ENGI E1112 Departmental Project Report: Computer Science/Computer Engineering

Dimitri Dyatlov, Kevin Roark, Nick Duckwiler

December, 2011

Abstract

The computer science gateway lab was pitched to students as a semester devoted to the hacking of calculators; for the most part it lived up to this claim. We will report on the process of software development for the platform (an HP-20b financial calculator), the substance of each piece of code, valuable lessons learned, and provide a user guide for the finished product of our efforts-a functional, custom-firmware Reverse Polish Notation calculator.

The process of development consisted of four sequential projects, each of which appropriately built upon the work of the previous project. After code was written for each sub-project, we critically analyzed the code as a class, and then optimized the code for better performance in the future.

The end project was purposefully ambiguous, allowing us to exercise engineering principles of design in the implementation of our software. For instance, we chose to simultaneously display the user's input and the calculator's most recent operation result on opposite sides of the LCD. Furthermore, by maintaining our same group of students throughout the semester, we gained firsthand experience in engineering collaboration and learned how to use one another's strengths to work efficiently. Ultimately the entire project was a successful introduction to first-level computer science from an engineering perspective.

1 Introduction

This report includes both our work in the Gateway lab and speculations on the process. It is divided into a user guide for the calculator (Section 2), an explanation of the social implications of our calculator (Section 3), a description of

the hardware and software we were given to begin our lab (Section 4), a brief description of the code we wrote to program the calculator (Section 5), a more in-depth walk-through of the code we wrote to program the calculator (Section 6), a section on lessons learned during the course (Section 7), and finally, criticisms of the course (Section 8).

The information covered in this report is from our work in the computer science design project lab of the class: Design Fundamentals using Advanced Computational Techniques. The professor for this project lab is Professor Stephen Edwards. The lab students were divided into groups of three or four and assigned the same lab. The ultimate goal of the lab was to develop the software for a functioning reverse polish notation (RPN) calculator on the platform of a HP 20b calculator. The lab was divided into four steps: (1) make a phrase scroll continuously across the screen of the calculator, (2) design a program to scan the keyboard for user input (keys being pressed), (3) connect the user input to memory storage and display, and (4) design the software to build stacks and do operations based on the user input (in other words, make an RPN calculator). This report is the capstone to our group's work.

2 User Guide

2.1 Reverse Polish Notation

Hopefully this was made evident in the product description, but this calculator works in Reverse Polish Notation (RPN). This simply means that operators are placed after numbers when writing/typing expressions.
Example 1

To express the addition of 1 and 2, (1 2 +) is written, rather than the traditional (1 + 2).

Example 2

To express the subtraction of 1 from 2, (2 1 -) is written, rather than (2 - 1).

2.2 The Stack

An RPN calculator is approached in computer science via a "stack." The stack can be envisioned as a column of numbers, as the provided figure shows. Whenever a number is input to the calculator, it is placed on the lowest layer of the stack not already occupied. Any understanding of the programming beyond

that is not particularly helpful in learning how to operate the calculator.

2.3 The Keyboard

Next, please examine the beautifully constructed keypad on this custom HP 20b, either in Figure 1 or by buying it from the *hpcalc* website [3]. The functional buttons of interest to us are:

• The 10 keys labeled with integers 0-9. These keys, as is likely evident, are used to input numbers for calculation.

• The 4 keys labeled with +, -, $\times$, $\div$. These keys, as is hopefully also obvious, are used to tell the calculator which operators to use on the numbers we input.

• The key labeled "INPUT." This key is treated as another operator used to facilitate our use of RPN. It places the input into the current layer of the stack, and moves the stack pointer to the next layer.

• The key labeled with $+/-$. This key is used to tell the calculator if an input number is positive or negative.

• The key labeled with a backwards arrow like so: "$\Leftarrow$". This key is used to delete any accidentally pressed digit.

• The key labeled "ON/CE." This key is simply used to turn the calculator on.

Any other button on the calculator will have no effect when pressed.

2.4 The Display

As has certainly been observed, there is a fairly large LCD display placed above the calculator's keypad. This screen is used to display both inputs and results. On the right side of the screen, the integer currently at the top of the stack is displayed. On the left side, the number currently being input will be displayed.

Example 1

If the number at the current top of the stack was 12 and the user was inputting the number 500, the screen would appear as follows:

```
[500          12]
```

Example 2

If the user had just calculated the addition of 1 and 2 and not yet input anything, the screen would appear as follows:

```
[0             3]
```

2.5 Using the Calculator

The first step to using our calculator (or any electronic device for that matter) is to provide a power supply. Either insert the provided power adapter or, if portability is desired, insert two AA batteries into the battery compartment.

