Embedded Systems

Ascii Translator

Università degli studi di Palermo 2019-2020

Francesco Paolo Castiglione

Prof. Daniele Peri

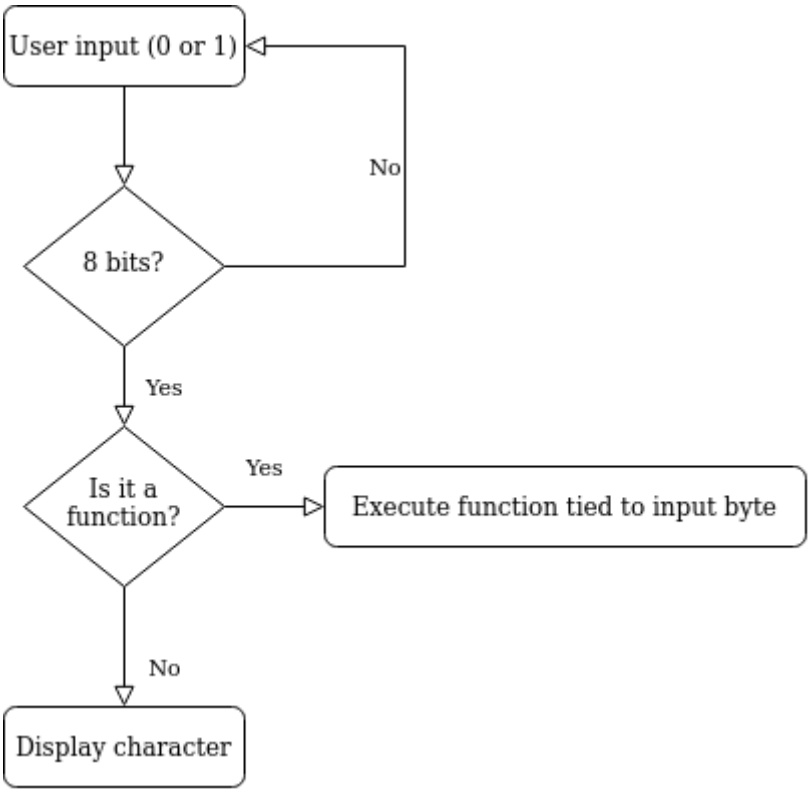
Table of contents

Introduction

The project is an ASCII translator. User input is retrieved using two push buttons that provide respectively a 0 and a 1 . After 8 bits have been provided , if the input corresponds to a display function then the LCD will act accordingly. Otherwise, the corresponding ASCII character will be shown on the display.
The user can choose between the following display functions:

- Clear the display
- Left shift the cursor
- Right shift the cursor
- Go to the beginning of the first line (without clearing the display)
- Go to the end of the first line (without clearing the display)
- Go to the beginning of the second line (without clearing the display)
- Go to the end of the second line (without clearing the display)

All the ASCII characters that the LCD datasheet illustrates can be shown by providing the corresponding value through the push buttons.
The logical overview is shown below with a flowchart:



Hardware

The following hardware is used in the project:

- Raspberry Pi 3b +
- Breadboard
- FTDI FT232RL USB (Serial UART)
- PCF8574AT I/O Expander I2C
- 16x02 QAPASS LCD
- 2 Qteatak push buttons
- 10 jumper wires

Raspberry Pi 3b +

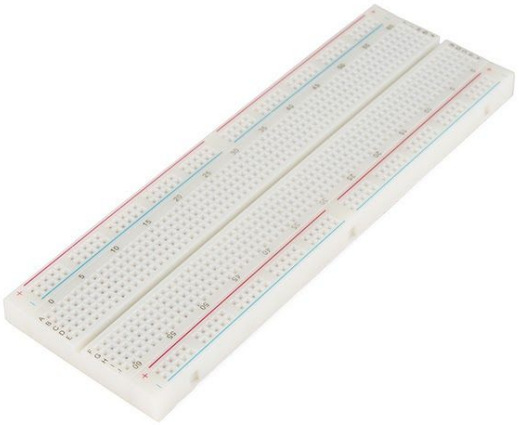
The hardware target for the project is the Raspberry Pi 3b+, which boasts the following specs:

