ENGI E1112 Departmental Project Report: Computer Science/Computer Engineering

Arushi Gupta
Jake Kazimir
Serena Shah-Simpson

December, 2012

Abstract

The purpose of this final report is to present the assigned project for the Computer Science project lab of ENGI E1112. For this project, we were required to write firmware for an HP 20b calculator that had been cleared of program code. The project was split up into four different labs that consisted of getting the calculator to display numbers on the screen and do calculations. We completed the first two labs, which dealt with getting numbers stored and displayed correctly.

This report begins with a thorough description of how the calculator is meant to be used, when correctly programmed. The report then goes on to describe elements of the hardware in the calculator and gives an in-depth break down of the software that was written for each lab assignment that we completed.

1 Introduction

In general, when people refer to computers, they are talking about personal computers such as desktops or laptops. However, these computers make up only a small fraction of the computers in existence. Computers exist almost everywhere in developed countries. They are present in the majority of electronic devices purchased by consumers, as embedded systems. These embedded systems are created to perform specific tasks, and so may not seem very relatable to what computers are generally seen as- user-friendly and flexible.

The problem that we were faced with was that an HP 20b calculator was wiped clean of all firmware that related numbers entered on the keyboard to memory and display. Using the C language, we wrote some code to reinstate these processes. We were able to transfer our code from the computer (where we wrote it) to the processing chip of the calculator to become embedded programming. This embedded programming instructed the calculator what to do under certain circumstances. Our writing process is detailed in the sections below, along with a description of the workings of an HP 20b calculator.

2 User Guide

2.1 Turning on the calculator

To turn the calculator on and off, press the on/ce (8) key.

2.2 Entering Numbers

Unfortunately, currently the calculator prints numbers several times per button push, but the following describes our intended use for the calculator after finishing all labs. Pressing a key will display the corresponding symbol on the lcd. Numbers and symbols will be displayed next to each other unless the user presses input or exceeds the 12 character display. If the user exceeds the 12 character limit, the screen does not display additional numbers and will not change until the user presses input(4/5). To enter a negative number, press the '-' (9) key and then enter the desired number.

2.3 Clearing the Screen

Input can be used to put the number onto the stack and will clear the screen.

2.4 Performing operations

We were not able to complete Lab 3 and Lab 4, but our intention was to have the backspace key remove the rightmost character on the display. The user would input numbers and put them onto the stack by either pressing the multiplication, division, addition, subtraction, or equals symbol(9). Pressing one of these symbols would clear the screen of characters and display the operation which was most recently called. The calculator would then clear the screen and display the results of the operation.

3 The Platform

3.1 The Processor

The Atmel AT91SAM7L128 processor (aka SAM7L chip) is the embedded processor of this calculator. It was built around a 32-bit ARM7 RISC processor

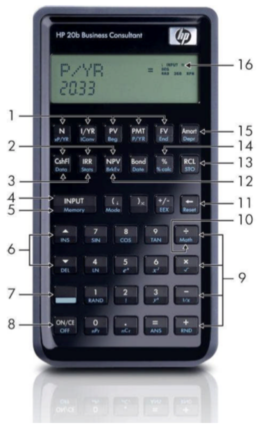

Figure 1: HP 20b Keyboard & Display Reference [**?**]

core to perform as a low energy processor in embedded systems, as described by the ARM7TDMI technical reference manual [**?**]."AT" refers to Atmel, "SAM" to "smart ARM core," and "128" to the flash memory, 128 Kilobytes.

The processor of the SAM7L chip is located at the top of the block diagram shown in Figure 2, and is surrounded mainly by memory and peripherals. The chip contains a system controller (located at the left hand side of the chip in the schematic), that controls the clocks and power supply of the processor's peripherals in order to ensure that each peripheral gets power only when it needs it.

3.2 The LCD Display

The calculator communicates with the user through an LCD display screen, which is connected to the SAM7L chip. The display consists of two lines. We did not write firmware for the first line, which under HP's programming showed operation status and symbols up to eight characters. The bottom line, which we did write code for, can show up to 12 digits, namely the numbers entered by the user and the results from performing a calculation.