Next, to turn the calculator on, simply press the button labeled "ON/CE" at the bottom left corner of the calculator's keypad. If the calculator was successfully turned on, the number 0 will appear on the left side of the LCD display.

Now the calculator is ready to perform operations. The best way to explain the use of the calculator will be through extensive use of examples, so many examples are to follow. The most important thing to keep in mind is that the calculator uses RPN and the stack, and that RPN will be the main factor in how the user tells the calculator to do what it is supposed to do.

Example 1: The user wishes to add 1 and 2.

• Step 1: Press the "1" key, then press the "INPUT" key. Pressing the "INPUT" key will add 1 to the bottom layer of the stack. If this sequence is correctly performed, the "1" shown on the input side of the LCD display will shift to the stack side when "INPUT" is pressed.

• Step 2: Press the "2" key, then press the "+" key. Pressing the "+" key will add the input number 2 to the number in the layer of the stack below the current layer, 1. If this sequence is correctly performed, the number "3" will be shown on the stack side of the display after the "+" key is pressed. This number 3, as we know, is our result.

Example 2: The user wishes to subtract 6 from 10.

• Step 1: Just like last time, input the number 10 into the bottom layer of the stack by pressing the "1" key, then the "0" key, and then "INPUT." If successful, a "10" will be displayed on the right side of the LCD.

• Step 2: Press the "6" key, and then the "-" operator. This sequence will subtract the input, 6, from the layer below it, in this case 10. If successful, "4" will be displayed on the LCD. The calculator's performance can again be checked with basic arithmetic-we know that 10 - 6 = -4. Example 3: The user wishes to multiply 500 and -2.

• Step 1: As always, input the first number, 500, and use the input operator to place it in the stack.

• Step 2: First, to indicate that the input will be negative, press the $+/-$ key on the keyboard. When pressed, a "-" sign will appear on the left side of the LCD, showing that the input will be negative. Then, input the number as normal, by pressing "2." Finally, press the "$\times$" key to multiply the input with the layer below

it on the stack, "500." If everything is entered correctly, "-1000" should appear on the right side of the LCD.

Example 4: The user wishes to add 2 and 5, multiply the sum by 12, and then subtract 6 from the product three times. The traditional expression looks like this: (2+5)*12 - 6 - 6 - 6.

• Step 1: Place 2 in the stack by pressing "2," then "INPUT."

• Step 2: Add 5 to 2 by pressing "5," then "+."

• Step 3: Multiply the sum (should read "7") by 12 by pressing the sequence of keys "1," "2," and "$\times$."

• Step 4: To subtract 6 from the product three times, press the sequence "6", "-" three times in succession. If everything has been entered correctly, the final result should read "68."

NOTE: Expressions with more than one operator can be handled in another way thanks to the way RPN and the stack work. If an operator is pressed without any number input, the calculator performs the operation with the two highest layers of the stack. See this next example if interested.

Example 5: The user wishes to calculate this expression: 1 + 2 + 3 + 4 + 5

• Step 1: Place 1 in the bottom layer of the stack with the sequence "1" then "INPUT."

• Step 2: Place 2, 3, 4, and 5 in the next four layers of the stack in the same way, by pressing the desired number key then "INPUT."

• Step 3: Press "+." This will add the two uppermost layers of the stack, namely 5 and 4. As we should realize, if we have done this correctly, the result displayed will be 9.

• Step 4: Press "+" three more times to add the remaining digits-3, 2, and 1-to our summation. The end result should read "15."

As we can see, problems with multiple operators can be approached in multiple ways and with varying levels of efficiency.

NOTES/EXTRAS:

The calculator will display an error if the result of an operation is greater than 2,147,483,647 or less than -2,147,483,647. After the error, the stack will remain intact and other operations can be performed if desired.

There is currently no functional divide operator–not because of developer error, but because of an issue with the library of functions and the calculator's memory limit.

Backspace can intuitively be used to correct any accidental error in the input of a user's number.

The negative $+/-$ key functions as a toggle. If a user wishes for the input to

Figure 1: The HP 20b

be negative, the $+/-$ key must be pressed before any digit keys.

3 Social Implications

Calculators are used in every technology and in every field of study- businesses use calculators to maximize profit and compete to ultimately provide consumers with the best product possible, governments use calculators to most efficiently build roads, provide water, and produce energy, non-profit organizations use calculators to maximize the provision of their services to those in need. In education, students use calculators in practicing mathematics to learn to do the previous actions even more efficiently. The calculator we programmed is not the calculator used in all of these fields, but it is a necessary stepping stone in producing the calculators that educate students, save resources and build a better future.