- 1.4GHz 64-bit quad-core processor

- dual-band wireless LAN
- Bluetooth 4.2/BLE
- faster Ethernet (compared to the previous iteration)
- Power-over-Ethernet support (with separate PoE HAT)

Breadboard

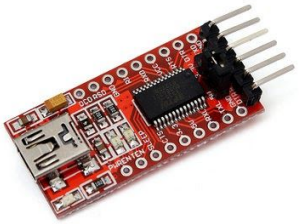
A breadboard is a solderless device used for temporary prototype with electronics and test circuit designs. Most electronic components in electronic circuits can be interconnected by inserting their leads or terminals into the holes and then making connections through wires where appropriate. The breadboard has strips of metal underneath the board that connect the holes on the top of the board. The top and bottom rows of holes are connected horizontally and split in the middle while the remaining holes are connected vertically. In the project two push buttons with two pins are placed in a vertical row (+) and current is provided through an electric wire connected to a 3,3V RPi pin.



FTDI FT232RL

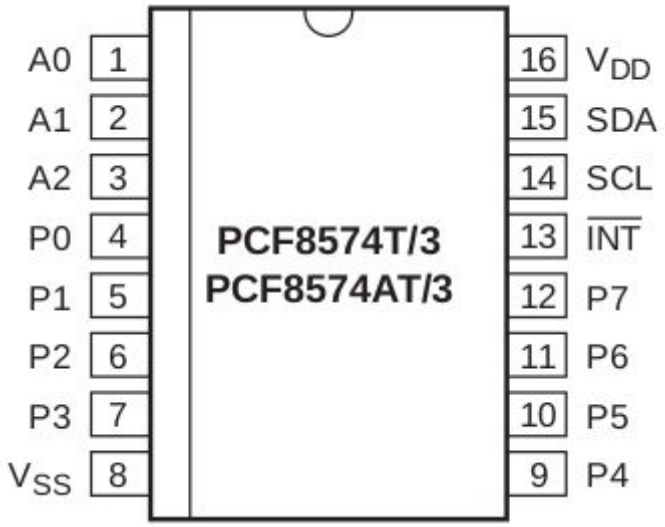
The FT232R is a USB to serial UART interface with optional clock generator output.
The FTDI module is connected to the computer through a USB port and to the RPi in the following fashion:

- FTDI-RX to RPi-GPIO14 (TX)
- FTDI-TX to RPi-GPIO15 (RX)
- FTDI-Ground to RPi-GND



PCF8574AT I/O Expander I2C

This 8-bit input/output(I/O) expander for the two-line bidirectional bus (I2C) is designed for 2.5-V to 6-V V_{cc} operation. The PCF8574AT provides general-purpose remote I/O expansion for most microcontroller families by way of the I2C interface. The expander is soldered to the LCD pins.
SDA(serial Data) is connected to GPIO3 and SCL(serial clock) to GPIO4

