Politecnico di Torino

Electronic Engineering

Electronics for Embedded Systems Final Project

Author:
Nicola Vianello

Professor:
Claudio Passerone

March 8, 2019

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

The goal of this project is to design a parking sensor using all the topics seen during the course, i.e. programmable logic devices, memories, interconnections, processor peripherals, AD and DA conversions, and power management[10].

The core of the system is a PIC24FJ64GB004 microcontroller that has the task of managing four ultrasonic transmitter-receiver couples HC-SR04, thought up to cover both the front and the rear of the vehicle, as shown in figure 1.1. By cyclically enabling the four sensors and computing for each of them the time elapsed from the emission of the ultrasounds to their subsequent detection following the reflection on the surface of a potential obstacle (see figure 1.2), it is possible to trace the distance between the sensor and this obstacle[1]. Consequently the four distances measured in this way are displayed on an alphanumeric LCD with a Hitachi HD44780-like controller,

[1] From elementary kinematics calculations we find $d = (V_s \times \Delta t)/2$, where d is the distance between the sensor and the obstacle, V_s is the speed of sound, Δt is the measured time interval, and factor 2 appears because the wave travels in both directions before being detected by the sensor[6].

Figure 1.1: Sensor positioning on a car.

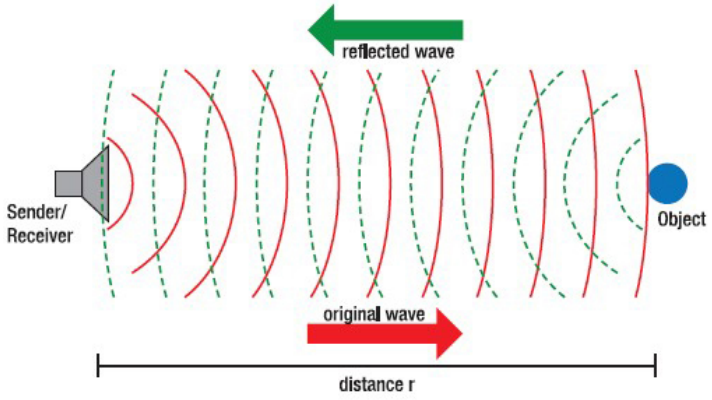


Figure 1.2: Emission, reflection and detection of an ultrasound.

where the most critical of the four distances will also be highlighted. The shorter distance will also be used to drive a buzzer intermittently with a frequency that increases as this distance decreases, up to become a continuous sound.

Since the speed of sound varies depending on the ambient temperature, in order to increase the accuracy of the measured distances there will be also an analog temperature sensor LM35 whose output, converted into digital format by an ADC0804 digital to analog converter, will be periodically read by the microcontroller which can compute the speed of sound at that specific temperature[2].

Although the task of a usual parking sensor can be carry out using only these components, it will also be included the possibility to request from a PC via RS232 interface all the temperature readings made in the last minutes, to do this a DE0 board will be used which contains an FPGA where a RAM memory and a finite state machine will be implemented, the latter will continuously store the temperature digital values according to a FIFO policy and, as soon as a control signal is received from the PC, the FSM will transmit them using the UART protocol. Since the used board has not already implemented a circuit for adapting the voltages and an RS232 connector yet, instead of this interface, an adaptor that reads and writes

[2]In linear approximation the speed of sound in the air varies according to the law: $V_s(T) = 0.62\,\mathrm{m/(s \times ^\circ C)} \times T + 331.45\,\mathrm{m/s}$, where V_s is the speed of sound and T is the temperature in Celsius degree, so for example from a temperature of 0 °C to a temperature of 30 °C, easily detectable excursion in the earth, there is a variation of the speed of sound and consequently of the measured distance of about 5.6%.

the data using a voltage compatible with the GPIO pins of the FPGA (0V-3.3V) and that get connected to the PC via a USB port will be used. This adaptor is seen from the PC as a normal serial port therefore there isn't any difference both from the point of view of the written VHDL code and the data reading from the PC.

Regarding the power supply, since the circuit is designed to be installed on board a vehicle it will be powered with 12V (like the voltage of a medium charged car battery), from this voltage 5.1 V will be obtained through a L4960 switching regulator to power all the components except the microcontroller, which will be powered with 3.3V obtained from the 5.1V through a LD1117V33 low-dropout linear regulator.

In figure 1.3 it is possible to see a block diagram of the used components and their connections.

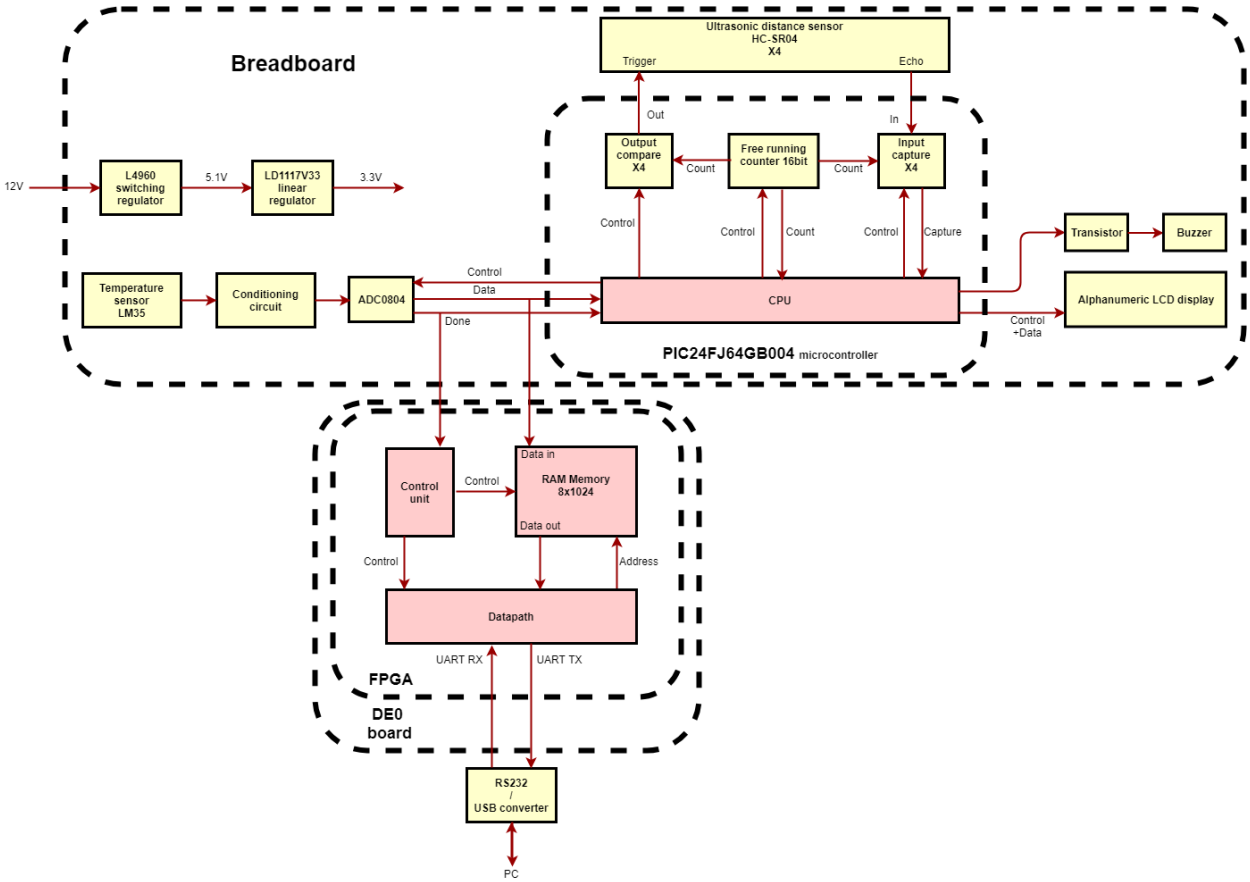


Figure 1.3: Block diagram of the used components and their connections.

Chapter 2

Used components

2.1 Microcontroller

The chosen microcontroller is a PIC24FJ64GB004, 16 bit PIC24 family device produced by Microchip Technology Inc.[8]. It is not produced in DIP format, therefore it will be used in a 44-TQFP package and soldered on a small PCB that allows the connection to the breadboard through some ad-hoc made connectors. This device has been chosen as it has a sufficient amount of peripherals, program memory, and input/output pins to implement the system and at the same time it minimizes the unused resources. In fact inside it we find:

- 64 kB of flash memory for the program and 8 kB of RAM memory for the data, both largely sufficient for the source code.
- 33 remappable input/output pins: 6 pins are used for the LCD display (as shown below: four for the data, one for the enable, and one for the register selector); 2 pins for each ultrasonic sensor (one to enable the transmitter and one to read the receiver) for a total of 8 pins; 10 pins are for the ADC (eight for the data, one for the start control, and one for the done signal); 1 pin is used to drive the buzzer; 2 pins are needed to make a 24 MHz quartz oscillate.
- 5 input capture channels, of which four are used to read the sensors.
- 5 output compare channels, of which four are used to cyclically enable the four sensors, and one is used to run a periodic polling routine every 50 ms.

- 5 timers, of which one has 16 bit and the remaining four can be used as four 16 bit timers or as two 32 bit timers: only the first is used and it is configured to be the reference timer of all input capture and output compare channels.
- 3 sources of external interrupt, only one is used and it is connected to the ADC done signal in order to read the converted value as soon as it is available (actually the microcontroller already implements a 10 bit ADC inside itself, but the external one is used for didactic reasons).

Moreover, one of the peculiar characteristics of this microcontrollers family is the possibility to map in runtime most peripherals to any input/output pin. In fact, in figure 2.1 it is possible to notice how in the pinout shown in the datasheet of the component there are no pins associated to the input capture or the output compare.

In conclusion, to connect the 5 V outputs of the sensors and the ADC directly to the inputs of the microcontroller (which works at 3.3V) without using signal conditioning circuits, the pins marked in gray in the pinout (Fig. 2.1) is used, in fact they already implement this circuit inside the microcontroller and support input voltages up to 5.5 V.

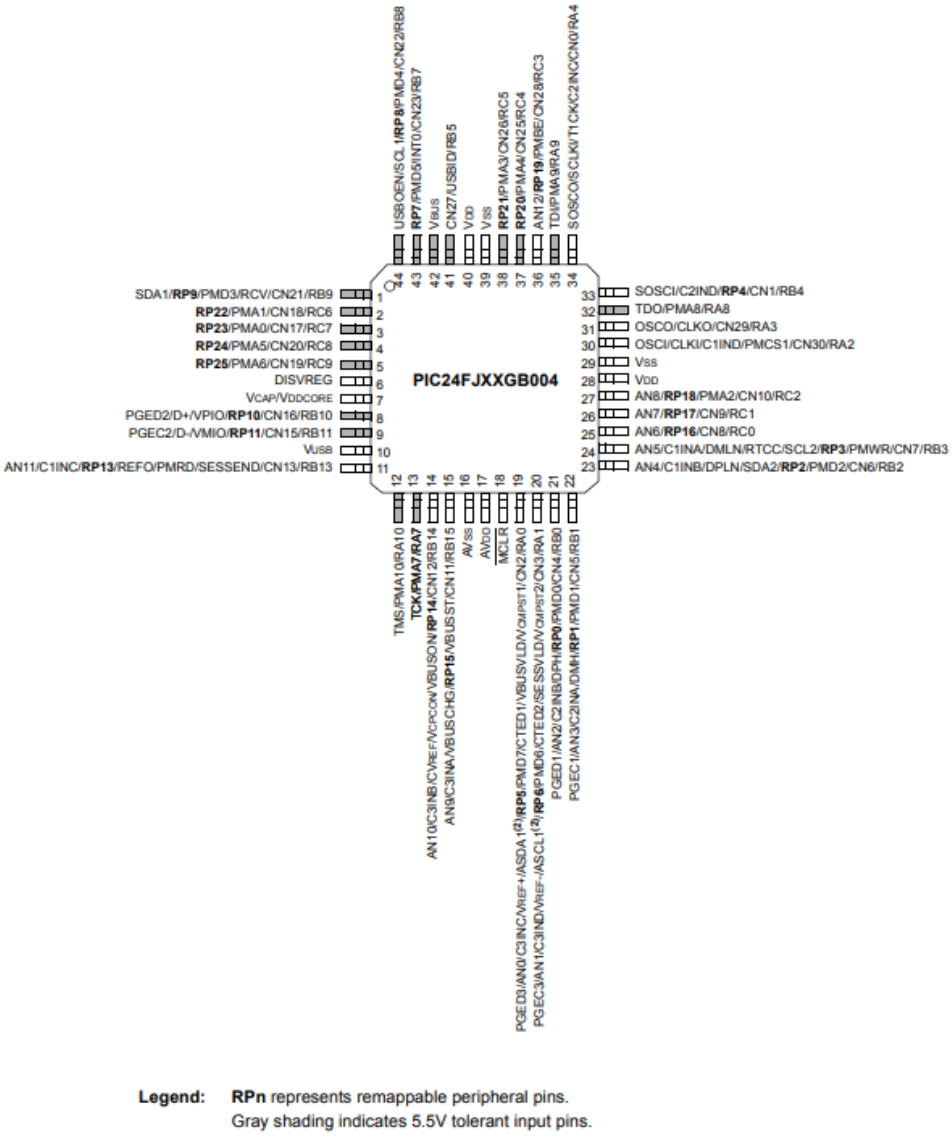
The developed source code are explained and shown in Section 4.1.

