Università degli Studi di Palermo
Computer Engineering

Embedded Systems

IR Receiver Project
on bare-metal Raspberry Pi 4 Model B

By Davide Sferrazza
February 20, 2023

Contents

1 Introduction

The proposed project consists of using an IR receiver to capture commands sent by a remote controller and displaying them on an LCD display, along with the name of the pressed buttons.
This document is organized as follows:

- Firstly, I will discuss the hardware used for the implementation, by describing the physical architecture where the software will run and the external necessary harware components that will interact with it;
- Then, I will explain the environment used for project development and how to obtain the parameters needed for the project to function properly;
- Lastly, I will illustate how the code was designed and organized.

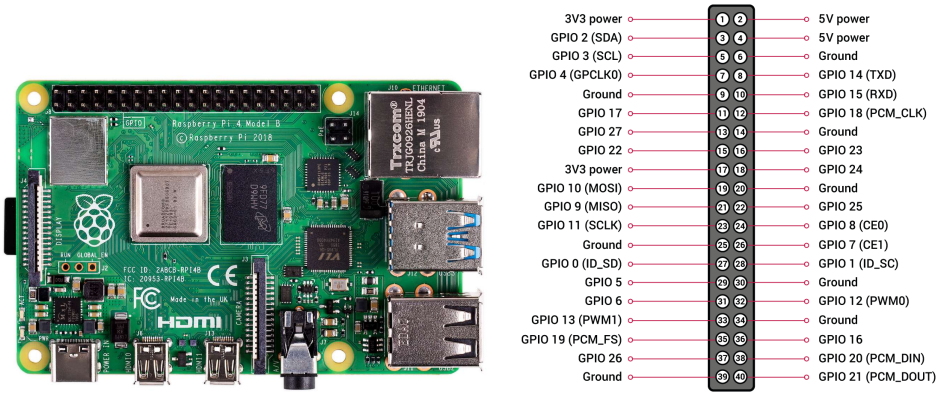
On the last pages you can find the entire code. It is also available online using GitHub (insert link).

2 Hardware

2.1 Raspberry Pi 4 Model B

The target of this project is a general-purpose Single-Board Computer (SBC): the Raspberry Pi 4 Model B.

It is a member of a series of products, which are developed in the United Kingdom by the Raspberry Pi Foundation in association with Broadcom.
It was released in June 2019[1] and replaced the well-known **Raspberry Pi 3 Model B** and **Raspberry Pi 3 Model B+**.



(a) Hardware (b) 40-pin Header

Figure 1: Raspberry Pi 4 Model B

It is built with the following specifications[2]:

- Broadcom BCM2711, Quad core Cortex-A72 (ARM v8) 64-bit SoC @ 1.5GHz;
- 1GB, 2GB, 4GB or 8GB LPDDR4-3200 SDRAM (depending on model);

- 2.4 GHz and 5.0 GHz IEEE 802.11ac wireless, Bluetooth 5.0, BLE;
- Gigabit Ethernet;
- 2 USB 3.0 ports; 2 USB 2.0 ports;
- Raspberry Pi standard 40 pin GPIO header (fully backwards compatible with previous boards);
- 2 × micro-HDMI ports (up to 4kp60 supported);
- 2-lane MIPI DSI display port;
- 2-lane MIPI CSI camera port;
- 4-pole stereo audio and composite video port;
- H.265 (4kp60 decode);
- H264 (1080p60 decode, 1080p30 encode);
- OpenGL ES 3.1, Vulkan 1.0;
- Micro-SD card slot for loading operating system and data storage;
- 5V DC via USB-C connector (minimum 3A*);
- 5V DC via GPIO header (minimum 3A*);
- Power over Ethernet (PoE) enabled (requires separate PoE HAT);
- Operating temperature: 0 – 50 degrees C ambient.

The model I used comes with 4GB of RAM.

2.2 FTDI Adapter FT232RL

Since modern computers do not expose serial ports to program the SBC, a UART (Universal Asynchronous Receiver-Transmitter) serial adapter is required.

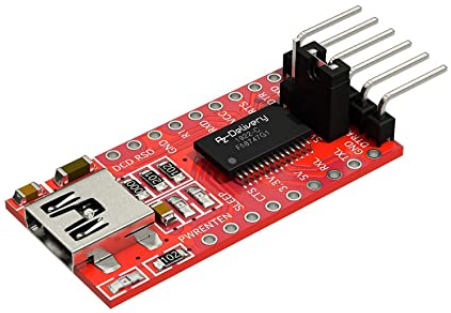


Figure 2: FTDI Adapter FT232RL from Mini-USB to TTL

I adopted the Mini-USB to TTL serial adapter provided by AZ-Delivery[4].
To avoid voltage problems power is supplied from the terminal used to develop the project and is shared between the Pi 4 and the FT232RL.
The connection is made in the following way:

- FT232RL RX pin is connected to GPIO 14 (TXD);
- FT232RL TX pin is connected to GPIO 15 (RXD);
- FT232RL GND pin is connected to the GND of the Pi 4.

2.3 KY-022 Infrared receiver module

IR detectors[5] are little microchips with a photocell that are tuned to listen to infrared light. They are almost always used for remote control detection - every TV and DVD player has one of these in the front to listen for the IR signal from the clicker. Inside the remote control is a matching IR LED, which emits IR pulses to tell the TV to turn on, off or change channels. IR light is not visible to the human eye, which means it takes a little more work to test a setup.

Figure 3: KY-022

The module is able to detect frequencies ranging from about 35 KHz to 41 KHz, but the peak frequency detection is at 38 KHz.

The IR receiver comes with 3 pins: the digital signal output pin (S) used to read the value of infrared light, the power pin (+) and the ground pin (-).

When it detects a 38KHz IR signal, the output is low. When it detects nothing, the output is high.
It requires a supply voltage in [2.7, 5.5]V, so it is powered by the Pi 4 using one of the 3V3 pins. Its GND pin is connected to the GND of the Pi 4.

Its output pin is connected to GPIO 25.

2.4 Samsung AA59-00584A

The TV remote controller I used is produced by Samsung.

Figure 4: Samsung AA59-00584A

After several timing trials, I observed that the implemented protocol conforms to the timings found online on the page cited in the references [9].

For convenience, I report the picture of the timings here:

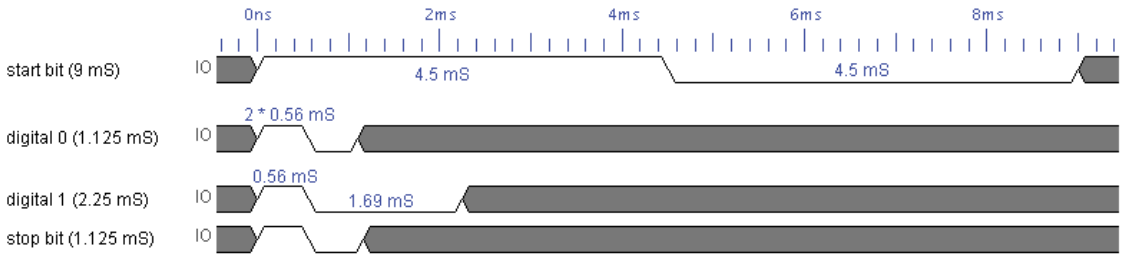


Figure 5: Samsung protocol timings

2.5 LCD 1602

The LCD 1602[6], [7] is an industrial character LCD that can display 16×2 or 32 characters at the same time, with a display font of 5×8 dots. The principle of the LCD 1602 liquid crystal display is to use the physical characteristics of the liquid crystal to control the display area by voltage, that is, the graphic can be displayed.

Figure 6: LCD 1602

It is controlled through a parallel interface with:

- 8-bit/4-bit data bus;
- 3 control signals.

