NXP3878

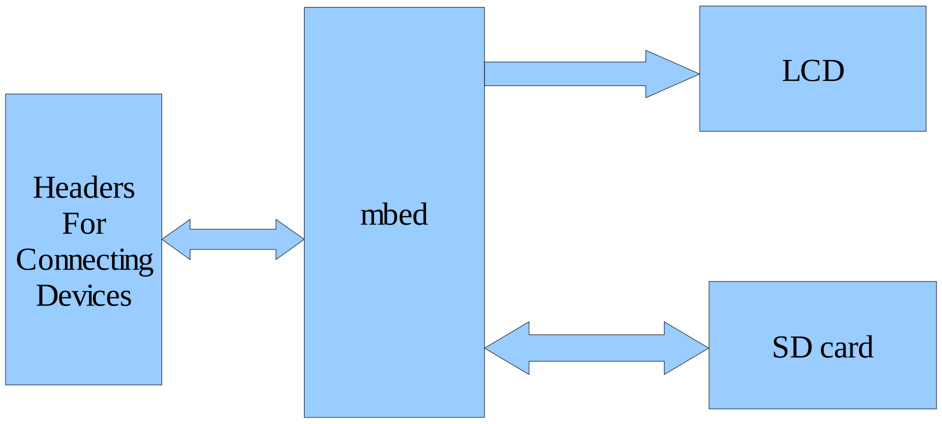
Reconfigurable Computing For Embedded System Devices

This project aims to be a quick prototyping unit, in spirit of the mbed platform itself. The setup consists of a mbed board, a LCD and few other components that demonstrate that how easy it is to develop a prototype using this platform. This project uses a scripting language called Forth for its scripting purpose. It was designed by Chrales H. Moore. Forth is an interpreted computer language. It uses post fix method for performing arithmetic's. Forth is a structured, stack based computer language. The Forth language acts as a mediator between the hardware and the user. User can interactively develop new functions (referred to as words in Forth jargon), test these new words and improve upon them. The other side of this interactive nature of the Forth language is that you can create prototypes on the hardware test them and get the feed back immediately. If the working of the new function is not as you expected, you can quickly create a new word or redefine the word to match your expectation and test it immediately without going through edit, compile and load loop. The decision to use Forth against other interpreted languages such as Python/Ruby/Perl etc... are
(a) Python/Ruby are very high level languages even though Pyhton for microcntrollers is available, it can be a daunting task to add a new function to the system for a new comer. Say if you wanted to add a function to access I2C bus, it can be tedious to do so in Python.
(b) Forth is a low level language and its constructs are very simple in nature. Forth requires you to divide your complex task to many small words, which must be tested and then combined to form the complete solution. This helps in localizing the errors.

The Hardware:

The hardware setup for this project is minimal. The hardware consists of a mbed module, an LCD, a touch screen controller, a SD card holder and a few other components. The LCD used is based on ILI9331 controller used in 8-bit mode and 16-bit color mode. The SD card is used to store Forth scripts which can be loaded and executed. You can have an “init” file on SD card which should contain name of the forth script to be executed as its first line entry. If the system finds an “init” file, then the file is opened and the first line of this file is interpreted as name of the Forth file to be executed and if the file named in the “init” was found then the file is loaded and executed. The “init” file is to be used if you require a Forth script to be executed at start up. The SD card access module uses sford's mbed SDFileSystem Library. All the port pins that are free are brought out on the board.

Block Diagram:



The Basic Language structure:

Let us start with a simple example. Enter the following in the Forth interpreter environment.

```
3 2 + . <Cr>
```

And hit enter, you should get the following output:

```
5
```

When you enter 3 and 2, it is pushed onto the stack which is called as the parameter stack. All operations in Forth language are performed on this stack. '+' is an operator, which takes Top of stack (TOS) element and Next on stack (NOS) as two parameters and leaves the result on top of stack. The dot (.) operator Pops out the TOS and displays it. The OK message is printed out by the interpreter to indicate that the interpreter is ready for the next input. Similarly, you can perform '-', '*' and '/' on stack. All arithmetic operations must be in Post Fix format .

Let us look at some more language tokens (referred to as words in Forth) available in Forth.

