ENGI E1112 Departmental Project Report: Computer Science/Computer Engineering

Charlie Wu, Michael Yang

May, 2011

Abstract

This report documents our process through which we develop a functioning HP 20b calculator from its constituent hardware and software components using C. We first demonstrate how the end product works and its possible social implications before moving on to describing the calculator's hardware and software.

Finding the main components of the hardware, and how each of these, the processor, LCD, and keyboard function, we come to an understanding of the embedded system's physical components, which helps us think about how to write our software later.

The software section then details how we successively build upon the core hardware interactions with the software to create new functionality and bring the calculator closer to functioning as expected. We first build a function to output data, then one for checking for input. Building on this, we write functions that interactively aggregate input meaningfully and evaluate it arithmetically.

Concluding, we address our experiences with the course.

1 Introduction

Computing devices are integrated into almost every aspect of the modern lifestyle. From reading books to playing games, computers have become more diverse and pervasive, almost to the extent that we no longer actively think of them as computing devices. The fact that many of the electronics we interact with now have computational power manifests itself in this semester's Gateway project: programming the HP20b calculator.

While an embedded system like the HP20b shares much in common with the computer and other computing devices, there are limitations that it imposes that gives the sense of programming for it a different twist. It brings you back into the world before everything was already written and packaged in some library, and you didn't have to think too much about efficiency, memory, or basic device limitations.

In this experience, we describe the process of stepping off of the shoulders of the giants before us, building a calculator up from its basic functions, and how the understanding of the hardware we use gives rise to how we program and how we interact with systems in which hardware and software are so closely intertwined.

2 User Guide

Our calculator is programmed based on Reverse Polish Notation (RPN). In RPN, each individual expression consists of two operands, followed by their operator. This is different from the format we are used to, which placed the operator between the two operands.

In order to begin your calculation, input any number. As you enter your number, the calculator LCD display will display the number. To move on to the next input, press the input key. This tells the calculator that the current number is accepted, and to move on to the next number. To signal that the number has been accepted, an R character is displayed at the far left of the LCD display. From here, you can input your next number, and it will be saved. The new number will displace the original number displayed on the LCD.

After inputting at least two numbers, you may perform operations. Rather than press Input to signal the end of a number, you press the operation to act on the current number and the past number.

From here, we can create more complex expressions from our simple individual expression. Each individual expression can be treated as an operand, and be operated on by another operator. For example, in the expression "5, input, 4, +", the operands 5 and 4 belong to the operator "+". However, in the expression "5, input, 4, + , 3, input, 2, -, *" there are two sub-expressions "5, input, 4, +" and "3, input, 2, -" that belong to the operator "*". Note: the "input" key is denoted by the letter "r".

You may notice that there are multiple ways to format more complex expressions. For instance, the expression 4+3+2 can be written in these two ways: "4 , input , 3 , input , 2 , + , +" and "4 , input , 3 , + , 2 , +". As long as you remember that the current operation acts on the two most recent input values, you can structure expressions in multiple valid formats.

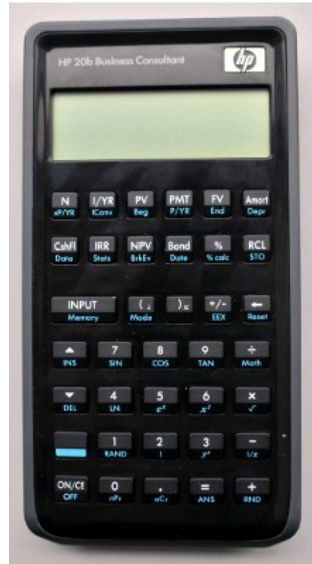


Figure 1: The HP 20b

Two key details are that the two operands must have been input prior to the operation. Furthermore, the operation is performed immediately and the result is displayed to the LCD screen.

To address the problem of underflow, we ignore the operation if only one number is present in the stack. If any operation is input following the number in the first position of the stack, the number is still displayed in the LCD. The stack is cleared if too many (>100) items are being stored.

3 Social Implications

The HP 20b is a compact and light calculator which we programmed to be able to perform accurate arithmetic calculations. The calculator is viable for everyday school use. Furthermore, as a teaching tool of embedded programming, the calculator is a relatively cheap piece of hardware with vast potential for firmware development.

4 The Platform

We initially learned about hacking the 20b calculator through the first lab in this course[1]. Further readings include the HP20b Repurposing Project website[2].