The interface signals reach the two controller chips that drive the LCD panel:

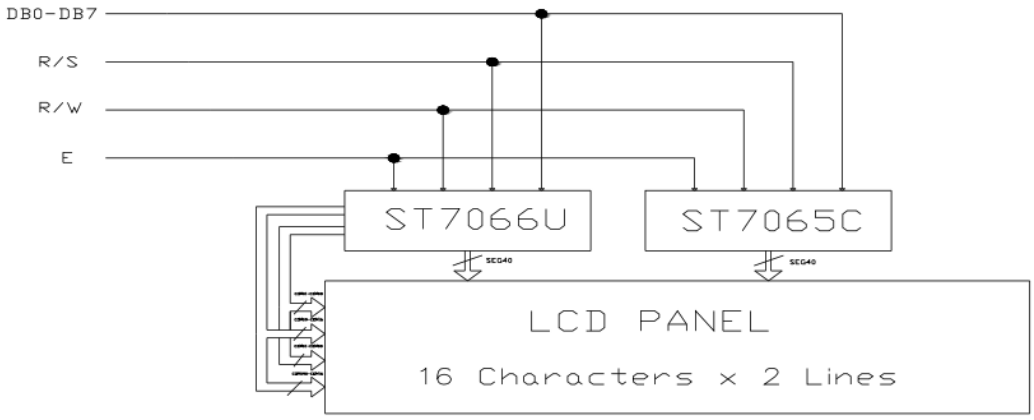


Figure 7: Block Diagram

Pin assignments are summarized in this table:

No.	Symbol	Level	Function	
1	Vss	--	0V	Power Supply
2	Vdd	--	+5V	
3	V0	--	for LCD	
4	RS	H/L	Register Select: H:Data Input L:Instruction Input	
5	R/W	H/L	H--Read L--Write	
6	E	H,H-L	Enable Signal	
7	DB0	H/L	Data bus used in 8 bit transfer	
8	DB1	H/L		
9	DB2	H/L		
10	DB3	H/L		
11	DB4	H/L	Data bus for both 4 and 8 bit transfer	
12	DB5	H/L		
13	DB6	H/L		
14	DB7	H/L		
15	BLA	--	BLACKLIGHT +5V	
16	BLK	--	BLACKLIGHT 0V-	

Figure 8: LCD pin assignments

To properly read or write data, some timing constraints must be observed. Independently of whether we are reading or writing, the Enable signal must start its falling edge while DB0-DB7 are stable. If it is not, then erroneous data are sampled. The read or write mode is chosen only by the R/W signal, so only one case of these modes can be represented.

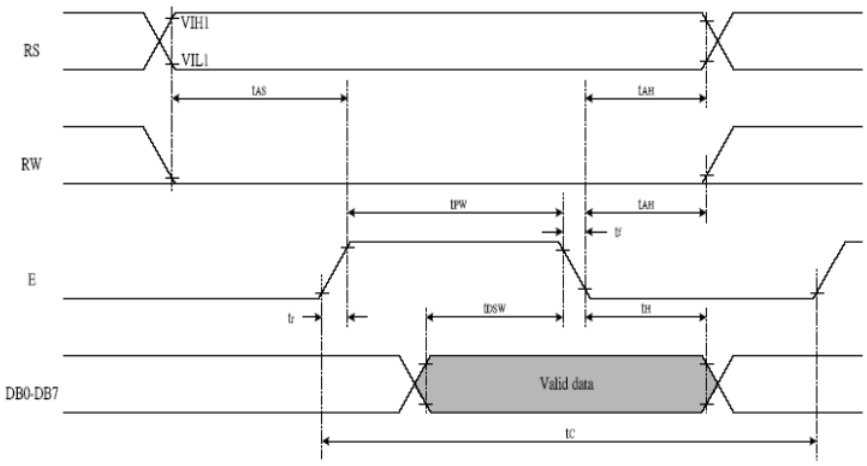


Figure 9: Writing timing characteristics

There are four categories of instructions that:

- set display format, data length, cursor move direction, display shift etc.;
- set internal RAM addresses;
- perform data transfer from/to internal RAM;

- others.

A detailed description of how they can be realized is as follows:

Instruction	Instruction Code										Description	Description Time (270KHz)
	RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0		
Clear Display	0	0	0	0	0	0	0	0	0	1	Write "20H" to DDRAM. and set DDRAM address to "00H" from AC	1.52 ms
Return Home	0	0	0	0	0	0	0	0	1	x	Set DDRAM address to "00H" from AC and return cursor to its original position if shifted. The contents of DDRAM are not changed.	1.52 ms
Entry Mode Set	0	0	0	0	0	0	0	1	I/D	S	Sets cursor move direction and specifies display shift. These operations are performed during data write and read.	37 us
Display ON/OFF	0	0	0	0	0	0	1	D	C	B	D=1:entire display on C=1:cursor on B=1:cursor position on	37 us
Cursor or Display Shift	0	0	0	0	0	1	S/C	R/L	x	x	Set cursor moving and display shift control bit, and the direction, without changing DDRAM data.	37 us
Function Set	0	0	0	0	1	DL	N	F	x	x	DL:interface data is 8/4 bits N:number of line is 2/1 F:font size is 5x11/5x8	37 us
Set CGRAM address	0	0	0	1	AC5	AC4	AC3	AC2	AC1	AC0	Set CGRAM address in address counter	37 us
Set DDRAM address	0	0	1	AC6	AC5	AC4	AC3	AC2	AC1	AC0	Set DDRAM address in address counter	37 us
Read Busy flag and address	0	1	BF	AC6	AC5	AC4	AC3	AC2	AC1	AC0	Whether during internal operation or not can be known by reading BF. The contents of address counter can also be read.	0 us
Write data to RAM	1	0	D7	D6	D5	D4	D3	D2	D1	D0	Write data into internal RAM (DDRAM/CGRAM)	37 us
Read data from RAM	1	1	D7	D6	D5	D4	D3	D2	D1	D0	Read data from internal RAM (DDRAM/CGRAM)	37 us

Figure 10: Instruction Table

2.6 IIC PCF8574AT Interface

To ease the communication to the LCD display a serial I^2C module[7], [8] has been used. The interface connects its serial input and parallel output to the LCD, so only 4 lines can be used to do the job.

The I^2C bus was invented by Philips Semiconductor (now NXP Semiconductors). It can be described as:

- synchronous;
- multi-master;
- multi-slave;

- packet switched;
- single-ended.

Each device connected to the bus is software-addressable by a unique address.

Two wires carry data (SDA - Serial DAta) and clock signals (SCL - Serial CLock), with the bus clock generated by the master.

It makes use of open-drain connections for bidirectional communication which allows us to transmit a logic low simply by activating a pull-down FET, which shorts the line to ground. To transmit a logic high, the line is left floating, and the pull-up resistor pulls the voltage up to the voltage rail.

The PCF8574AT I/O expander for I^2C-bus contains:

- 8-bit remote I/O pins (indicated as P0, P1, ..., P7) used to transfer data;
- 3 address pins (indicated as A0, A1, A2) used to address the slave.

In this project, it is not necessary to read data from the I^2C LCD, so only the writing mechanism will be described.

2.6.1 Writing mechanism

To allow a master to send data to a slave device:

1. the master, which acts as the transmitter, initiates communication by sending a START condition and addressing the slave, which acts as the receiver;
2. the master sends data to the slave-receiver;
3. the master terminates the transfer by sending a STOP condition.

A high-to-low transition on the SDA line while the SCL is high is interpreted as a START condition.
In a reverse manner, a low-to-high transition on the SDA line while the SCL is high is interpreted as a STOP condition.

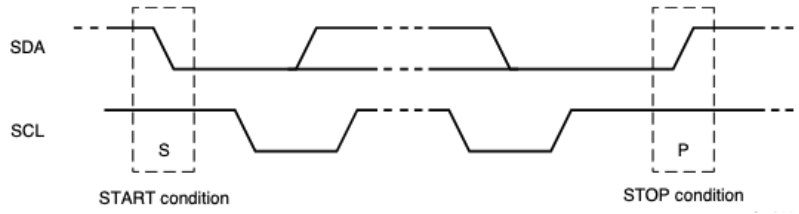


Figure 11: START and STOP conditions

A sent byte can be a device address, a register address, or data written to a slave.

One data bit is transferred during each clock pulse. The data on the SDA line must remain stable during the HIGH period of the clock pulse as changes in the data line at this time will be interpreted as control signals.
Any number of data bytes can be transferred from the master to the slave between

the START and STOP conditions.
Data is transferred starting from the MSB (Most Significant Bit).

After each byte of data has been transmitted, the master releases the SDA line to allow the slave-receiver to signal a successful transfer with an ACK (Acknowledge) or a failed transfer with a NACK (Not Acknowledge).
The receiver sends an ACK bit if the SDA is stable low during the high phase of the 9th period of the clock. If the SDA line remains high, a NACK is sent.

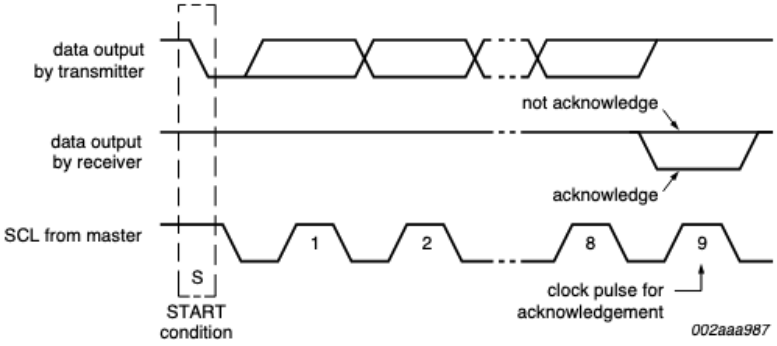


Figure 12: Acknowledgements

Thus, there are 6 steps for the writing mechanism:

1. the master sends the START condition and the slave address setting the last bit of the address byte to logic 0 for the write mode;
2. the slave sends an ACK bit;
3. the master sends the register address of the register it wishes to write to;
4. the slave possibly acknowledges again;
5. the master starts sending data;
6. the master terminates the transmission with a STOP condition.