2.2 Ultrasonic sensors HC-SR04

The used sensors[4], shown in figure 2.2, integrate in the same device an ultrasonic transmitting capsule, an ultrasonic receiving capsule, and some integrated circuits that manage the sensor. Externally there are four pins, of which two are needed for the 5 V power supply, one is an input called *Trigger*, and one is an output called *Echo*. The functioning is the following one (also shown in the timing diagram of figure 2.3):

1. A TTL-compatible positive pulse of at least 10 µs must be applied to the *Trigger* input to enable the sensor.
2. The sensor will emit eight 40 kHz ultrasound pulses.
3. The Echo signal is raised up to the supply voltage.
4. When the ultrasounds return to the sensor the *Echo* signal is cleared.

This means that to measure the time elapsed from the emission of the ultrasounds upon their return it is sufficient, once enabled the sensor, to



Legend: **RPn** represents remappable peripheral pins.
Gray shading indicates 5.5V tolerant input pins.

Figure 2.1: PIC24FJ64GB004 pinout.

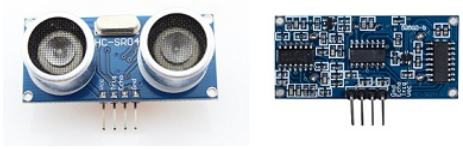


Figure 2.2: Front and back of the sensor HC-SR04.

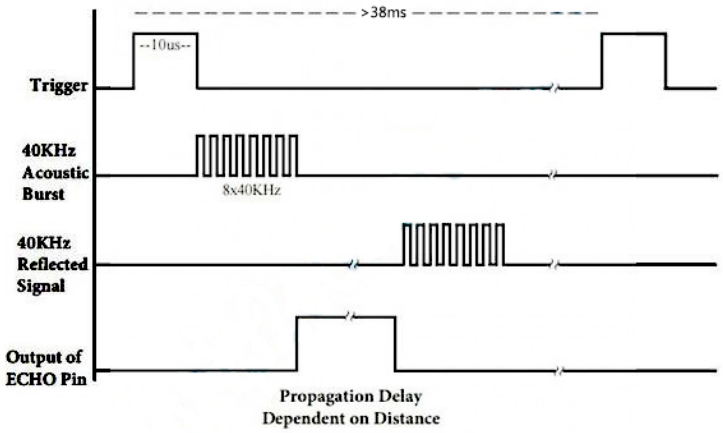


Figure 2.3: HC-SR04 functioning timing diagram.

make a measurement on the width of the *Echo* pulse. Furthermore, in case that no soundwave returns to the sensor within 38 ms, it will lower the *Echo* signal down in any case after this period of time.

As already mentioned, to read the *Echo* signal the 5.5 V tolerant microcontroller pins will be used, whereas to drive the *Trigger* no precautions are necessary because it can read 3.3 V inputs also.

2.3 Alphanumeric LCD

The display used in this project is a 2004A[11] produced by ShenZhen Etong ElectronicsCo. Ltd., it is equipped with 4 rows and 20 columns and uses a driver completely compatible with the Hitachi HD44780 driver[5] which is nowadays standard de facto to drive the liquid crystals in the alphanumeric LCD. Obviously any other display with the same driver can be used instead of this, without any adaptation if the number of rows and columns remains the same, or with some minor adjustments in the microcontroller source code if the number of rows and columns is different.

The driver can be driven in two modes: the 4 bit mode, and the 8 bit mode. The first one, used in this project, has the advantage of requiring fewer pins but requires a slightly more complicated writing process, as will be explained below.

Externally, the display has 16 pins whose function is explained in the table 2.1. Since D0, D1, D2, D3 are not used and RAM reading is not required in this specific application, these four data inputs and the RW

Table 2.1: LCD pinout.

Pin	Name	Function
1	V_{ss}	GND logic power supply.
2	V_{dd}	+5 V logic power supply.
3	V_0	Contrast adjust: by connecting a variable voltage from a trimmer it is possible to adjust the contrast of the display.
4	RS	Register selector: if set to 0 it allows to send commands to the driver, if set to 1 it allows to write in the RAM memory where the characters are stored.
5	RW	Read/Write: if set to 0 it is possible to write in the display or send commands, if set to 1 it is possible to read the written characters.
6	E	Enable: when a pulse is detected on this input, the data bus is sampled and the corresponding action is performed depending on the value of RS and RW.
7	D0	Used only in 8 bit mode.
8	D1	Used only in 8 bit mode.
9	D2	Used only in 8 bit mode.
10	D3	Used only in 8 bit mode.
11	D4	Bit 0 (LSB) of the data bus in the 4 bit mode.
12	D5	Bit 1 of the data bus in the 4 bit mode.
13	D6	Bit 2 of the data bus in the 4 bit mode.
14	D7	Bit 3 (MSB) of the data bus in the 4 bit mode.
15	A	Backlight LEDs anode.
16	K	Backlight LEDs cathode.

input will be hardwired to GND.

After configuring the driver to work in 4 bit mode (see lcd_init function in Listing 4.6 for the specific procedure), it will be necessary, both to write in the display (RS=1) and to execute commands (RS=0), to send a sequence of two nibbles in the data bus starting from the most significant and confirming both nibbles with a pulse on the enable pin (as shown in Figure 2.4). In the first case it will be necessary to send the two nibbles that make up the character to be written in ASCII code, in the second case you will have to send one of the commands shown in table 2.2 also divided into two nibbles.

Each character written in the display is stored in a RAM memory at the address pointed by a register inside the driver and every time a character is written this pointer is auto-increased of one unit. This RAM memory will be read by the driver in order to control the liquid crystals according to an injection function that associates a specific memory cell to each position in the display. This function varies depending on the number of rows of the display (For the exact correspondence according to the number of rows, refer to the lcd_goto function in Listing 4.6), but in any case the memory

Table 2.2: All the possible instructions of the HD44780 driver. X=don't care.

Instruction	RS	RW	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	Description
Clear display	0	0	0	0	0	0	0	0	0	1	Resets RAM and writing address.
Return Home	0	0	0	0	0	0	0	0	1	X	Resets writing address.
Entry mode set	0	0	0	0	0	0	0	1	I/D	S	S enables the shift of the entire display, I/D sets the cursor moving direction.
Display ON/OFF control	0	0	0	0	0	0	1	D	C	B	D switches on/off the display, C activates/deactivates the cursor, B enables the blinking of the cursor.
Cursor or display shifted	0	0	0	0	0	1	S/C	R/L	X	X	S/C enables the shift of the cursor, R/L selects the shift direction of the characters.
Function set	0	0	0	0	1	DL	N	F	X	X	DL sets the display in the 4 bit mode or in the 8 bit mode, N sets the number of display lines, F sets the font type of the display.
Set CGRAM address	0	0	0	1	AC5	AC4	AC3	AC2	AC1	AC0	Allows to draw characters not natively supported.
Set DDRAM address	0	0	1	AC6	AC5	AC4	AC3	AC2	AC1	AC0	Sets the RAM memory pointer to the desired value.
Read busy flag and address counter	0	1	BF	AC6	AC5	AC4	AC3	AC2	AC1	AC0	BF is 1 if the driver is busy, and the content of the RAM memory pointer can also be read.
Write data to RAM	1	0	D7	D6	D5	D4	D3	D2	D1	D0	Writes a character in RAM memory to the pointed address.
Read data from RAM	1	1	D7	D6	D5	D4	D3	D2	D1	D0	Reads the pointed character from the RAM memory.

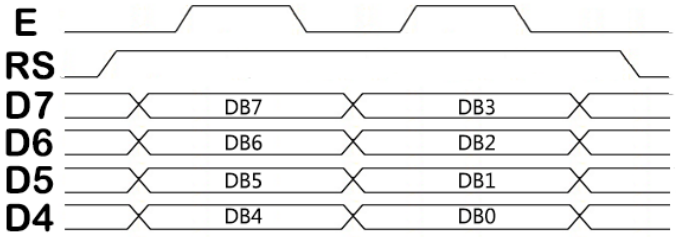


Figure 2.4: Timing diagram for the writing of a character in an alphanumeric LCD display with the HD44780 driver in 4 bit mode, the timing diagram for sending a command is the same but setting RS = 0.

cells associated with a row are sequential, so, in order to write a string in the display, you need to set the pointer by a special command to make the writing begin in the desired position and then send in sequence all the characters of the string, taking care to don't write a number of characters greater than the number of positions remaining in the line.

Also this component supports 3.3 V inputs, so adapting circuits are not necessary.

2.4 Buzzer and driving transistor

The used buzzer is a PCB-mounting magnetic one and it is active, this means that it is sufficient to supply its two pins with a voltage of about 5 V to make it oscillate at a frequency predetermined by the manufacturer. This choice is due to the fact that it is not necessary to be able to vary the frequency of the sound in this application. Since the buzzer must be powered at 5 V and the microcontroller output is at 3.3 V it is necessary to use a driving transistor. Actually, as already mentioned, the power supply is a bit higher than 5 V and equal to 5.1 V but still within the tolerance limits of the buzzer. With this supply voltage we can measure a current flowing in the buzzer of 27 mA, so a BJT 2N3904[9] in low side driving (see Figure 2.5) can be fine for the current, the voltage, and the power involved.

From the datasheet it is possible to read that the minimum beta of the transistor is 60 in these conditions, rounding up the current in the buzzer to 30 mA (actually the voltage drop between collector and emitter makes the load current a little smaller than 27 mA but it is negligible because what we will find is an upper bound for the base resistance) and forcing a base current 10 times higher than in linearity it is found that the saturation base

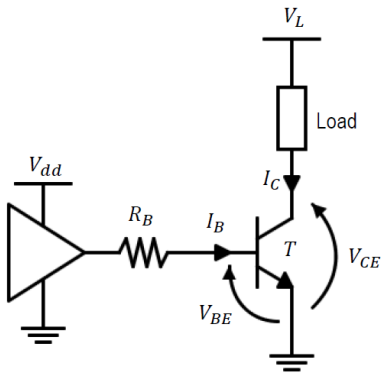


Figure 2.5: Buzzer driving circuit.

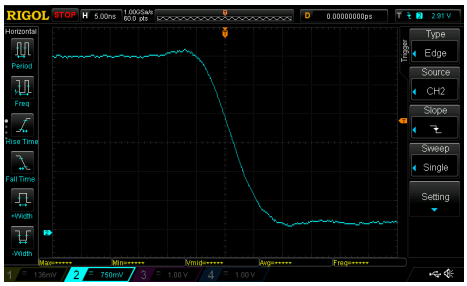
Table 2.3: Measured transistor delays and response times.

Falling time	Rising time	High-to-low delay	Low-to-high delay
10 ns	7 µs	10.8 ns	2.5 µs

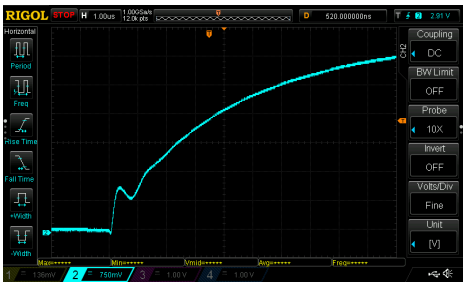
current must be at least 5 mA. The minimum voltage that can be output by the microcontroller is 2.4 V when high, while the maximum voltage between base and emitter of the saturated transistor is 0.95 V, this means that the base resistance must not be higher than $(2.4\,\text{V} - 0.95\,\text{V})/5\,\text{mA} = 290\,\Omega$. To find the lower bound we read again from the datasheets that the maximum current that can be output from a pin is 25 mA and the minimum base-emitter saturation voltage is 0.65 V, so the base resistance should not be lower than $(3.3\,\text{V} - 0.65\,\text{V})/25\,\text{mA} = 106\,\Omega$. Since a strong saturation is not necessary a 270 Ω resistor will be used in order to stress as little as possible the switching regulator that will have to work despite breadboard's parasitic impedances. With this value the maximum dissipated power by the resistance is $(3.3\,\text{V} - 0.65\,\text{V})^2/[270\,\Omega \cdot (1 - 5\%)] = 27\,\text{mW}$ so a normal 1/4 W resistor can be fine.