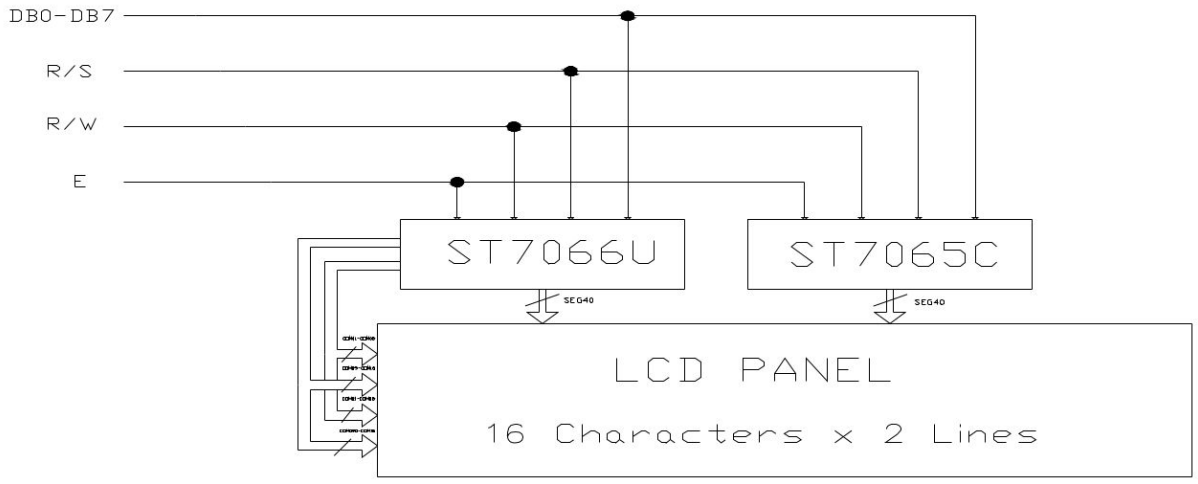


QAPASS LCD 16x02

The display boasts the following features:

- Display Mode: STN, BLUB
- Display Format: 16 Character x 2 Line
- Viewing Direction: 6 O’Clock
- Input Data: 4-Bits or 8-Bits interface available
- Display Font : 5 x 8 Dots
- Power Supply : Single Power Supply (5V±10%)
- Driving Scheme : 1/16Duty, 1/5Bias
- BACKLIGHT(SIDE):LED(WHITE)

Block diagram:



Push buttons

Qteatak two pins 6x6x5 mm tactile push buttons. These buttons boast a mechanical life of 100,000 times so they are more than adequate for the project. The rated load is 12V DC so we can expect no issues with the RPi 3,3V GPIOs. In the project we make use of two such buttons.

Environment

---

pijFORTHos

A bare-metal operating system for Raspberry Pi, based on Jonesforth-ARM. *x86 JonesForth* is a Linux-hosted FORTH presented in a Literate Programming style by Richard W.M. Jones .

Each dialect of FORTH has its own definitions. Development in FORTH involves extending the vocabulary with words specific to the application. Most predefined words in *pijFORTHos* follow traditional standards and conventions. In the file se-ans.f (Daniele Peri, Università di Palermo) some words are added for ANSI compliance (they are needed in the input.f module) .

Minicom, picocom and ASCII-XFR

Minicom is a terminal emulator software for Unix-like operating systems. It is commonly used when setting up a remote serial console.

Picocom is shares with minicom many similarities . It was designed as a simple, manual, modem configuration, testing, and debugging tool.

ASCII-XFR is used to send the source file to the Raspberry because it allows a delay between each character and line sent, a much needed feature since if avoids overrun errors and lost characters (the UART is asynchronous) .

Picocom is launched on the development machine through the command --b 9600 /dev/tyyUSB0 --imap delbs -s "ascii-xfr -sv -l100 -c10" :

- --b 9600: 9600 bit/s bit rate.
- --imap delbs: allows the use of backspace to delete a character.
- -s "ascii-xfr -sv -l100 -c10": specifies ascii-xfr as the external command to use for transmitting files.
    - -sv: verbose send mode.
    - -l100: sets a 100 milliseconds delay after each line is sent, this usually gives enough time to run a command and be ready for the next line in time.
    - -c10: waits 10 milliseconds between each character sent.

Script

Since it is not possible to automatically load the source code from storage at startup, the code has to be sent via serial connection. A simple script is developed to merge the various project modules and remove brackets and their content.

```
#!/bin/bash

cd Documents/Embedded\ Systems/

cat utils.f lcd.f se-ans.f input.f |                    #source files

sed '/^[[:blank:]]*(/d;s/(.*//'  > /home/frank/merged_src.f   #merged source
```

Code structure

We now dive into a more extensive analysis of the forth code. The code is divided into three modules: utils.f, lcd.f, se-ans.f. and input.f.
To develop the project a bottom-up methodology was followed, that is the most general words were actually defined after a testing phase. Such a way of developing the project appears to be more natural and better suited to the forth programming language.
The code presented here is therefore the final result of the testing and redesign process.