For this project, we were provided with the following library functions:

- *lcd_init*: Turned on the display's power supply
- *lcd_put_char7*: Displayed a specified character or number (entered in ASCII code) in a specified position on the LCD display

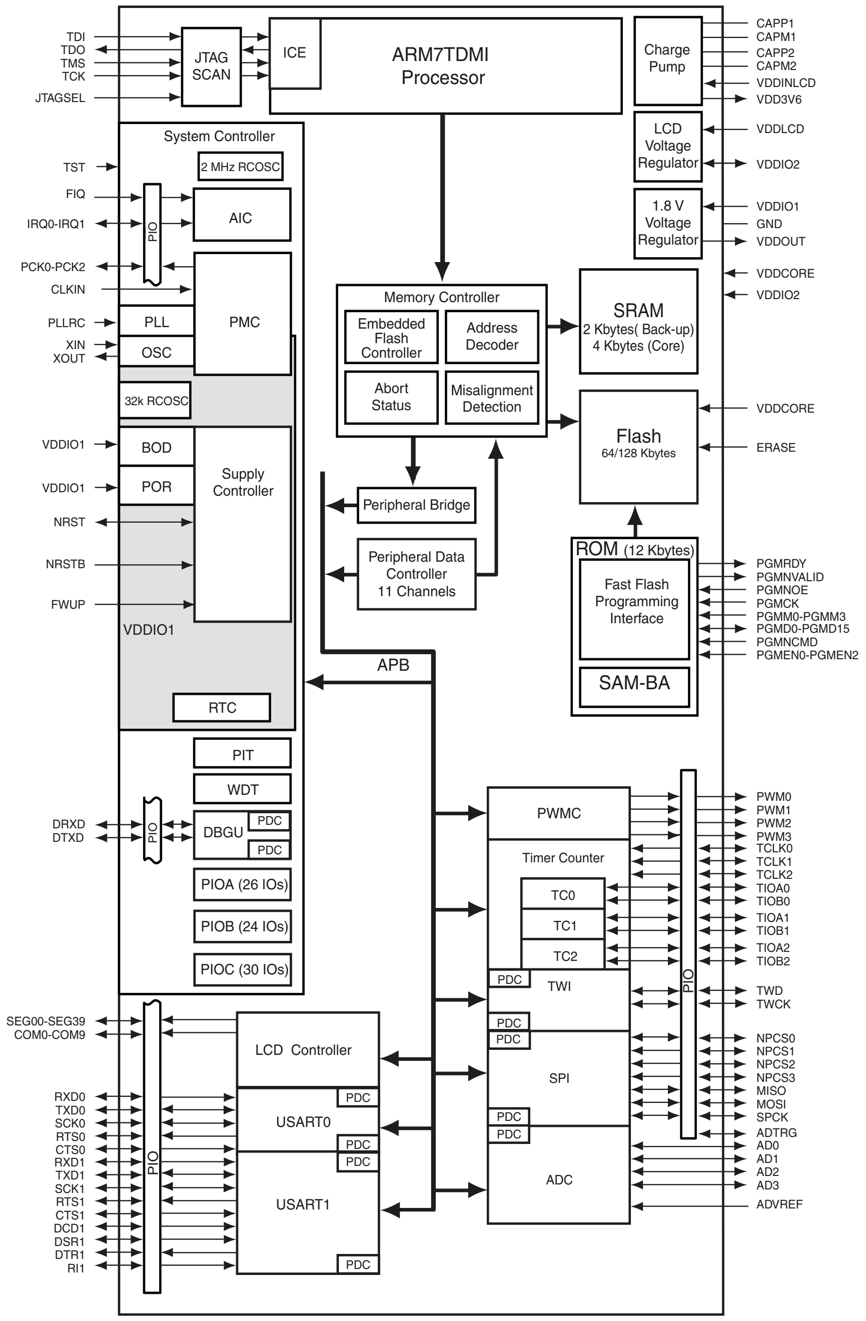

Figure 2: Diagram of the AT91SAM7L128 processor that controls the functioning of the HP 20b, from Atmel's released datasheet [**?**]

.

3.3 The Keyboard

The HP 20b's keyboard is connected to pins attached to the SAM7L chip. Figure 3 shown below details the row and column wires that are shorted by the keys. When a certain key is pressed, both the row and the column of the key will be shorted, and so then a certain pair of pins are shorted, one for the column, one for the row. The state of the pins is read by software in a peripheral and then interpreted as a number or command to the firmware of the processor.

We were provided with these library functions for the keyboard:

- *keyboard_init*: Set all the columns high with pull-up resistors on the rows
- *keyboard_column_high*: Set a specified column high
- *keyboard_column_low*: Set a specified column low
- *keyboard_row_read*: Returned true if specified row was high, return false if low

4 Software Architecture

Keyboard_key returns an integer if a key is pressed. *DepressedKey* takes this integer as input and returns it only when the key as gone from being pressed to being not pressed. *Print_screen* takes the value from *depressedKey* and prints the appropriate symbol to the display. This process is repeated inside an infinite loop in the main method.