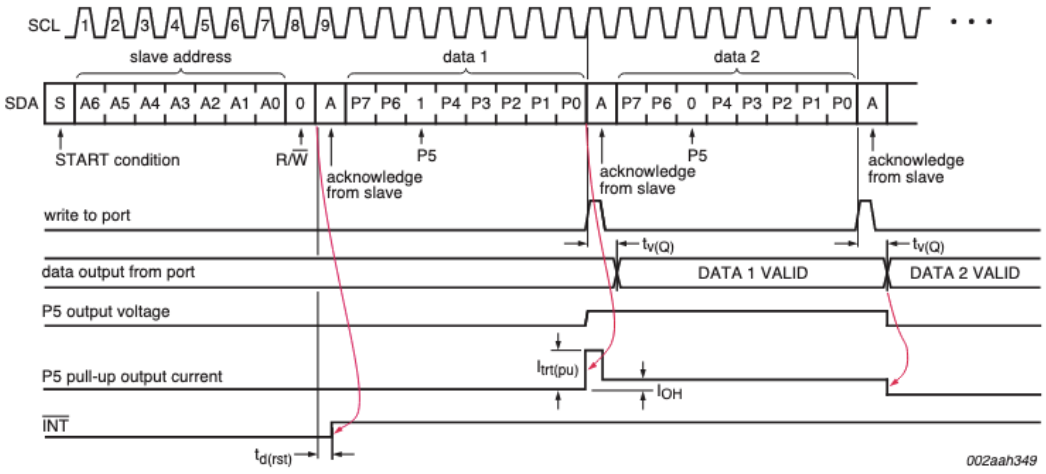


Figure 13: Write mode (output)

All this work is done by the PCF8574AT interface. All we need to do is to send the data byte for pins P7 to P0 after the slave address has been set correctly.

2.6.2 PCF8574AT to LCD connection

The PCF8574AT expander is soldered to the LCD pins according to the following scheme:

PCF8574AT pins	LCD pins
P0	RS
P1	R/W
P2	E
P3	Backlight
P4	D4
P5	D5
P6	D6
P7	D7

Table 1: PCF8574AT to LCD pin connections

Hence, the first time the LCD is turned on, it must be set to operate in 4 bit mode by sending the sequence:

D7	D6	D5	D4	Backlight	E	R/W	RS
0	0	1	0	1	1	0	0

D7	D6	D5	D4	Backlight	E	R/W	RS
0	0	1	0	1	0	0	0

Table 2: FUNCTION SET for 4 bit mode

After this setting, instructions can be sent by transmitting the most significant nibble first.
As an example, suppose you want to display an E, whose ASCII code is 0x45. To accomplish this task, the sequence to be sent is as follows:

D7	D6	D5	D4	Backlight	E	R/W	RS
0	1	0	0	1	1	0	1
0	1	0	0	1	0	0	1

D7	D6	D5	D4	Backlight	E	R/W	RS
0	1	0	1	1	1	0	1
0	1	0	1	1	0	0	1

Table 3: Sending an ASCII E

which means that the sequence to transmit is: 0x4D, 0x49, 0x5D, 0x59.

Command instructions are sent in the same way, except for the RS bit, which must be set to zero.

2.7 Schematics

All the hardware components shown in the previous sections have been connected with a breadboard to the Pi 4 as follows:

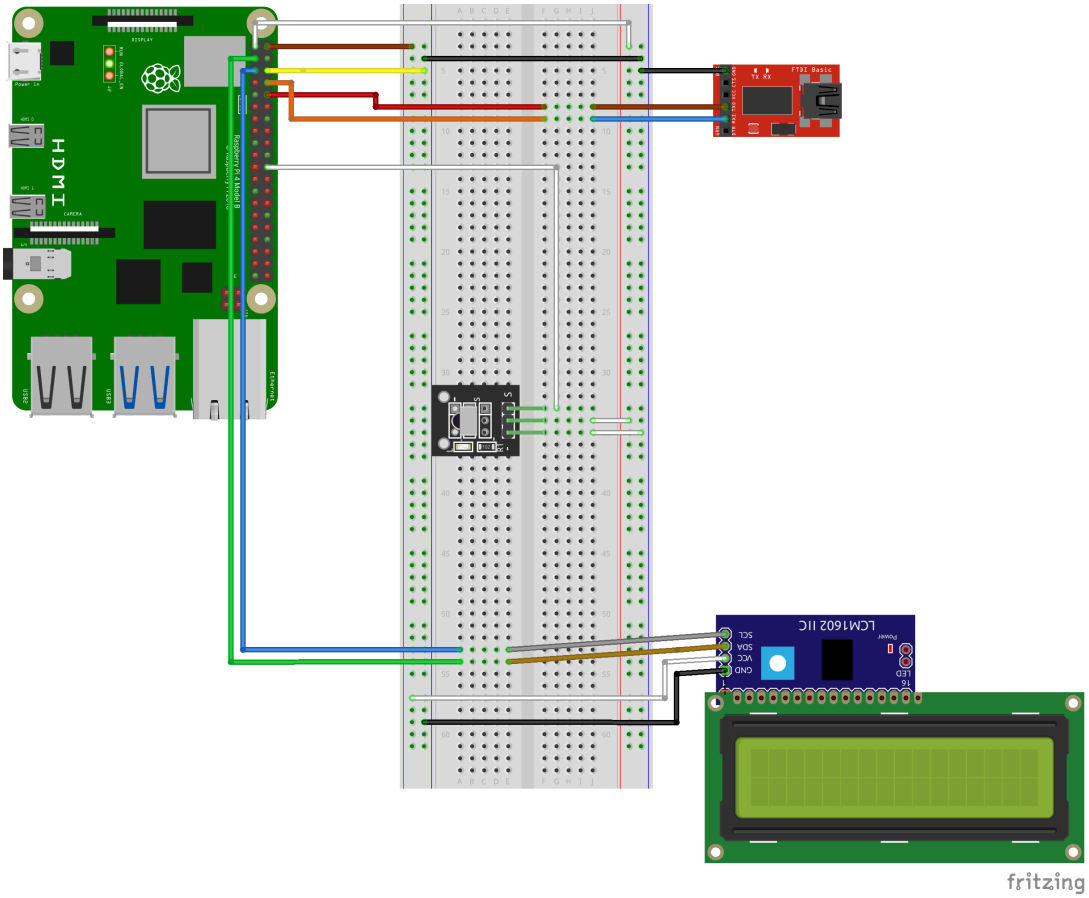


Figure 14: Schematics

The following table summarizes all the connections between them and the Pi:

Pi 4 Model B	KY-022	FT232RL	IIC PCF8574T
GPIO 25	S		
3V3 power	+		
GPIO 14 (TXD)		RX	
GPIO 15 (RXD)		TX	
5V power			VCC
GPIO 2 (SDA)			SDA
GPIO 3 (SCL)			SCL
Ground	-	GND	GND

3 Environment

3.1 pijFORTHos

Forth is a programming language designed by *Charles H. "Chuck" Moore* which can be implemented easily for resource-constrained machines due to its semplicity.
It is a procedural language heavely based on the use of the stack.

There are several dialects, each with its own definitions. The development involves the definition of new words, which added to the pre-existing vocabulary, can create the final application incrementally and naturally.

pijFORTHos[11] is a bare-metal FORTH interpreter for the Raspberry Pi, based on Jonesforth-ARM[10].

Jonesforth-ARM is an ARM port of x86 JonesForth, which is a Linux-hosted FORTH presented in a Literate Programming style by *Richard W.M. Jones*.

pijFORTHos represents the base on which the application is built.

3.2 Minicom and Picocom

Minicom[12] is a terminal emulation program for Unix-like operating systems, used for communication and typically to set up a remote serial console.

Picocom[13] is, in principle, very much like minicom. It was designed to serve as a simple, manual, modem configuration, testing, and debugging tool.

ASCII-XFR[14] transfers files in ASCII mode. It is used to send the source file to the Raspberry because it allows a delay between each character and line sent.
Given that the UART is asynchronous, this is a needed feature because it avoids overrun errors and lost characters if the receiver is busy while executing or compiing FORTH words.