DUP : This word copies the top of stack and leaves TOS with another copy of what was on TOS.

```
1 2 3 4 <cr> OK
.S <cr>
1 2 3 4 OK
DUP <cr> OK
.S <cr>
1 2 3 4 4 OK
```

.S : is a word in forth which lists the contents of stack. It is non stack destructive call unlike the dot operator.

```
1 1 2 3 5 8 <cr> OK
.S <cr>
1 1 2 3 5 8 OK
```

SWAP: This word swaps the Top of stack with next on stack.

```
5 10 <cr> OK
.S <cr>
5 10 OK
SWAP <cr> OK
.S <cr>
10 5 OK
```

Words:

Every valid token that is not a number, is a word in Forth language. Even the operators, +, -, *, and / are all words. Forth maintains an internal data structure called dictionary in which these words are stored. You can define you're own word which then becomes a part of the dictionary. In other words, it becomes a part of the Forth Language. You can create a word by entering compile mode while in interpreter mode. You enter compile mode by entering colon operator (:) which in itself is a word which switches the forth interpreter to compiled mode. The definition of a word should end with semi colon (;). Now you can use your newly created word just as any other Forth word. If you want to define a word which computes the square of a number, you could enter the following in an interactive Forth session.

```
: square dup * . ; <cr>
```

This word when executed, would take TOS and duplicate it, multiply these two numbers and then pop out the result using dot operator. Below is square usage demonstration:

```
4 square <cr>
16 OK
```

Comments:

There are two types of comments supported in Forth. One is known as parenthesized comments and another one is drop line comments. Parenthesized comments start with '(' and end with ')'. Everything in between '(' and ')' are treated as comments.

```
1 2 3( push 1 2 and 3 onto stack ) <cr>
```

The second type of comment is known as drop line comment. These types of comment start with '\'. String that follows '\' is treated as comment until a new line is encountered.

```
: Square           \ define a word by name square
      dup          \ make a copy of TOS
      * ( multiply it with itself).\ pop out the result for display
;
```

Decisions:

This version of Forth supports IF...ELSE...THEN construct which is available only in compile mode. In this construct the interpreter checks if the TOS is true(!=0). If it is true, then it executes statements between 'IF' and 'ELSE'. If TOS is false(0) then the else part is executed and then whatever words follow 'THEN' are executed. In the processes, this construct consumes top of stack .

```
: test IF." True "ELSE." False "THEN ; <cr> OK
-1 test <cr>
True OK
0 test <cr>
False OK
4 test <cr>
True OK
```

." is another word which is used to print out strings. This word prints the string starting with ." and ending with ". Note that there is a space after ." this is required since Forth interprets statements word by word .

Looping:

Only BEGIN...UNTIL of ANS Forth is implemented in this version of Forth. General format of BEGIN...UNTIL is

BEGIN xxx f UNTIL

Where XXX is some words that are to be executed f is flag, if it is false the loop is executed else if the flag is true, control flows out of the loop and onto the next word in the definition. This construct can be used only in compile mode. As an example, let us consider following word which implements count down timer.

```
: CntDwn
      begin        \ print out the number
            dup .  \ subtract it with 1
            1 -    \ make another copy of the number for comparison
            dup    \ is it equal to zero ?
            0 =    \ if not loop again
      until
  drop             \ drop the left over
;
```

Variables:

You can create named variables in Forth language using the word 'VARIABLE'. If you want to create a variable with name 'temp' then:

```
variable temp
```

This will create a variable by name 'temp'. To store a variable in temp, you say

```
55 temp ! <CR>
```

The above statement when executed would store the number 55 into memory location allocated to temp. To retrieve value from temp, enter

```
temp @ <Cr>
```

This would leave top of stack with 55. In reality, '!' And '@' words are for memory operations. '!' is for write and '@' is for read/fetch from memory. When you entered 'temp' in command line, a memory location address was left in top of stack. Which was read by the '!' word and the next on stack was interpreted as the data and written to the memory location as indicated by 'temp' similarly, '@' word reads the data from memory location as indicated by the Top of the stack and leaves TOS with contents of that memory location.

Printing strings:

The word used for printing strings is ." It can only be used in compile mode for example:

```
: hello." Hello world " ; ( print hello world ) <Cr>
hello <Cr>
Hello world OK
```

Relational operators:

The relational operators supported in this version of Forth are

- = Tests if Top of stack and next on stack are equal. If equal, leaves TOS with TRUE value else leaves TOS with FALSE value.
- < Tests if next on stack is greater than top of stack. If so, leaves TOS with TRUE value else leaves FALSE value on TOS.
- > Tests if next on stack is greater than top of stack. If next on stack is greater than TOS TRUE value is left on TOS else FALSE value is left on TOS.
- <= Compares next on stack with TOS. If next on stack is less than or equal to TOS then TRUE value is left else FALSE value is left on TOS.
- >= Compares next on stack with TOS. If next on stack is greater than or equal to TOS then TRUE value is left else FALSE value is left on TOS.

As an example, consider the following:

```
: ?equal-to-zero 0 = If." Equal to zero"Else." Not equal to zero"Then ; <Cr>
OK
0 ?equal-to-zero <Cr>
Equal to zero OK
1 ?equal-to-zero <Cr>
Not equal to zero OK
```

In the word '?equal-to-zero', first zero is pushed onto stack. Then TOS and next on stack are compared to see if they are equal. If they are equal, TRUE(-1) is left on top of stack or else FALSE(0) is left on top of stack.