Regarding the transistor cut-off, the maximum output voltage from a microcontroller pin is 0.4 V when low, always less than the minimum $V_{be(on)}$, so the switching off is also guaranteed.

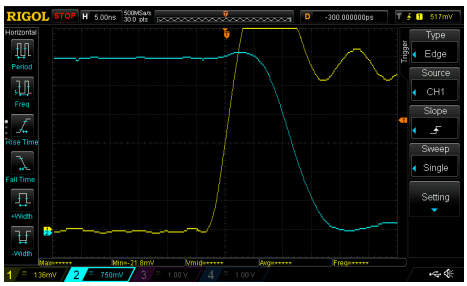
Once the transistor is selected and the base resistance is dimensioned, the rise and fall times and the high-to-low and low-to-high delays have been measured using the oscilloscope screenshots in Figure 2.6, the results are shown in Table 2.3.



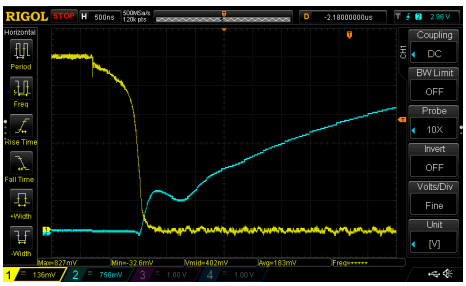
(a) *Falling time.*



(b) *Rising time.*



(c) *High-to-low delay.*



(d) *Low-to-high delay.*

Figure 2.6: Transistor delays and response times seen on the oscilloscope.

2.5 Temperature sensor and conditioning circuit

The chosen temperature sensor is a LM35[15] in TO-92 package. It has two power supply pins and an output pin in which there is a voltage of 10 mV for each Celsius degree. Since there is only a single supply and the sensor output is very close to 0 V, the selected operational amplifier to adapt the signal is an OPA347[16] which has a rail-to-rail input and output and can be powered with a single supply of 5.1 V, moreover, in this way we can have a wider output dynamics and therefore higher precision.

The employed configuration is the non-inverting one shown in Figure 2.7. We choose to read a temperature range from 0 °C to 50 °C, this means that when in input is present a voltage of 0.5 V the output must be equal to 5 V and therefore the gain of the circuit must be 5 V/0.5 V = 10. Recalling that the gain of an operational amplifier in non-inverting configuration is equal to A=1+Rf/R1 and choosing Rf=22 kΩ (which is a reasonable value taking into account the offset current and the maximum output current of the op-amp) we find R1=2.44 kΩ, rounding up to the nearest normalized value we finally have R1=2.7 kΩ, for an effective gain of A=1+22 kΩ/2.7 kΩ=9.15.

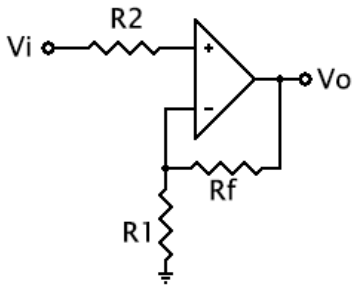


Figure 2.7: Signal conditioning circuit.

R2 is used to limit the effects of the bias current and it must have a value as close as possible to the parallel between R1 and Rf, since R1//Rf=2.4 kΩ is not a normalized value, a 2.2 kΩ resistor is chosen for this role.

2.6 Analog-to-digital-converter

ADC0804[14] is an 8 bit successive approssimation converter that use a potentiometric ladder as internal DAC. The clock frequency can be set externally through an RC network, but in this case the values recommended by the datasheet is maintained, that are 10 kΩ and 150 pF which allow to obtain a frequency of 640 kHz, more than enough for this application.

Since the microcontroller V_{OH} is not high enough to control the start signal, whereas the V_{IL} of the ADC has a wide margin, the circuit in figure 2.8 is used which shifts the output voltage up of about 0.7V making it fall into both input ranges[7]. In Table 2.4 are reported the voltage thresholds for the 5 V input and the 3.3 V output with and without the shifting circuit. As D_1 and D_2 we can choose a 1N4148[17] signal diode, which supports a maximum forward current of 300 mA. We know from the datasheet that in order to have a V_{OH} of 3 V the maximum current sinkable by a microcontroller pin must be 3 mA, considering a voltage drop on the diode of 0.6 V, it means that $R_1 > (5.1\,V - 3\,V - 0.6\,V)/3\,mA = 500\,\Omega$. On the other hand, when the ADC input voltage is as high as possible, it is necessary to make the current flowing in the resistor negligible with respect to the maximum input current, which is 1 µA. The maximum diode forward voltage is 1 V, so: $R_1 << (5.1\,V - 3.3\,V - 1\,V)/1\,\mu A = 800\,k\Omega$. In conclusion 68 kΩ can be fine.

Before mounting the circuit, all the input thresholds that make the ADC output change have been measured, in order to characterize it and to cal-

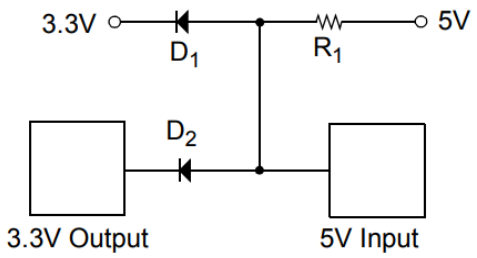


Figure 2.8: Output voltage shifting circuit.

Table 2.4: Voltage threshold with and without shifting circuit, assuming V_{D_2}=0.7V.

	ADC input	Microcontroller output	Microcontroller output with shifting circuit
High threshold	>3.5 V	>3 V	>3.7 V
Low threshold	<1.5 V	<0.4 V	<1.1 V

culate all the static errors. To measure them the ADC was connected to a breadboard in free-running configuration, and a voltage generator have been connected to its input and to a voltmeter, while the oscilloscope showed the least significant bit of the output. Starting from 0 V the input voltage was changed in steps of 1 mV until an oscillation of the least significant bit (like the one shown in figure 2.9) was displayed on the oscilloscope, at that point the voltage read by the voltmeter was recorded and so on for all 255 thresholds. The taken measures are reported in the Table 2.5 and in the graph of Figure 2.10 where the ideal transfer function and the best linear approximation are also shown. Starting from these measures and the ADC voltage reference, equal to 5.126 V, it was easily possible to find the voltage values at the center of the analog segments, and from these, the DNL, the INL, the offset error and the gain error. The results are shown in Table 2.6 where they are compared to the ideal and the datasheet values. It's possible to see that m, q, and A_d are quite close to the ideal values, DNL is within the limit established by the manufacturer, but the INL is not compatible with what is declared by the datasheet. This may be due to the poor quality of the used voltmeter and partially also to the inability of the voltage generator to change its output by an amounts smaller than 1 mV.

Regarding the dynamic properties, it has been possible only to measure the time elapsed from the start signal to the done signal, since for the other characteristics a logic analyzer would have been necessary. This period, as can be seen in Figure 2.11, is 168 µs. The datasheet reports a maximum conversion time of 114 µs with a clock frequency of 640 kHz, this difference

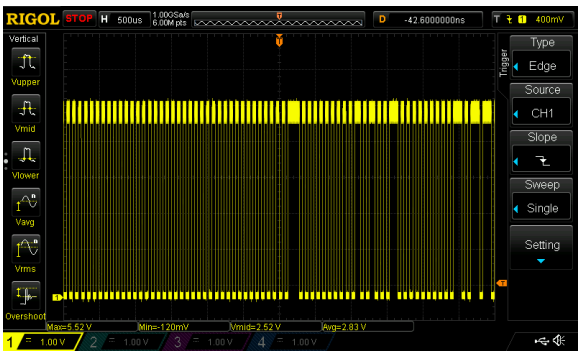


Figure 2.9: Oscillation of the ADC least significant bit at which the voltage threshold is measured.

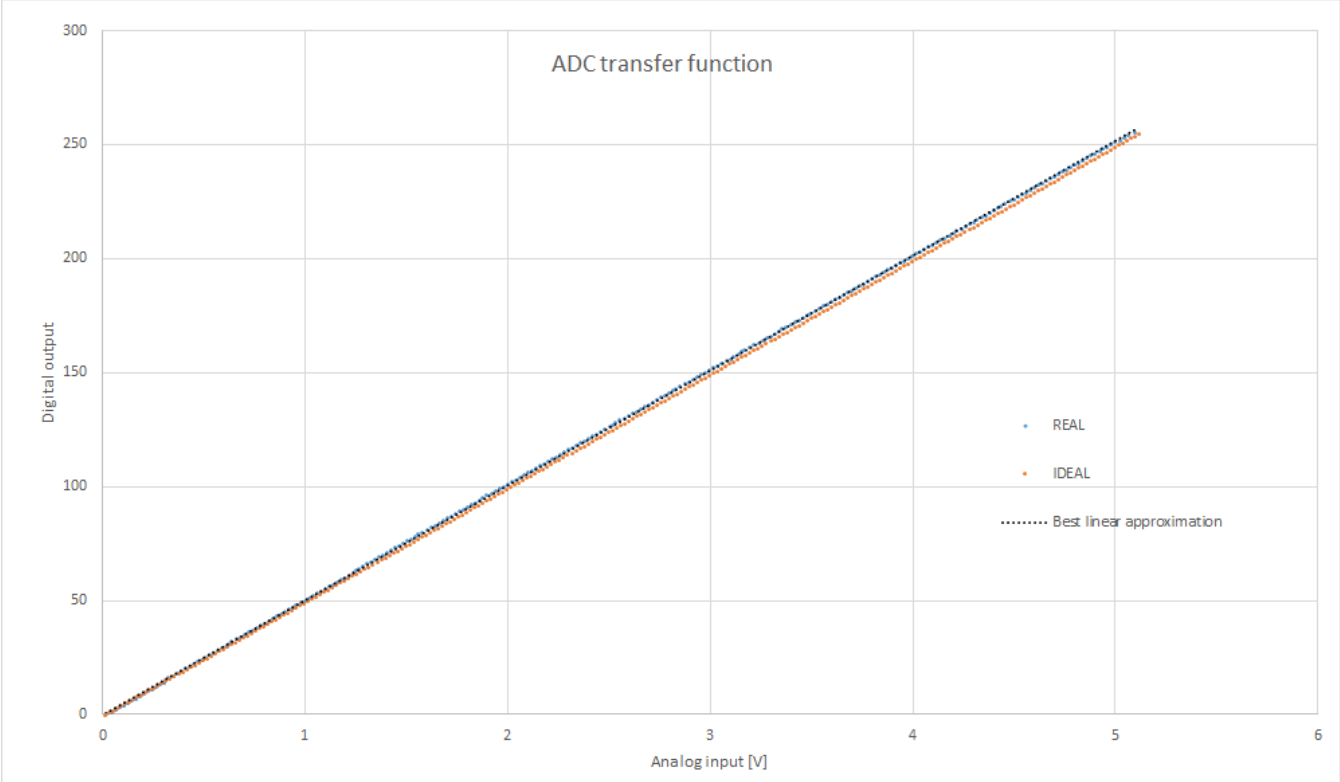


Figure 2.10: Real and ideal ADC transfer functions and best linear approximation.

Table 2.5: ADC voltage thresholds measurements.