4.1 The Processor

The HP 20b is little more than a keyboard and LCD, connected to an Atmel AT91SAM7L128(SAM7L) processor. The SAM7L is a low power member of Atmel's AT91SAM series of chips, which are all built around an ARM processor core ("AT" is for Atmel; "SAM" is "smart ARM core;" 91 appears to be arbitrary). The 7L series of microcontrollers are designed for low power (hence the L), and the final 128 is a reminder that it includes 128K of flash program memory. Figure 2 shows a block diagram of the SAM7L chip. It looks complicated, but is essentially a single standard processor surrounded by memory and a wide variety of peripherals, most of which are unused for this project. The system controller controls the clock and power supply of each peripheral through software. This makes it possible to save energy by not powering on unnecessary peripherals, but can also make a peripheral appear not to work if the power is not turned on when needed.

4.2 The LCD Display

The calculator's LCD display, shown in Figure 3, is driven by complex ac waveforms, which are generated from the LCD controller. The LCD has 15 digit display positions, which represented in segments, as shown in Figure 3; of these, 12 are for regular digits, and 3 are for exponent digits. For both regular and exponential digits, there is a negative sign position. Although there are comma and decimals present on the LCD, we did not include either in our calculator firmware considerations. Furthermore, in the space above the digit positions, there is a block of pixels for more advanced calculator functions, including the financial functions. Lastly, there are various extra indicators for RPN, angle units (degree or radian), battery life, and others.

4.3 The Keyboard

The keyboard is a row/column matrix of switches. The rows and columns are connected to general Input/Output (I/O) port pins. See the Figure 4 for the specific row/column combinations.

The matrix is connected to pins on the processor chip that can by driven by a parallel I.O controller, which is a peripheral that enables software to control and read the state of each pin. In our second lab, we explored this peripheral by writing a function that returned which key was being pressed, or if no key was currently being pressed.