Logical operators:

The logical operators supported in this version of Forth are 'not', 'and', 'or', 'xor', '?bitset', '?bitclear'.

Not	Inverts the truth value of Top of stack
And	performs logical 'and' on TOS and NOS. Leaves the result on TOS
or	performs logical 'or' on TOS and NOS. Leaves the result on TOS
xor	performs logical 'Xor' on TOS and NOS. Leaves the result on TOS
?BitSet	Tests if a particular bit of next on stack number is set. TOS should

	have bit position to be tested. Leaves TOS with the TRUE value if bit was set or else leaves FALSE value.
?BitClear	Tests if a particular bit of next on stack is reset. TOS should have bit position to be tested. Leaves TOS with the TRUE value if bit was reset or else leaves FALSE value.

The following examples illustrates the usage of the logical operators.

```
1 2 3 and <Cr>
OK
.S <Cr>
1      2      OK
2 2 xor .s <Cr>
1      2      0  OK
4 2 ?bitset <Cr>
OK
.S
1      2      0      -1     OK
4 3 ?bitset <Cr>
OK
.S
1      2      0      -1     0      OK
4 0 ?bitclear .S <Cr>
1      2      0      -1     0      -1     OK
1      2      0      -1     0      -1     0      OK

: ?test-2-bit2 ?bitset If." Second bit is one "
                        Else." Second bit is zero "
                        Then ; <cr>
OK
4 ?test-2-bit <cr>
Second bit is one OK
6 ?test-2-bit <cr>
Second bit is one OK
2 ?test-2-bit <cr>
Second bit is zero OK
```

Now we will move onto words that are specific to the mbed and reconfigurable computing platform. Before we proceed, let us see what stack diagrams are and what they mean. Consider the following code snippet:

```
: SQUARED( a -- a*a ) DUP * ;

: SUM-OF-SQUARES
      SQUARED ( a b -- a*a+b*b )
      SWAP
      SQUARED
      +
;
```

Base:

You can switch between various base systems using the word “BASE”.

```
16 Base
```

Executing the above line causes the Forth to accept hexadecimal values. You can use this feature to access hardware registers by directly entering the hex values given in the data sheet. You can switch back to decimal mode from hexadecimal input mode by entering

```
0A Base
```

The comment (a -- a*a) is called stack notation which indicates what stack would look like before and after the word has been executed. In the above example, SQUARED has a stack comment (a -- a*a) which states that this word takes in 'a' and leaves stack with 'a*a'. In the example above, first we create a word called SQUARED which calculates square of a number. It expects the number whose square has to be calculated on Top of stack. This word takes top of stack, duplicates it and multiplies these two and leaves the result on TOS. SUM-OF-SQUARES uses this word to calculate the sum of squares of two numbers. As the stack diagram shows, this word expects two numbers on the stack. It takes each of these two numbers, calculates square and stores back the result in the stack. First SQUARED is performed on TOS then using SWAP, Next on stack is brought to top while top of stack becomes the new next on stack. SQUARED is performed on TOS number. Addition is performed on these modified stack elements to leave the result on top of stack

AddTicker

This word provides an interface for the Ticker object of the mbed platform. Even though mbed provides unlimited number of ticker objects, this project provides only one ticker. This word expects delay in seconds in stack and this word must be followed by the string that indicates the call back word.

```
delay AddTicker call_back_word_name
```

The following script prints “Hello world” every second.

```
: hello." Hello world " ;
1 AddTicker hello
```

DigitalOut

This word expects the port position on top of stack and the logic level to be output on that port pin next on stack. The port position can be any value between 8-30 (since p5-p7 are used by touch screen's SPI interface) both inclusive. Port values from 31-34 are special in the sense, they control the on board LEDs. Any values outside the above said range are invalid. If the port value argument is zero, then the corresponding port pin as mentioned in port position is set to logic level zero. Non-zero port values sets the pin to logic level '1'.

DigitalIn

This word reads digital logic level at a given port and pushes either 1 or 0 on to stack depending on the logic voltage at that pin.
Stack diagram: (port_position DigitalIn -- logic_level_at_that_port)
Valid port_position are from 8-30.

AnalogIn

This word reads analog voltage at given port pin and pushes the read data to the stack.
Stack diagram: (ch_index AnalogIn -- read_value)

ch_index can be 0-4 corresponding to p15-p20.

<u>AnalogOut</u>
Outputs a given analog value at p18 pin. The analog value to be out put must be pushed on the stack and AnalogOut must be called to output analog value that is represented digitally by the number pushed to stack.

<u>Fload:</u>
To load and execute a forth script stored in a SD card, you can use fload word. This word requires its arguments to be quoted. The file to be loaded for execution must be specified after the fload word.

```
fload "ha.fs"
```

This word uses SD card code interface by sford.