4 The Platform

The HP 20b is rare amongst consumer-oriented electronics-its producer has provided hardware schematics and a software development kit, giving the average user the ability to repurpose the calculator. Thus, it must have been an obvious choice for the construction of our custom-firmware RPN calculator. Certainly limitations are inherent in the platform; the calculator is essentially comprised of only a keypad, an LCD display, an Atmel AT91SAM7L128 embedded processor, a JTAG port for communication with another computer, and, at least on the modi-

fied version we used for software development, a power connector.

4.1 The Processor

The AT91SAM7L128 (SAM7L) is a member of Amtel's SAM7L series of microcontrollers, which are all based on the 32-bit ARM7TDMI processor. The AT91 has 128 kb of flash memory, 6 kb of SRAM, and a maximum operating frequency of 36 MHz. The processor was engineered to run under low power conditionshence the "L" in SAM7L-so it operates in single-supply mode at 1.8 V and consumes a mere .5 mA/MHz while active and 100 nA while powered off according to the Amtel website [1].

Also included on the microcontroller are a number of peripherals like a 40-segment LCD controller, two USARTS, and an SPI, most of which went unused in our software development. Of particular interest, however, is the system controller, which considered as a sort of puppeteer of the operation. Through software it controls the power supply to every other peripheral and manages the system's clock. It is used effectively to manage power consumption, but must be kept in mind during development-programmers must remember to turn each individual peripheral on.

The heart of the AT91 is the ARM7TDMI. According to the ARM website [2], the ARM7 is the world's most widely used 32-bit embedded processor, having shipped over 10 billion units since 1994. The ARM7 is likely so widely used because it is cheap to produce. As shown in Figure 2, also found on the ARM website [2], the ARM7 is both the least efficient and least feature-rich member of the ARM family. This fact attests to the idea that our calculator has been developed on a bare core of a platform.

4.2 The LCD Display

The display is arguably the most important component of any electronic gadget because it is the only means the device has of reporting back to its operator with feedback; if our calculator had more advanced capabilities but an unreadable display, it would be useless. The LCD display of the HP 20b is basic, but adequately functional. As shown in the diagram below (found in HP's Software Development Kit [4]), its primary body consists of twelve sequential large seven-segment displays separated by period and comma segments, and then three smaller seven-segment displays in its upper right corner meant for exponents. These are the portions of the LCD we utilized in the development of our calculator.

