ENGI E1112 Departmental Project Report:

Computer Science/Computer Engineering

By Yiming Ge, Anna Teng, Kaiven Zhou

Abstract

This report serves to document the ENGI E1112 Departmental Project. In this project, we used C to program the HP 20b calculator with the goal of turning it into an RPN calculator, one that operates using Reverse Polish Notation. Essentially, we are writing a new firmware for this calculator. We worked on the final product with algorithm efficiency and user friendliness in mind. This project teaches the important lesson of how to work on a limited platform with confined methods and a strict memory limit. Thus, it was paramount that we planned accordingly and wrote the right methods to achieve a functioning RPN calculator.

We started out by writing a program that makes a message scroll across the screen. This gave us the necessary background on working with the LCD display and manipulating the output of characters. Then, we worked on a program that scans for a key being pressed, and if so, returns that key. With this knowledge, we were able to proceed by writing a program that lets the user input a number. Finally, there was a sufficient foundation to make a functioning calculator, and in this case we implemented stacks to make an RPN calculator.

1 Introduction

In today's world, processors are found everywhere, whether you realize it or not. While there seems to be a big distinction between something as simple as a four function calculator and a top of the line desktop PC, the difference is actually not that great. Both devices have a processor, memory, and some sort of input and output. Most importantly, both run on firmware and software that has been pre-programmed into memory.

The HP 20b calculator contains all of the above; it features a liquid crystal display, a 6 by 7 matrix keyboard, and an Atmel AT91SaM7L128 processor with 128K of flash program memory. In this flash memory is the firmware that allows the calculator to perform all sorts of functions, in this case specialized finance functions which is undoubtedly why this calculator is dubbed the 'business consultant.' In this project, we attempted to reprogram this calculator into an RPN calculator, which required several steps that gradually lead to the final product. This type of programming is called 'embedded programming,' as the calculator doesn't appear to be a computer at first glance but in fact is. We used a JTAG port to communicate with the Atmel processor, and on the other end, we used a USB port to connect to Linux workstations running openOCD software to edit and compile the necessary files. This project attempted to manipulate the LCD screen and other input/output peripherals such as the keyboard to make the calculator behave like a real RPN calculator.

In the first lab, we created a scrolling message across the screen. This required knowledge of how for loops and character arrays in C work. In the second lab, we constructed a program that takes input from the keyboard and displays it on the screen. In the third lab, we used the knowledge and methods from the previous programs to make entering numbers possible. Lastly, in the fourth lab, we went one step further by allowing the user to operate on numbers in Reverse Polish Notation.

2 User Guide

In order to enter a number, simply press the digits from left to right on the keyboard, then hit the return key. Our calculator has the capacity to store and display any integer, both positive and negative, up to twelve digits long. Leading zeros, however, will not be displayed. In order to toggle between positive and negative, enter the number and then hit the +/- key.

Once you have entered a number (A) and pressed the return key, enter a second number (B). Now that there are two numbers, one of the four binary operations we support can be performed on them. Once you have finished entering the second number, press either the addition (+), subtraction (-), multiplication (x) or division (÷) symbol. The calculator will then perform A **__** B, and display the result.

The calculator we have reprogrammed is an RPN calculator. RPN stands for "Reverse Polish Notation," and is characterized by entering the number first followed by the operation. Additionally, numbers can be stored in what is called a "stack". A way to visualize it is to image that every time a number is pressed and then return hit, it is added to the top of a literal, vertical stack of numbers. When an operation is pressed, it is performed on the topmost two items of the stack in the format written above. The resulting answer becomes the new top of the stack. In the case of our calculator, the maximum stack height is 30.

An RPN calculator provides alternate, and often simpler alternatives to entering long expressions with many terms. For example, if one were trying to perform (9 + 8 + 7 + 6 + 5), he could enter the following keys: 5 return 6 return 7 return 8 return 9 (adding each number to the stack) and then + + + +, adding all of the numbers in correct order of operations.