Picocom is launched on the development machine through the command:

```
picocom --b 115200 /dev/cu.usbserial-A50285BI --send "ascii-xfr
↪  -sv -l100 -c10" --imap delbs
```

It is launched with the same VCP (Virtual COM Port) parameters as the Pi UART:

- `--b 115200`: 115200 bit/s bit rate;
- `/dev/cu.usbserial-A50285BI`: the serial UART device;
- `--send "ascii-xfr -sv -l100 -c10"`:

  specifies ascii-xfr as the external command to use for transmitting files.
  The options used are:

  - `-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.
- `--imap delbs`: allows the use of backspace to delete a character.

3.3 Script

Since the serial UART has a limited operating speed, one way to speed up the loading of the entire project is to delete all comments and merge the files into a single file.

As not everyone has a Unix-like operating system, the first part of the script (`create_program.sh`) is made for Zsh, but can be easily changed to other shells because it simply prints the files to the stdout.

```
#! /bin/zsh

rm program.f 2> /dev/null
cat jonesforth.f    \
    se-ans.f        \
    utils.f         \
    timer.f         \
    i2c.f           \
    lcd.f           \
    ir_receiver.f   \
    lookup_table.f  \
    main.f          \
    | ./unify_and_uncomment.py
```

The second part (`unify_and_uncomment.py`) is done in Python, because today almost everyone has a Python interpreter in their system. It reads from stdin, deleting comments and joining all words by separating them with a single space.

```
#! /usr/bin/python3

import sys
import re

with open('program.f', 'a') as f:
    for line in sys.stdin:
        # delete leading and trailing extra spaces
        line = line.strip()
        # skip comment lines
        if line and line[0] != '\\':
            # delete inline comments such as ( i1 i2 ... -- o1 o2 ... )
            reg = re.search("\s+\(\s+[^-]*--[^-]*\s+\)", line)
            if hasattr(reg, 'start') and hasattr(reg, 'end'):
                idx1 = reg.start()
                idx2 = reg.end()
                line = line[:idx1] + line[idx2:]
            # delete inline comments such as \ ...
            line = line[: line.find('\\') - 1 ] if line.find('\\') != -1 else line
```

```
20            # unify line by deleting extra spaces and skipping
21            # empty strings or strings containing only whitespaces
22            skip_whitespaces = lambda x: x and not x.isspace()
23            line = ' '.join( filter( skip_whitespaces,  re.split('\s+', line) ) )
24            print(line, file=f, end=' ')
```

Script files must have execution permissions to run. If you are using a Unix-like operating system you can grant them by typing in a terminal window (assuming you are in the project folder):

```
chmod u+x unify_and_uncomment.py
chmod u+x create_program.sh
```

and then build the project by simply writing (assuming that you have changed the shell and python interpreter):

```
./create_program.sh
```

4 Software

4.1 Prerequisites

The files needed to be placed in the micro SD card to run the software are as follows:

- `bootcode.bin`;
- `config.txt`;
- `fixup.dat`;
- `start4.elf`;
- `kernel7.img` (which contains pijFORTHos);
- `bcm2711-rpi-4-b.dtb`.

An empty micro SD card with only those files is sufficient.
If you do not have these files, you can download them from the official repository[1] or operate in this way: you can format the micro SD card for the Pi 4 using the official software called Raspberry Pi Imager[2], released by Raspberry Pi.
When you open the program, you are asked to choose an operating system. You can select Raspberry Pi OS (32 bit) and continue with formatting.

Once the formatting is complete[3], you need to remove the `kernel*.img` and insert the `kernel7.img` that contains pijFORTHos.

Your `config.txt` file must contain the following uncommented options:

```
dtparam=i2c_arm=on
enable_uart=1
```

[1] `https://github.com/raspberrypi/firmware`
[2] `https://www.raspberrypi.com/software/`
[3] Please read the following subsection first to avoid repeating the procedure

which enable the I^2C and the UART, respectively.

4.1.1 How to get IIC LCD slave address

For the project to work, it is important to know the correct slave address.

A visual method to recognize the address is to analyze the expander structure as follows[7]:

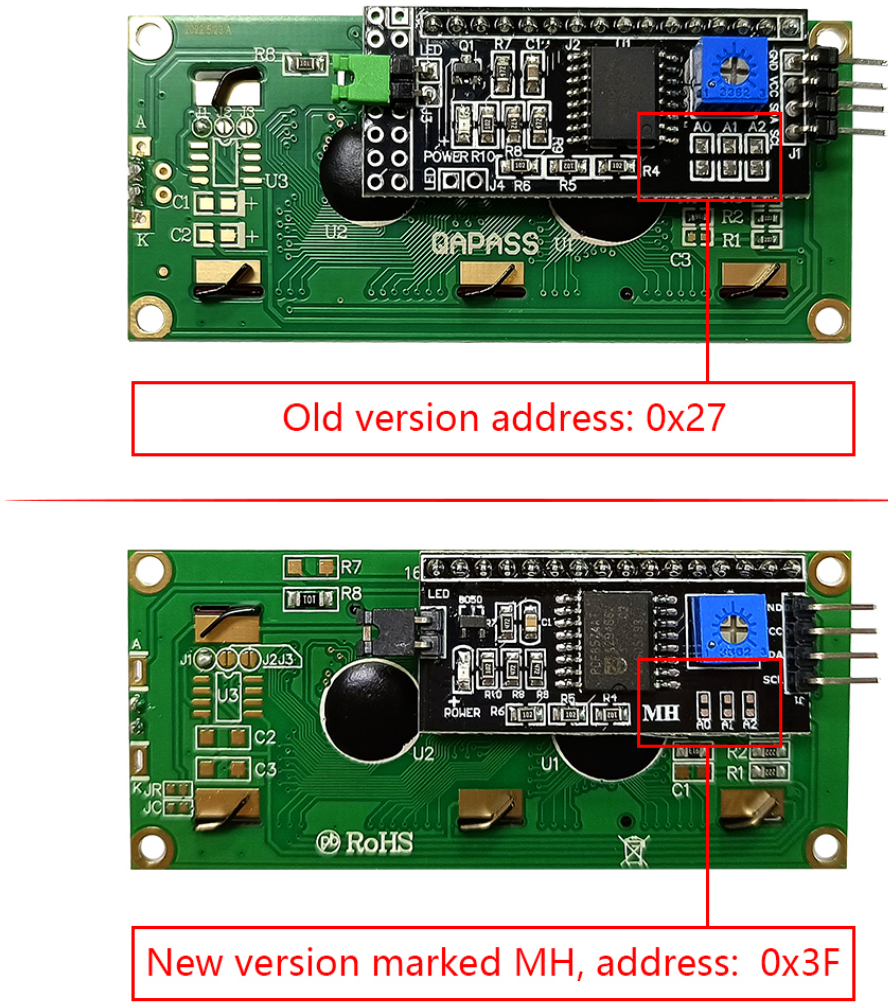


Figure 15: PCF8574AT I/O expander

If it doesn't work and you still can't figure out what the address is you can operate using Raspbian.
Open a terminal window and install `i2c-tools`:

```
sudo apt-get install i2c-tools
```

Once the download is complete, you can simply type:

```
i2cdetect -y 1
```

Now, if the connection on the breadboard is correct, you should see the slave address in the center of the array.
In my case, the I^2C slave address is 0x27.

4.1.2 How to run

Once you have downloaded the project, you can port it to **Raspberry Pi 3 Model B** and **Raspberry Pi 3 Model B+** by changing the `PERI_BASE` constant to `0x3F000000` (`utils.f`)[4].
After possibly performing this operation and reading the previous subsection, having changed if necessary the literal used to set the slave address into the word `INIT_I2C` (`i2c.f`), you can run the script and load the program as follows:

1. execute `./create_program.sh` and run

```
picocom --b 115200 /dev/cu.usbserial-A50285BI --send
↪  "ascii-xfr -sv -l100 -c10" --imap delbs
```

2. load the file `program.f` generated by the script using ctrl+A and ctrl+S, and writing the path to the file, e.g. `./program.f`;
3. start the program by typing `MAIN` and pressing ⏎.

4.2 Code structure

The project is composed of 9 files:

- `jonesforth.f`;
- `se-ans.f`;
- `utils.f`;
- `timer.f`;
- `i2c.f`;
- `lcd.f`;
- `ir_receiver.f`;
- `lookup_table.f`;
- `main.f`;

The development process followed a bottom-up methodology, so the files are listed in an ascending order of abstraction.

Each file consists of a module, which contains all the constants and words necessary for the proper functioning of that module.

[4]For the project to work, the boot files must also be changed

Given the information on the previous pages, it should be clear how the project works. In addition to the defined words, there are comments explaining what may not be perfectly clear.

4.2.1 ANSI Compliance

JonesForth does not conform to ANSI[15] standards, which means that some words do not behave as expected.

For this reason, the first file `jonesforth.f`, which contains words such as `S"`, is loaded as the first file, to ensure that `OF` and `ENDOF` used in `lookup_table.f` can function properly.