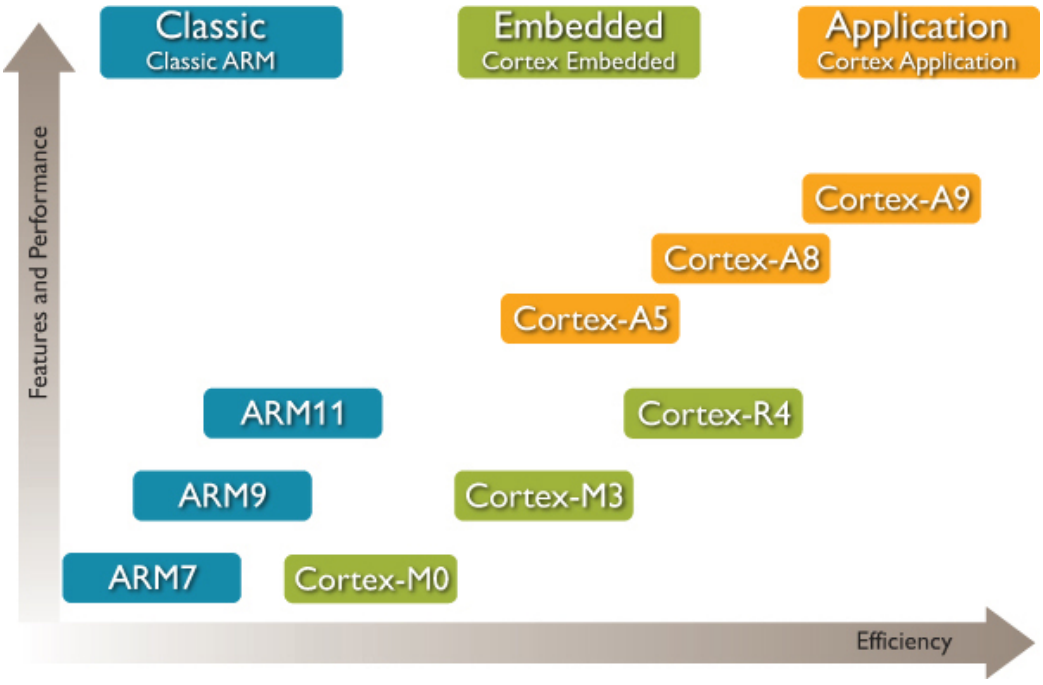

Figure 2: ARM7 Compared to the Rest of the ARM family

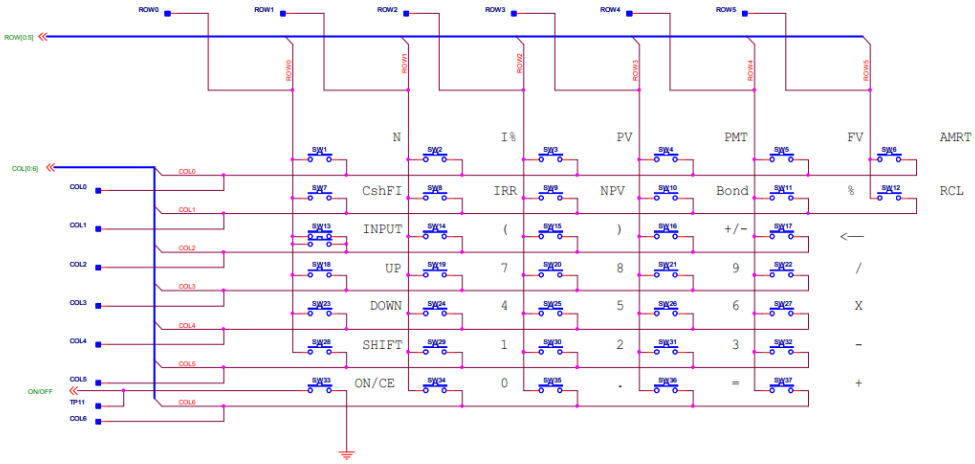

Figure 3: The Keyboard Diagram for the HP 20b

Two library functions, *lcd put char7* and *lcd print7*, were provided to us by Professor Edwards in order to facilitate our utilization of the LCD display. Constructing unique functions linking certain combinations of segments with particular digit and place values would have required a stronger background in C and more time than we were allotted for the semester. Rather than fumbling around trying to make the display work, we were able to spend our time trying to make the calculator itself operate properly. The function *lcd print7* takes an input of any string of up to 15 characters and displays it on the LCD from left to right. The function *lcd put char7* requires an input of a single character and the desired place value of that character, and then displays said character in its desired place on the screen.

4.3 The Keyboard

One of the first labs of the semester was dedicated to tearing apart old keyboards in an attempt to understand the way they work. What we discovered was that the basis of all keyboards is a circuit matrix, much like the one shown in the figure above. Each key corresponds to a specific row-column combination; when a certain key is pressed, it completes the circuit in its location. Software that constantly scans this matrix for a closed circuit is used to determine if a key is pressed.

Figure 3 is the schematic for the calculator's keyboard given in the HP soft-

ware development kit (available for download on the HP website [4]). Rows and columns are inexplicably playing one another's roles, but otherwise the schematic is fairly straightforward. The keyboard consists of six rows and 7 columns, and a total of 37 keys. Every key corresponds to an intersection of a specific row and column; for example, when the "4" key is pressed, we change the circuit at the intersection of row 2 and column 4. Then software we have implemented detects this change and connects this change to the fact that the "4" key was pressed.

It may be worth noting that the labels on keys are totally arbitrary; the meaning of a key press is decided entirely in software. What this essentially means is that the functionality of our calculator is not limited by its keyboard in any way, but instead by its computing power, display, and the coding capabilities of its developers. For instance, if more time or experience had been available, we could have set the UP key to represent exponents, or the NPV key to represent a decimal-to-fraction conversion.

5 Software Architecture

Our RPN software is composed of two main collections of code, *main.c*, and *keyboard.c*. The script keyboard.c is used to obtain input from the user via the calculator's keyboard. Within it are two functions, *keyboard key*, and *keyboard get entry*. *keyboard key* is used to scan the keyboard matrix until a user presses a key. Then, *keyboard get entry* translates the keys that the user inputs into an output of an integer and an operation.

Next, *main.c* accepts the output of *keyboard get entry* and works to perform the input operation on the input values, maintain the stack, and display results to the user. *main.c* is comprised of the functions *do operation*, *display int*, and *main*. *main* is like the heart of the code; after reading the output from *keyboard get entry* it interprets the meaning of the operator and acts accordingly. If the operator is mathematical it performs that operation on the top two layers of the stack with *do operation*, displays the result to the LCD with *display int*, and then repeats the loop. If the operator is INPUT, then it places the input integer into the stack and repeats the loop.