- It scrolls from right to left? YES
- If you hit an operation first it returns INTMAX
- If you keep hitting an operation after the stack only has 1 term nothing happens

Source: http://www.hpmuseum.org/rpn.htm

- key in a number
- hit "return" to tell the calculator that you have finished entering the number
- Key in another number
- hit an operation key. The first number will perform that operation on the second number (for example if you entered A return B -, it would perform A-B not B-A
- There is a stack, so you can do it for more than 2 numbers. A return B return C + +. Or, A return B + C +. You do not need an enter after the B because the operation key makes clear that you are done entering that number.
- The only functions our calculator supports are multiplication, division, addition, and subtraction. Negative and positive integers are supported as well.

3 Social Implications

The calculator we used is extremely useful because it is easily re-programmable. We were able to turn this calculator into an RPN calculator by writing the necessary code. There is so much flexibility in the use of this calculator because by programming it, we can change its function. The calculator is valuable to people who require technical calculators with specialized functions. We reduce the cost by condensing these many possible calculators into one package.

The result of re-programming the calculator is a RPN calculator that is effective and user-friendly. This calculator can accurately and quickly perform the simple arithmetic that people need in day to day life: it encompasses multiplication, division, addition, and subtraction, and also has the ability to process negative numbers. Because it neglects the more complicated math that most people do not need, it is also a useful tool for children who are only beginning to learn arithmetic. Because Reverse Polish Notation does not take into account the complicated operation orders but instead focuses purely on the operations between numbers, this RPN calculator has definite use for children early in their education.

4 The Platform

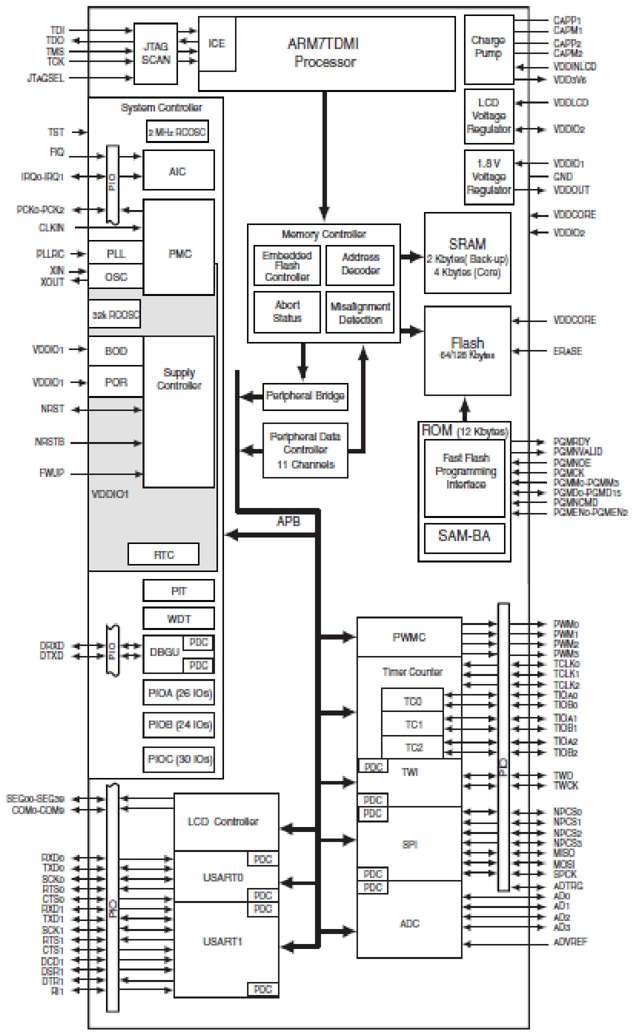
HP 20b Calculator

The calculator we used for this project is the HP 20b. The original purpose of this calculator was for finance, insurance, real estate, accounting, and statistics functions. However, for this project, Professor Edwards stripped the calculator of all of its features.