Next, we look into words that are related to the GUI part of this project. For all the GUI elements mentioned below, each of the GUI elements must be assigned with an unique id. Since id must be a number which uniquely identifies a GUI control, it must assigned carefully to each elements.

Let us now look at a simple Forth script which blinks on board LEDs of mbed every 1 second.

```
: pattern1 1 31 DigitalOut 0 32 DigitalOut 1 33 DigitalOut 0 34 DigitalOut ;
: pattern2 0 31 DigitalOut 1 32 DigitalOut 0 33 DigitalOut 1 34 DigitalOut ;

variable flag

0 flag !                    \ reset flag to start with

: blink-loop
      flag @                      \ read the flag value
      0 =                         \ is it equal to zero
      if
            pattern1              \ if so, time to trun on the LED1 and LED3
            -1 flag !             \ set the flag
      else
            pattern2              \ turn off LEDs LED1 and LED3 an turn on LED2 and LED4
            0 flag !
      then
;

1 addTicker blink-loop            \ start blinking
```

In the above code snippet, we first define two words which blink the on-board LEDs in two distinctive patterns and they are named as “pattern1” and “pattern2”. Within definitions of “pattern1” and “pattern2”, we make use of the word “DigitalOut” which as said earlier uses port index form 31-34 to refer to on board LEDs. Next we make use of a flag called variable which is used to switch between two different patterns. The word “blink-loop” actually handles the pattern switching. Next, we create a ticker with 1 second period which executes “blink-loop” approximately every 1 second.

<u>CRT_BTN</u>
This word adds a button to the GUI elements list.
Word call: (x y id crt_btn "button_name" "button_lbl" call_back_wrd)

x and y are the coordinates of the button and id is the id for the button. These parameters must be pushed onto stack before executing this word. This word must be followed by the button_name, button_lbl which will be displayed on the button and the last argument is the call back word which you want to execute when there is an event on the button. This word can only be used in interpreted mode. On success, this word leaves -1 (True) on the stack and 0 (False) in case of error.

CRT_P_BAR
This word creates a progress bar.
Word call: (x y id crt_p_bar "p_bar_name")
As in crt_btn, the first there parameters pushed on stack are x location, y location and id for the crt_p_bar the argument that follows crt_p_bar is the name of the progress bar. This word can only be used in interpreted mode. On success, this word leaves -1 (True) on the stack and 0 (False) in case of error.

SET_P_BAR
This word sets the level in progress bar to given volume.
Word call: (vol id set_p_bar)
For this word to work, you need to push the vol and id to the stack. Depending on the id you have entered, it looks out for the GUI element with given id. If it is a progress bar then using the vol argument, the volume is set. On success, this word leaves -1 (True) on the stack and 0 (False) in case of error.

CRT_ST_TXT
This word creates a static text GUI item.
Word call: (f_Color b_bolor x y id crt_sttxt "name" "text")
f_color and b_color are fore ground and back ground colors for the static text they are color coded as shown below. x and y define the location of the control. The id is a number which identifies the GUI item. "name" is the name of the static text and "text" is the text you want this control to display when created. This word can only be used in interpreted mode. On success, this word leaves -1 (True) on the stack and 0 (False) in case of error.

SET_ST_TXT
This word sets a new text for given static text control.
Word call : (id SET_ST_TXT ." New text ")
This word searches for GUI element with given id. If the element is found and if it is a static text, then the text of this particular static text element is changed to the new text and redrawn with the new text on the LCD. This word can only be used in compile mode.

SET_ST_CLR
This word sets fore ground and background color for a static text control with given id.
Word call: (f_color b_color id SET_ST_CLR)
This word searches for GUI element with given id. If the element is found and if it is a staic text, then the fore ground and back ground colors of this particular static control text is changed based on the values of the parameters f_color and b_color. These colors are color coded as shown in table below. On success, this word leaves -1 (True) on the stack and 0 (False) in case of error.

CRT_BMP

This word adds a bmp image to the GUI list. The source for this bit map control is the sd card.
Word call: (x y id crt_bmp "name" "file_name")
The x and y parameters on stack define the location for this control and the id uniquely identifies the this control. “name” argument assigns a name to the GUI control and the “file_name” indicates the location of the file on the sd card. Since the bit map is fetched from the SD card and drawn, execution of this word takes considerable amount of time. This word can only be used in interpreted mode. On success, this word leaves -1 (True) on the stack and 0 (False) in case of error.

SET_BMP
This word can be used to change the image being displayed by the bmp control and this word can be used only in compiled mode.
Word call: (id set_bmp ." new_file_location ")
This word sets new bit map for given bit map control (as indicated by the id) and redraws the image. On success, this word leaves -1 (True) on the stack and 0 (False) in case of error.