6 Software Details

6.1 Lab 1: A Scrolling Display

Figure 4 shows our code for the scrolling display. It consists of two loops inside an infinite loop. The first loop inside of the infinite loop steps through each

```
#include "AT91SAM7L128.h"
#include "lcd.h"
#define SLOWDOWN 50000
#define DISPLAY_LENGTH 11
int main(){
  lcd_init();
  int n, counter=1;
  char printScreen[] = "␣␣␣␣␣␣␣␣␣␣␣", name[] = "Badboyz",
    *pname = &name[0];
  while (1){
    lcd_print7(printScreen);
    for (n = 0; n <= DISPLAY_LENGTH; n++){
      printScreen[n] = printScreen[n+1];
    }
    int x = 1;
    while (x < SLOWDOWN)
      x++;
    if (*pname == '\0')
      printScreen[DISPLAY_LENGTH] = "␣";
    else{
      printScreen[DISPLAY_LENGTH] = *pname;
      pname++;
    }
    if (counter == DISPLAY_LENGTH){
      pname = &name[0];
      counter = 0;
    }
    counter++;
  }
  return 0;
}
```

Figure 4: Our solution for lab 1: the scrolling message

```
int keyboard_key(){
  int i,j,n;
  for(j=0;j<COLUMNS;++j){
    for(i=0;i<ROWS;++i){
      for (n = 0; n < COLUMNS; n++) keyboard_column_high(n);
      keyboard_column_low(j);
      if(!keyboard_row_read(i))  return i+10*j;
    }
  }
  return -1;
}
```

Figure 5: Our solution for lab 2: scanning the keyboard

character of the display except for the last character, changing the value of the character to the value of the character that is ahead of it in the array. After the first loop ends, the second loop does nothing in order to insert a delay so that the text scrolls at a reasonable speed (this loop could be placed anywhere inside of the infinite loop, but we happened to put it in the middle).

After the second loop ends, the last character of the string that will be displayed is added, based on the value of **pname*, and *pname* is reset if the counter reaches 12, which is the number of characters on the calculator screen.

After this is finished, the outer *while* loop restarts the process.

6.2 Lab 2: Scanning the Keyboard

Figure 5 shows our code for the *keyboard key* function. It consists of two loops which together step through each row and column of the keyboard, testing whether the key at each position is pressed. When the program finds that a key has been pressed, it returns the location of the key.

Figure 6 shows our code for the *integer to character* function and *main*. The *integer to character* function works by converting an integer that was returned from *keyboard key* so that the new integer corresponds to an element in an array of characters (the array is not shown because of lack of space). The function then returns this array element.

In *main*, an infinite *for* loop makes the program switch between two *while* loops. The first *while* loop waits until a key has been pressed, and the second *while* loop converts the value returned by *keyboard key* into a string to display and

```
#include "AT91SAM7L128.h"
#include "lcd.h"
#include "keyboard.h"
void integer_to_character(int key_value, char *display){
  int i,j;
  if(key_value<29)
  j=(key_value/10)*6+(key_value%10);
  else
    j=17+(key_value/10-3)*5+(key_value%10);
    for(i=0;i<10;++i){
      *display=VALUES[j][i];
      display++;
    }
  };
int main(){
  // Turn off the watchdog timer
  *AT91C_WDTC_WDMR = AT91C_WDTC_WDDIS;
  int last_key;
  lcd_init();
  keyboard_init();
  char display[10];
  for(;;){
    while(last_key==-1){
      last_key=keyboard_key();
      lcd_print7("no key      ");
    }
    lcd_print7("            ");
    while(last_key!=-1){
      last_key=keyboard_key();
      integer_to_character(last_key,&display);
      lcd_print7(display);
    }
  }
  return 0;
}
```

Figure 6: Our solution for lab 2: displaying what was scanned

then displays it, looping until the key has been released.

These functions were tested simply by running the program and pressing all of the keys one at a time.

6.3 Lab 3: Entering and Displaying Numbers

Figure 7 shows our code for the *keyboard get entry* function. It consists of an infinite loop in which the *keyboard key* function is called and then its output is used to determine which actions to take. If the key returned is an integer, the key is displayed on the screen and the value of *num* is updated. If the key returned is the backspace key, the right-most number on the display is removed, and *num* is updated. If the only thing displayed was a negative sign, the negative sign is removed and *poscheck* is set to 1 to show that the final value of *result->number* will be positive. If the key returned is the $+/-$ key (while *num* is equal to 0), the sign of *poscheck* is reversed and either a negative sign or a space is placed on the left side of the display when *poscheck* is -1 and 1, respectively. If the key returned is an operation key (this is checked by looking at each element in the *operations* array), the operation is displayed on the right side of the screen and the two parts of *result* are filled. After that, the function returns.

Figure 8 shows our code for *main* and *keyboard key*. In main, a *struct entry* type is initialized and then *keyboard get entry* is called to do the rest of the work. *keyboard key* does the same thing as in Section 6.2. The only major difference is that, to increase efficiency, all of the columns are kept high and lowered when necessary, instead of making all of the columns high on each loop iteration even when some are already high.