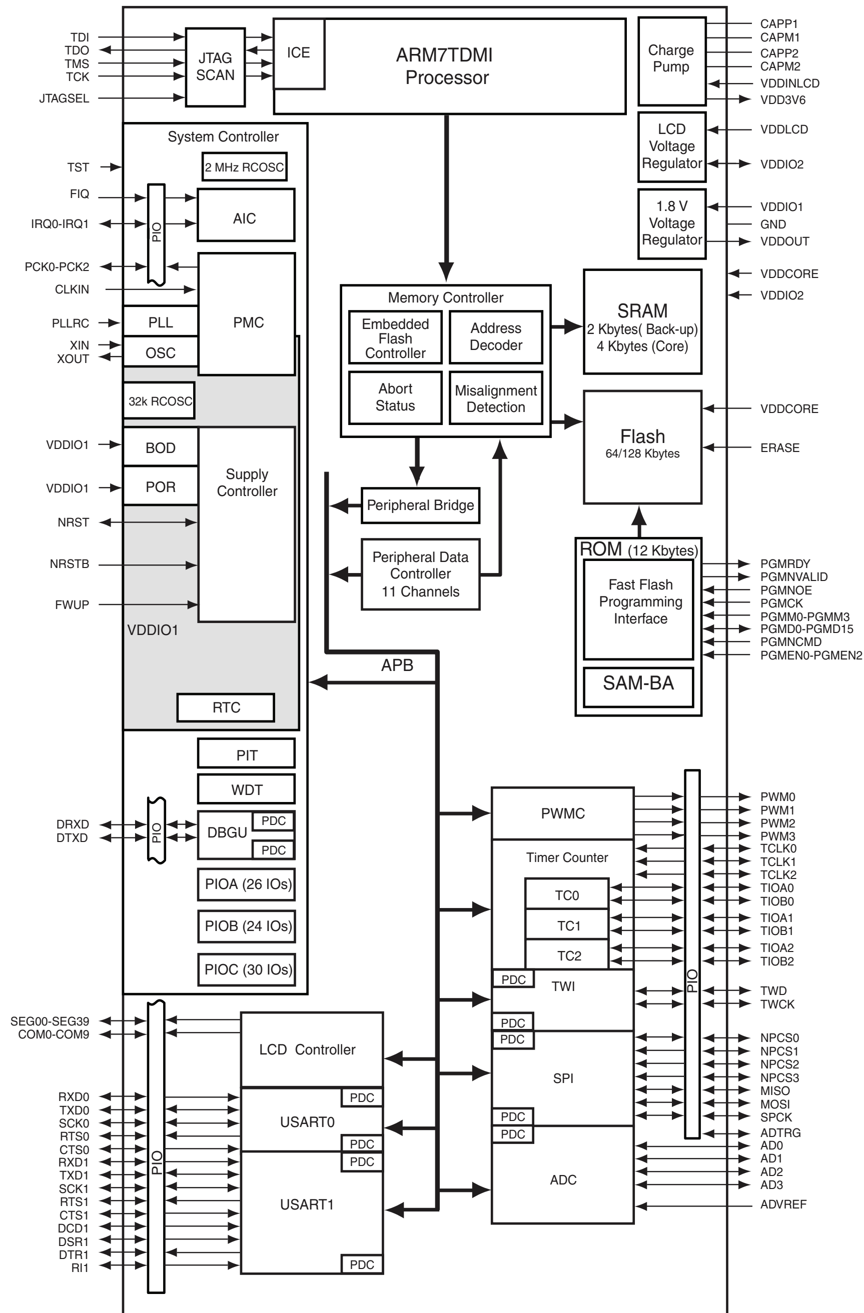
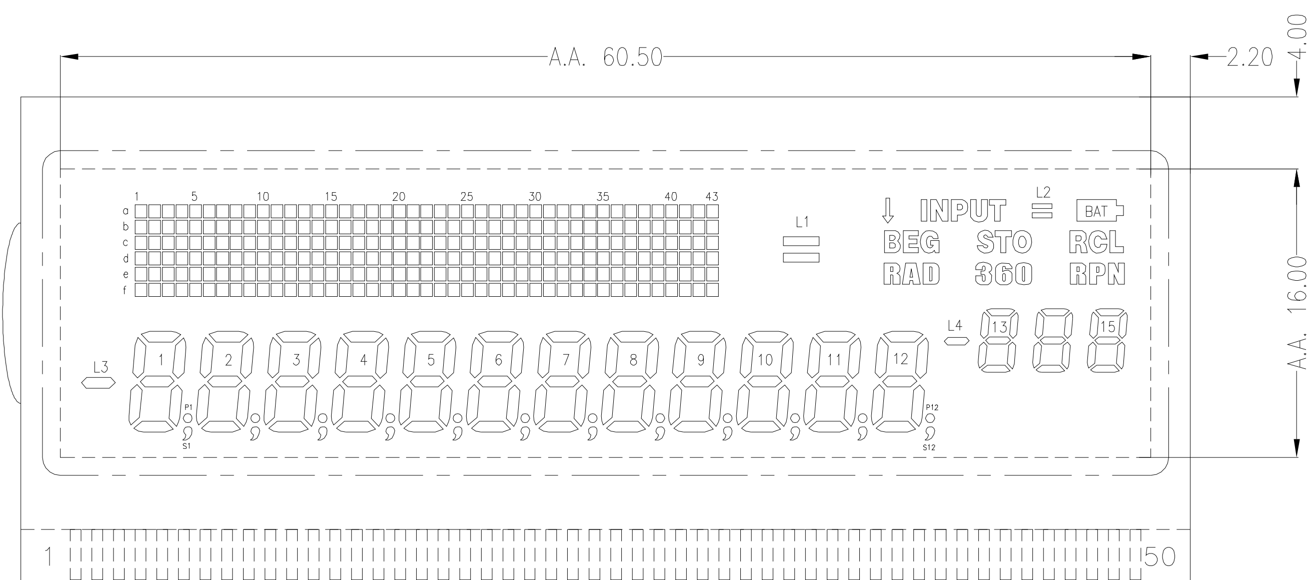
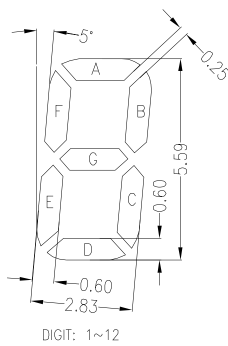
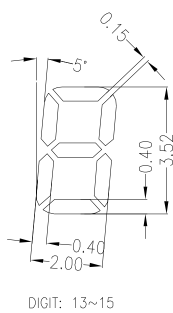


Figure 2: A block diagram of the AT91SAM7L microcontroller that is at the heart of the HP 20b