5 Software Details

Software Details includes cleaned-up listings of every bit of code we wrote for each lab. Followed by each code segment is a brief paragraph explaining the intent of each block of code as well as other comments. Unfortunately, our final code would not run with all the previous coding. We therefore began to condense the code and keep only what was necessary. The beginnings of that code are the last block of code in this segment.

5.1 Lab 1: A Scrolling Display

In this lab we were instructed to edit hello.c and create a function that takes an integer argument and displays it in decimal on the calculator. We had up to 12 digits to work with.

Figure 4 shows our code for lab 1. *Main* simply calls our print function on a specific integer number. It can be positive or negative, however, the only constraint is that it must be an integer value. The program immediately clears the previous

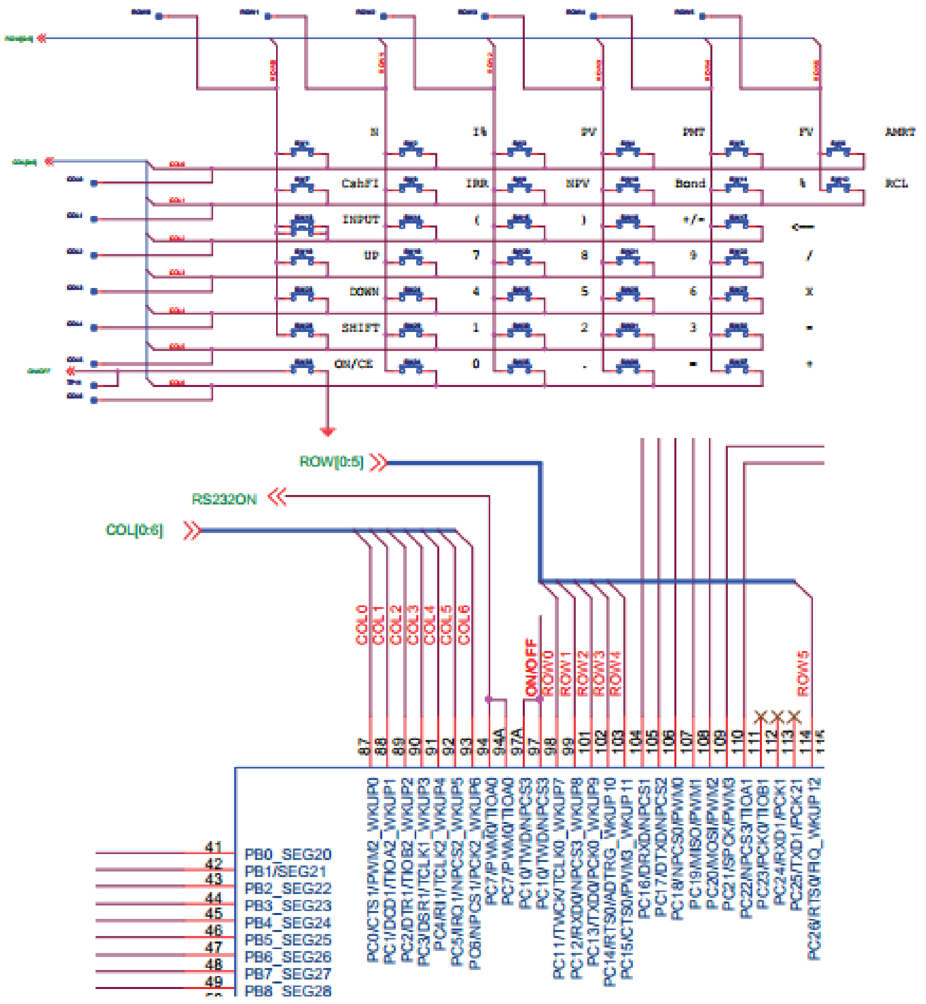

Figure 3: Diagram of the keyboard row/column wiring

.

```
int main(){
  lcd_init();
  printFunction(-123523);
  return 0;
}
void printFunction(int NUM) {
  clearScreen();  //clear screen
  int num_position [11];        //int array to hold nums
  int n;                        //initialize index for printing NUM
  int x;                        //initialize index for storing NUM
  int ASCII_CORRECTION = 48;    //correction
  if(NUM<0){
    lcd_put_char7('-',0);       //place negative sign
    NUM=-NUM;
//make positive
  }

  x=1;
  n=1;

  while(NUM >= 1){
    num_position[x] = NUM%10;     //next char
    NUM = NUM/10;                 //since int will cut off
    x++;                          //add 1 to index
  }
  if(NUM==0){
    lcd_put_char7(0+ASCII_CORRECTION, 11);
  }
  for(n; n<x; n++){
    lcd_put_char7(num_position[n]+ASCII_CORRECTION,12-n); //print
  }
}
```