Utils.f

At the very beginning of the module we define constants for all the needed registers and for added clarity to the code (**INPUT,OUTPUT,UP,DOWN**).

It is deemed useful to first define some words that allow us to wait for some specific amount of time. The wait word is therefore defined. It makes use of the system timer on the RPi, but we really only need the lower 32 bits (SYSTIMER_CL0 register).

```
: CURRENT_TIME                    ( -- time )
   SYSTIMER_CLO @ ;               \ We only use the lower 32 bits of the system timer

: WAIT                            ( microseconds -- )
   CURRENT_TIME                   \ We define a word that requires the time to wait in microseconds on the stack
   BEGIN
        DUP CURRENT_TIME          \ Now on the stack: microseconds start_time start_time current_time
        - ABS                     \ Now on the stack: microseconds start_time elapsed_time
```

```
   >R OVER R>  <=
   UNTIL
   DROP DROP ;
```

In this module the most general words are defined. A noteworthy word is **MASK_REGISTER**

```
: MASK_REGISTER                    ( address starting_bit_position
bits_num -- masked_register )
   MASK 1 -                        \ Sets a bits_num of bits to 1
   SHIFT INVERT                    \ Shifts those bits by
starting_bit_position and then inverts the bits
   SWAP @ AND ;                    \ Performs a logic AND between the
current content of the register and the mask
```

This word provides the masked register, that is the register content remains the same except for bits_num of bits that start from starting_bit_position which are set to 0.
The end goal here is to avoid rewriting the same word every time we need to set a GPIO as input or output or as an alternate function or even change just specific bits in a register. We need a more general set of words that will later allow us to write cleaner and more maintainable code.
We later define **SET_REGISTER**

```
: SET_REGISTER                     ( value starting_bit_position
bits_num address -- )
   DUP >R ROT ROT MASK_REGISTER    \ We define a general
word for any field of any register
   OR R> ! ;                       \ We then define more specific words for
ease of use
```

The reasoning behind this word is the same as the preceding. For example, with **SET_BSC1** and **SET_GPFEN0** we define two words that allow us to work with any field of the chosen register (BSC1 and GPFEN0 respectively). The value that we need to put on the stack is simply the hex value of the register related only to the bits we wish to change since **MASK_REGISTER** handles the masking logic

---

```
: SET_BSC1                    ( value starting_bit_position bits_num
-- )
   BSC1_BASE SET_REGISTER ;          \ We define a general word for
any field of the BSC1_BASE register

: SET_GPFEN0                  ( value starting_bit_position
bits_num -- )
   GPFEN0 SET_REGISTER ;             \ We define a general word for
any field of the GPFEN0 register
```

The result is cleaner and simpler specific words that define our actions. If we need to set other bits we can now easily define a new word.
As we can see in **START_TRANSFER**, we only need to provide the value , the starting bit and the number of bits

---

```
: START_TRANSFER                    ( -- )
   80 7 1 SET_BSC1    ;             \ Starts a new transfer
```

If we want to set the GPIO function it can be useful to notice that we can easily find the register address from the pin number. For this very reason the **GPFSEL_ADDRESS** word is defined

```
: GPFSEL_ADDRESS                    ( pin_number -- GPFSEL_address )
   A / 4 *                          \ We put the GPIO pin number on the stack and get
   GPIO_BASE  + ;                   \ the address of the appropriate GPFSEL function select register
```

By putting the pin_number on the stack we get the corresponding register address. We can also notice that it is possible,given the pin_number,to get the starting_bit_position of the three bits to change to set a GPIO function, hence we define

```
: GPFSEL_STARTING_BIT_POSITION      ( pin_number -- starting_bit_position )
   A MOD 3 * ;                      \ starting_bit_position for the appropriate FSEL field in the appropriate GPFSEL register
```

We can now define a general word that, given value and pin_number, sets the GPIO function, **SET_GPFSEL**

```
: SET_GPFSEL                        ( value pin_number -- )
   DUP GPFSEL_ADDRESS DUP >R SWAP   \ We define a general word that only requires the pin_number and value to set the appropriate GPFSEL register
   GPFSEL_STARTING_BIT_POSITION     \ We then define more specific words for ease of use
   3 MASK_REGISTER OR R> ! ;
```

For convenience, we then define specific words for the GPIOs we are interested in for the project. That being said, all we would need to define specific words for other GPIOs is the value of the appropriate register related only to the bits we wish to change, because the pin_number is all we need to find the register address and the starting_bit_position.