REV.	DESCRIPTION	DATE
A	REVISED WIRING	08-NOV-2007



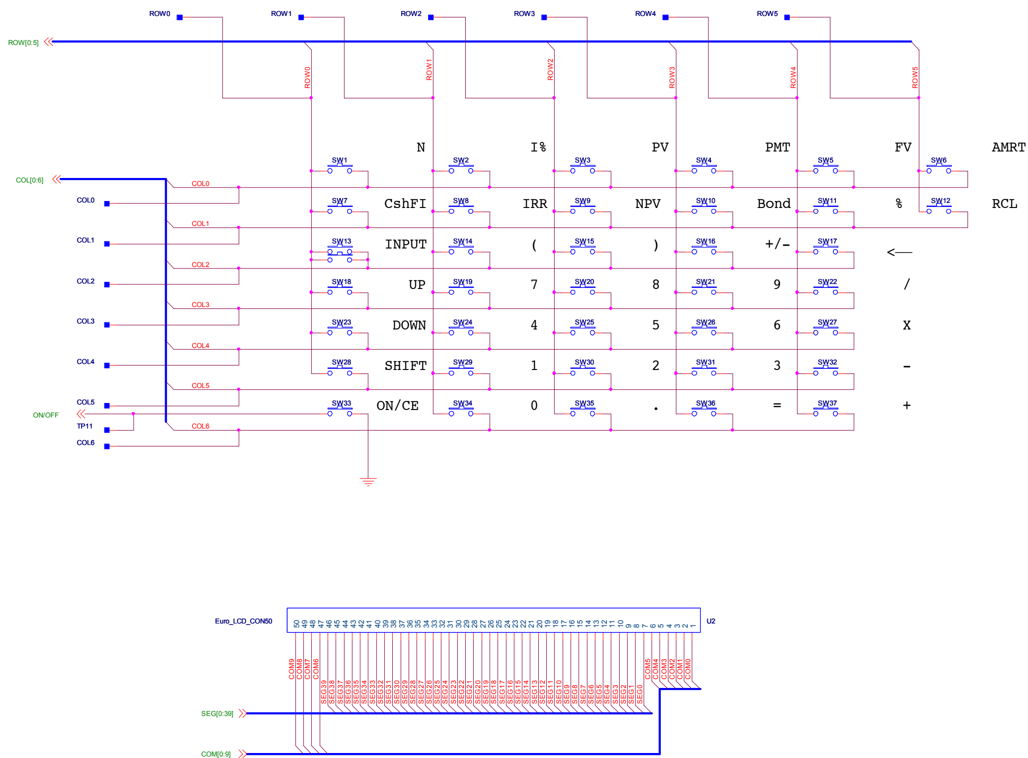




SCALE: NOT TO SACLE	ELEC & ELTEK 依利安達		
PAGE: 4 OF 7			
GENERAL TOL. 0.20 / UNITS MM	字划/图案尺寸(ICON/SEGMENT DIMENSION)		
APPROVALS / DATE	MODEL NUMBER :		DRAWING NO: 999465G
APP			
CHK	922-151A-0484		DATE: 31-OCT-2007
DWN YFL 31/OCT		PROJECTION	

Figure 3: A diagram of the HP20b's LCD display

PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn



reated with pdfFactory trial version www.pdffactory.com

Figure 4: A circuit diagram of the keyboard and its keys

5 Software Architecture

The calculator consists of four basic actions that build upon each other to function: displaying numbers, detecting keys, entering numbers, and evaluating expressions. The first component is the LCD integer printing function. This allows us to display the results of our work on the LCD and represents the output portion of the calculator's user interface. For the input half, we turn to the detection component, which tells us which key is pressed at any given time. Combining the input and output components, we are able to enter numbers and operation, and display the as they are entered by aggregating the results of the keyboard input. Wrapping this number and operation data together, and leveraging the previous components allows us to further aggregate these data into arithmetic expressions, which can then be evaluated.

6 Software Details

6.1 Lab 1: A Scrolling Display

In the first lab we were asked to write a function to take an integer and display its decimal representation on the LCD. Because each of the LCD's columns function independently of each other, we decided that we had to split the integer accordingly. In order to accomplish this, we took advantage of the C language's integer division and modulus operations.

We used the desired base (10 for decimal digits) to successively extract digits by using the modulus operation to take the least significant digit, and the division operation to remove the taken digit, shifting the next significant digit into place. This process occurs, displaying each digit as it's taken off, until the taken digit is a zero, indicating the "end" of the digit.

This algorithm works because of how integer division is accomplished in C, which truncates the floating-point portion of the result accomplished with normal mathematical division. For example, the number 123 divided by 10 would seem to result in 12.3, but by truncating the numbers after the decimal, it becomes 12, allowing it to "cut off" the end digit of a number. By successively performing these operations, we eventually narrow the integer down to one digit, which integer division will evaluate to zero, indicating the endpoint of the algorithm.

Additional conditions and minor modifications were added to support "0" and negative numbers. The function checks for "0" or a negative number before beginning the normal algorithm. If the number is a "0", a "0" is printed and function finishes. If the number is negative, the sign is stored in another variable, and the algorithm is applied to the absolute value of the number. After this, a minus sign

is added in front of the number to indicate that it is negative.

Lab 1

```
#define MAX_POS 11
#define BASE 10

void displayInt(int number)
{
    int positionLimit = MAX_POS;

    if(number < 0)
    {
        // leave space for '-'
        positionLimit--;
        // get absolute value
        number *= -1;
    }

    int position = 0;

    // continues while digits and spaces are left (or number is initially 0)
    while((number != 0 && position <= positionLimit) || position == 0)
    {
        // get least sig digit
        int digit = number % BASE;

        // print character form of digit
        lcd_put_char7('0' + digit, MAX_POS - position);

        // cut least sig digit
        n /= BASE;
        position++;
    }

    // positionLimit is MAX_POS if positive, MAX_POS - 1 if negative
    if(positionLimit == MAX_POS - 1)
        lcd_put_char7('-', MAX_POS - p);
}

int main()
{
    lcd_init();

    displayInt(0);


    return 0;
}
```

6.2 Lab 2: Scanning the Keyboard

Lab 2 required that we provide a function to scan the keyboard for input, and return some code indicating the key that was pressed. We were provided with functions that controlled the voltages across the columns of the keyboard, and another function that determined whether a key in a row was being pressed based

on those voltages. To check if a key is being pressed, we set all of the column voltages to high ("1"), except the column we want to check. We then use the row reading function on each row to determine which of the rows is being pressed, in which case it will return "0".