After this necessary operation, regardless of the files containing the above words, all subsequent words are ANSI-compliant thanks to the file `se-ans.f`, created by Professor Daniele Peri.

4.2.2 jonesforth.f

```
\ Some word definitions taken from pijFORTHos.
\ Link:
↪  https://github.com/organix/pijFORTHos/blob/master/annexia/jonesforth.f.txt

\ These words are necessery in order to PRINT_STRING (in lcd.f)
\ to work properly.
\ After a number of attempts trying to rencocile these definitions with
\ those contained in se-ans.f, I observed that lookup_table.f stop working.
\ The problem resides in the use of OF - ENDOF.
\ This file must be uploaded as the first one.

\ We can use [ and ] to insert literals which are calculated at compile time.
\ (Recall that [ and ] are the FORTH words that allow you to exit and enter
↪  compilation mode.)
\ Within definitions, use [ ... ] LITERAL anywhere that '...' is a constant
↪  expression
\ which you would rather only compute once (at compile time, rather than
↪  calculating
\ it each time your word runs).

\ Returns the ASCII code of ".
: '"' ( -- ascii_code )
    [ CHAR " ] LITERAL ;

\ Takes an address and rounds it up (aligns it) to the next 4 byte boundary.
: ALIGNED ( c_addr -- a_addr )
    3 + 3 INVERT AND ;

\ Aligns the HERE pointer, so the next word appended will be aligned properly.
: ALIGN ( -- )
    HERE @ ALIGNED HERE ! ;

\ Appends a byte to the current compiled word.
: C, ( byte -- )
    HERE @ C! 1 HERE +! ;
```

```
\ S" string" is used in FORTH to define strings.  It leaves the address of the
↪  string and
\ its length on the stack, (length at the top of stack).  The space following S"
↪  is the
\ normal space between FORTH words and is not a part of the string.
\ This is tricky to define because it has to do different things depending on
↪  whether
\ we are compiling or in interpret mode.  (Thus the word is marked IMMEDIATE so
↪  it can
\ detect this and do different things).
\ In compile mode we apppend the string length and the string rounded up 4 bytes
\ to the current word.
\ In interpret mode there isn't a particularly good place to put the string, but
↪  in this
\ case we put the string at HERE (but we _don't_ change HERE). This is meant as a
\ temporary location, likely to be overwritten soon after.
: S" IMMEDIATE ( -- addr len )
        STATE @ IF
                ' LITS , HERE @ 0 ,
                BEGIN KEY DUP '"'
                <> WHILE C, REPEAT
                DROP DUP HERE @ SWAP - 4- SWAP ! ALIGN
        ELSE
                HERE @
                BEGIN KEY DUP '"'
                <> WHILE OVER C! 1+ REPEAT
                DROP HERE @ - HERE @ SWAP
        THEN ;
```

4.2.3 se-ans.f

```
\ Sistemi Embedded 18/19
\ Daniele Peri - Universita' degli Studi di Palermo
\
\ Some definitions for ANS compliance
\
\ v. 20181215

: JF-HERE   HERE ;
: JF-CREATE   CREATE ;
: JF-FIND   FIND ;
: JF-WORD   WORD ;

: HERE   JF-HERE @ ;
: ALLOT   HERE + JF-HERE ! ;

: [']   ' LIT , ; IMMEDIATE
: '   JF-WORD JF-FIND >CFA ;

: CELL+  4 + ;

: ALIGNED   3 + 3 INVERT AND ;
: ALIGN JF-HERE @ ALIGNED JF-HERE ! ;
```

```
: CREATE   JF-WORD JF-CREATE DOCREATE , ;
: (DODOES-INT)  ALIGN JF-HERE @ LATEST @ >CFA ! DODOES> ['] LIT ,  LATEST @ >DFA
↪  , ;
: (DODOES-COMP)  (DODOES-INT) ['] LIT , , ['] FIP! , ;
: DOES>COMP   ['] LIT , HERE 3 CELLS + , ['] (DODOES-COMP) , ['] EXIT , ;
: DOES>INT   (DODOES-INT) LATEST @ HIDDEN ] ;
: DOES>   STATE @ 0= IF DOES>INT ELSE DOES>COMP THEN ; IMMEDIATE
```

4.2.4 utils.f

```
\ Changes base from decimal to hex.
HEX

\ Creates constant for peripherals base memory address.
FE000000    CONSTANT PERI_BASE

\ Creates constant for offset GPIO registers base address.
200000      CONSTANT GPIO_OFFSET

\ Creates constant for offset GPIO output set register.
1C          CONSTANT GPIO_SET_OFFSET

\ Creates constant for offset GPIO output clear register.
28          CONSTANT GPIO_CLR_OFFSET

\ Creates constant for offset GPIO pin level registers.
34          CONSTANT GPIO_LEV_OFFSET

\ Creates constant for LOW bit value.
00          CONSTANT LOW

\ Creates constant for HIGH bit value.
01          CONSTANT HIGH

\ Creates constant for input function.
00          CONSTANT INPUT

\ Creates constant for output function.
01          CONSTANT OUTPUT

\ Creates constant for alternate function 0.
04          CONSTANT ALT0

\ Creates constant for alternate function 1.
05          CONSTANT ALT1

\ Creates constant for alternate function 2.
06          CONSTANT ALT2

\ Creates constant for alternate function 3.
07          CONSTANT ALT3

\ Creates constant for alternate function 4.
03          CONSTANT ALT4

\ Creates constant for alternate function 5.
```

```
02          CONSTANT ALT5

\ Creates constant for GPIO register base address.
PERI_BASE GPIO_OFFSET + CONSTANT GPIO_BASE

\ Fetches the contents of the return stack.
: R@ ( -- top_of_return_stack )
    R> R> TUCK >R >R ;

\ Clears the specified bits of a given word using a pattern.
: BIC ( word pattern -- word_with_cleared_bits )
    INVERT AND ;

\ Returns -1 if a value in contained within an interval, 0 otherwise.
: IN_RANGE ( value low high -- truth_value )
    ROT DUP ROT                             \ low value value high
    <=                                      \ low value value<=high
    -ROT                                    \ value<=high low value
    <= AND ;                                \ low<=value<=high

\ Multiplies a number by 4 to refer to word offsets.
: >WORD ( number -- word_aligned_number )
    02 LSHIFT ;

\ Returns GPFSEL register address for a given GPIO pin.
: GPFSEL ( gpio_pin_number -- gpio_pin_address )
    0A /                                    \ GPFSEL register number
    >WORD GPIO_BASE + ;

\ Creates mask for a given GPIO pin.
: MASK ( gpio_pin_number -- mask )
    0A MOD                                  \ Offset (base 10) for
    ↪ gpio_pin_number in GPFSEL contents
    DUP 2* +                                \ Multiplies by 3 to get the real
    ↪ offset
    07 SWAP LSHIFT INVERT ;                 \ Returns the mask

\ Returns a configuration for a GPFSEL contents update given a functionality in
↪ 0-7 and a GPIO pin number.
: CONFIGURATION ( functionality_number gpio_pin_number --
↪ configuration_for_GPFSEL )
    0A MOD                                  \ Offset (base 10) for
    ↪ gpio_pin_number in GPFSEL contents
    DUP 2* +                                \ Multiplies by 3 to get the real
    ↪ offset
    LSHIFT ;                                \ Returns contents to update the
    ↪ functionality of a pin

\ Configures a specific functionality for a GPIO pin given its number and the
↪ functionality in 0-7.
: CONFIGURE ( gpio_pin_number functionality_number -- )
    SWAP DUP GPFSEL >R                      \ Gets GPFSEL register address and
    ↪ stores in the return stack
    DUP MASK                                \ Gets the mask for gpio_pin_number
    R@ @ AND                                \ Cleans up the GPFSEL register
    ↪ contents for gpio_pin_number
```

```
    -ROT CONFIGURATION OR                   \ Updates the contents to set up the
    ↪ functionality
    R> ! ;                                  \ Stores the new value in the GPFSEL
    ↪ register address

\ Returns GPSET register address for a given GPIO pin.
: GPSET ( gpio_pin_number -- gpio_pin_address )
    20 /                                    \ GPSET register number
    >WORD GPIO_BASE + GPIO_SET_OFFSET + ;

\ Returns GPCLR register address for a given GPIO pin.
: GPCLR ( gpio_pin_number -- gpio_pin_address )
    20 /                                    \ GPCLR register number
    >WORD GPIO_BASE + GPIO_CLR_OFFSET + ;

\ Returns a 32 bit word with just one bit high in the proper positon for a GPIO
↪ pin.
: BIT>WORD ( gpio_pin_number -- bit_word )
    20 MOD                                  \ Converts to base 32
    01 SWAP LSHIFT ;

\ Returns 0 or 1 depending on the value of the bit in a given position of a given
↪ 32 bit word.
: WORD>BIT ( bit_word bit_number -- bit_value )
    RSHIFT 01 AND ;

\ Sets a GPIO output high for a given GPIO pin.
: SET_HIGH ( gpio_pin_number -- )
    DUP BIT>WORD                            \ Gets the right bit position to
    ↪ update the contents of GPSET
    SWAP GPSET ! ;

\ Sets a GPIO output low for a given GPIO pin.
: SET_LOW ( gpio_pin_number -- )
    DUP BIT>WORD                            \ Gets the right bit position to
    ↪ update the contents of GPCLR
    SWAP GPCLR ! ;

\ Returns GPLEV register address for a given GPIO pin.
: GPLEV ( gpio_pin_number -- gpio_pin_address )
    20 /                                    \ GPLEV register number
    >WORD GPIO_BASE + GPIO_LEV_OFFSET + ;

\ Returns 0 or 1 depending on the level (low or high) of a specific GPIO pin when
↪ set as input.
: READ ( gpio_pin_number -- )
    DUP GPLEV @                             \ Gets the contents of GPLEV
    ↪ register
    SWAP WORD>BIT ;
```