4.1 The Processor

The HP 20b uses the Atmel AT91SAM7L128 chip, containing the ARM7TDMI processor. The chip is composed of the single processor, surrounded by memory and a multitude of peripherals. The 128 at the end of the name represents the amount of programmable flash memory it posses (128 KB). The ARM7TDMI is a 32 bit microprocessor designed to run at low power. It uses a three-stage instruction pipeline: Fetch → Decode → Execute. During operation, while one command is being executed, the next command is being decoded, and a third command is being fetched. The ability for the processor to allow these three actions simultaneously greatly increases the efficiency.

Two important peripherals for our purposes are the System Controller and the LCD controller. The System controller has software that controls both the internal clock and the power supply to each of the other peripherals. The LCD controller generates AC waveforms which control the calculator's LCD display.



4.2 The LCD Display

The LCD display of the HP 20b contains 2 rows. For this lab, we only utilized the bottom row. The bottom line contains twelve squares to display characters. Each of the digits reserved for characters is composed of 9 possible line segments:

- The top center horizontal
- the top left vertical
- the middle center horizontal
- the top right vertical
- the bottom left vertical
- the bottom center horizontal
- the bottom right vertical
- the decimal point
- the comma.

Each digit and operation is composed of these nine line segments.

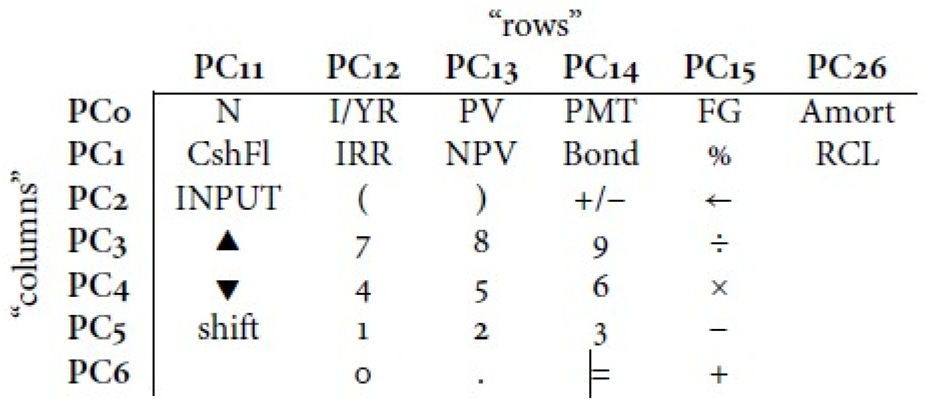
To take care of the complicated task of making the LCD display characters, Professor Edwards gave us a library of three functions to work with. The first, *lcd_init* initializes the LCD display and turns on its power supply. The next function is called *lcd_put_char7* prints a specified character in the specified column of the LCD display. The 7 in the name stands for the 7 main line segments that make up most of the characters (numbers 1-7 above). The third function was *lcd_print7*. This function prints an array of characters, always starting in the first column of the display.

4.3 The Keyboard

Before beginning our lab, we took time to disassemble some assorted keyboards, including computer keyboards, a fax machine, and a very simple calculator. One commonality we observed among each of the keyboards was an underlying two dimensional matrix of wires. Each key corresponded to an intersection of row and column.

Like all of the keyboards that examined, the keyboard of the HP 20b is organized in a matrix of size 6x7. Confusingly, in this keyboard matrix, vertical stacks conventionally referred to as columns are referred to as rows, and vice versa. The action of pressing a key shorts together a row and a column, equalizing the voltage between the two. The matrix is connected to pins on the SAM7L chip which lead to a parallel I/O controller. This controller enables software to both control and set the state of each pin.

Before we began the second phase of our lab (Listening to the Keyboard), Professor Edwards bestowed upon us some functions which allowed us to scan the matrix of the keyboard for a key being pressed. The first of these is called *keyboard_init*, which initializes the keyboard, setting all pins to high. The next function, *keyboard_column_high*, sets the pins corresponding to a given column to “high” voltage. The corresponding function is the function *keyboard_column_low*, which just does the opposite. The last function, *keyboard_row_read* checks the voltage on the pin of a specified row, and if it reads high, returns true.