SHOW
Once all the GUI items are created, to display it on the LCD, you have to use the 'SHOW' word. This word displays all the GUI elements added to the GUI list using the words discussed above.
Word call: (show)
The GUI elements are drawn in order of their creation. This word only displays the GUI controls you need use 'ml' word to allow the system to handle and dispatch events.

ML
This word is responsible for handling and dispatching events to the call back words.
Word call: (ml)
'ml' stands for main loop. Execution stays in main loop until 'exit_ml' is executed.

EXIT_ML
This word causes the execution to come out of main loop (ml).
Word call: (exit_ml)
This can be used as a part of definition of a word which happens to be call back word for a button which quits the application. This word has significance only when ml is under execution.

CLR_GUI:
This word clears all the GUI elements and clears the LCD.
Word call : (clr_gui)

Following script illustrates the usage of some of the words that are related to GUI part of this project.

```
\ This is a simple script which prints "You clicked on hello " on serial console
\ when you click on "hello" button on LCD

: hello            \ this word prints "You clicked on hello" and a new line
      ." You clicked on hello  " Cr
;
\ draw a "wall paper" :)
0 0 5 crt_bmp "hi" "test.bmp"
Drop
\ create a button and register call back
```

```
30 40 1 crt_btn "hello" "Hello" hello
Drop
\ word to exit the main loop
: exit_loop exit_ml ;
30 150 2 crt_btn "exit" "Exit" exit_loop
Drop
show                 \ show the GUI items
ml                   \ enter the main loop
clr_gui
```

The above script first defines a word called “Hello” which simply prints out the message “You clicked on hello” on the serial console using the string display operator, ." . Then we add a bitmap control to the GUI list. The data for displaying image is fetched from the file “test.bmp” located on the SD card as mentioned in the parameter passed to the word “crt_bmp”. Since we are not checking for the return values from the word crt_bmp or crt_btn, we “Drop” the return value from stack. Then we create a button with name “hello”, caption/text “Hello” and call back word “hello”. Then we create a word called “exit_loop” which when executed, causes the execution to flow out of main loop (ML) since “exit_loop” has “exit_ml” as its part of definition. Next we create another button with label/caption “Exit” and “exit_ml” as its call back word. Hence when you click on “Exit”, the execution would floe out of “ml” and execute “clr_gui”. Next we call show to display the GUI items. Then we enter the main event handling loop, “ml”. When “exit_loop” gets executed by means of clicking on “Exit” button, the execution would flow out of the main loop and “clr_gui” word would get executed which clears all the GUI elements.

Color Table:

0	Black
1	White
2	Red
3	Green
4	Blue
5	Yellow
6	Magenta
7	Orange
8	Cyan
9	Gray
10	Silver
11	Gold

Extending the Forth Interpreter:

It is fairly simple to extend the Forth interpreter. We will now add Forth interface for accessing the SPI bus on the mbed platform. As we have seen earlier, In Forth, stack is used to pass parameters to the words. We are going to write a 'C' function which accepts arguments from the stack and then takes necessary steps to access the SPI bus. All 'C' call back functions are defined in the file “forthFunctions.cpp”. We will be adding our newly created function to this file. The SPI block that we are going to use is the one whose signals are on p11, p12 and p13. The CS signal will be p14. Before we add the function, we need to decide on what arguments are needed and in what order we need to push the data onto the stack. We will configure this word to accept parameters starting from top of stack, frequency, mode, bits, number of bytes to be written followed by data that is to be written over SPI bus as many in number as indicated by numbers of bytes to be written argument. All call back functions are of type void func(void). We will name the newly created function as `SpiWrite()`. The in spi write function, we create an object of type SPI with instance name spi and declare cs pin as p14.

```
    SPI spi(p11, p12, p13);
    DigitalOut cs(p14);
```

Next we define variables that would be required by the function.

```
    int cond, i, data, bits, no_bytes, mode, freq;
```

Access to the parameter stack is provided by the functions in “stack.c”. PushDs() and PopDs() are the two functions which perform push and pop operations on the data stack. Both PushDs() and PopDs() take parameter which is a pointer to the integer type. On exit, from these functions, the parameter will have the result of operation performed on the stack. If the operation was successful then this parameter will have value STACK_ERR_SUCCESS. If the stack is empty (During pop operation), this parameter will have value STACK_ERR_EMPTY. If the stack is full (push operation), this parameter will have value STACK_ERR_FULL. We are using cond(condition) variable during push and pop operations. Since frequency is in top of stack, we pop out the frequency parameter first.

```
    freq = PopDs(&cond);
    if (cond == STACK_ERR_EMPTY) {
        printf (ERR_TABLE[INSUFF_PARAMS]);
        return;
    }
```

While fetching the data for the frequency parameter, if we find that there is no data in stack that can be used as parameter which sets the frequency, we print out an error message and bail out. You can either use the messages from “ERR_TABLE” or you can print out your own error messages.
Similarly, rest of the parameters can be fetched from the stack and you can catch errors on the way and take appropriate actions.

```
    mode = PopDs(&cond);
    bits = PopDs(&cond);
    no_bytes = PopDs(&cond);
```

