing the operation button (without having to press "INPUT" first).

```
        } else if(((c == '\r') || (c == '/')) || (((c == '*') || (c
 == '-')) || (c == '+'))) {
                switch (c){
                case '\r': // INPUT
                        stack[ptr] = num;
                        ptr++;
                        break;
            case '+':
                        if (ptr >= 1) {
                                ptr--;
                                num = stack[ptr] + num;
                        }
                        break;
                case '-':
```

```
                                if (ptr >= 1){
                                        ptr--;
                                        num = stack[ptr] - num;
                                }
                                break;
                        case '*':
                                if (ptr >= 1){
                                        ptr--;
                                        num = stack[ptr] * num;
                                }
                                break;
                        case '/':
                                if (ptr >= 1) {
                                        ptr--;
                                        num = divide(stack[ptr],num);
                                }
                                break;
                        }
                reset = 1;
                }
                display(num);
        }
}
```

After the key press is processed by the calculator, it displays *num* on the LCD using the display() function (Section 5.1). It then restarts the loop, and since they key has not yet been released (again because of relative speeds), it waits for the key to be released.

6 Lessons Learned

This lab was about working with people at least as much as it was about working with computers. Programming with a team requires that all members of the team understand what is being done at each step, which in turn requires each idea to be clearly communicated by the programmer who conceived it - a difficult task for many computer scientists. Working in a group of four, of course, helped to develop this skill, and also made its value obvious. Verbalizing an idea before rushing in to write code helps to clarify the idea, making flaws more visible and improvements more easily conceivable.

This raises another important point that we learned - there are many possible

solutions to any programming problem, and there are almost always methods of making a given piece of code better, faster, and simpler. Periodically presenting our work to the professor and TAs, as well as watching other groups explain their solutions, was an extremely constructive experience; it exposed us to innovations that were often many times more efficient than ours, which opened our eyes to new techniques and motivated us to try to come up with more inventive approaches.

We learned about the unique challenges associated with programming an embedded system. None of us had previously written code for any device that was not a traditional computer, so we were not accustomed to, for instance, expending a great deal of effort to simply make a number appear on the screen. We used C, which is a high-level programming language, but we were performing low-level tasks, such as reading input from a keyboard, which are usually taken for granted by C programmers. Thus, in addition to learning new skills, we learned entirely new ways of applying our existing skills.

We would advise future students to periodically print copies of their code and review them over the course of the week, before returning to the lab. Although group work is important, there is no substitute for individual focus, and the extra time can yield valuable insights. It also keeps the project fresh in one's mind, so it is not necessary to become reacquainted with the code every time one enters the lab.

7 Criticism of the Course

Our only criticism is minor, and it is of the students, not the project or the professor: all dead batteries should be set aside or discarded when it is discovered that they are dead. We lost nearly three hours one day, trying to figure out what was wrong with the hardware we were using and replacing different parts; it turned out that every one of the four batteries we tried had been dead, and our code worked perfectly when we finally tried a fifth battery. To avoid wasting precious time, everyone ought to avoid putting dead batteries back in the same box as the functioning batteries.

With that aside, we all had very high opinions of the course. Since each of the labs built on the knowledge we'd gained from doing the previous ones, all of the work was quite manageable. It was challenging enough to keep us thinking and working hard throughout our time in the lab, but not so challenging as to be insurmountable. Further, none of the labs were isolated; each one was an extension of the one that came before it, culminating in a reasonably impressive piece of working technology. This provided a valuable experience in setting engineering goals and achieving them by a series of steps.

The code reviews, as mentioned in the previous section, were an incredibly helpful part of the course. The knowledge that any member of the group might be asked to justify any programming decision was motivation for everyone to understand what was going on at all times, and provided a reason to be as efficient as possible.

At least two of the group's four members, as a result of this course, decided to pursue computer science or computer engineering as a major. This, perhaps, best indicates our favorable opinion of the CS/CE Lab.

References

[1] Atmel Microcontroller Summary. Online. http://www.keil.com/dd/docs/datashts/atmel/at91sam7l128_64_ds.pdf, February 2008.

[2] HP 20b Business Consultant Financial Calculator Manual. Online. http://h10010.www1.hp.com/wwpc/pscmisc/vac/us/product_pdfs/HP_20b_Online_Manual.pdf.

[3] Mcllroy, Mark. "Reverse Polish Notation." Online. http://mathworld.wolfram.com/ReversePolishNotation.html.

[4] SAM7L ARM-based Microcontroller Overview. Online. http://www.atmel.com/products/microcontrollers/arm/sam7l.aspx.