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

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

5 Software Architecture

We have 3 methods that interact together to let the calculator perform arithmetic using Reverse Polish Notation. The first method *keyboard_key()* scans the keyboard for keystrokes and returns the pressed key. The second method *keyboard_get_entry(struct)* gets an entered number and operation and stores them in the passed struct. The third method *rpn(int *stack, int, struct)* uses a stack, the location of the top of the stack, and a struct to perform the four binary operations.

The first lab was used as an introduction to the platform, as well as an introduction to programming in C.

6 Software Details

6.1 Lab 1: A scrolling display

We start by creating a function *strlen* which computes the length of a character array by iterating through the array until a null character is reached, and counts and returns how many characters there are. Then we create a character array *message* to be displayed. We find and store its length into *len* using function *strlen* and set *startingPosition* = 0. In an infinite loop we display our message by using a for loop that places the character *message[i]* at *placeNext* mod *NUM_DISP_DIGITS* (modulus is used so that any characters displayed off the screen reappear at the left of the screen), then increments *placeNext*. The end result is that all the characters of *message* are placed on the screen. Then we pause the system by using do-nothing while loop which increments *dummyCounter*. The screen is wiped of all its characters, the *startingPosition* is incremented, and the loop restarts, and *dummyCounter* is reset to 0 again.

Our Solution for Lab 1: A scrolling display

```
//By: Abhinav Mishra, Andrew Pope, Yiming Ge, Anna Teng, Will VanArsdall, Kaiven Zhou
#define NUM_DISP_DIGITS 12
#define NUM_TO_WAIT 50000
int strlen(const char *s)
{
	int n;
	for(n=0; *s!='\0';s++)
		n++;
	return n;
}
int main()
{
	lcd_init();
	char message[] = "test";			//the message
	int len = 0;
	len = strlen(message);			//length of the message
	int startingPosition=0;

	while(1)
	{
		int i, placeNext, dummyCounter=0;
		placeNext=startingPosition;
		for(i=0; i < len; i++)
		{
			lcd_put_char7(message[i],placeNext% NUM_DISP_DIGITS);
			placeNext++;
		}
		while(dummyCounter< NUM_TO_WAIT)	dummyCounter+=1; //dummy loop
		lcd_print7("            "); //clears the screen
		startingPosition++;
	}
	return 0;
}
```

6.2 Lab 2: Scanning the Keyboard

We created a character array *keys* which stores every possible keystroke, along with some empty values because the upper part of the calculator has more keys per row than the lower part. Columns and rows have two states: 0 and 1 (or, low and high), and by default all the columns are set to 1. We define an integer *index,* to represent the index of the key that is pressed. In a *for* loop, we set each of the columns to 0 using *keyboard_column_low()* and use another *for* loop and check each row with *keyboard_row_read()*. If the key is pressed in the row we are checking, *keyboard_row_read()* returns true, and otherwise it will return false. If true, then we set the column back to 1 using *keyboard_column_high()* (in order to return the calculator back to its default state), and return *key[index],* the key pressed. If false, *index* increments. If the row checking *for* loop ends without finding the key pressed, then the column is set back to 1, and we start checking the next column. If no key is pressed, 0 is returned.

Our Solution for Lab 2: Scanning the Keyboard

```
#define NUM_COLUMNS 7
#define NUM_ROWS 6
const char* keys[44] = {"", "N", "I/YR", "PV", "PMT", "FG", "Amort",
        "CshFl", "IRR", "NPV", "Bond", "%", "RCL",
        "INPUT", "(", ")", "+/-", "<-", "",
        "UP", "7", "8", "9", "/", "",
        "DOWN", "4", "5", "6", "x", "",
        "SHIFT", "1", "2", "3", "-", "",
        "", "0", ".", "=", "+", ""};
int keyboard_key()
{
        int i, j;
        int index = 1;
        for(i = 0; i < NUM_COLUMNS; i++)
        {
                keyboard_column_low(i);

                for(j = 0; j < NUM_ROWS; j++)
                {
                        if(!keyboard_row_read(j))
                        {
                                keyboard_column_high(i); //Resets the current column
                                return keys[index];
                        }
                        index ++;
                }

                keyboard_column_high(i);
        }
        return 0;
}
```