To check if any, not just an individual key, is being pressed, we iterate over each column, setting its voltage to low, and then iterating over each row, checking its value. We do this by using a nested loop to visit each key and check its pressed status. Once we find a key that is pressed, we return the column and row in the form of a two-digit number, the first digit representing the column and the second representing the row, and display this on the LCD. If no key is being pressed, it will return a "NO_KEY" (set to -1, a number chosen because no combination of columns or rows could result in a negative number). By placing this function in an infinite loop, we are able to continuously check which keys are being pressed, and display the "NO_KEY" value otherwise.

After finishing this lab, we decided to use Professor Edwards' implementation of it because it returned more useful output (characters rather than encoded coordinates).

Lab 2

```
int keyboard_key_read(int row, int column)
{
  // reset columns except for target column
  int i;
  for(i = 0; i < MAX_COL; i++)
      keyboard_column_high(i);
  keyboard_column_low(column);

  // return if the row is low
  return !keyboard_row_read(row);
}

int keyboard_key()
{
  // initialize to no key
  int coordinate = NO_KEY;

  // check each key in keyboard matrix
  int column, row;
  for(row = 0; row < MAX_ROW; row++)
      for(column = 0; column < MAX_COL; column++)
          if(keyboard_key_read(row, column))
              // return a code containing pressed row and column
              coordinate = 10 * row + column;

  return coordinate;
}
```

6.3 Lab 3: Entering and Displaying Numbers

For lab 3, we were instructed to devise a function to enter and display user-inputted numbers, followed by an operation key. To accomplish this, we contain the number information in an integer buffer variable, and successively append digits until an operation key is pressed. The input key is first contained in the "key" variable, obtained by using a solution from the previous lab provided to us. Additionally, a count variable containing the number of digits in the buffer helps prevent integer overflow and a sign variable handles negative numbers.

If the input is a digit, we append the digit to the end of our number (by multiplying the existing digit by 10, which shifts each digit to one higher significance, and then add the incoming digit), refresh the LCD, increment our count, set the "key" variable to our continuation condition ("NO_KEY"), and insert the variable into the "last_key" variable. The "key" variable is set to this continuation condition for all keys except the operation keys, so this lets us know when the number portion of the input is finished (in this case, it's not), and the "last_key" variable is used to ensure a gap of "NO_KEY" between key presses. This "NO_KEY" is analogous to the gap in key presses the user performs when his or her finger is lifted from the keyboard. This is necessary because the processor runs much faster than human input, so this prevents erroneous input from occurring because the processor still thinks the same key is being pressed when the finger has not actually been lifted.

Various other keys are supported, such as the backspace and sign change keys, which set similar continuation conditions for the number section of the input. When the sign change key is pressed, the internal sign variable is flipped, the LCD is updated (removing or adding a negative sign), and the continuation conditions are set. The LCD representation code (in the latest version) so that the negative sign appears before the most significant digit. If there are no digits, a negative sign by itself is displayed. For the backspace key, we use integer division to remove the last digit in the buffer and decrement the count, followed by the continuation conditions. In the case that the backspace character removes all visible digits, even though the value of the buffer is "0", we display no digits (as expected from the functionality of current calculators).

When the user finally does input an operation key, the loop that scans for input exits, and the last value taken was the operation. If the length of the buffer is "0", indicating that no number was input, the buffer is set to a value of "INT_MAX" to indicate this.

When the function ends, it sets the operation field of the struct to the last key

(which contains the operation) and the number field to the buffer, multiplied by its sign.