When we set GPIOs as input we need to make sure to set the internal pull resistor since if we don't the pin will capture electromagnetic noise and not have a set logic state. The BCM2837 datasheet illustrates exactly how to set the pull up/down resistors:

1. Write to GPPUD to set the required control signal (i.e. Pull-up or Pull-Down or neither to remove the current Pull-up/down)

2. Wait for 150 cycles – this provides the required set-up time for the control signal

3. Write to GPPUDCLK0/1 to clock the control signal into the GPIO pads you wish to modify

4. Wait for 150 cycles – this provides the required hold time for the control signal

5. Write to GPPUD to remove the control signal

6. Write to GPPUDCLK0/1 to remove the clock

Hence, we can define the **SET_PUD** word

```
: SET_PUD                    ( GPPUDCLK0_MASK UP/DOWN --
)

   GPPUD !

   150 DELAY
```

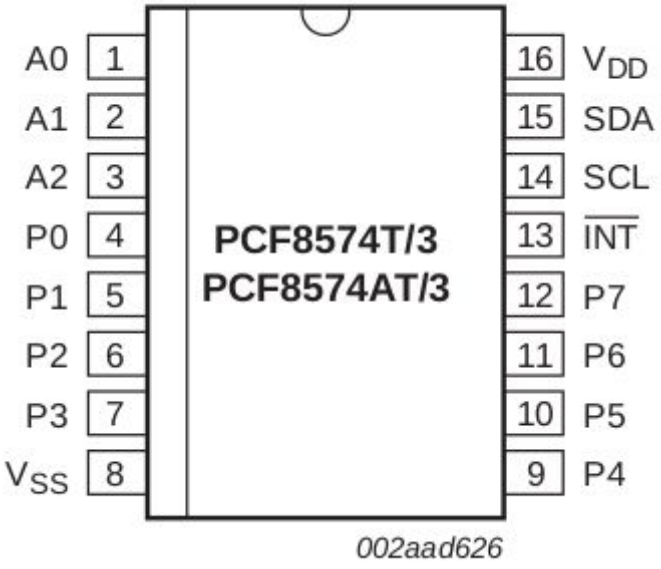
```
    DUP INVERT SWAP
    GPPUDCLK0 @ OR GPPUDCLK0 !
    150 DELAY
    0 GPPUD !
    GPPUDCLK0 @ AND GPPUDCLK0 ! ;
```

We can then define a more specific **SET_PULL** word. At the beginning of the module we had defined the UP and DOWN constants to make the code more readable. Now we only need to put UP/DOWN and the pin_number on the stack to set the UP/DOWN pull for the related GPIO.

```
: SET_PULL                    ( UP/DOWN pin_number -- )
    GPPUDCLK0 SWAP 1 MASK_REGISTER          \ We define a general word that works with any pin_number
    SWAP SET_PUD ;
```

Lcd.f

This module handles the I2C operations needed to actually send data to the LCD.
The two-wire bidirectional I2C bus allows for communication between a master (or multiple masters) and a slave (or multiple slaves).
In our instance, our master is the RPi, and more specifically the Broadcom Serial Controller (BSC1), and our slave is the PCF8574AT I/O Expander, soldered to the LCD. The LCD makes use of the HD44780 controller.
In order to send data to the LCD, we must then not merely follow the I2C protocol but also make sure to send the data correctly to the I/O expander. The following image shows the corresponding pin assignment between the expander and the display.