6.4 Lab 4: An RPN Calculator

Figure 9 shows our code for the *do operation* function. Given an operation and two integers, the function uses a switch statement to choose which operation to perform.

Since the library for division by non-constant integers does not work, there is no divide case, and the function returns *INT MAX* to show that there has been an error.

The multiply case checks for overflow by comparing two values that are always equal when there is no overflow (the last 5 digits of the product of the last 5 digits of each number and the last 5 digits of the product of the two numbers). This method is not ideal, since 1/100,000th of the time when there is an overflow the two sets of digits will happen to coincide, and the calculator will

```
void keyboard_get_entry(struct entry *result)
{
  int i, last_key, key_before_last_key, num = 0,
    count = 0, poscheck = 1;
  for(;;){
    last_key=keyboard_key();
    if(last_key<'0'+10 && last_key>='0' && (num)<INT_MAX/10
        &&last_key!=key_before_last_key){
      lcd_put_char7(last_key,count++);
      num=num*10+(last_key-'0');
    }
    if(last_key=='\b' && last_key!=key_before_last_key && count>0){
      lcd_put_char7('␣',--count);
      num/=10; // Drops last digit of integer.
      if (poscheck==-1 && count==0)  poscheck=1;
    }
    if(last_key=='~' && num==0 && last_key!=key_before_last_key){
      poscheck*=-1;
      if (poscheck==-1)  lcd_put_char7('-',count++);
      else if (poscheck==1)  lcd_put_char7('␣',--count);
    }
    for(i=0;i<NUM_OPERATIONS;++i)
      if(last_key==operations[i]){
        lcd_put_char7(last_key,11);
        result->operation=last_key;
        result->number=num*poscheck;
        return;
      }
    // Used to prevent issues of holding a button for too long
    key_before_last_key=last_key;
  }
}
```

Figure 7: Our solution for lab 3: entering numbers

```
#include "AT91SAM7L128.h"
#include "lcd.h"
#include "keyboard.h"
const char operations[] = {'\r','/','*','-','+','='};
#define NUM_LINES 12
int main(){
  // Disable the watchdog timer
  *AT91C_WDTC_WDMR = AT91C_WDTC_WDDIS;
  lcd_init();
  keyboard_init();
  struct entry theEntry;
  keyboard_get_entry(&theEntry);
  return 0;
}
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;
}
```

Figure 8: Our solution for lab 3: main and keyboard key

```
#include "AT91SAM7L128.h"
#include "lcd.h"
#include "keyboard.h"
#define NUM_LINES 11
#define NUM_OPERATION 4
#define MAX_STACK 10
const char operation[] = {'/','*','-','+'};
const char operations[] = {'\r','/','*','-','+','='};
int do_operation(char operation, int int1, int int2){
  switch(operation){
    case '*':
      if(((int1%100000)*(int2%100000))%100000==
          (int1*int2)%100000)
        return int1*int2;
      else
        return INT_MAX;
    case '-':
      if((int1<0 && int2>0) || (int1>0 && int2<0)){
        if(((int1%100)-(int2%100))%100==(int1-int2)%100)
          return int1-int2;
        else
        return INT_MAX;
      }
      return int1-int2;
    case '+':
      if((int1>0 && int2>0) || (int1<0 && int2<0)){
        if(((int1%100)+(int2%100))%100==(int1+int2)%100)
          return int1+int2;
        else
          return INT_MAX;
      }
      return int1+int2;
    }
  return INT_MAX;
};
```

Figure 9: Our solution for lab 4: doing an operation

```
void display_int(int the_int){
  int intval = the_int, place=0;
  if(intval<0)
    the_int*=-1;  // Makes a negative value positive
    do{
      lcd_put_char7(the_int % 10 + '0',NUM_LINES-place++);
      the_int/=10;
    } while(the_int);
    if(intval<0)
      lcd_put_char7('-',NUM_LINES-place);
};
```

Figure 10: Our solution for lab 4: displaying an integer

not display an error message. If division by non-constant integers worked, however, it would be possible to use a more ideal method, that of checking whether *int1*int2/int2==int1*. This method will always alert the user of an overflow, since it is impossible for an integer that is lower than *INT MAX* to be divided by *int2* and produce *int1* when there is an overflow. If there is an overflow, *INT MAX* is returned; otherwise the product of the two numbers is returned.

The minus case first checks to see if *int1* and *int2* have the same sign, to see if it is even possible for there to be an overflow (if they have different signs, it is impossible). After that, it performs a test similar to the one in the multiply case. In this case, however, it is impossible for the calculator to not display an error message when there is an overflow, since the maximum difference between two *int* values is $+/-$*2*INT MAX*, and the difference that would be required for the two values being compared to coincide is $+/-$*100*INT MAX*. If there is an overflow, *INT MAX* is returned; otherwise the difference of the two numbers is returned.