Lab 3

```
void keyboard_get_entry(struct entry *result)
{
  // initialize to defaults
  int key = NO_KEY;
  int last_key = NO_KEY;
  int sign = 1;
  int num_buffer = 0;
  int count = 0;

  // run while there is no operation
  while(key == NO_KEY)
  {
    // get a key
    key = keyboard_key();

    // ensure a gap between presses
    while(last_key != NO_KEY)
    {
      last_key = key;
      key = keyboard_key();
    }

    // handle sign change and continue
    if(key == '~')
    {
      // flip sign
      sign = -1 * sign;

      // handle LCD changes
      if(count == 0)
      {
        if(sign == -1)
          lcd_put_char7('-', LCD_LAST_COLUMN);
        else
          lcd_put_char7(' ', LCD_LAST_COLUMN);
      }
      else
        lcd_print_int_neg(sign == -1, num_buffer);

      // continue
      last_key = key;
      key = NO_KEY;
    }

    // handle backspaces if there are digits
    if(key == '\b' && count > 0)
    {
      // cut least significant digit
      num_buffer /= 10;
      count--;

      // handle LCD changes
      if(count == 0)
```

```
54          {
55            if(sign == -1)
56            {
57              lcd_put_char7(' ', LCD_LAST_COLUMN - 1);
58              lcd_put_char7('-', LCD_LAST_COLUMN);
59            }
60            else
61              lcd_put_char7(' ', LCD_LAST_COLUMN);
62          }
63          else
64            lcd_print_int_neg(sign == -1, num_buffer);
65
66          // continue
67          last_key = key;
68          key = NO_KEY;
69        }
70
71        // handle digits
72        if(key >= '0' && key <= '9')
73        {
74          // ensure no overflow
75          if(count < MAX_COUNT)
76          {
77            // append digit
78            num_buffer = (num_buffer * 10) + (int)(key - '0');
79            count++;
80          }
81
82          // handle LCD changes
83          lcd_print_int_neg(sign == -1, num_buffer);
84
85          // continue
86          last_key = key;
87          key = NO_KEY;
88        }
89
90        // if none of the above were processed, exit
91      }
92
93      // marker for no number is INT_MAX
94      if(count == 0)
95          num_buffer = INT_MAX;
96
97      // loop exits when operation is pressed
98      // key contains the operation
99      result->operation = key;
100     result->number = sign * num_buffer;
101   }
```

6.4 Lab 4: An RPN Calculator

To implement the RPN calculator, we used the entry function from the last lab, which builds upon the reading and displaying functions from the previous labs. To store the input information, we used a stack, implemented by using an array and an integer indicating the head (point of insertion, and also the length) of the

stack. When a number is pushed onto the stack, the number is added at the head position and the head is incremented. When a number is popped off the stack, the head is decremented and the number at the new head is removed (this shifts the head and removes the number). In the actual code, I use C-style post-increment and pre-increment to shorten and in some way abstract the function of pushing and popping. Additionally, the stack uses lazy deletion, so the numbers in the stack are never actually "erased", but rather written over if necessary. The head variable tells us which of the stack items are accessible.

We then repeatedly take in inputs. Similar to how the keyboard reading function forced gaps between keys, the calculator forces "gaps" between entries by setting the operation to "NO_OP", which is represented as the null value after each valid entry.

Moving on to the actual evalution, if the input is a number, followed by the "Input" operation, the number is pushed onto the stack and the loop continues. If the number is accompanied by an arithmetic operator, then the number is pushed onto the stack and the operator is evaluated by popping two numbers off of the stack, performing the operation on them, and then placing the result into the stack. This algorithm acts on the definition of RPN input to evaluate the expression.

Underflow is handled when only one number is in the stack. In this case head (which also is the length of the stack) is 1, and the operation is ignored.

Overflow is NOT HANDLED AT THE MOMENT???, but the stack size of 100 is so large that the human entering the numbers will likely forget the expression before this occurs. The stack only increases in size from a continuous string of numbers, because operations reduce the size of the stack, so only 100 numbers or evaluated expressions entered in sequentially will result in overflow.

Lab 4

```
1  void calculator_rpn(void)
2  {
3    // initialize stack
4    int stack[STACK_SIZE];
5    int head = 0;
6
7    int last_key = NO_KEY;
8
9    struct entry entry;
10
11   while(1)
12   {
13     keyboard_get_entry(&entry);
14
15     // handle valid input (non-empty num, op pair)
16     if(entry.operation != NO_OP)
17     {
18       if(entry.number != INT_MAX)
```

```
      {
        // handle LCD output
        lcd_print_int(entry.number);
        if(entry.operation == INPUT)
          lcd_put_char7('i', 0);
        else
          lcd_put_char7(entry.operation, 0);

        // handle overflow
        if(head == STACK_SIZE)
        {
          //clear stack
          head = 0;
          clear_lcd();

          // print error
          lcd_put_char7('O', 0);
        }

        // push number onto stack
        stack[head++] = entry.number;
      }

      // perform non-input operation with check for underflow
      if(entry.operation != INPUT && head > 1)
      {
        // pop two entries from stack
        int val1 = stack[--head];
        int val2 = stack[--head];



        int result = 0;

        // perform operation
        switch(entry.operation)
        {
          case '+':
            if (val2/2 + val1/2 <= INT_MAX/2 || -1 * (val2/2 + val1/2) <= INT_MAX/2)
              result = val2 + val1;
            else
            {
              //clear stack
              head = 0;
              clear_lcd();

              // print error
              lcd_put_char7('E', 0);
            }
            break;
          case '-':
            if (val2/2 - val1/2 <= INT_MAX/2 || -1 * (val2/2 - val1/2) <= INT_MAX/2)
              result = val2 - val1;
            else
            {
              //clear stack
              head = 0;
```

```
76                clear_lcd();
77
78                // print error
79                lcd_put_char7('E', 0);
80              }
81              break;
82            case '*':
83              result = val2 * val1;
84              break;
85            case '/':
86              result = val2 / val1;
87              break;
88            default:
89              break;
90          }
91
92          // push result onto stack and print
93          stack[head++] = result;
94          lcd_print_int(result);
95
96          last_key = entry.operation;
97        }
98
99        // clear
100       if(entry.operation == CLEAR)
101       {
102         head = 0;
103         clear_lcd();
104       }
105
106       // continue
107       entry.operation = NO_OP;
108     }
109
110     // ensure a gap
111     while(last_key != NO_KEY)
112       last_key = keyboard_key();
113   }
114 }
```