No.	Symbol
1	Vss
2	Vdd
3	V0
4	RS
5	R/W
6	E
7	DB0
8	DB1
9	DB2
10	DB3
11	DB4
12	DB5
13	DB6
14	DB7
15	BLA
16	BLK

A **SEND** word is defined. It is written following the RPi datasheet and the I2C protocol. The BSC FIFO is first cleared. After setting the length of data to send to the FIFO (1 byte), we put data into the FIFO, set the slave address, set the write bit, start the transfer, and wait until DONE bit in the STATUS register is set.

```
: SEND                          ( data -- )
   CLEAR_FIFO                   \ It clears the FIFO
   1 SET_DLEN                   \ We will write 1 byte into the FIFO
   WRITE_FIFO                   \ We put data in the FIFO
   SET_SLAVE                \ We set the slave address
   WRITE_TRANSFER                   \ We want to write
   START_TRANSFER                   \ It starts the transfer
   CHECK_STATUS ;                   \ It checks that the transfer is
done
```

In order to use the display functions , we now need to take a look at the LCD datasheet. The LCD makes use of the HD44780 controller developed by Hitachi.
The LCD's interface consists of 11 bits. 8 of these carry data (DB0-DB7), the remaining 3 are control bits (RS, R/W, E).
The RS (data/instruction select) bit tells the LCD whether we are going to send an instruction or a piece of data.
The R/W (read/write) bit tells the LCD whether we wish to read from or to write to RAM.
The E (enable) bit tells the LCD when it should read the data lines.
When the display is first powered on, it operates in 8 bit mode,so we first wish to set it to 4 bit mode. The **FUNCTION_SET** word is then defined.

```
: FUNCTION_SET                    ( -- )
   2C SEND                        \ D7-D6-D5-D4=0010 (MSB) BL=1
EN=1 RW=0 RS=0 Initialize Lcd in 4-bit mode 2C SEND
   88 SEND ;                      \ D3-D2-D1-D0=1010 (LSB) BL=1 EN=0
RW=0 RS=0
```

From now on, we can operate in 4 bit mode, which means that to send an instruction we will need to **SEND** 4 times, twice for the most significant bits and twice for the least significant bits.

For this very reason we define the words **4LSB** , **4MSB** and **SEND_CMD**. We simply wish to send the 4 most significant bits first, once with the EN bit enabled and once with the EN bit set to 0. Defining a **SEND_CMD** word makes the code a lot more readable and also allows us to then just work with the instruction codes from the LCD datasheet without worrying about the I/O expander. For the word **SEND_CHAR** the same reasoning is applied, the only difference is the RS bit (set to 1).

```
: 4LSB                            ( -- )
   F AND 4 LSHIFT ;               \ These are the 4 least significant bits

: 4MSB                            ( -- )
   F0 AND ;                       \ These are the 4 most significant bits

: SEND_CMD                        ( cmd -- )
   DUP 4MSB                       \ We operate in 4 bit mode so we have to
send the 4MSB and 4LSB twice
   DUP C +    SEND                \ D7-D6-D5-D4=MSB BL=1 EN=1
RW=0 RS=0 ( 0xC )
   8 + SEND                       \ D7-D6-D5-D4=MSB BL=1 EN=0 RW=0
RS=0 ( 0x8 )
   DUP 4LSB
```

```
DUP C + SEND
8 + SEND
DROP ;
```

The HD44780 type controller chip is used with a wide variety of Liquid Crystal Displays. These LCDs come in many configurations each with between 8 and 80 viewable characters arranged in 1, 2, or 4 rows.

The problem is that there is no way to inform the controller of the configuration of the display that it is driving. The controller operates exactly the same way for all displays. This introduces an issue, because it means that, since the DDRAM (*Display Data Random Access Memory)* configuration of our 16x02 LCD is the same as a 40x02 display, some memory addresses in our LCD will not be mapped to the display. So if the user keeps sending characters, the address counter will be automatically incremented by 1 and at some point the characters will no longer be displayed until the AC reaches a the address 0x40 (first position of the second line).Here is a diagram showing how the two rows of the display are mapped into the two lines of memory.

We solve this issue by tracking the position of the cursor on the display with a variable, and we also define some constants for the boundaries of the display so that the code would be easily portable to other display sizes.