The plus case works just like the minus case, except that at the beginning it checks to see if *int1* and *int2* have different signs instead of the same sign.

Figure 10 shows our code for the *display int* function. It first makes a copy of the integer that was input, and then makes the integer that was input positive. After that, it displays the left-most number of the integer on the left-most part of the display that is not already occupied by an integer, and then removes the left-most number by dividing the integer by 10. Then, it checks whether the copy of the integer is negative and if so, it places a negative sign on the left-most part of

the screen that is not occupied by an integer.

Figure 11 shows our code for *main*. It consists of an infinite *for* loop. In this loop, *keyboard get entry* is called, and if the number returned by it is not equal to *INT MAX*, the number is put on the stack. Otherwise, the stack pointer is decremented in order to cancel out an increment later in the loop.

If the input key was pressed, the value on the top of the stack is displayed (if it exists). If one of the operation keys in the array *operation* (distinct from *operations*) was pressed, the *do operation* function is called, and if it does not return *INT MAX*, the result is displayed. Otherwise, ERROR is displayed.

Figure 12 shows our code for *keyboard key*. It is exactly the same as in Section 6.3 (Figure 8).

Figure 13 shows our code for *keyboard get entry*. It is similar to the code in Section 6.3 (Figure 7), but has a few major differences.

In the beginning, there is a *while* loop which sets *continue loop* to 0 whenever it is 1, then calls *keyboard key* and checks whether the return value is an operation key. If it is, *continue loop* is set back to 1, and the loop continues. The purpose of this loop is to get rid of any operation key presses at the beginning. Otherwise, pressing an operation key such as + will cause all of the numbers on the stack to be added, since the function will have enough time to run multiple times before the user is able to release the button.

After that, there is a *while* loop which calls *keyboard key* until a key has been pressed. This makes it so that, if the key that causes this loop to terminate is an operation key, the main *for* loop (i.e. the largest one) will never loop and therefore *key before last key* will remain at -2. This causes the function to return *INT MAX*.

7 Lessons Learned

The lab was an educational experience in computer programming in a realistic setting- we were given assignments that built on each other, we worked with hardware and user interface and we were required to both present and review our work. In this setting, we learned information useful in both an educational and professional computer science setting.

We learned the best ways to write effective and efficient code. In the third lab, we had completed the requirements, but had planned to revise our work, and so did not submit it. Unfortunately when we came back the next week, we accidentally deleted our entire code. Re-writing all of our work seemed impossible at the on-set, but the actual re-writing went many times more quickly than the first attempt. This was because we fully understood what we wanted to achieve and

```
int main(){
  *AT91C_WDTC_WDMR = AT91C_WDTC_WDDIS;
  lcd_init();
  keyboard_init();
  lcd_print7("0");  // Initializes screen with 0 display.
  struct entry theEntry;
  int stack[MAX_STACK], si=0, i, stackvalue;
  for(;;){
    keyboard_get_entry(&theEntry);
    if (theEntry.number != INT_MAX)
stack[si]=theEntry.number;
    else  si--; // If only an operation is pressed
    if(si<MAX_STACK){
      if(si++)
        lcd_print7("0          ");
      if (theEntry.operation=='\r'){
        lcd_print7("0          ");
        if(si) display_int(stack[si-1]);
      }
      for(i=0;i<NUM_OPERATION;++i)
        if(theEntry.operation==operation[i]&&si>1){
          si--;
          stackvalue=
            do_operation(theEntry.operation,stack[si-1],stack[si]);
          if(stackvalue!=INT_MAX){
            stack[si-1] = stackvalue;
            display_int(stack[si-1]);
          }
          else  lcd_print7("     ERROR");
        }
    }
    else  lcd_print7("     ERROR");
  }
  return 0;
}
```

Figure 11: Our solution for lab 4: main

```
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;
}
```

Figure 12: Our solution for lab 4: keyboard key

our fundamental approach to do so. From this we learned that programming is much easier when one has set out the goals and methods of a project, instead of working piece by piece to discover a good strategy of design. We also learned that one's code becomes much more efficient when all is planned out before. Additionally, the best code is the code with the fewest lines.

The lab taught us not only efficiency and effectiveness of software, but clarity. Programs require multiple reviews and revisions, especially when a lab is based on the work of the previous one. We first believed that more comments meant more clarity. In truth, the greatest clarity is achieved through proper spacing, functional variable names and the avoidance of magic numbers (predefined constants are more clear).