We now set the attributes for the SPI communication.

```
    spi.format(bits, mode);
    spi.frequency(freq);
```

Using an index variable and the no_bytes we loop until all the data that are to be transmitted are retrieved from the stack and transmitted on the SPI bus.

```
    cs = 0;
    for (i=0; i<no_bytes; i++) {
        data = PopDs(&cond);
            if (cond == STACK_ERR_EMPTY) {
                cs = 1;
                return ;          // no data in the stack to write
        }
        spi.write(data);
    }

    cs = 1;               // deselct the spi device
```

Now we need to register this call back function with the Forth interpreter against a string which when encountered while interpreting would call our newly created function. Scroll to the top of the “forthFunctions.cpp” file there you will find a function called `init_dictionary()` this function registers words such as Dup, +, - etc that are available at start up. To the list of words already defined in this function, we are going to add a new word. This is done by using the function which `AddDicEntry()` registers the words with the dictionary. The first parameter to this function is a string which identifies our newly created word we will name this word as “SPIWRITE”. Note that Forth is insensitive to case and all word names that you register using `AddDicEntry()` must be in caps. Next parameter is the FORTH_FLAG this flag defines the nature of the word for most of the words defined using 'C' functions, FORTH_WORD_INBUILT will suffice. If you want your word to be included only in compiled mode, then you can use FORTH_COMPILE_ONLY. You can combine these flags to create desired effect using or(|) operator. Care must be taken so as to not conflict the nature of a word. For example, do not specify a word both FORTH_WORD_INBUILT and FORTH_WORD_USER. Finally, add a function prototype for newly created function in the “forthFunctions.h” file.

The Forth scripting language for reconfigurable computing is implemented mainly in the 'C' language. This Forth is inspired by jonesforth. It is an Forth implementation written in x86 assembly language. This version of Forth in many places is mainly conversion of this assembly code to C code.
The Forth is implemented in following 5 files:
CoreForth.c
stack.c
interprter.c
forthFunctions.cpp
CoreForth.c

<u>CoreForth.c</u>
This file contains foundation for implementing the Forth. Function called AddDicEntry() which makes entry to a dictionary is defined in this file. Dictionary is a linked list whose each node is a word and each node contains all the information required for executing a word. This function allocates enough memory to each node to hold the addresses of compiled words. This file also contains a function called Find() which looks for a word int the dictionary by name and if it finds one, it returns the address of the node within which the word was found.

stack.c
The operations on parameter stack (DatStack) and return stack (RetStack) such as push and pop are implemented in this file.

interprter.c
Core interpreter is implemented in this file. The Forth virtual machine is implemented here. Forth is interpreted word by word. Hence this file contains a function called Word() which extracts a word from input buffer. Forth maintains an internal flag called CompileMode if it is true a word is compiled if it is false, the word is executed immediately. This file has a pointer called CurrentAddr which acts as the instruction pointer. This file also has function that converts number to various base system and pushes it to the stack.

forthFunctions.cpp
All the call back functions are defined here. Also the init_dictionary() is defined here which sets up the dictionary with some basic Forth words which can be used by other words. If you want to extend this platform you need to add the function here to this file and add a reference to this c/c++ function in the init_dictionary() function using the function AddDicEntry(). The first parameter (name) for this function is the string which identifies the word. The second parameter (ForthFlags) is the flag which indicates nature of the word. It can be any one of the following

FORTH_WORD_INBUILT	Indicates that this particular word is inbuilt, functionality defined by c/c++ function
FORTH_WORD_IMED	Flag for indicating immediate word, used in compiled mode. Used to execute words when in compile mode
FORTH_WORD_USER	Flag for indicating user defined word
FORTH_COMPILE_ONLY	Indicates that word can be executed only in compile mode
FORTH_WORD_VAR	Word variable to which memory has been allocated

The next parameter (func) registers the c/c++ call back function against given word. These c/c++ functions must be of type void func_name(void). The next parameter (CodeList) is an array which should contain compiled addresses of words which are part of definition of word under consideration. This is only used to register user defined words at run time. The last argument is length of the compiled word array which has significance only if CodeList is non NULL.