VARIABLE **LINE_COUNTER** \
This variable keeps track of the cursor position on the display, 1 to 16(decimal) for the first display line, 17 to 32 (decimal) for the second display line

```
1                               CONSTANT
START_OF_FIRST_LINE
11                              CONSTANT
START_OF_SECOND_LINE
10                              CONSTANT
END_OF_FIRST_LINE
20                              CONSTANT
END_OF_SECOND_LINE
```

The display functions are implemented following the lcd datasheet and making sure the **LINE_COUNTER** is properly updated. In this context we can analyze the word **DISPLAY_CHAR**. The user-provided ASCII_code is on the stack, then the **LINE_COUNTER** is checked to see if we are at the end of one of the display lines. If we are, we wish to move to the start of the other display line, otherwise the **LINE_COUNTER** is simply updated. In the first case the **FIRST_LINE** and **SECOND_LINE** words take care of updating the display counter(they do not need to fetch the value and update it, but they simply store it for increased efficiency).

```
: DISPLAY_CHAR                  ( ASCII_code -- )
   SEND_CHAR
   LINE_COUNTER @
   DUP END_OF_FIRST_LINE =
   IF
      DROP
      SECOND_LINE               \ Sets the cursor position and
LINE_COUNTER to first position of the second line
   ELSE
   END_OF_SECOND_LINE =
   IF
      FIRST_LINE            \ Sets the cursor position and
LINE_COUNTER to first position of the first line
   ELSE
      LINE_COUNTER++
```

```
    THEN
    THEN ;

: FIRST_LINE                    ( -- )
    80 SEND_CMD                 \ Sets the cursor position counter to
the first position of the first line without easing RAM data
    START_OF_FIRST_LINE LINE_COUNTER ! ;    \ Sets
LINE_COUNTER to first position (1)

: SECOND_LINE                       ( -- )
    C0 SEND_CMD                 \ Sets the cursor position counter to
the first position of the second line without easing RAM data
    START_OF_SECOND_LINE LINE_COUNTER ! ;    \ Sets
LINE_COUNTER to position 17(dec)
```

Se-ans.f

This file was provided in the course material and contains some definitions to ensure ANSI compliance and needed for words used in the input.f module.

Input.f

This section of the code handles the user input.

USER_FUNCTIONS makes a dictionary entry , the word **'** (“tick”) finds the execution token (xt) of the following word, and the word **,** (“comma”) stores it in the data field of USER_FUNCTIONS.

```
CREATE USER_FUNCTIONS ' CLEAR_DISPLAY , ' DISPLAY_LSHIFT
, ' DISPLAY_RSHIFT , ' FIRST_LINE , ' LAST_FIRST_LINE , '
SECOND_LINE , ' LAST_SECOND_LINE ,
```

A self-explanatory **SETUP** word is defined to set all the proper GPIOs, the BSC and make sure that a falling edge transition in GPIO 9 or 10 sets a bit in the event detect status registers.
A noteworthy word here is **?IS_PRESSED** . It checks that the appropriate bits of the event register GPEDS0 are set and if they are it introduces a 1ms delay and checks the GPLEV0 register. During development, sometimes undesired falling edges were detected. By checking the GPLEV0 register the ones registered while pressing the button can be avoided. After releasing , though, a phenomenon called bouncing would sometimes occur. Bouncing is the tendency of any two metal contacts in an electronic device to generate multiple signals as the contacts close or open. This is solved with a simple heuristic, that is to wait 1 ms, that guarantees optimal results for our non time-constrained application.

```
: ?IS_PRESSED                          ( -- )
   BEGIN                        \ Waits until the button has been released
        STATUS_MASK
        GPEDS0 @ AND
        DUP 0 <>
        IF
                1 MILLISECONDS DELAY    \ Makes sure the button has
properly been released, avoiding double reads
                GPLEV0 @ INVERT AND
        THEN
   UNTIL ;
```