Figure 4: Our solution for Lab 1: displaying numbers on the LCD screen

entry on the screen using the *clearScreen* function. Once the number has been declared and the screen has been cleared, *printFunction* decides if the number is positive or negative. If the number is negative, *printFunction* will input a negative sign into slot 0 of the calculator screen. The most important part of the code is the while statement (while(*NUM* >= 1)) because it breaks up the number into individual digits. It does this using the modulus function and then setting the number equal to the number divided by 10. This will truncate the number because *NUM* is defined as an integer. The modulus function will get the farthest digit to the right. We then store this digit into an array of integers. For example: *NUM* =1234, *NUM* %10=4, *NUM* /10=123, 4 is stored in the array, *NUM* becomes 123, x=x+1 (x is simply used to keep track of the size of the array). The *while* loop will keep iterating until *NUM* is less than one.

One flaw with this design is that the number 0 will not print. Therefore, we inserted an if-statement to catch this error (if(*NUM* ==0)) and then print zero. A key note to this part is that the program does not store zero in the array. Therefore the *for* -loop will not execute. The final step in the program is to print the number. We do this using a *for* -loop (*for* (n; n<x; n++)) and the array of digits. We print the first number (first modulated into array) into the 12th spot, then the 11th, and so on and so forth. This will continue as long as the array still contains another number (can tell by the index x).

5.2 Lab 2: Scanning the Keyboard

In Lab 2, we had to find a way to read the keyboard and display which key is being pressed. To do this we began thinking of ways to scan the keyboard and thought it best to use an infinite loop (this proved later to be a bad method) that would continuously scan the rows and columns looking for which key has been pressed. We use two methods *keyboard_key* which identifies the key being pressed and *pressed_key* which assigns a value to that key.

The code in Figure 5 allows the calculator to know which button has been pressed. It continuously scans the keys one row at a time. Initially all the rows and columns are set to high. Then, the column being checked is set to low. If a button in this column is pressed, its row will also become low. Therefore, we can tell which button has been pressed because the row and column will be low (similar to cartesian coordinates). We then save these coordinates as dummy variables. In the method *pressed_key*, we use the 2x2 array of integers to discover the value of the pressed key and print it. When nothing is being pressed, the screen will display “nada”. Unfortunately, this program will have the number continuously flash because it constantly clears the screen and checks to see if it is being pressed

```
int keyboard_key(){
 int key[2]={-1,-1};        //key coordinate array to be returned
 int j=0;
 int i=0;
 for (;;) {                        //infinite loop
   keyboard_init();                 //set all to high
   for(j=0; j<7; j++){
       keyboard_column_low(j);     //"look here" column low
       for (i = 0 ; i < 6 ; i++){    //iterate through rows
         if (!keyboard_row_read(i)){
           key[1]=i;                //i=row
           key[0]=j;                //j=column
         }
       }
       keyboard_column_high(j);    //reset column high
   }
   int returnkey=key[1]*10+key[0]; //keeps track of column, row
   if(key[1]!=-1){
     return returnkey;             //returns [column, row] as an integer
   }
   else{
     lcd_print7("NADA");           //nothing is being pressed
   }
}
}
void pressed_key(int x){
   int i=x/10; //i is row
   int j=x%10; //j is column
   int CALC_KEYBOARD[7][6]={{0,0,0,0,0,0},
                    {0,0,0,0,0,0},
                    {0,0,0,0,0,0},
                    {0,7,8,9,0,0},
                    {0,4,5,6,0,0},
                    {0,1,2,3,0,0},
                    {0,0,0,0,0,0}};
   int y=CALC_KEYBOARD[j][i];
   lcd_put_char7(y+'0', 11);  //prints
   clearScreen();              //clears screen
}
```

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

again. Not being able to hold the value on the screen will become an issue later.

5.3 Lab 3: Entering and Displaying Numbers

Due to time constraints and several problems with our code, we were unable to complete Lab 3. A major problem with our code is that it does not stop recording what has been pressed. By this we mean that if 4 is pressed, 44444 will appear on the calculator screen. This is because the calculator is extremely fast, and our button pressing is extremely slow in comparison. The problem with our code is that it is running an infinite for-loop and there is no "stop" condition that lets the keyboard know that the key is still depressed. We tried to solve this problem by writing the function *depressedKey*. Unfortunatley, this did not work. We then began to add a struct which complicated things further. After exponentially increasing the problem with our code, we were forced to completely restart from Lab 2.