4.2.5 timer.f

```
\ Creates constant for the offset of the System Timer registers.
3000 CONSTANT TIMER_OFFSET

\ Creates constant for the System Timer Counter Lower bits.
```

```
PERI_BASE TIMER_OFFSET + 04 +    CONSTANT TIMER

\ Creates variabile to store last time read.
VARIABLE LAST_TIME

\ Starts the timer by storing the time read in LAST_TIME.
\ Usage: TIMER START
: START ( timer_address -- )
    @ LAST_TIME ! ;

\ Stops the timer by subtracting the time stored in LAST_TIME to the current
↪  time.
\ Usage: TIMER STOP
: STOP ( timer_address -- time_in_us )
    @ LAST_TIME @ - ;

\ Delays by a certain amount of time.
: DELAY ( delay_amount_in_us -- )
    TIMER START
    BEGIN
        DUP
        TIMER STOP
        <                          \ delay_amount_in_us < current_elapsed_time
    UNTIL DROP ;
```

4.2.6 i2c.f

```
\ There are 8 Broadcom Serial Control (BSC) controllers, numbered from 0 to 7
\ Only 6 of these masters can be used, because BSC masters 2 and 7 are not
↪  user-accessible.
\ Since GPIO pins 2 and 3 are used, BSC1 is the reference master.
804000 CONSTANT BSC1

\ There are 8 I2C registers, each of which is at an address obtained by applying
↪  an
\ offset to BSC1 (this process is the same for all BSCs).
\ I2C registers are:
\ - C register          The control register is used to enable interrupts, clear
↪  the FIFO,
\                       define a read or write operation and start a transfer;
\ - S register          The status register is used to record activity status,
↪  errors
\                       and interrupt requests;
\ - DLEN register       The data length register defines the number of bytes of
↪  data
\                       to transmit or receive in the I2C transfer. Reading the
\                       register gives the number of bytes remaining in the
↪  current
\                       transfer;
\ - A register          The slave address register specifies the slave address
↪  and cycle type.
\                       The address register can be left across multiple
↪  transfers;
\ - FIFO register       The Data FIFO register is used to access the FIFO. Write
↪  cycles
\                       to this address place data in the 16-byte FIFO, ready to
```

```
\                       transmit on the BSC bus. Read cycles access data received
↪  from
\                       the bus;
\ - DIV register        The clock divider register is used to define the clock
↪  speed of the
\                       BSC peripheral;
\ - DEL register        The data delay register provides fine control over the
\                       sampling/launch point of the data;
\ - CLKT Register       The clock stretch timeout register provides a timeout on
↪  how long the
\                       master waits for the slave to stretch the clock before
↪  deciding that
\                       the slave has hung.

\ The following constants are defined to point to the registers above.
\ DIV, DEL and CLKT can be left without changes.
BSC1 PERI_BASE +             CONSTANT CTRL
BSC1 PERI_BASE + 04 +        CONSTANT STATUS
BSC1 PERI_BASE + 08 +        CONSTANT DLEN
BSC1 PERI_BASE + 0C +        CONSTANT SLAVE
BSC1 PERI_BASE + 10 +        CONSTANT FIFO
BSC1 PERI_BASE + 14 +        CONSTANT DIV
BSC1 PERI_BASE + 18 +        CONSTANT DEL
BSC1 PERI_BASE + 1C +        CONSTANT CLKT

\ Sets slave address.
\ It can be left across multiple transfers.
: SET_SLAVE ( slave_address -- )
    SLAVE ! ;

\ Sets number of data bytes to transfer.
: SET_DLEN ( length -- )
    DLEN ! ;

\ Places 8 bits at a time in FIFO in order to transmit them on the BSC bus.
: APPEND ( 8_bit_data -- )
    FIFO ! ;

\ Resets the control register without touching the reserved bits.
\ Reserved bits are in positions: 31:16, 14:11, 6 and 3:1.
\ Interrupts are disabled.
: RESET_CTRL ( -- )
    CTRL @ 87B1 BIC CTRL ! ;

\ Resets status for subsequent transfers without touching the reserved bits.
\ Only CLKT (9), ERR (8) and DONE (1) can be cleared (W1C type), all other flags
↪  are read-only (RO).
\ Reserved bits are in positions: 31:10.
: RESET_STATUS ( -- )
    STATUS @ 302 OR STATUS ! ;

\ Clears FIFO without touching the reserved bits.
\ - CLEAR (5:4) set to X1 or 1X in order to clear the FIFO before the new frame
↪  is started.
\ Interrupts are disabled.
: CLEAR_FIFO ( -- )
```

```
    CTRL @ 10 OR CTRL ! ;

\ Modifies control register to trigger a transfer.
\ To start a new transfer, all bits are zero except for:
\ - I2CEN (15) set to 1 to enable the BSC controller;
\ - ST (7) set to 1 to start a new transfer (one-shot operation).
\ Interrupts are disabled.
: TRANSFER ( -- )
    CTRL @ 8080 OR CTRL ! ;

\ Transfers data through the I2C bus interface.
\ Since communication is established to the LCD panel, 8 bits at a time are sent.
: >I2C
    RESET_STATUS
    RESET_CTRL
    CLEAR_FIFO
    01 SET_DLEN
    APPEND
    TRANSFER ;

\ Sets up the I2C bus interface and the slave address.
\ Configures GPIO pin 2 for Serial Data Line.
\ Configures GPIO pin 3 for Serial Clock Line.
\ Sets the slave address to 0x27.
: INIT_I2C
    02 ALT0 CONFIGURE
    03 ALT0 CONFIGURE
    27 SET_SLAVE ;

\ Consider the following structure of data transfer:
\           D7 D6 D5 D4 Backlight Enable Read/Write Register-Select
\ equivalently:
\           D7 D6 D5 D4 BL EN RW RS
\ In order to send a byte, it must be decomposed in two nibbles, upper nibble and
↪  lower
\ nibble. Each nibble represented by D7 D6 D5 D4 must be followed by a
↪  combination of
\ BL EN RW RS.
\ For each data or command to transfer RW = 0.
\ Given a byte B = HIGH LOW (upper-nibble lower-nibble), if it is part of a
↪  command,
\ then transfer is obtained by sending:
\   HIGH 1 1 0 0 -> HIGH 1 0 0 0 -> LOW 1 1 0 0 -> LOW 1 0 0 0
\ If it part of a data transfer then:
\   HIGH 1 1 0 1 -> HIGH 1 0 0 1 -> LOW 1 1 0 1 -> LOW 1 0 0 1
\ RS is equal to 0 for command input and it is equal to 1 for data input.


\ Returns setting parts to be sent based on a truth value that indicates whether
↪  the input
\ is part of a command or data.
: SETTINGS ( truth_value -- first_setting second_setting )
    IF
        0C 08                       \ Setting parts for a command
    ELSE
        0D 09                       \ Setting parts for data
```

```
    THEN ;

\ Returns a nibble aggregated with the first setting part and the second setting
↪  part.
: AGGREGATE ( first_setting second_setting nibble -- nibble_second_setting
↪  nibble_first_setting )
    04 LSHIFT DUP                   \ first_setting second_setting byte byte
    ROT OR                          \ first_setting byte nibble_second_setting
    -ROT OR ;

\ Returns the lower nibble of a byte.
: >NIBBLE ( byte -- lower_nibble )
    0F AND ;

\ Divides a byte into two nibbles.
: BYTE>NIBBLES ( byte -- lower_nibble upper_nibble )
    DUP >NIBBLE                     \ Gets lower_nibble
    SWAP 04 RSHIFT >NIBBLE ;        \ Gets upper_nibble

\ Sends a nibble to LCD aggregated with settings.
: SEND_NIBBLE ( nibble truth_value -- )
    SETTINGS ROT
    AGGREGATE                       \ Gets the 2 bytes to send
    >I2C 1000 DELAY
    >I2C 1000 DELAY ;

\ Transmits input to LCD given an instruction or data.
: >LCD ( input -- )
    DUP 08 WORD>BIT >R              \ Stores the command/data bit in the return
    ↪  stack
    BYTE>NIBBLES R@                 \ Creates the two nibbles to be sent
    SEND_NIBBLE R>                  \ Sends the most significant nibble
    SEND_NIBBLE ;                   \ Sends the least significant nibble
```