N	Measure [V]	N	Measure [V]	N	Measure [V]	N	Measure [V]	N	Measure [V]
1	0.0347	52	1.027	103	2.032	154	3.046	205	4.068
2	0.0552	53	1.047	104	2.052	155	3.068	206	4.088
3	0.0758	54	1.067	105	2.071	156	3.090	207	4.108
4	0.0946	55	1.087	106	2.091	157	3.107	208	4.121
5	0.1137	56	1.108	107	2.112	158	3.129	209	4.148
6	0.1348	57	1.127	108	2.134	159	3.150	210	4.168
7	0.1558	58	1.147	109	2.151	160	3.155	211	4.190
8	0.1748	59	1.167	110	2.172	161	3.187	212	4.209
9	0.1934	60	1.188	111	2.192	162	3.206	213	4.228
10	0.2134	61	1.207	112	2.207	163	3.228	214	4.250
11	0.2345	62	1.228	113	2.230	164	3.248	215	4.270
12	0.2541	63	1.247	114	2.250	165	3.267	216	4.290
13	0.2727	64	1.253	115	2.271	166	3.288	217	4.308
14	0.2933	65	1.272	116	2.291	167	3.308	218	4.328
15	0.3116	66	1.292	117	2.310	168	3.328	219	4.350
16	0.3144	67	1.313	118	2.331	169	3.346	220	4.370
17	0.3407	68	1.332	119	2.351	170	3.366	221	4.388
18	0.3608	69	1.352	120	2.372	171	3.387	222	4.409
19	0.3815	70	1.373	121	2.390	172	3.409	223	4.430
20	0.4004	71	1.393	122	2.410	173	3.427	224	4.440
21	0.4195	72	1.413	123	2.430	174	3.448	225	4.469
22	0.4406	73	1.432	124	2.452	175	3.469	226	4.488
23	0.4608	74	1.452	125	2.471	176	3.484	227	4.510
24	0.4808	75	1.474	126	2.492	177	3.507	228	4.530
25	0.4989	76	1.493	127	2.512	178	3.527	229	4.549
26	0.5179	77	1.513	128	2.523	179	3.548	230	4.571
27	0.5400	78	1.534	129	2.544	180	3.567	231	4.590
28	0.5591	79	1.554	130	2.564	181	3.587	232	4.610
29	0.5771	80	1.565	131	2.586	182	3.608	233	4.630
30	0.5986	81	1.591	132	2.605	183	3.629	234	4.649
31	0.618	82	1.611	133	2.625	184	3.649	235	4.671
32	0.622	83	1.633	134	2.646	185	3.667	236	4.691
33	0.652	84	1.652	135	2.666	186	3.686	237	4.709
34	0.670	85	1.671	136	2.685	187	3.707	238	4.730
35	0.692	86	1.692	137	2.704	188	3.729	239	4.750
36	0.711	87	1.713	138	2.724	189	3.747	240	4.767
37	0.730	88	1.733	139	2.745	190	3.768	241	4.789
38	0.752	89	1.751	140	2.766	191	3.788	242	4.810
39	0.772	90	1.771	141	2.784	192	3.811	243	4.831
40	0.791	91	1.793	142	2.806	193	3.827	244	4.850
41	0.810	92	1.814	143	2.826	194	3.847	245	4.870
42	0.830	93	1.832	144	2.839	195	3.869	246	4.891
43	0.851	94	1.853	145	2.865	196	3.888	247	4.911
44	0.872	95	1.873	146	2.886	197	3.907	248	4.932
45	0.890	96	1.882	147	2.907	198	3.929	249	4.950
46	0.911	97	1.910	148	2.927	199	3.949	250	4.970
47	0.932	98	1.930	149	2.946	200	3.968	251	4.990
48	0.944	99	1.951	150	2.968	201	3.987	252	5.011
49	0.967	100	1.971	151	2.988	202	4.007	253	5.031
50	0.987	101	1.991	152	3.008	203	4.029	254	5.051
51	1.008	102	2.011	153	3.026	204	4.048	255	5.071

Table 2.6: ADC linear coefficients and static errors.

	Real	Ideal	Datasheet [max]
m [V^{-1}]	50.343	49.941	-
q [LSB]	-0.176	-0.5	-
A_d [mV]	19.864	20.023	-
ε_g [V^{-1}]	-0.401	-	Non reported
ε_{off} [LSB]	-0.324	-	Non reported
DNL [A_d]	0.899	-	1
INL [LSB]	1.2556	-	1

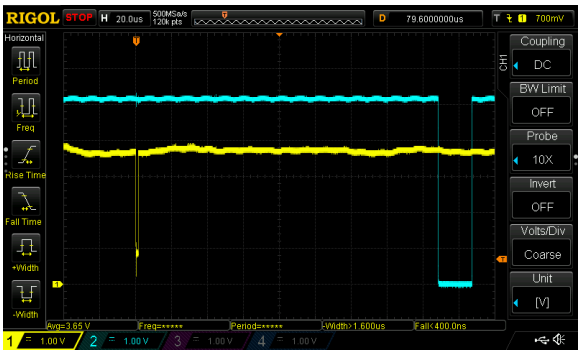


Figure 2.11: ADC start and done signal.

may be due either to a slightly different clock frequency (it hasn't been possible to measure it because the oscilloscope probe drastically change the oscillation frequency of the RC network), and to the asynchronous start pulse, which can take up to 8 additional clock cycles before starting the conversion (see page 5 of the datasheet [14]).

2.7 5.1V switching regulator

L4960[12] is a monolithic switching regulator delivering 2.5 A at a voltage variable from 5.1 V to 40 V in step down configuration. Since it can not supply exact 5 V, we will use the lower possible voltage, that is 5.1 V, connecting directly the output of the filter to the voltage reference feedback, insured by the fact that this voltage is within the absolute maximum ratings of all the used components.

The total current sinked by the circuit was measured equal to 66 mA when the buzzer is continuously active. We choose a 50% safety margin,

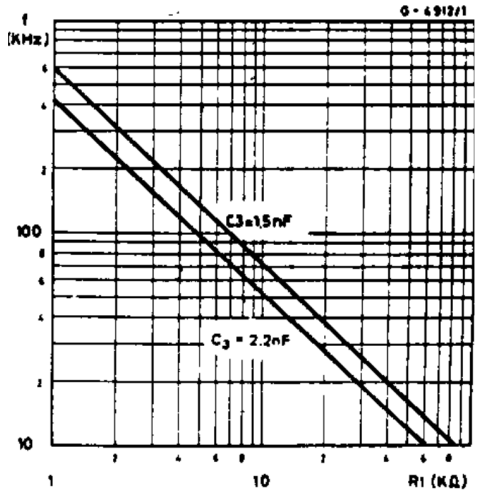


Figure 2.12: Switching frequency VS parallel RC network.

so the components will be dimensioned for a current of 100 mA. As output ripple we choose 1% of the voltage, that is 50 mV, a so low value is easily to reach because the involved current and voltage are quite low. Regarding the switching frequency we could choose a rather high value, for the same reasons why it is possible to choose a low ripple, but since the maximum supported frequency is 150 kHz we choose exactly this value. To established this frequency, it is necessary to connect a parallel RC network between a pin of the integrated circuit and ground, according to the graph of Figure 2.12. Using a 2.2 nF capacitor the resistance value for which we have a switching frequency as close as possible to 150 kHz but without exceeding this value is 3.3 kΩ, with these components we measure a frequency of 135 kHz using the oscilloscope (Figure 2.13). As low side switching diode we can choose a 1N5819[3] that is a schottky diode and it support until 40 V and 1 A, and it has a typical reverse recovery time of 10 ns.

From these specifications we can finally dimension the inductance and the output capacitor in order to make the regulator work constantly in CCM mode and to not exceed the required ripple:

$$L > \frac{5.1\,\text{V}}{2 \cdot 100\,\text{mA} \cdot 135\,\text{kHz}} = 188.9\,\mu\text{H}$$

$$C > \frac{100\,\text{mA}}{4 \cdot 50\,\text{mV} \cdot 135\,\text{kHz}} = 3.7\,\mu\text{F}$$

Obviously these values are not normalized, so 220 µH and 4.7 µF are used.

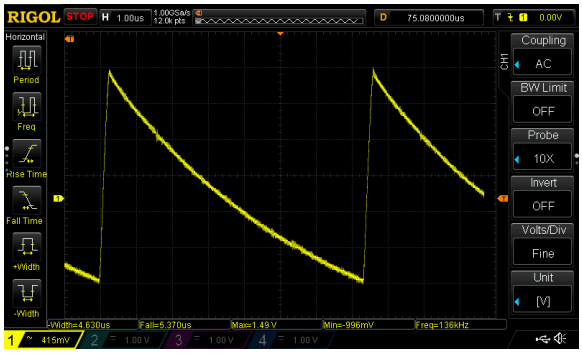


Figure 2.13: Switching frequency measurement.

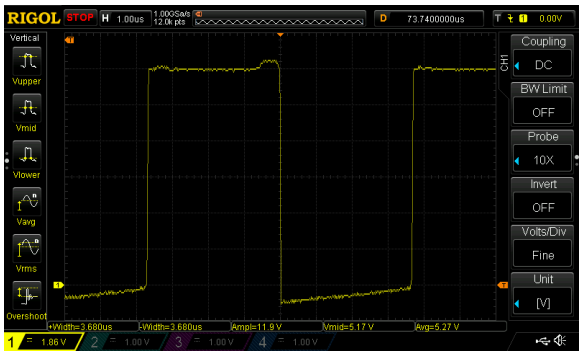


Figure 2.14: Regulator output before the filter.

The input capacitor does not have particular requirements because the input voltage already comes from a continuous source therefore the 100 µF value used in the datasheet test circuit has been maintained.

In Figure 2.14, where is shown an oscilloscope view of the regulator output (before the filter), it is possible to see that the regulator works in CCM mode, in fact, if it worked in DCM mode, there would be evident oscillations at the switching. In Figure 2.15 there is a screenshot of the output ripple (after the filter), with which it is possible to verify that it is about 38 mV and absolutely within the established limit.

We can also compute the RMS current in the inductor and in the output capacitor to be sure that the used components are supported:

$$I_{inductor}^{RMS} = \frac{2}{\sqrt{3}} \cdot 100\,\text{mA} = 115.470\,\text{mA}$$

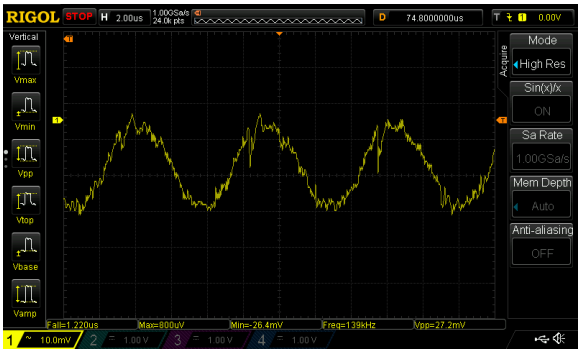


Figure 2.15: Output ripple after the filter.

$$I_{capacitor}^{RMS} = \frac{1}{\sqrt{3}} \cdot 100\,\text{mA} = 57.735\,\text{mA}$$

2.8 3.3V linear regulator

Since there is already 5.1 V generated by the switching regulator, to generate the 3.3 V for the microcontroller we will use a low dropout linear regulator. The voltage drop is $5.1\,\text{V} - 3.3\,\text{V} = 1.8\,\text{V}$ so an LD1117V33[13] is fine, which has a maximum dropout of 1.1 V with an output current of 100 mA. The values of the input and output capacitors will be maintained equal to those suggested by the datasheet: 100 nF for the input and 10 µF for the output.