The word **BUTTONS** checks to see which of the buttons has been pressed. Pushing and releasing one button leaves 0 on the stack, while pushing and releasing the other leaves 1.

```
: BUTTONS                       ( -- 0/1 )
   ?IS_PRESSED                  \ Waits for the button to be released
```

```
   GPEDS0 @
   400 CHECK                    ( v1 v2 -- flag )
   IF
       DROP                     \ We make sure not to leave the GPEDS0
content on the stack
       0                    \ Leaves on the stack either 0 or 1
   ELSE
   200 CHECK
   IF
       DROP
       1
   THEN
   THEN ;
```

LCD_HANDLE handles the user choice. **IS_VALID** checks that the value is within the valid range (10-16) , and if it is, we subtract 10 from the **CURRENT_VALUE** (the value given by the user via the buttons) so that **USER_CHOICE** can execute the execution token of the correct user function.

```
: USER_CHOICE                           ( nth -- )
   CELLS USER_FUNCTIONS + @ EXECUTE ;    \ We execute the xt of
the nth user function in USER_FUNCTIONS
```

If the **CURRENT_VALUE** does not correspond to a function it must be an ASCII code and we just display the character.

```
: LCD_HANDLE                            ( -- )
   CURRENT_VALUE @
   DUP ?IS_VALID                    \ Checks if CURRENT_VALUE is
within the specified range
   IF
```

```
        10 - USER_CHOICE            \ We subtract 10 so that we can
choose the correct user function in the USER_FUNCTIONS table
    ELSE
        DISPLAY_CHAR                \ If it's not a function it displays the
character
    THEN ;
```

The code of the input module is the word **INPUT**. It is an infinite cycle that makes use of the **SIZE** and **CURRENT_VALUE** variable and the constant **SIZE_REQUESTED**.

When the user pushes and releases the button a value is left on the stack, either 0 or 1. This value is left shifted every time by **SIZE**, a variable that keeps track of how many input values have been provided. The **CURRENT_VALUE** is then updated with the properly shifted user input. After updating the **SIZE** and clearing the status register, we check if **SIZE** is a multiple of **SIZE_REQUESTED**. If it is , it means 8 bits have been provided by the user, and we call **LCD_HANDLE** to handle the behavior associated with such input.

```
        : INPUT                             ( -- )
    BEGIN
        BUTTONS SIZE @ LSHIFT                   \ Left shifts the value on
the stack by size places
        CURRENT_VALUE @ + CURRENT_VALUE !     \ Adds the
shifted value to the CURRENT_VALUE
        SIZE @ 1 + DUP SIZE !               \ Adds 1 to SIZE
        CLEAR_STATUS                        \ Clears the STATUS register
        SIZE_REQUESTED MOD 0 =                  \ If size is a multiple of 8
(so we put 8 bits on the stack)
        IF
            LCD_HANDLE                      \ Call the LCD handler
            0 CURRENT_VALUE !               \ Resets the current value
```

```
            0 SIZE !            \ Resets the size
        THEN
    AGAIN ;
```

Finally, to start the whole program a **START** word is defined. The only word not already analyzed here is **WELCOME**, which simply displays text to welcome the user before starting the **INPUT** loop.

```
: START                         ( -- )
    SETUP                       \ Starts the whole program
    LCD_INIT
    WELCOME
    INPUT ;
```

Possible improvements

Even though it’s not within the scope of the project, a possible feature to add would be defining some custom characters.
All LCD displays based on the Hitachi HD44780 controller have two types of memories that store characters called CGROM and CGRAM (Character Generator ROM & RAM). CGROM is used for storing all permanent fonts that can be displayed by using their ASCII code. CGRAM can be used to store user defined characters, but it’s limited to 64 bytes. Therefore, for a 5×8 pixel based LCD up to 8 user-defined characters can be stored in the CGRAM.

Code availability

The project is available on github at
https://github.com/Frankie690II/Embedded-Systems

Resources

BCM 2835 datasheet :
https://www.raspberrypi.org/app/uploads/2012/02/BCM2835-ARM-Peripherals.pdf
EONE 1602 LCD datasheet :
https://www.openhacks.com/uploadsproductos/eone-1602a1.pdf
PCF8574 datasheet :
https://www.nxp.com/docs/en/data-sheet/PCF8574_PCF8574A.pdf