CoreForth.c file has the functions which form the basis for Forth. The Function called AddDicEntry, makes entries to the dictionary, which is a linked list of following c structure:

```
struct Node
{
        char WrdName[FORTH_NAMEMAX];     /**< Holds the word name */
        int flag;                        /**< To hold various conditions such as
                                              FORTH_WORD,FORTH_INBUILT etc */
        int *code;                       /**< Array to hold the address of various code word
                                          */
        NodePtr next;                    /**< To point to next entry in the dictionary */
        int WrdLen;                      /**< length of word name */
```

```
      func_ptr func;                    /**< Function pointer to inbuilt function */
};
```

As you can see this struct has all the information that is required to execute words. func_ptr is a typedef for function pointer of type void func(void). The AddDicEntry() function adds entries to this linked list with proper flags set and function pointer added, if any. The function prototype of AddDicEntry() is:

```
AddDicEntry ( char      *name,
              int       ForthFlags,
              func_ptr  func,
              int       *CodeList,
              int       len
)
```

“name” is the string which holds the name of the array. “ForthFlags” set various attributes of
the word that is being entered into the dictionary. If this flag is set as FORTH_WORD_INBUILT then, it is assumed that the function is inbuilt and it expects a function pointer in variable “func”. If the word is user defined, then “CodeList” should have list of addresses of words that is to be associated with the name and “len” should hold the length of the array CodeList. Next important function which is used in interpreter as well as compile mode is the Find()
function. The prototype of the function is :

```
int Find(char* name, int* addr);
```

'name' is a string which has the name you are looking for in the dictionary. If the word is found, the address of the entry is stored in 'addr' argument. If the word is not found, appropriate error code is returned.

Stack.c implements functions operating on two stacks, data stack and return stack. Data stack is where all the real things happen. The return stack is used to hold the return addresses when words are being executed.

Next important file in is the interpreter.c This file has a function called Word() which fetches the next word form the input stream. This file also contains one of the most complicated function , the Interpret(). The function prototype for Interpret()
is:

```
int Interpret(void);
```

This function is responsible for executing the words. This function can be viewed as control unit of the Forth machine. It takes no argument. It returns error values depending upon the state of the interpreter. It fetches words from input buffer using function Word(). Forth maintains an internal flag called CompileMode, if this flag is true, then the word is compiled. If this flag is false then words are immediately executed. When you enter anything at the Forth terminal, the interpreter first tries to parse the input as a word. If it finds this word, it executes it if it does not find the word, it tries to parse the input as a number if it succeeds, then the number is pushed to stack else it throws an error message saying that it did not recognize the word. When in Interpreted mode (i.e. CompileMode = FALSE), Interpreter() function Find()s the word, when found, checks the flag. If the function is inbuilt, the function pointer is used to call the function which is associated with a word. In other word, it is a call back function for a word. This technique is similar to function call back or signal handling used in some GUI toolkits. If the current word being executed is not an inbuilt function, it looks in CodeList[] of the word and tries to execute it sequentially. If a word has another word as part of its definition, it stores the address of current word in return stack and jumps to the new memory address and tries to execute it. If it is not an inbuilt function, this process continues until the word cannot to be broken down or in other words until an inbuilt word is found which is attached with a c/c++ function call back. Once the function is executed, the address is poped out from return stack and execution continues. In compile mode (i.e. CompileMode = TRUE), the interpreter() finds each word and records their addresses into an array and when it sees CompileMode = FLASE (set FALSE by ';' word), it adds this entry into the dictionary. You can execute certain functions at compile time as well. This technique is used for calculating offsets in branching instructions also, this technique is used in various other places. This is accomplished by setting the flag FORTH_WORD_IMED while making an entry to the dictionary.

Next file in the list is ForthFunctions.cpp This file contains all the inbuilt and immediate word functions defined here. Also this file contains init_dictionary() which initializes the dictionary. The procedure to add a new inbuilt word is to, first define your function in ForthFunctions.cpp and then declare a prototype for this function in ForthFunction.h. Next, in init_dictionary() function, make an entry to the dictionary of newly created word with appropriate name using AddDicEntry() function. Set the attributes for the new word as desired. The attribute FORTH_INBUILT would be ideal for most of the words. If you want your word to be part of only a compiled word then use flag FORTH_COMPILE_ONLY. This file is easily the largest file in this project. The procedure for adding immediate call back functions is similar only the difference being that you need to set FORTH_WORD_IMED flag while entering this word into the dictionary. You can combine flags to set multiple attribute in a sensible way.