2.9 FPGA

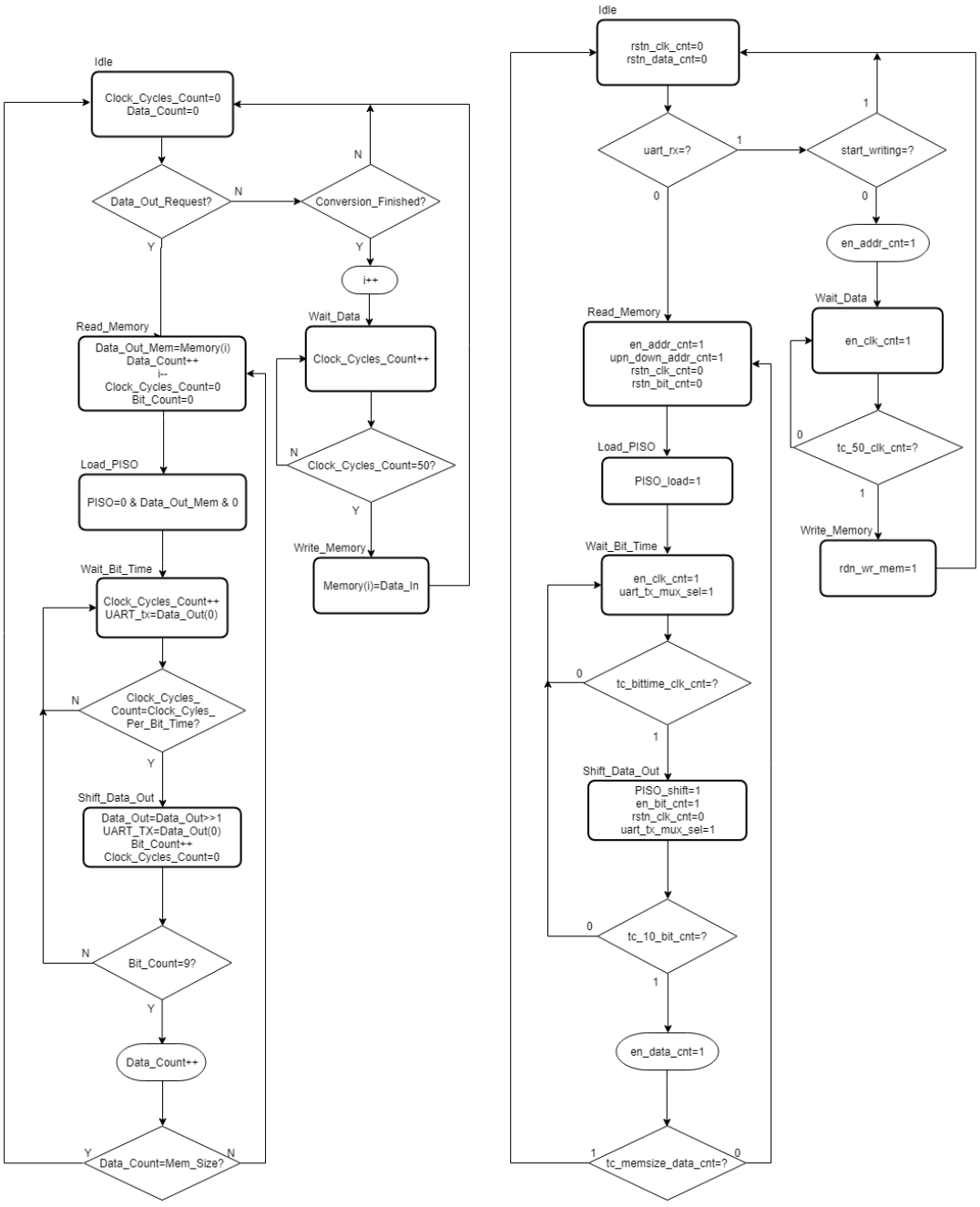
The FPGA of the DE0 Board[2] is a 5CSEMA4U23C6 of the Cyclone 5 family[1], clocked by 50 MHz.

Essentially three interconnected components will be implemented:

1. The control unit, that is an FSM developed using the algorithmic state machine method, Figure 2.16-a shows the ASM chart and Figure 2.16-b shows the control chart.

2. The datapath driven by the control unit outputs and from which some control signals are read, the scheme is shown in figure 2.17.

3. An 8 bit RAM memory with 1024 locations, whose connection with the previous two modules is also illustrated in Figure 2.17.



(a) *Algorithmic state machine chart.*

(b) *Control chart.* "&" *represent the concatenation operator.*

Figure 2.16: ASM chart and control chart.

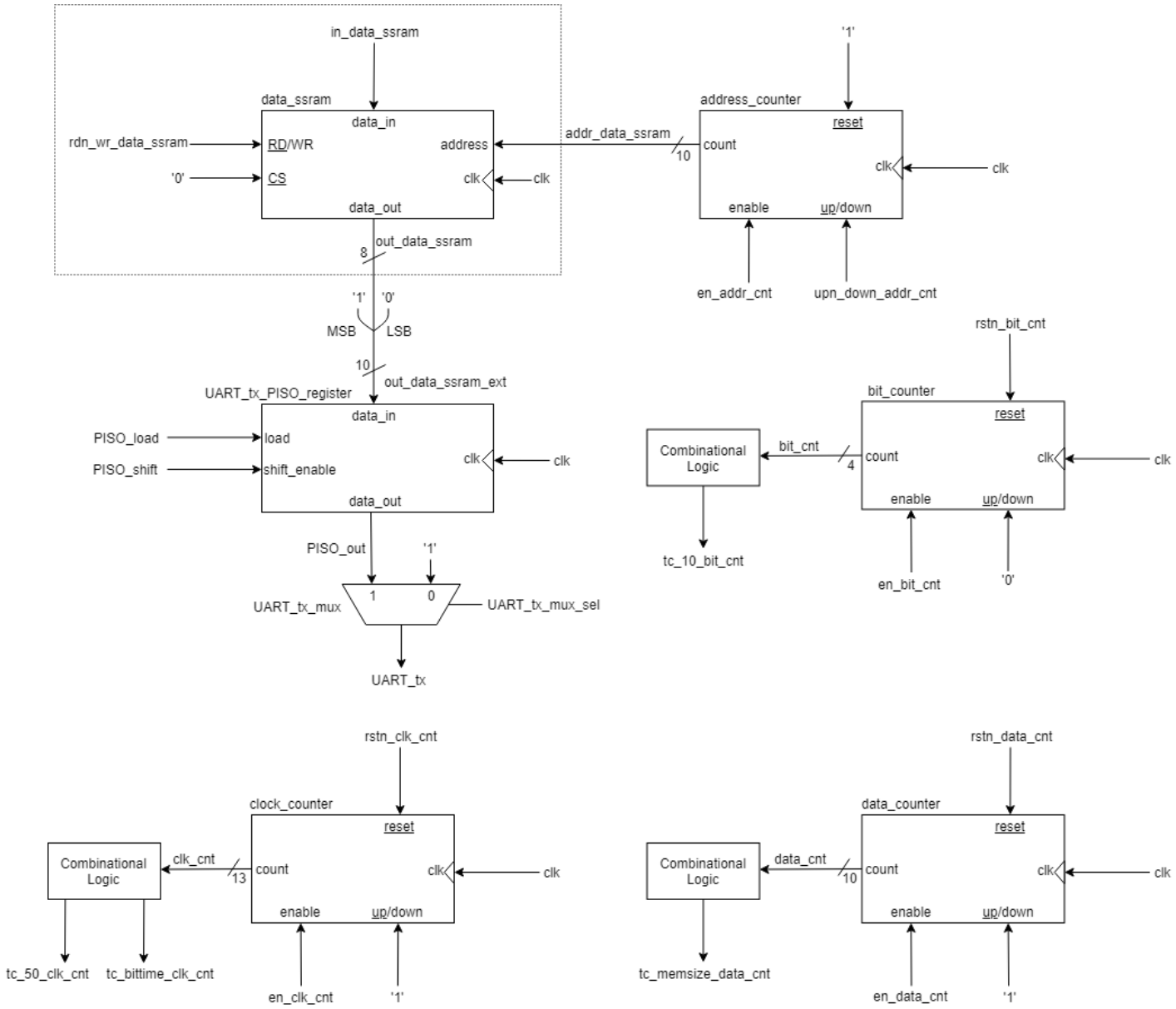


Figure 2.17: Datapath and RAM memory RTL connections. The combinational logic blocks work as terminal count, their implementation is visible in Listing 4.16.

During normal execution, whenever the ADC performs a conversion, the done signal is read by the control unit, which after waiting 200 ns writes this value into RAM to an address pointed by a register, and this pointer is increased by one. Since the pointer has 10 bits, after the value 1023 it will start again from 0. Recalling that the conversions take place every second, the temperature of the last 1024 seconds (17 minutes and 4 seconds) will be stored after which the first entered data will be rewritten by the incoming one. When any byte is received from the UART receiver (what byte doesn't matter because only the start bit will be read) it will stop writing the data in the memory and will start to send to the UART transmitter one by one all the written data, starting from the last written one up to the oldest.

The number of transmitted data bits is clearly eight, plus one bit for the start and one bit for the stop. No bits are used for parity. With this configuration the highest possible baud rate allowed by the PC was tested, which is 1280000, and no transmission errors were committed.

To connect the ADC output data and done signal, which work at 5 V, to the GPIO pins of the FPGA, which support voltages up to 3.8 V, a voltage divider will be used for each signal, like the one in Figure 2.18. First of all to choose the value of the resistances we neglect R_S, its value is not reported in the ADC datasheet but typically an output resistance should be $<10\,\Omega$, so if R_1 is sufficiently higher than this value the approximation will be correct. Not even R_L is reported in the FPGA datasheet, but in this case we can read the leakage current, equal to 30 µA. In order to make R_L negligible, it is necessary to force the current flowing in R_1 to be much greater than 30 µA. Consequently it is found that R_1 must be much smaller than 60 kΩ. 5.6 kΩ could be a good value to make the output and input resistances negligible and at the same time to maintain a low power dissipation. Using the voltage divider formula it is found: R_2=10 kΩ.

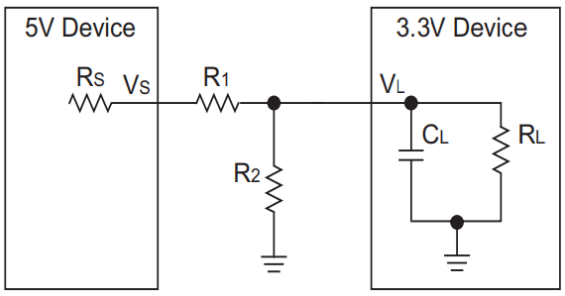


Figure 2.18: Voltage divider used to connect the 5 V ADC output to the 3.3 V FPGA input.

Actually, according to the ADC datasheet, these values do not guarantee the correct functioning. The reported cases are two:

1. V_{OH}=4.5 V with I_{OH}=10 µA.
2. V_{OH}=2.4 V with I_{OH}=360 µA.

In the first case it is not necessary to make any calculation to verify that it is not compatible with our case because only the leakage current is sufficient to overcome that current value. In the second case the maximum current is fine, in fact even with an output voltage of 5.1 V the current would be equal to (neglecting the leakage current): $5.1\,\text{V}/(10\,\text{k}\Omega + 5.6\,\text{k}\Omega) = 327\,\mu\text{A}$, but the minimum voltage is not high enough, in fact at the FPGA input there would be: $2.4\,\text{V} \cdot 10\,\text{k}\Omega/(10\,\text{k}\Omega + 5.6\,\text{k}\Omega) = 1.538\,\text{V}$, but its V_{IH} is 1.7 V. Despite this, the selected resistors have been used anyway, and the correct functioning was verified both functionally and by measuring the FPGA inputs voltages with a voltmeter, they were about 3.1 V.

In figure 2.19 you can see the beginning of the transmission: the purple signal is the UART receiver (in this case a byte of only zeros is sent as start signal), and the blue signal is the UART transmitter. With the screenshot of the Figure 2.20 it was possible to measure the latency of the response, equal to 75 ns and equivalent to three clock cycles as can be expected by seeing the ASM chart. In the unfortunately case in which the request is made exactly at the moment when the FSM is busy to writing a data in memory, the latency could increase further.

All the HDL listings and the simulation are shown in Section 4.2.

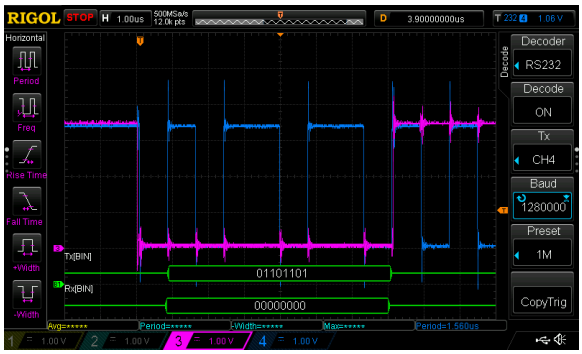


Figure 2.19: Data request and start of the transmission.

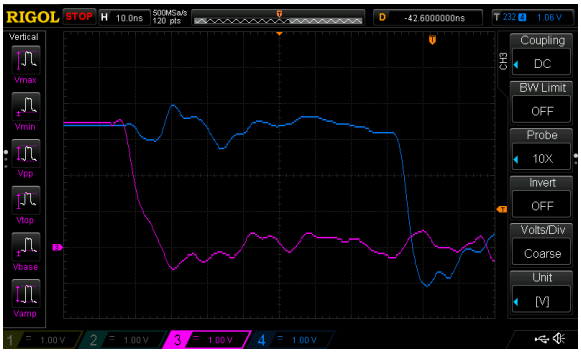


Figure 2.20: Transmission latency.