6.3 Lab 3: Entering and Displaying Numbers

We use a struct to hold the user's input: the number entered as well as the operation entered. We define integers *inputNumber,* the current number that the user has entered; *lastKey,* which will store the last key pressed; *inputOperation,* the user's inputted operation; and *numOfDigits,* the number of digits in the user's number. By default, *inputNumber* is *INT_MAX*. Then, in an infinite loop, we get attempt to get the user's keystroke while avoiding a problem: since the loop iterates faster than the user can react, a pressed keystroke will register in multiple iterations of the loop. Therefore, we start the loop by ensuring no key is being pressed. Once we are sure that no key is pressed, we get the key pressed and store it in *lastKey*. Several cases are now checked:

If the *lastKey* is a digit, and the total number of digits in the *inputNumber* is less than the predefined maximum number of digits *MAX_NUM_DIGITS*, then we proceed. If this keystroke is the very first, *inputNumber* is *INT_MAX*, so we set *inputNumber* to 0. We then "append" a digit to *inputNumber* by converting *lastKey* into an integer, multiply *inputNumber* by 10, and add the converted number to *inputNumber*. We display *lastKey* on screen, and increment the number of digits.

If '~' is pressed, we check if any number has been inputted already (as in the previous case), then change the sign of *inputNumber,* and display a '+' or '-'.

If one of the four operations (+,-,/,*) or enter (\r) is pressed, then we set *inputOperation* to equal *lastKey*, set the struct's number to equal *inputNumber* and the struct's operation to *inputOperation*, and return.

If none of the above is pressed, nothing happens. Any key that is not a number or operation is ignored.

Our Solution to Lab 3: Entering and Displaying Numbers

```
//By Kaiven Zhou, Yiming Ge, Anna Teng
void keyboard_get_entry(struct entry *result)
{
        int inputNumber=INT_MAX;

        int lastKey=-1,inputOperation=-1,numOfDigits=0;
        lcd_put_char7(0,'+'); //by default the number is positive
        while(1)
        {
                while(keyboard_key());          //ensure no key is pressed

                while(lastKey==-1) lastKey = keyboard_key();        //get key

                if('0'<=lastKey && lastKey <= '9' && numOfDigits<=MAX_NUM_DIGITS)
                {
                        if(inputNumber==INT_MAX) //excutes if lastKey is the first key pressed
                                inputNumber=0;
                        int integerOfLastKey = lastKey-'0'; //char to int
                        inputNumber*=10;
                        inputNumber+=integerOfLastKey;
                        lcd_put_char7(lastKey,numOfDigits+1);
                numOfDigits++;
                }
                else if(lastKey == '~')
                {
                        if(inputNumber==INT_MAX)
                                inputNumber=0;
                        inputNumber*=-1;
                        if(inputNumber<0)
                                lcd_put_char7(0,'-');
                        else
                                lcd_put_char7(0,'+');
                }
                else if(lastKey=='+'||lastKey=='-'||lastKey=='*'||lastKey=='/'||lastKey=='\r')
                {
                        result->number = inputNumber;
                        result->operation = inputOperation;
                        return;
                }
        }
}
```

6.4 Lab 4: An RPN Calculator

We create an array **stack* which will be used as a stack, and then we keep track of the index in the array that is the position of where the next element would be placed in **top*. The size of the array is limited by the constant *STACK_SIZE.*

If a number has been entered, then the first *if* statement executes. The number is added to the top of the stack and **top* is incremented.