Mistakes are always made along the way. After the hundreds of mistakes made in this lab, we learned the best methods of revision. Instead of making single changes and running the program until error messages were gone, we learned to ask for peer reviews of our code, attempt to explain our code to others and operate the program with pen and paper. These methods proved to be much more effective in fixing errors. Finally, we learned that revision is constant. Even if a function operates, there are always better solutions.

```
void keyboard_get_entry(struct entry *result){
  int i, last_key=-1, key_before_last_key=-2, num = 0,
    count = 0, poscheck = 1, continue_loop = 1;
  while(continue_loop--){ // Gets rid of operation key presses
    last_key=keyboard_key();
    for(i=0;i<NUM_OPERATIONS;++i)
      if(last_key==operations[i])  continue_loop=1;
  }
  last_key=-1;
  while(last_key==-1)  last_key = keyboard_key();
  for(;;){
    if(last_key<'0'+10 && last_key>='0' && (num)<INT_MAX/10 &&
        last_key!=key_before_last_key){
      lcd_put_char7(last_key,count++);
      num=num*10+(last_key-'0');
    }
    if(last_key=='\b' && last_key!=key_before_last_key && count>0){
      lcd_put_char7('␣',--count);
      num/=10; // Drops last digit of integer.
      if (poscheck==-1 && count==0)  poscheck=1;
    }
    if(last_key=='~' && num==0 && last_key!=key_before_last_key){
      poscheck*=-1;
      if (poscheck==-1)  lcd_put_char7('-',count++);
      else if (poscheck==1)  lcd_put_char7('␣',--count);
    }
    for(i=0;i<NUM_OPERATIONS;++i)
      if(last_key==operations[i]){
        result->operation=last_key;
        if(key_before_last_key==-2)  result->number=INT_MAX;
        else  result->number=num*poscheck;
        return;
      }
    key_before_last_key=last_key;
    last_key=keyboard_key();
  }
}
```

Figure 13: Our solution for lab 4: getting user input

8 Criticism of the Course

Ultimately, the calculator programming lab was a useful experience to us as engineering neophytes. One of the most important lessons we learned in this course is to never be satisfied with the quality of the results of our first attempts. In following this lesson, we will critically consider the quality of the first attempt the Dean's Office made in redesigning this lab. The lab excelled in its freedom of design and review system. We enjoyed the ability to choose our methods of solving the problems presented to us. For example, we were asked to display numbers on the screen as we entered them, but it was up to us to decide where to display them on the screen, how long to display them, and how to go about writing the code to implement those decisions. Once we had completed each of the tasks, the following code reviews were very useful. The code reviews involved each group presenting their code to the class and describing their methods of programming the calculator. This was important to us because it forced us to look at our code critically and it provided a venue to receive feedback from both students, TA and professor on our work. Not only did we learn from our achievements and mistakes, but we also learned from the work of others.

Unfortunately, the lab faltered in technical aspects. The calculators which we programmed were re-designed to use an external power source. The attachments for this were unreliable, so that we lost a lot of time re-attaching the cable when it fell out (which happened often) or testing our code on another calculator when we could not tell if ours was still functioning. In addition, the computer lab in which we worked was far too small to accommodate all of the students, not to mention the other people who came in and out of the lab to do other projects at the same time.

The class was designed to introduce us to engineering, specifically in the field of computer science. In that regard, it was a very good experience- in problem-solving, decision-making, communication and teamwork. Although, the class failed to actively educate us in those areas. We did learn through trial and error in our personal experiences, but we could have learned even more (in the aspects of problem-solving, decision-making, communication and teamwork) if we were taught and advised by Professor Edwards. For example, Professor Edwards midway through the course gave us the analogy of the blind man and the Rubix cube. Imagine a blind man holding a Rubix cube and a sighted man sitting on a bench. The blind man makes one turn on the Rubix cube, hands it to the sighted man who checks to see if it is solved, says no, and hands it back to the blind man to try again. The blind man was us, who made one change in the code, then ran the

program to see if it worked. In most real engineering problems, it is not so easy to test and test again every time a change is made. The most effective way to fix our code was to simulate it ourselves and make changes accordingly, instead of letting the compiler vaguely identify errors for us. The advice was very helpful to all of the students in the lab. That kind of teaching would not only help us with the task at hand, but it would also help us in solving the myriad other problems we will face in our future engineering careers.

References

[1] Amtel product description. Online http://www.atmel.com/dyn/products/product_card.asp?part_id=4293, July 2008.

[2] Arm7 processor family. Online http://www.arm.com/products/processors/classic/arm7/index.php.

[3] Eric Rechlin. Hp 20b financial calculator. Online http://commerce.hpcalc.org/20b.php, 1997.

[4] Hp software development kit. Online http://preview.tinyurl.com/649bws, October 2009.