2.9.1 PC interfacing

In order to have a better control on the data request and a better formatting of the received data, a Python script was written, which requests the data whenever the user wants, and writes them in binary format on the terminal and into a file, adding also for each data the time at which it was recorded and the corresponding temperature in Celsius degree. The source code is shown in Listing 4.22.

Chapter 3

Complete circuit

After all the components have been chosen and if need dimensioned, the complete circuit has been mounted on a breadboard and the proper connections to the DE0 board have been made, in this way it was possible to test, and when necessary to correct and to improve, the source code for the microcontroller and the HDL for the FPGA. Figures 3.1 and 3.2 show the complete wiring diagram, where, however, the connections to the DE0 board through the voltage dividers are not shown, as they would weigh down the diagram reading excessively. For completeness the used FPGA pins are reported in table 3.1.

Table 3.1: DE0 board connections.

Signal	DE0 Board pin
ADC Done	GPIO_0[7]
ADC Data 0	GPIO_0[11]
ADC Data 1	GPIO_0[13]
ADC Data 2	GPIO_0[15]
ADC Data 3	GPIO_0[17]
ADC Data 4	GPIO_0[19]
ADC Data 5	GPIO_0[21]
ADC Data 6	GPIO_0[23]
ADC Data 7	GPIO_0[25]
UART TX	GPIO_0[9]
UART RX	GPIO_0[8]

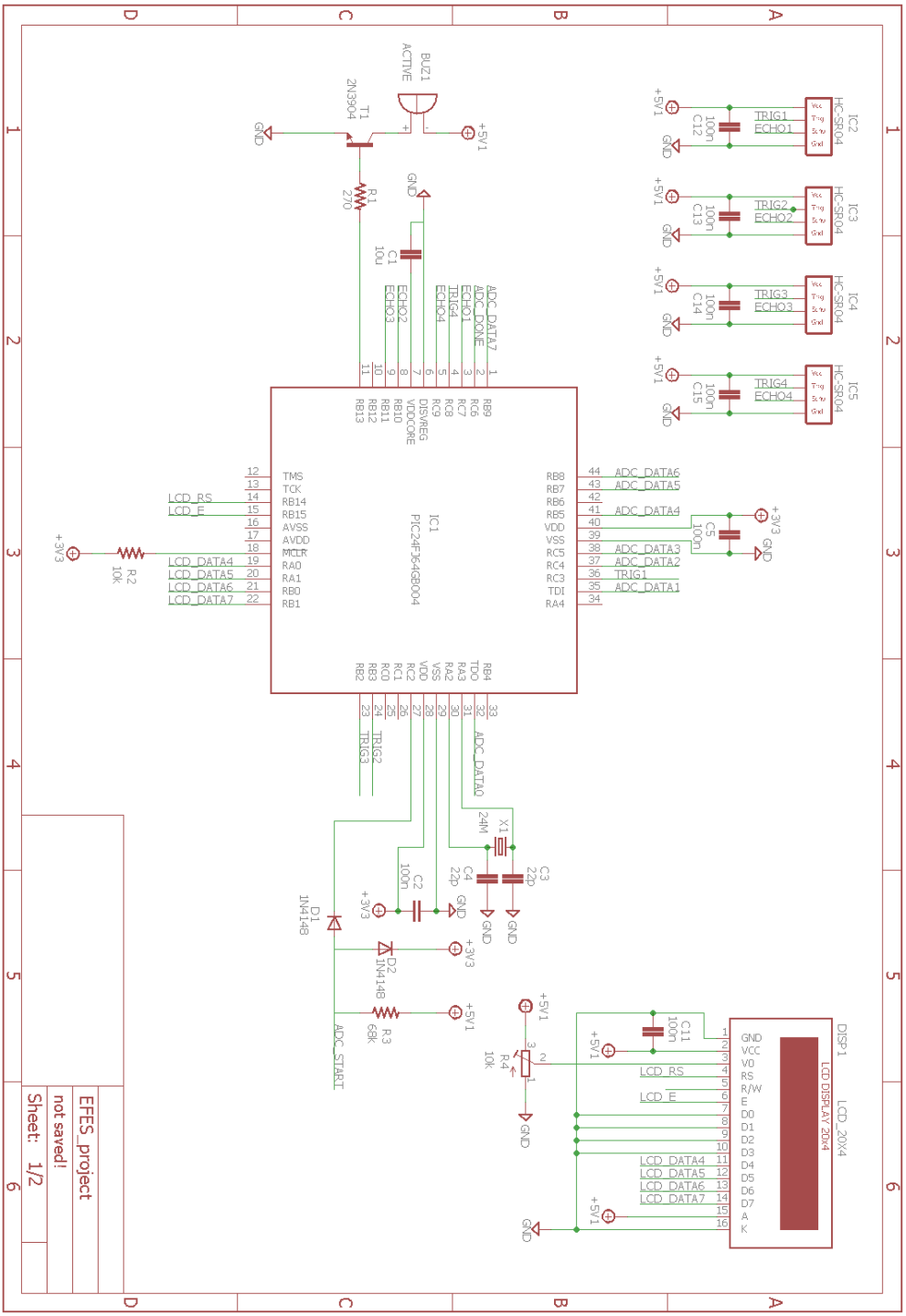


Figure 3.1: Complete circuit wiring diagram, page 1.

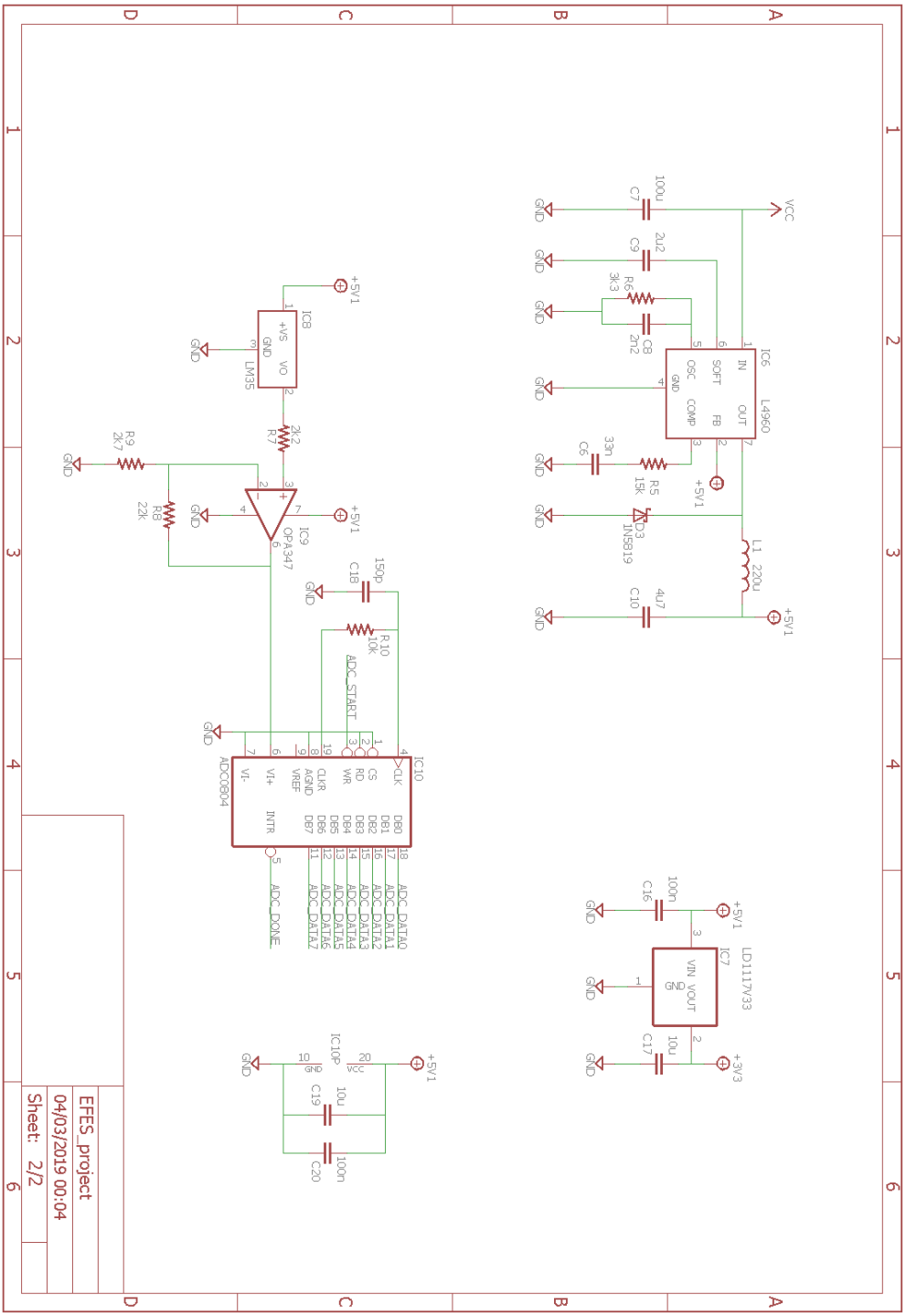


Figure 3.2: Complete circuit wiring diagram, page 2.

Chapter 4

Listings

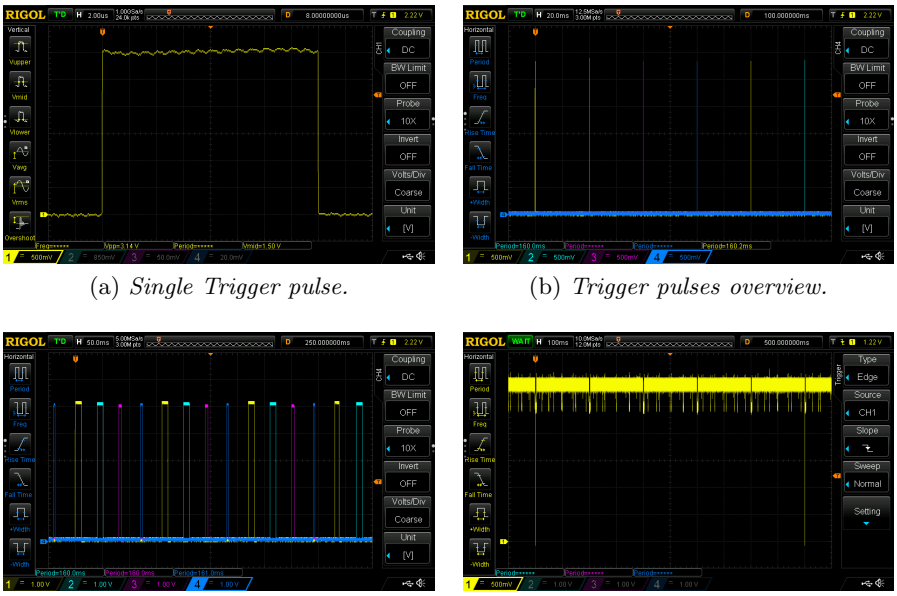
4.1 Source code

All the source code is written in C language and it is divided in independent modules. In the main module there is the main function, in which all the used peripherals are initialized and then the micro is set in idle mode, therefore all the tasks are computed by the interrupt routines. The first four output compare channels are set to send every 40 ms a pulse of 16 µs every time to the *Trigger* of the next ultrasonic sensor. In this way every instant only a single sensor can be active and it can never happen that the ultrasounds emitted from a sensor are received by another sensor. Within the 40 ms, the input capture channel associated with the sensor that has been activated measures the width of the *Echo* pulse and calculates the corresponding distance.

The fifth output compare channel is set to start a periodic polling routine every 50 ms where the display, the buzzer, and the ADC are managed. The four distances and the current temperature are displayed in different position of the LCD and the most critical distance is blinked with a higher or lower frequency according whether it is lower or higher than 10 cm. If a distance is higher than 1 m the word "out" will be shown in its place instead. The Buzzer is also drived according to the most critical distance and to the same thresholds: if the distance is lower than 10 cm the sound is incessant; if the distance is higher than 10 cm but lower than 1 m the intermittence frequency is variable; if the distance is greater than 1 m the buzzer is always off. Furthermore, every 20 times the routine is run (in other words every second), the ADC conversion is started. When the conversion will be done, the external interrupt will be triggered, which reads the conversion value,

calculates the corresponding temperature, and consequently the speed of the sound at that temperature.

In Figure 4.1 there are some oscilloscope measures that verify the correct software functioning and timing.