If there are at least 2 elements in the stack, operations can occur. We use a *switch* to determine what to do depending on the operation. If operation is one of the 4 mathematical operations (+,-,/,*), then the highest two numbers in the stack (at positions *(*top)-1* and *(*top)-2*) are acted upon, where the first number in the operation is *(*top)-1*. If enter is pressed, nothing happens, because first if statement has already handled all the necessary actions.

Our Solution to Lab 4: An RPN Calculator

```
//By Kaiven Zhou, Alex Ge, Anna Teng

void rpn(int *stack, int *top, struct entry *input)
{
        if(input->number != INT_MAX) //if there is a number, add it to the stack
        {
                stack[(*top)] = input->number;
                (*top)++;
        }
        if( (*top)>1 ) //only do an operation if there are at least 2 numbers in the stack
        {
                switch(input->operation)
                {
                        case '\r‘:      break;          //done in the first if statement
                        case '+‘:       lcd_print_int( stack[(*top)-2] += stack[(*top)-1] );
                                (*top)--;
                                break;
                        case '-‘:       lcd_print_int( stack[(*top)-2] -= stack[(*top)-1] );
                                (*top)--;
                                break;
                        case '/':   lcd_print_int( stack[(*top)-2] /= stack[(*top)-1] );
                                (*top)--;
                                break;
                        case '*‘:       lcd_print_int( stack[(*top)-2] *= stack[(*top)-1] );
                                (*top)--;
                                break;
                }
        }
}
```

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

#define STACK_SIZE 30

int main()
{
	int i;
	struct entry entry;
	// Disable the watchdog timer
	*AT91C_WDTC_WDMR = AT91C_WDTC_WDDIS;

	lcd_init();
	keyboard_init();

	int stack[STACK_SIZE];
	int top = 0;
	while(top<STACK_SIZE)
	{
		keyboard_get_entry(&entry);
		rpn(&stack,&top,&entry);
	}
	lcd_print7("Overflo"); //only shown if more numbers are added to the
					//calculator than can be stored in the stack

	return 0;
}
```

7 Lessons Learned

This project taught us C, an important programming language. We have learned valuable problem solving skills through writing and fixing code to achieve a working product. Most importantly, we learned that the best way to program is not by trial and error, but by using logic and sound reasoning to proofread our code beforehand. This method allows us to consciously realize our mistakes so that we won't make the same mistakes later on. It also forces us to put more thought and focus into what we're writing.

This project also made us realize the important role of planning and organization in programming. On numerous occasions, we were confused by our nebulous variable names, causing us to lose track of the purpose of those variables. Because of a lack of planning beforehand, some of our code could be much more efficient by condensing two variables into one. Furthermore, some 'magic numbers' are better off replaced with constants. These are just some of the lessons we have gleaned from this project.

8 Criticism of the Course

Going into this lab, nearly none of the members of our group had any experience programming in C. So although jumping in and trying to learn the language in a day was challenging, it was a rewarding experience to work through it together. However, although we started at about the same level in the language C, experience programming in general ranged from nearly none to extensive. Because of this discrepancy, members of the group who were skilled in writing code ended up carrying most of the programming. It was often difficult for members who did not understand basics about loops, functions, and syntax to follow what was being written.

In addition to learning how to program in C, we had to understand the hardware that we were working with. The calculator was connected to the computer by a JTAG connector. Additionally, the calculator had to be connected to a power source. In the beginning stages of the project, when the calculator malfunctioned, sometimes it was due to a disconnected chord, not a bug in the program. Getting used to the platform made it harder to focus on the code itself. On a larger scale, the room in which we worked was cramped and lacked chairs. Occasionally, calculators would go missing. These physical inconveniences made the lab trivially more confusing and difficult.

Code reviews were a helpful way to see a different solution to the same problem. Once we got started on writing a function, our group would often get stuck in a single track of reasoning and problem solving. Instead of stepping back and trying an approach which avoided our problem, we would try to hack through the roadblock we faced. Reading the code of different groups, who often approached the programming from a different angle, showed us simpler, more elegant ways to write the programs.