4.2.7 lcd.f

```
\ Creates constant for 0 ASCII code.
30 CONSTANT 0ASCII

\ Creates constant for A ASCII code.
41 CONSTANT AASCII

\ Sets up the LCD.
\ By sending this command to the LCD, we set up the data interface to 4 bits,
↪  instead
\ of 8 bits, and turn the cursor and cursor position off.
: INIT_LCD ( -- )
    102 >LCD                    \ Sets 4 bit mode using FUNCTION SET
    10C >LCD ;                  \ Disables cursor and cursor position

\ The following words can be used only after the LCD has been set up.

\ Clears the LCD display.
: CLEAR_DISPLAY ( -- )
    101 >LCD ;
```

```
\ Moves the cursor to the first cell in the first row of the LCD display.
\ It stands for Return Home Line 1.
: RH_LINE1 ( -- )
    102 >LCD ;

\ Returns the cursor to the first cell in the second row of the LCD display.
\ It stands for Return Home Line 2.
: RH_LINE2 ( -- )
    1C0 >LCD ;

\ Prints a string to the LCD.
\ Usage: S" embedded systems" PRINT_STRING
: PRINT_STRING ( address length -- )
    OVER + SWAP                 \ address_last_char+1 address_first_char
    BEGIN                       \ While there are chars to send
        DUP C@ >LCD             \ Sends the char of the current position
        1+                      \ Increment address
        2DUP =                  \ Checks whether the string is terminated
    UNTIL 2DROP ;

\ Converts a 4-bit hexadecimal number to the corrisponding ASCII code.
: HEX>ASCII ( 4_bit_number -- ascii_code )
    DUP 09 >
    IF                          \ If it is greater than 9, it is a letter
        0A -                    \ Then subtracts 10 and adds the ASCII code of A
        AASCII +
    ELSE
        0ASCII +                \ Adds the ASCII code of 0 otherwise
    THEN ;

\ Prints a hexadecimal number to the LCD.
: PRINT_HEX ( number -- )
    1C                          \ The amount of shift for sending the current
    ↪ nibble
    BEGIN
        DUP
        0 >=                    \ While there are nibbles to send
    WHILE
        2DUP RSHIFT
        >NIBBLE                 \ Gets the current nibble
        HEX>ASCII >LCD          \ Sends it to the LCD
        04 -                    \ Decrements the amount of current shift
    REPEAT 2DROP ;
```

4.2.8 ir_receiver.f

```
\ Creates constant for IR receiver GPIO pin.
19 CONSTANT RECEIVER

\ Creates constant for lower bound time for a START_BIT detection (4.35 ms).
10FE CONSTANT LB_START_BIT

\ Creates constant for upper bound time for a START_BIT detection (4.65 ms).
122A CONSTANT UB_START_BIT

\ Creates constant for lower bound time for a 0 bit detection (0.44 ms).
```

```
1B8 CONSTANT LB_0_BIT

\ Creates constant for upper bound time for a 0 bit detection (0.68 ms).
2A8 CONSTANT UB_0_BIT

\ Creates variable to store the sampled command.
VARIABLE COMMAND

\ Sets up the receiver by configuring its GPIO pin as input.
: INIT_RECEIVER ( -- )
    RECEIVER INPUT CONFIGURE ;

\ Awaits a transition from the given value to its negation and returns the
↪  elapsed time.
\ Usage: RECEIVER LOW (or HIGH) AWAIT
: AWAIT ( receiver value -- elapsed_time )
    TIMER START
    BEGIN                                   \ Repeats the loop while the read
    ↪  input for
        OVER READ                           \ RECEIVER is equal to the current
        ↪  value i.e.
        OVER <>                             \ awaits a transition to the
        ↪  negation of value
    UNTIL
    2DROP
    TIMER STOP ;

\ Detects if a start bit arrives.
\ -1 is left on the stack if a start bit is detected, 0 otherwise.
\ A start bit is detected if after a transition from HIGH to LOW, LOW is held for
↪  4.5 ms
\ and after a transition from LOW to HIGH, HIGH is held for 4.5 ms.
\ Since perfect timing cannot be achieved, I sample a timing contained within
↪  4.35 ms and 4.65 ms.
\ Usage: RECEIVER START_BIT
: START_BIT ( receiver -- good_or_fail )
    BEGIN
        DUP READ                             \ Waiting for the transition to
        ↪  begin
        LOW =
    UNTIL
    DUP LOW AWAIT                            \ Times how long input is kept LOW
    LB_START_BIT UB_START_BIT IN_RANGE       \ Checks if it can be considered
    ↪  valid
    OVER HIGH AWAIT                          \ Times how long input is kept
    ↪  HIGH
    LB_START_BIT UB_START_BIT IN_RANGE       \ Checks if it can be considered
    ↪  valid
    AND NIP ;


\ Detects if a new arrived bit is a 0 or a 1.
\ -1 is left on the stack if a 0 is detected, 1 if a 1 is detected, 0 otherwise.
\ A 0 is detected if after a transition from HIGH to LOW, LOW is held for 0.56 ms
↪  and
\ after a transition from LOW to HIGH, HIGH is held for 0.56 ms.
```

```
\ A 1 is detected if after a transition from HIGH to LOW, LOW is held for 0.56 ms
↪  and
\ after a transition from LOW to HIGH, HIGH is held for 1.69 ms.
\ Since perfect timing cannot be achieved, I sample a timing contained within
↪  0.44 ms and 0.68 ms.
\ Usage: RECEIVER DETECT_BIT
: DETECT_BIT ( receiver -- sampled_value )
    DUP LOW AWAIT                               \ Times how long input is kept LOW
    LB_0_BIT UB_0_BIT IN_RANGE                  \ Checks if it can be considered
    ↪  valid
    OVER HIGH AWAIT                             \ Times how long input is kept
    ↪  HIGH
    LB_0_BIT UB_0_BIT IN_RANGE                  \ Checks whether it is
    IF
        -1                                      \ a 0
    ELSE
        1                                       \ or a 1
    THEN
    AND NIP ;

\ Add a new bit to the command sampled.
\ If value is -1 then add a 0, 1 otherwise.
: ADD_BIT ( value -- )
    COMMAND @
    2* SWAP                                     \ Makes room for the new bit
    1 =
    IF
        01 OR                                   \ Adds a 1 if it is a 1
    THEN
    COMMAND ! ;

\ Detects if a stop bit arrives.
\ -1 is left on the stack if a stop bit is detected, 0 otherwise.
\ A stop bit is detected if after a transition from HIGH to LOW, LOW is held for
↪  0.56 ms
\ and after a transition from LOW to HIGH, HIGH is held for  0.56 ms.
\ Since perfect timing cannot be achieved, I sample a timing contained within
↪  0.44 ms and 0.68 ms.
\ Usage: RECEIVER STOP_BIT
: STOP_BIT ( receiver -- )
    DUP >R                                      \ Stores the RECEIVER pin number
    ↪  in the return stack
    LOW AWAIT                                   \ Waits for a transition from LOW
    ↪  to HIGH
    DROP UB_0_BIT
    TIMER START
    BEGIN                                       \ Repeats while
        R@ READ                                 \ the input is HIGH
        OVER TIMER STOP <                       \ and not enough time has elapsed
        ↪  to
        AND                                     \ declare the arrival of a
        ↪  STOP_BIT
    UNTIL
    R>
    2DROP ;
```

```
\ Samples a command sent to the receiver.
\ If command detection fails then 0 is returned.
: DETECT_COMMAND ( -- command_hex )
    0 COMMAND !                                 \ Resets the sampled command
    RECEIVER START_BIT                          \ Detects whether a START_BIT has
    ↪ arrived.
    NOT IF                                      \ If not
        0 EXIT                                  \ returns 0 and exits
    THEN
    20                                          \ Number of bits to be detected
    BEGIN
        1- DUP
    WHILE                                       \ While there are bits to detect
        RECEIVER DETECT_BIT                     \ Detects the bit
        DUP
        NOT IF                                  \ If it is not valid
            2DROP
            0 EXIT                              \ returns 0 and exits
        THEN
        ADD_BIT                                 \ adds the bit to the current
        ↪ command otherwise
    REPEAT
    DROP
    RECEIVER STOP_BIT                           \ Detects the STOP_BIT
    COMMAND @ ;                                 \ Returns the sampled command
```