6.5 Lab 4+: An Infix Calculator

We decided to try out working on an infix calculator in order to improve the usability of the calculator.

To evaluate infix expressions, two stacks are maintained. One is an expression stack, which contains an RPN expression and the other is the operator stack, which is necessary in using the chosen algorithm (Shunting-Yard) to convert infix to RPN. The expression stacks consists of entry structs, which are actually just numbers and operators, and the struct is just used to differentiate them. If the operator field is "NO_OP", then the entry is just a number, and if the number field is a "INT_MAX", then the entry is just an operator.

The function first takes the number portion of the entry and places it in the input stack. For the operator part, the function unwinds the operator stack into the expression stack until an operator of lower precedence appears. At this point, the input operator is pushed onto the stack and evaluation continues. If the operator in question is an equals sign, then the operator stack is entirely unwound and the expression stack is evaluated as a postfix expression. The result of this postfix expression is then printed, and the expression stack is reset with the result at the head.

Lab 4+

```

void calculator_infix(void)
{
  // initialize stacks
  struct entry expr_stack[STACK_SIZE];
  char op_stack[STACK_SIZE];
  int expr_head = 0;
  int op_head = 0;
  int last_key = NO_KEY;
  struct entry entry;

  while(1)
  {
    keyboard_get_entry(&entry);

    // handle valid input (non-empty num, op pair)
    if(entry.operation != NO_OP)
    {
      if(entry.number != INT_MAX)
      {
        // handle LCD output
        lcd_print_int(entry.number);

        // handle overflow
        if(expr_head == STACK_SIZE || op_head == STACK_SIZE)
        {
          //clear stacks
          expr_head = 0;
          op_head = 0;

          clear_lcd();

          // print error
          lcd_put_char7('O', 0);
        }

        // push number onto expression stack
        struct entry num_expr = {.number = entry.number, .operation = NO_OP};
        expr_stack[expr_head++] = num_expr;
      }

      if(entry.operation == INPUT)
        lcd_put_char7('i', 0);
      else
```

```
45              lcd_put_char7(entry.operation, 0);
46
47            // pop operator stack onto expression stack until lower priority operator
48            // push the current operator into the stack
49            switch(entry.operation)
50            {
51              case '(':
52                op_stack[op_head++] = entry.operation;
53                break;
54              case ')':
55                while(op_head != 0)
56                {
57                  char top = op_stack[op_head - 1];
58
59                  if(top ==' (')
60                  {
61                    op_stack[--op_head];
62                    break;
63                  }
64
65                  struct entry op_expr = {.number = INT_MAX, .operation = op_stack[--op_head]};
66                  expr_stack[expr_head++] = op_expr;
67                }
68                break;
69              case '+':
70                while(op_head != 0)
71                {
72                  char top = op_stack[op_head - 1];
73                  if(top ==' (')
74                    break;
75
76                  struct entry op_expr = {.number = INT_MAX, .operation = op_stack[--op_head]};
77                  expr_stack[expr_head++] = op_expr;
78                }
79                op_stack[op_head++] = entry.operation;
80                break;
81              case '-':
82                while(op_head != 0)
83                {
84                  char top = op_stack[op_head - 1];
85                  if(top ==' (')
86                    break;
87
88                  struct entry op_expr = {.number = INT_MAX, .operation = op_stack[--op_head]};
89                  expr_stack[expr_head++] = op_expr;
90                }
91                op_stack[op_head++] = entry.operation;
92                break;
93              case '*':
94                while(op_head != 0)
95                {
96                  char top = op_stack[op_head - 1];
97
98                  if(top == '+' || top == '-' || top == '(')
99                    break;
100
101                 struct entry op_expr = {.number = INT_MAX, .operation = op_stack[--op_head]};
```

```
102            expr_stack[expr_head++] = op_expr;
103          }
104          op_stack[op_head++] = entry.operation;
105          break;
106        case '/':
107          while(op_head != 0)
108          {
109            char top = op_stack[op_head - 1];
110
111            if(top == '+' || top == '-' || top == '(')
112              break;
113
114            struct entry op_expr = {.number = INT_MAX, .operation = op_stack[--op_head]};
115            expr_stack[expr_head++] = op_expr;
116          }
117          op_stack[op_head++] = entry.operation;
118          break;