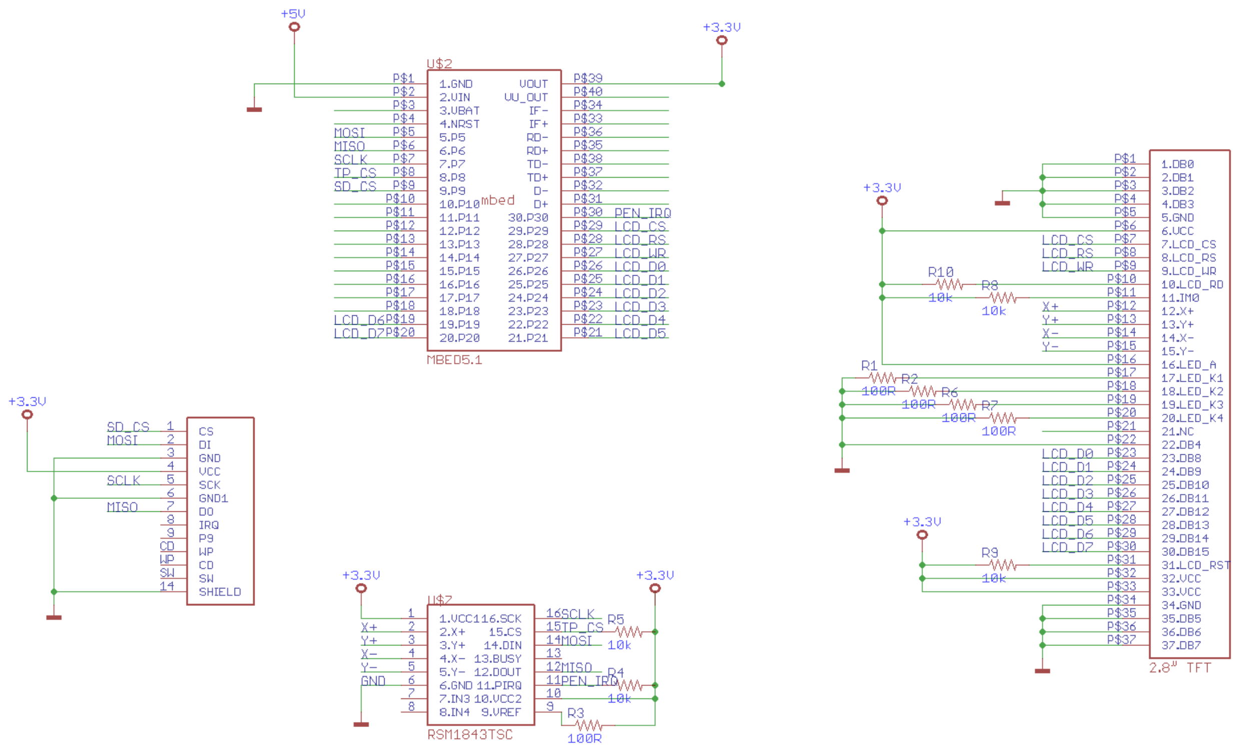
GUI implementation:

The files which implement the GUI functionality of the project can be found in folder name “GUI”. In the file “lcd.c” you will find the code that drives the LCD and procedure for drawing some basic drawing primitives such as lines and rectangles. The file “gui_controls.c” make use of functions provided by “lcd.c” to draw GUI objects such as buttons, progress bar and static text controls. Functions used in this file make use of structures that are defined in “gui_controls.h”. This file contains structure that define GUI controls and a structure called “gui_elem”

```c
struct gui_elem
{
  uint32_t id;                      /*< Id of this element */
  char name[MAX];                   /*< name of the GUI element */
  uint32_t x;                       /*< x coordinate of the element */
  uint32_t y;                       /*< y coordinate of the element */
  uint32_t  width;                  /*< Width of the element */
  uint32_t  height;                 /*< Height of the element */
  enum gui_elem_type gui_type;      /*< Type of the GUI element */
  void* gui_struct;                 /*< info specific to this GUI type */
  gui_elem_ptr next;                /*< To hold the reference to the next type */
};
```

This structure defines all the parameters such as id, name, x, y, width, height that are common to all GUI elements. The enum 'gui_type' indicates type of GUI element. The gui_struct pointer holds pointer to information that are specific to a particular GUI element type such as text. Using 'gui_type' member we can type cast the 'gui_struct' correctly to get the information that are specific to GUI element type. The 'next' pointer points to the next element in the GUI list. The file “gui.c” has functions that manage the GUI elements in much easier manner by using functions provided by “gui_controls.c”. Functions related to event handling are defined in file called “ts.c”. The x,y coordinates are obtained from the touch screen controller over SPI bus and this information is pushed to buffer which is named as “event_table”. The function named “Dispatcher()” define in “gui.c” extracts the information from “event_table” and decides whether or not to dispatch the event on a button (can be extended to any other GUI element). If you look at “gui_controls.c” file you will realize that none of the GUI constructs in this project use bit map images for GUI controls. This was done to reduce the complexity and moreover, GUI part of this application is meant to demonstrate how the Forth+mbed platform can be adopted according to the project requirements.

Schematic:


+5V
+3.3V
U$2
MBED5.1
mbed
MOSI
MISO
SCLK
TP_CS
SD_CS
LCD_D6
LCD_D7
PEN_IRQ
LCD_CS
LCD_RS
LCD_WR
LCD_D0
LCD_D1
LCD_D2
LCD_D3
LCD_D4
LCD_D5
2.8" TFT
RSM1843TSC
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
10k
100R
X+
Y+
X-
Y-
GND
CS
DI
VCC
SCK
DO
IRQ
SHIELD