(a) *Single Trigger pulse.* (b) *Trigger pulses overview.*

(c) *Echo responses.* (d) *ADC start signal.*

Figure 4.1: Software functionig verification: the *trigger* pulse is high for 16 µs, the time between two *Trigger* pulse is 40 ms, for every *Trigger* pulse there is an *Echo* response (also every 40 ms), and the ADC is started every 1 s.

Listing 4.1: main.c

```
/************************************/
/*                                  */
/*  file:       main.c              */
/*  author:     Nicola Vianello     */
/*  last edit:  Turin 04/03/2019    */
/*                                  */
/************************************/

#include "main.h"

volatile float distance[NUMBER_OF_SENSORS];
volatile float celsius;

void port_mapping(){
    __builtin_write_OSCCONL(OSCCON & ~(1<<6)); //unlock PPS registers
    _INT1R=22;  //connect external interrupt 1 to pin RP22
    _IC1R=23;   //connect channel 1 of input capture to pin RP9
    _IC2R=10;   //connect channel 2 of input capture to pin RP8
```

```
    _IC3R=11;   //connect channel 3 of input capture to pin RP7
    _IC4R=25;   //connect channel 4 of input capture to pin RP10
    __builtin_write_OSCCONL(OSCCON | (1<<6)); //lock PPS registers
}

void external_interrupt_init(){
    _INT1EP=1;  //interrupt on negative edge;
    _INT1IP=5;  //interrupt priority is 5
    _INT1IF=0;  //clear flag
    _INT1IE=1;  //enable interrupt
}

void timer_init(){
    //timer1
    //used for input capture and output compare, period 5.333us, overflow every 349.5ms
    T1CONbits.TSIDL=0;      //continue count in idle mode
    T1CONbits.TGATE=0;      //gated time accumulation is disabled
    T1CONbits.TCKPS=0b10;   //prescaler 1:64
    T1CONbits.TCS=0;        //clock source from internal clock
    T1CONbits.TON=1;        //start count
}

void input_capture_init(){
    unsigned garbage;

    //channel 1, use for echo1
    IC1CON1bits.ICSIDL=0;           //continues to operate in CPU idle mode
    IC1CON2bits.IC32=0;             //16bit
    while(IC1CON1bits.ICBNE==1)     //remove garbage from the FIFO
        garbage=IC1BUF;
    IC1CON2bits.SYNCSEL=0b01011;    //synchronization source from timer1
    IC1CON1bits.ICTSEL=0b100;       //clock source from timer1
    IC1CON1bits.ICI=0b00;           //interrupt on every capture event
    IC1CON2bits.ICTRIG=0;           //synchronous mode
    IC1CON1bits.ICM=0b001;          //capture on every edge
    _IC1IP=7;                       //interrupt priority is 7
    _IC1IF=0;                       //clear flag
    _IC1IE=1;                       //enable interrupt

    //channel 2, use for echo2
    IC1CON1bits.ICSIDL=0;           //continues to operate in CPU idle mode
    IC2CON2bits.IC32=0;             //16bit
    while(IC2CON1bits.ICBNE==1)     //remove garbage from the FIFO
        garbage=IC2BUF;
    IC2CON2bits.SYNCSEL=0b01011;    //synchronization source from timer1
    IC2CON1bits.ICTSEL=0b100;       //clock source from timer1
    IC2CON1bits.ICI=0b00;           //interrupt on every capture event
    IC2CON2bits.ICTRIG=0;           //synchronous mode
    IC2CON1bits.ICM=0b001;          //capture on every edge
    _IC2IP=7;                       //interrupt priority is 7
    _IC2IF=0;                       //clear flag
    _IC2IE=1;                       //enable interrupt

    //channel 3, use for echo3
    IC1CON1bits.ICSIDL=0;           //continues to operate in CPU idle mode
    IC3CON2bits.IC32=0;             //16bit
    while(IC3CON1bits.ICBNE==1)     //remove garbage from the FIFO
        garbage=IC3BUF;
    IC3CON2bits.SYNCSEL=0b01011;    //synchronization source from timer1
    IC3CON1bits.ICTSEL=0b100;       //clock source from timer1
    IC3CON1bits.ICI=0b00;           //interrupt on every capture event
    IC3CON2bits.ICTRIG=0;           //synchronous mode
    IC3CON1bits.ICM=0b001;          //capture on every edge
    _IC3IP=7;                       //interrupt priority is 7
    _IC3IF=0;                       //clear flag
    _IC3IE=1;                       //enable interrupt

    //channel 4, use for echo4
    IC1CON1bits.ICSIDL=0;           //continues to operate in CPU idle mode
    IC4CON2bits.IC32=0;             //16bit
    while(IC4CON1bits.ICBNE==1)     //remove garbage from the FIFO
        garbage=IC4BUF;
    IC4CON2bits.SYNCSEL=0b01011;    //synchronization source from timer1
    IC4CON1bits.ICTSEL=0b100;       //clock source from timer1
```

```
    IC4CON1bits.ICI=0b00;           //interrupt on every capture event
    IC4CON2bits.ICTRIG=0;           //synchronous mode
    IC4CON1bits.ICM=0b001;          //capture on every edge
    _IC4IP=7;                       //interrupt priority is 7
    _IC4IF=0;                       //clear flag
    _IC4IE=1;                       //enable interrupt
}

void output_compare_init(){

    //channel 1, used for trigger1
    OC1CON1bits.OCSIDL=0;           //continues to operate in CPU idle mode
    OC1CON1bits.ENFLT2=0;           //comparator fault input is disabled
    OC1CON1bits.ENFLT1=0;           //OCFB fault is disabled
    OC1CON1bits.ENFLT0=0;           //OCFA fault is disabled
    OC1CON2bits.OCINV=0;            //output is not inverted
    OC1CON2bits.DCB=0b00;           //capture event occurs at start of the instruction
        ↪ cycle
    OC1CON2bits.OC32=0;             //16bit
    OC1CON2bits.OCTRIS=1;           //tri-stated output
    OC1R=0;                         //init value
    OC1CON2bits.OCTRIG=0;           //synchronous mode
    OC1CON2bits.SYNCSEL=0b01011;    //synchronization source from timer1
    OC1CON1bits.OCTSEL=0b100;       //based on timer1
    OC1CON1bits.OCM=0b011;          //single compare continuous pulse mode
    _OC1IP=6;                       //interrupt priority is 6
    _OC1IF=0;                       //clear flag
    _OC1IE=1;                       //enable interrupt

    //channel 2, used for trigger2
    OC2CON1bits.OCSIDL=0;           //continues to operate in CPU idle mode
    OC2CON1bits.ENFLT2=0;           //comparator fault input is disabled
    OC2CON1bits.ENFLT1=0;           //OCFB fault is disabled
    OC2CON1bits.ENFLT0=0;           //OCFA fault is disabled
    OC2CON2bits.OCINV=0;            //output is not inverted
    OC2CON2bits.DCB=0b00;           //capture event occurs at start of the instruction
        ↪ cycle
    OC2CON2bits.OC32=0;             //16bit
    OC2CON2bits.OCTRIS=1;           //tri-stated output
    OC2R=DELTA_TRIGGER+DELTA_ECHO;  //init value
    OC2CON2bits.OCTRIG=0;           //synchronous mode
    OC2CON2bits.SYNCSEL=0b01011;    //synchronization source from timer1
    OC2CON1bits.OCTSEL=0b100;       //based on timer1
    OC2CON1bits.OCM=0b011;          //single compare continuous pulse mode
    _OC2IP=6;                       //interrupt priority is 6
    _OC2IF=0;                       //clear flag
    _OC2IE=1;                       //enable interrupt

    //channel 3, used for trigger3
    OC3CON1bits.OCSIDL=0;           //continues to operate in CPU idle mode
    OC3CON1bits.ENFLT2=0;           //comparator fault input is disabled
    OC3CON1bits.ENFLT1=0;           //OCFB fault is disabled
    OC3CON1bits.ENFLT0=0;           //OCFA fault is disabled
    OC3CON2bits.OCINV=0;            //output is not inverted
    OC3CON2bits.DCB=0b00;           //capture event occurs at start of the instruction
        ↪ cycle
    OC3CON2bits.OC32=0;             //16bit
    OC3CON2bits.OCTRIS=1;           //tri-stated output
    OC3R=(DELTA_TRIGGER+DELTA_ECHO)*2;//init value
    OC3CON2bits.OCTRIG=0;           //synchronous mode
    OC3CON2bits.SYNCSEL=0b01011;    //synchronization source from timer1
    OC3CON1bits.OCTSEL=0b100;       //based on timer1
    OC3CON1bits.OCM=0b011;          //single compare continuous pulse mode
    _OC3IP=6;                       //interrupt priority is 6
    _OC3IF=0;                       //clear flag
    _OC3IE=1;                       //enable interrupt

    //channel 4, used for trigger4
    OC4CON1bits.OCSIDL=0;           //continues to operate in CPU idle mode
    OC4CON1bits.ENFLT2=0;           //comparator fault input is disabled
    OC4CON1bits.ENFLT1=0;           //OCFB fault is disabled
    OC4CON1bits.ENFLT0=0;           //OCFA fault is disabled
    OC4CON2bits.OCINV=0;            //output is not inverted
    OC4CON2bits.DCB=0b00;           //capture event occurs at start of the instruction
```

```
        ↪ cycle
    OC4CON2bits.OC32=0;             //16bit
    OC4CON2bits.OCTRIS=1;           //tri-stated output
    OC4R=(DELTA_TRIGGER+DELTA_ECHO)*3;//init value
    OC4CON2bits.OCTRIG=0;           //synchronous mode
    OC4CON2bits.SYNCSEL=0b01011;    //synchronization source from timer1
    OC4CON1bits.OCTSEL=0b100;       //based on timer1
    OC4CON1bits.OCM=0b011;          //single compare continuous pulse mode
    _OC4IP=6;                       //interrupt priority is 6
    _OC4IF=0;                       //clear flag
    _OC4IE=1;                       //enable interrupt

    //channel 5, used to start conversion every second
    OC5CON1bits.OCSIDL=0;           //continues to operate in CPU idle mode
    OC5CON1bits.OCTSEL=0b100;       //based on timer1
    OC5CON1bits.ENFLT2=0;           //comparator fault input is disabled
    OC5CON1bits.ENFLT1=0;           //OCFB fault is disabled
    OC5CON1bits.ENFLT0=0;           //OCFA fault is disabled
    OC5CON2bits.OCINV=0;            //output is not inverted
    OC5CON2bits.DCB=0b00;           //capture event occurs at start of the instruction
        ↪ cycle
    OC5CON2bits.OC32=0;             //16bit
    OC5CON2bits.OCTRIS=1;           //tri-stated output
    OC5R=DELTA_PERIODIC_POLLING;    //init value
    OC5CON2bits.OCTRIG=0;           //synchronous mode
    OC5CON2bits.SYNCSEL=0b01011;    //synchronization source from timer1
    OC5CON1bits.OCM=0b011;          //single compare continuous pulse mode
    _OC5IP=4;                       //interrupt priority is 4
    _OC5IF=0;                       //clear flag
    _OC5IE=1;                       //enable interrupt
}

unsigned get_min_array_index(float array[], unsigned size){
    unsigned i, min_index=0;
    for(i=0; i<size; i++)
        if(array[i]<array[min_index])
            min_index=i;
    return min_index;
}

void periodic_polling(){
    static unsigned char conversion_count=0, display_count=0;
    unsigned char n, min_distance_number;
    char buffer[8];
    min_distance_number=get_min_array_index((float*)distance, NUMBER_OF_SENSORS);

    //conversion
    conversion_count++;
    if(conversion_count>=20){   //every second
        conversion_count=0;
        ADC080x_start_conversion();
    }

    //display
    display_count++;
    if(display_count>=4){   //every 200ms
        display_count=0;
        for(n=0;n<NUMBER_OF_SENSORS;n++){
            switch(n){  //format string in the lcd
                case 0:
                    lcd_goto(7,0);
                    break;
                case 1:
                    lcd_goto(14,0);
                    break;
                case 2:
                    lcd_goto(14,3);
                    break;
                default:    //case 3
                    lcd_goto(7,3);
                    break;
            }
            if(distance[n]==HCSR04_OUT_OF_RANGE || (unsigned)distance[n]>=THRESHOLD_HIGH)
                lcd_print_string(" Out ");
```

```
            else{
                sprintf(buffer,"%2.0fcm  ",(double)distance[n]-0.5);
                if(min_distance_number==n){
                    if((unsigned)distance[n]<=THRESHOLD_LOW)
                        lcd_blink_print_string(buffer, 2, 1);
                    else
                        lcd_blink_print_string(buffer, 4, 1);
                }
                else
                    lcd_print_string(buffer);
            }
        }
        sprintf(buffer,"%3.0f%cC ", (double)celsius, LCD_DEGREE_SYMBOL);
        lcd_goto(0,1);
        lcd_print_string(buffer);
    }

    //buzzer
    if((unsigned)distance[min_distance_number]<=THRESHOLD_LOW)
        BUZZER=1;
    else if((unsigned)distance[min_distance_number]<THRESHOLD_HIGH)
        buzzer_blink(2,((unsigned)distance[min_distance_number]-THRESHOLD_LOW)/2);
    else
        BUZZER=0;
}

void main(){
    port_mapping();
    lcd_init();
    HCSR04_init();
    ADC080x_init();
    external_interrupt_init();
    input_capture_init();
    output_compare_init();
    buzzer_init();
    timer_init();
    _TRISC0=0;
    _TRISC1=0;
    while(1){
        Idle();         //set micro in idle mode
        ClrWdt();       //feed the dog
    }
}

/*********interrupt*********/

//output compare channel 1
void __attribute__((interrupt, no_auto_psv)) _OC1Interrupt(){
    _OC1IF=0;   //clear flag
    HCSR04_TRIGGER1=!HCSR04_TRIGGER1;
    if(HCSR04_TRIGGER1==1)  //rising edge
        OC1R+=DELTA_TRIGGER;
    else    //falling edge
        OC1R+=DELTA_ECHO*4+DELTA_TRIGGER*3;
}

//output compare channel 2
void __attribute__((interrupt, no_auto_psv)) _OC2Interrupt(){
    _OC2IF=0;   //clear flag
    HCSR04_TRIGGER2=!HCSR04_TRIGGER2;
    if(HCSR04_TRIGGER2==1)  //rising edge
        OC2R+=DELTA_TRIGGER;
    else    //falling edge
        OC2R+=DELTA_ECHO*4+DELTA_TRIGGER*3;
}

//output compare channel 3
void __attribute__((interrupt, no_auto_psv)) _OC3Interrupt(){
    _OC3IF=0;   //clear flag
    HCSR04_TRIGGER3=!HCSR04_TRIGGER3;
    if(HCSR04_TRIGGER3==1)  //rising edge
        OC3R+=DELTA_TRIGGER;
    else    //falling edge
        OC3R+=DELTA_ECHO*4+DELTA_TRIGGER*3;
```

```
}

//output compare channel 4
void __attribute__((interrupt, no_auto_psv)) _OC4Interrupt(){
    _OC4IF=0;   //clear flag
    HCSR04_TRIGGER4=!HCSR04_TRIGGER4;
    if(HCSR04_TRIGGER4==1)  //rising edge
        OC4R+=DELTA_TRIGGER;
    else    //falling edge
        OC4R+=DELTA_ECHO*4+DELTA_TRIGGER*3;
}

//output compare channel 5
void __attribute__((interrupt, no_auto_psv)) _OC5Interrupt(){
    _OC5IF=0;   //clear flag
    OC5R+=DELTA_PERIODIC_POLLING;
    periodic_polling();
}

//input capture channel 1
void __attribute__((interrupt, no_auto_psv)) _IC1Interrupt(){
    static unsigned T_rise, T_fall;
    _IC1IF=0;   //clear flag
    if(HCSR04_ECHO1==1) //rising edge
        T_rise=IC1BUF;
    else{   //falling edge
        T_fall=IC1BUF;
        distance[0]=HCSR04_get_distance(T_rise, T_fall);
    }
}

//input capture channel 2
void __attribute__((interrupt, no_auto_psv)) _IC2Interrupt(){
    static unsigned T_rise, T_fall;
    _IC2IF=0;   //clear flag
    if(HCSR04_ECHO2==1) //rising edge
        T_rise=IC2BUF;
    else{   //falling edge
        T_fall=IC2BUF;
        distance[1]=HCSR04_get_distance(T_rise, T_fall);
    }
}

//input capture channel 3
void __attribute__((interrupt, no_auto_psv)) _IC3Interrupt(){
    static unsigned T_rise, T_fall;
    _IC3IF=0;   //clear flag
    if(HCSR04_ECHO3==1) //rising edge
        T_rise=IC3BUF;
    else{   //falling edge
        T_fall=IC3BUF;
        distance[2]=HCSR04_get_distance(T_rise, T_fall);
    }
}

//input capture channel 4
void __attribute__((interrupt, no_auto_psv)) _IC4Interrupt(){
    static unsigned T_rise, T_fall;
    _IC4IF=0;   //clear flag
    if(HCSR04_ECHO4==1) //rising edge
        T_rise=IC4BUF;
    else{   //falling edge
        T_fall=IC4BUF;
        distance[3]=HCSR04_get_distance(T_rise, T_fall);
    }
}

//external interrupt 1
void __attribute__((interrupt, no_auto_psv)) _INT1Interrupt(){
    _INT1IF=0;  //clear flag
    __delay_us(1);  //at least 200ns
    celsius=temperature_get_celsius(ADC080x_read());
    HCSR04_set_temperature(celsius);
}
```