During the last two weeks we decided to completely rewrite our code (though not 100% from scratch). We decided simply have one method which would handle when a key is pressed and print it. We were then going to write individual methods as necessary for Lab 4 depending on what key was being pressed and the function that was going to be performed. Figure 6 shows what we came up with, though we should note that it is not complete and was never tested.

Rather than have a separate "getter" and "setter" function, we have combined *keyboard_key* and *pressed_key*. Therefore, once the key has been pressed we will know immediately know what it is.

5.4 Lab 4: An RPN Calculator

We did not begin Lab 4, but if we had completed Lab 3, Figure 7 shows a very rough outline as to our course of action in coding. Knowing when a key was pressed, we could construct a series of *if/elseif/else* conditions in which we can have it display a number, know to multiply two numbers, save a number, etc (all calculator functions). We can handle all these by simply writing methods for each case and have the method be called in the *if/elseif/else* statements. Unfortunately, the pressed key problem was not solved.

The ouline shown in Figure 7 should work because it will execute the find key method only while the key is not being held down. While pressed, the program will do nothing. Given more time, we believe that we could have completed this work.

```
void keyboard_key(){
  int j=0;
  int i=0;
  int x=-1;
  int y=-1;
  for (;;) {                           //infinite loop
    while(!keyboard_init());
      keyboard_init();                 //set all to high
      for(j=0; j<7; j++){
        keyboard_column_low(j);      //"look here" column low
        for (i = 0 ; i < 6 ; i++){     //iterate through rows
          if (!keyboard_row_read(i)){
            x=i;                     //i=row
            y=j;                     //j=column
            break;                   //exit for loop
          }
        }
        keyboard_column_high(j);   //"look here" column high
     }
     int count=0;
     if (x < 0 && y<0)
        return;
     else if(x==2 && y==5){
         //delete (reset number)
     }
     else if((x>=1 && x<=3)&&(y>=3&&y<=5)){
         //print number
     }
     else if (asdlf;asd;lf){
         //do math
     }
     else {
          count = count+1;
          number[11-count]=CALC_KEYBOARD[y][x];;
          int z=0;
          for(z;z<11;z++)
          {
            lcd_put_char7(number[z]+'0', z);  //prints
          }
      }
    }
  }
```

Figure 6: Our second revamped atempt at Lab 3

```
While(pressed) ;
While(unpressed){
    //Find key
    //Do function/print to screen
}
```

Figure 7: Possible course of action for Lab 4

6 Lessons Learned

Choosing the best return type for a method can make writing later methods a lot easier; having *keyboard_key* return an integer made it difficult to display a number only once per key push. Having *keyboard_key* return a Boolean would have made more sense, as *keyboard_key* should really keep track of the state of whether a key is being pressed, in other words, if it is being pressed or depressed. Having *pressed_key* return an array of characters also made it easier to print than if we had returned an array of integers.

Always have a working piece of code which can test if the code is buggy or if the calculator battery died. Remember to check all the links connecting the calculator to the computer before assuming that the battery died. The longer the code and the more necessary it is to copy paste, the greater the likelihood that there is a simpler solution. Ultimately trying to just get the calculator to do something is less fruitful than planning carefully the design for the code. Remember to look at edge cases. For example, what happens if the user is already holding down a key when *keyboard_key* is called? It is much easier to unit test smaller methods than figure out everything that went wrong at the end.

7 Criticism of the Course

This course was very entertaining, fun, and rewarding. Having the ability to program from scratch and see our code begin to work on an actual device yielded a huge feeling of accomplishment (though on the other end of the spectrum, compiling errors were infuriating). The labs were difficult but reasonable, as the instructions were quite clear. Given proper programming technique it would be apparent how they would all fit together to form a semi-functional calculator. The code reviews were quite helpful in that it was useful to see how other people would break up the problems. Also taking the time to explain our code forced us to re-assess certain areas and make changes to our code as necessary.

We did end up spending a lot of time simply trying to connect the calculator the computer. At times it was difficult to tell if the calculator was plugged in, since the light on the JTAG connector did not always light up, so more consistent equipment would have been helpful. Additionally, a little more background in C/C++ programming would have been extremely useful for these assignments, since we spent a lot of time looking online for the proper syntax/semantics. Other than these issues, there were no major problems or concerns. All in all, we really enjoyed this course and would not change anything major.