119        // pop all operators onto expression stack
120        case '=':
121          while(op_head != 0)
122          {
123            struct entry op_expr = {.number = INT_MAX, .operation = op_stack[--op_head]};
124            expr_stack[expr_head++] = op_expr;
125          }
126          // show result and reduce stack
127          int result = evaluate_postfix(expr_stack, expr_head);
128          expr_head = 0;
129          struct entry num_expr = {.number = result, .operation = NO_OP};
130          expr_stack[expr_head++] = num_expr;
131          lcd_print_int(result);
132          break;
133        default:
134          break;
135      }
136
137      last_key = entry.operation;
138
139      // continue
140      entry.operation = NO_OP;
141    }
142
143    // ensure a gap
144    while(last_key != NO_KEY)
145      last_key = keyboard_key();
146  }
147 }
148
149 int evaluate_postfix(struct entry *expr_stack, int length)
150 {
151   // initialize stack
152   int head = 0;
153   int num_stack[STACK_SIZE];
154   int i;
155
156   // iterate through input
157   for(i = 0; i < length; i++)
158   {
```

```
159       struct entry input = expr_stack[i];
160
161       // operation == NO_OP indicates a number
162       if(input.operation == NO_OP)
163         num_stack[head++] = input.number;
164       else
165       {
166         int val1 = num_stack[--head];
167         int val2 = num_stack[--head];
168         int result = 0;
169
170         switch(input.operation)
171         {
172           case '+':
173             if (val2/2 + val1/2 <= INT_MAX/2 || -1 * (val2/2 + val1/2) <= INT_MAX/2)
174               result = val2 + val1;
175             else //handle overflow
176             {
177               //clear stack
178               head = 0;
179               clear_lcd();
180
181               // print error
182               lcd_put_char7('E', 0);
183             }
184             break;
185           case '-':
186             if (val2/2 - val1/2 <= INT_MAX/2 || -1 * (val2/2 - val1/2) <= INT_MAX/2)
187               result = val2 - val1;
188             else //handle overflow
189             {
190               //clear stack
191               head = 0;
192               clear_lcd();
193
194               // print error
195               lcd_put_char7('E', 0);
196             }
197             break;
198           case '*':
199             result = val2 * val1;
200             break;
201           case '/':
202             result = val2 / val1;
203             break;
204           default:
205             break;
206         }
207
208         num_stack[head++] = result;
209       }
210     }
211
212     // return last entry in stack
213     return num_stack[--head];
214   }
```

7 Lessons Learned

Michael: I learned how to interact with embedded systems that might have more limitations than the machines that I'm used to working with. When the calculator can only display a limited amount of data, it's sometimes a bit difficult to understand what's going wrong. What I've learned is how to try and intelligently work with the information I'm given. (I also have a greater sense of appreciation for those people that write debuggers).

Future students may want to try and find patterns in their errors by varying input to debug their code.

I don't wish anything more than what was told to me at the beginning of class. Realizing something yourself, or having someone guide you through a process is much more helpful in the learning process than simply being told that something is true.

Charlie: I gained a greater appreciation of the computer science and engineering that goes into systems beyond your normal computer. The course was an introduction to C for me, and was structured rather well. The lessons built upon each other, and it was helpful to learn from the more experienced students in the course.

Also, I enjoyed learning of the stack data structure, and how it applies to the RPN and infix methods of arithmetic. Lastly, the introduction to LATEX opened up a new way to format word documents into a pdf form.

8 Criticism of the Course

Michael: While I do think that the labs weren't terribly challenging, I do realize that this is an introductory course, and I think relative to its stated purpose, that the course was sufficient. I did wish that there was more interaction with the computer engineering side of things though.

Charlie: While we did explore some of the hardware inside the calculator, the course was more heavily focused on the computer science label, as programming was the bulk of the course. While I did prefer focusing on programming, I felt that we could have benefitted from more exposure to hardware elements of not just the calculator.

References

[1] Gateway lab for computer science and computer engineering. Online `http://www.cs.columbia.edu/~sedwards/classes/2012/gateway-spring/index.html`.

[2] Hp-20b repurposing project. Online http://www.wiki4hp.com/doku.php?id=20b:repurposing_project, July 2008.