4.2.9 lookup_table.f

```
\ Returns the string representation corresponding to a given code.
\ Note that ENDCASE eliminates the top of the stack, so to return
\ the default string to print we have to do a rotation and let
\ ENDCASE discard the value used for the lookup.
: LOOKUP ( code -- address length )
    CASE
        7070205F OF S" POWER ON/OFF"        ENDOF
        7070403F OF S" SOURCE"              ENDOF
        70706897 OF S" HDMI"                ENDOF
        7070106F OF S" ONE"                 ENDOF
        7070502F OF S" TWO"                 ENDOF
        7070304F OF S" THREE"               ENDOF
        70700877 OF S" FOUR"                ENDOF
        70704837 OF S" FIVE"                ENDOF
        70702857 OF S" SIX"                 ENDOF
        70701867 OF S" SEVEN"               ENDOF
        70705827 OF S" EIGHT"               ENDOF
        70703847 OF S" NINE"                ENDOF
        7070443B OF S" ZERO"                ENDOF
        70701A65 OF S" TTX/MIX"             ENDOF
        7070641B OF S" PREVIOUS CHANNEL"    ENDOF
        7070700F OF S" VOLUME UP"           ENDOF
        70706817 OF S" VOLUME DOWN"         ENDOF
        70707807 OF S" MUTE"                ENDOF
        70706B14 OF S" CHANNEL LIST"        ENDOF
        7070245B OF S" CHANNEL UP"          ENDOF
        7070047B OF S" CHANNEL DOWN"        ENDOF
        70704F30 OF S" SMART MODE"          ENDOF
```

```
        70707906 OF S" GUIDE"               ENDOF
        70702C53 OF S" MENU"                ENDOF
        70706916 OF S" TOOLS"               ENDOF
        70707C03 OF S" INFO"                ENDOF
        7070037C OF S" UP"                  ENDOF
        7070433C OF S" DOWN"                ENDOF
        7070532C OF S" LEFT"                ENDOF
        7070235C OF S" RIGHT"               ENDOF
        70700B74 OF S" OK/ENTER"            ENDOF
        70705A25 OF S" EXIT"                ENDOF
        70700D72 OF S" RETURN"              ENDOF
        70701B64 OF S" RED"                 ENDOF
        7070146B OF S" GREEN"               ENDOF
        7070542B OF S" YELLOW"              ENDOF
        7070344B OF S" BLUE"                ENDOF
        707031CE OF S" FAMILY STORY"        ENDOF
        70706718 OF S" SEARCH"              ENDOF
        70707C83 OF S" 3D"                  ENDOF
        70707E01 OF S" SUPPORT"             ENDOF
        70707887 OF S" D (WHAT IS IT?)"     ENDOF
        7070522D OF S" AD/SUBT."            ENDOF
        7070512E OF S" PREVIOUS TRACK"      ENDOF
        70700976 OF S" NEXT TRACK"          ENDOF
        70704936 OF S" RECORD"              ENDOF
        7070710E OF S" PLAY"                ENDOF
        70702956 OF S" PAUSE"               ENDOF
        7070314E OF S" STOP"                ENDOF
                 S" UNKNOWN COMMAND" ROT
    ENDCASE ;
```

4.2.10 main.f

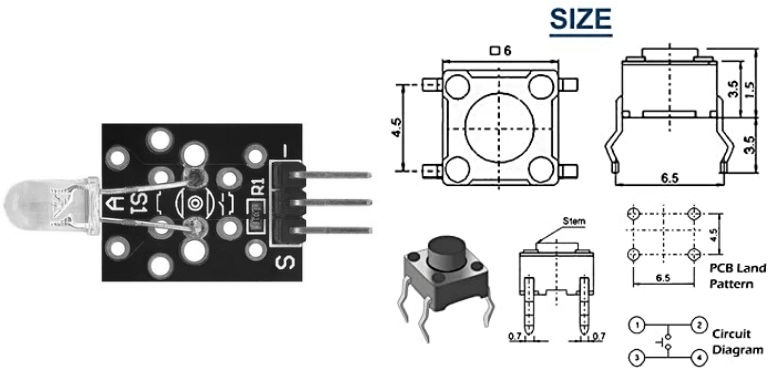
```
\ Creates variable to store the last sampled command.
VARIABLE LAST_COMMAND

\ Creates constant for 2 seconds delay.
1E8480 CONSTANT 2SECONDS

\ Welcome quote to start the program.
: QUOTE ( -- )
    S" `Learning never "    PRINT_STRING
    RH_LINE2
    S" exhausts the "       PRINT_STRING
    2SECONDS DELAY
    CLEAR_DISPLAY
    RH_LINE1
    S" mind.` "             PRINT_STRING
    RH_LINE2
    S" - Leonardo da "      PRINT_STRING
    2SECONDS DELAY
    CLEAR_DISPLAY
    RH_LINE1
    S" Vinci"               PRINT_STRING
    2SECONDS DELAY
    CLEAR_DISPLAY
    RH_LINE1
```

```
    S" Waiting for "        PRINT_STRING
    RH_LINE2
    S" a command... "       PRINT_STRING ;

\ The main program. It consists of a loop that prints a new command each time
\ one is encountered.
\ It prints the command in hexadecimal form in the first line of the LCD and its
↪  string
\ equivalent in the second line.
\ The display is updated only when a new command is found.
: MAIN ( -- )
    0 LAST_COMMAND !                        \ Resets the last sampled command
    INIT_I2C                                \ Initialize the I2C interface
    INIT_LCD                                \ Initialize the LCD
    INIT_RECEIVER                           \ Initialize the receiver

    QUOTE                                   \ Prints the quote

    BEGIN
        DETECT_COMMAND                      \ Detects the sampled command
        ?DUP IF                             \ If it is a valid command
            DUP
            LAST_COMMAND @
            <>                              \ Checks whether it is not equal to
            ↪  the last one
            IF                              \ If they are different
                CLEAR_DISPLAY               \ Clears the display
                RH_LINE1                    \ In the first line
                DUP PRINT_HEX               \ prints the new command in
                ↪  hexadecimal form
                DUP LAST_COMMAND !          \ Stores the new command
                RH_LINE2                    \ In the second line
                LOOKUP PRINT_STRING         \ prints the new command as a string
            ELSE
                DROP                        \ drops the sampled command,
                ↪  otherwise
            THEN
        THEN
    AGAIN ;
```

4.3 Possible improvements

The project could be extended to respond to IR signals, that is, to store the last sampled command and, through the use of a simple button (e.g. DAOKAI 4Pin button), decide when to send it to the TV using an IR transmitter (e.g. KY-005). In this way, the button acts as a trigger and the application is able to send and receive signals.

In addition to this improvement, one might consider using the Pi 4's green ACT LED to communicate with the user, for example, to signal that a command has been processed correctly or that something has gone wrong.



(a) KY-005 (b) DAOKAI 4Pin button

Figure 16: Additional hardware

References

[1] E. Upton. "Raspberry Pi 4 on sale now from $35." (2019), [Online]. Available: https://www.raspberrypi.org/blog/raspberry-pi-4-on-sale-now-from-35/.

[2] *Raspberry pi 4 product brief*, https://datasheets.raspberrypi.com/rpi4/raspberry-pi-4-product-brief.pdf.

[3] *Raspberry pi 4 gpio*, https://www.raspberrypi.com/documentation/computers/raspberry-pi.html.

[4] *Ftdi adapter ft232rl*, https://www.az-delivery.de/it/products/ftdi-adapter-ft232rl.

[5] *Elegoo documentation*, https://drive.google.com/file/d/1jfLQ6lDgM_gSl604r_zW7n1IvWrFUKN9/view.

[6] *Lcd 1602*, https://www.openhacks.com/uploadsproductos/eone-1602a1.pdf.

[7] *I2c lcd 1602*, https://wiki.52pi.com/index.php?title=Z-0234.

[8] *Pcf8574*, https://www.nxp.com/docs/en/data-sheet/PCF8574_PCF8574A.pdf.

[9] *Samsung protocol*, https://www.techdesign.be/projects/011/011_waves.htm.

[10] *Jonesforth-arm*, https://github.com/M2IHP13-admin/JonesForth-arm.

[11] *Pijforthos*, https://github.com/organix/pijFORTHos.

[12] *Minicom*, https://linux.die.net/man/1/minicom.

[13] *Picocom*, https://linux.die.net/man/8/picocom.

[14] *Ascii-xfr*, https://man7.org/linux/man-pages//man1/ascii-xfr.1.html.

[15] *Ansi compliance*, http://www.forth.org/svfig/Win32Forth/DPANS94.txt.