Listing 4.2: main.h

```
/***********************************/
/*                                 */
/*  file:       main.h             */
/*  author:     Nicola Vianello    */
/*  last edit:  Venice 27/12/2018  */
/*                                 */
/***********************************/


// CONFIG4
#pragma config DSWDTPS = DSWDTPSF       // DSWDT Postscale Select (1:2,147,483,648 (25.7
    ↪ days))
#pragma config DSWDTOSC = LPRC          // Deep Sleep Watchdog Timer Oscillator Select (
    ↪ DSWDT uses Low Power RC Oscillator (LPRC))
#pragma config RTCOSC = LPRC            // RTCC Reference Oscillator Select (RTCC uses Low
    ↪ Power RC Oscillator (LPRC))
#pragma config DSBOREN = ON             // Deep Sleep BOR Enable bit (BOR enabled in Deep
    ↪ Sleep)
#pragma config DSWDTEN = OFF            // Deep Sleep Watchdog Timer (DSWDT disabled)

// CONFIG3
#pragma config WPFP = WPFP63            // Write Protection Flash Page Segment Boundary (
    ↪ Highest Page (same as page 42))
#pragma config SOSCSEL = IO             // Secondary Oscillator Pin Mode Select (SOSC pins
    ↪ have digital I/O functions (RA4, RB4))
#pragma config WUTSEL = LEG             // Voltage Regulator Wake-up Time Select (Default
    ↪ regulator start-up time used)
#pragma config WPDIS = WPDIS            // Segment Write Protection Disable (Segmented code
    ↪  protection disabled)
#pragma config WPCFG = WPCFGDIS         // Write Protect Configuration Page Select (Last
    ↪ page and Flash Configuration words are unprotected)
#pragma config WPEND = WPENDMEM         // Segment Write Protection End Page Select (Write
    ↪ Protect from WPFP to the last page of memory)

// CONFIG2
#pragma config POSCMOD = HS             // Primary Oscillator Select (HS Oscillator mode
    ↪ selected)
#pragma config I2C1SEL = PRI            // I2C1 Pin Select bit (Use default SCL1/SDA1 pins
    ↪ for I2C1 )
#pragma config IOL1WAY = ON             // IOLOCK One-Way Set Enable (Once set, the IOLOCK
    ↪ bit cannot be cleared)
#pragma config OSCIOFNC = OFF           // OSCO Pin Configuration (OSCO pin functions as
    ↪ clock output (CLKO))
#pragma config FCKSM = CSDCMD           // Clock Switching and Fail-Safe Clock Monitor (Sw
    ↪ Disabled, Mon Disabled)
#pragma config FNOSC = PRI              // Initial Oscillator Select (Primary Oscillator (
    ↪ XT, HS, EC))
#pragma config PLL96MHZ = OFF           // 96MHz PLL Startup Select (96 MHz PLL Startup is
    ↪ enabled by user in software( controlled with the PLLEN bit))
#pragma config PLLDIV = DIV12           // USB 96 MHz PLL Prescaler Select (Oscillator
    ↪ input divided by 12 (48 MHz input))
#pragma config IESO = ON                // Internal External Switchover (IESO mode (Two-
    ↪ Speed Start-up) enabled)

// CONFIG1
#pragma config WDTPS = PS32768          // Watchdog Timer Postscaler (1:32,768)
#pragma config FWPSA = PR128            // WDT Prescaler (Prescaler ratio of 1:128)
#pragma config WINDIS = OFF             // Windowed WDT (Standard Watchdog Timer enabled,(
    ↪ Windowed-mode is disabled))
#pragma config FWDTEN = ON              // Watchdog Timer (Watchdog Timer is enabled)
#pragma config ICS = PGx3               // Emulator Pin Placement Select bits (Emulator
    ↪ functions are shared with PGEC2/PGED2)
#pragma config GWRP = OFF               // General Segment Write Protect (Writes to program
    ↪  memory are allowed)
#pragma config GCP = OFF                // General Segment Code Protect (Code protection is
    ↪  disabled)
#pragma config JTAGEN = OFF             // JTAG Port Enable (JTAG port is disabled)

#include <stdio.h>
#include <xc.h>
#include "lcd.h"
```

```
#include "ADC080x.h"
#include "temperature.h"
#include "HCSR04.h"
#include "delay.h"
#include "buzzer.h"

#define NUMBER_OF_SENSORS       4           //number of HC-SR04

#define THRESHOLD_HIGH          100U        //cm
#define THRESHOLD_LOW           10U         //cm

#define T_PERIODIC_POLLING      0.05        //second
#define T_ECHO                  0.04        //second
#define T_TRIGGER               0.000016    //second

#define TIMER1_PRESCALER        64

#define T_TIMER1                ((float)TIMER1_PRESCALER/((float)FCY))              //
    ↪ second
#define DELTA_ECHO              ((unsigned)((float)T_ECHO/T_TIMER1))
#define DELTA_TRIGGER           ((unsigned)((float)T_TRIGGER/T_TIMER1))
#define DELTA_PERIODIC_POLLING  ((unsigned)((float)T_PERIODIC_POLLING/T_TIMER1))

void port_mapping();

void external_interrupt_init();

void timer_init();

void input_capture_init();

void output_compare_init();

unsigned get_min_array_index(float array[], unsigned size);

void periodic_polling();

void main();
```

Listing 4.3: delay.h

```
/************************************/
/*                                  */
/*  file:       delay.h             */
/*  author:     Nicola Vianello     */
/*  last edit:  Venice 26/02/2019   */
/*                                  */
/************************************/

#define FXTL 24000000UL

#define FCY FXTL/2
#include <libpic30.h>   //for __delay_us and __delay_ms
```

Listing 4.4: HCSR04.c

```
/************************************/
/*                                  */
/*  file:       hcsr04.c            */
/*  author:     Nicola Vianello     */
/*  last edit:  Venice 27/12/2018   */
/*                                  */
/************************************/

#include "hcsr04.h"
```

```
float hcsr04_sound_speed; //sound speed in centimeters per seconds

void HCSR04_init(){
    //data direction
    HCSR04_TRIGGER1_DIRECTION=0;
    HCSR04_TRIGGER2_DIRECTION=0;
    HCSR04_TRIGGER3_DIRECTION=0;
    HCSR04_TRIGGER4_DIRECTION=0;
    HCSR04_ECHO1_DIRECTION=1;
    HCSR04_ECHO2_DIRECTION=1;
    HCSR04_ECHO3_DIRECTION=1;
    HCSR04_ECHO4_DIRECTION=1;

    //clear output
    HCSR04_TRIGGER1=0;
    HCSR04_TRIGGER2=0;
    HCSR04_TRIGGER3=0;
    HCSR04_TRIGGER4=0;

    //init speed of sound at 20 Celsius degree
    HCSR04_set_temperature(20);
}

//strobe on trigger of channel "channel"
void HCSR04_trigger_strobe(unsigned char channel){
    switch(channel){
        case 1:
            HCSR04_TRIGGER1=1;
            __delay_us(HCSR04_T_TRIGGER);
            HCSR04_TRIGGER1=0;
            break;
        case 2:
            HCSR04_TRIGGER2=1;
            __delay_us(HCSR04_T_TRIGGER);
            HCSR04_TRIGGER2=0;
            break;
        case 3:
            HCSR04_TRIGGER3=1;
            __delay_us(HCSR04_T_TRIGGER);
            HCSR04_TRIGGER3=0;
            break;
        default:    //channel 4
            HCSR04_TRIGGER4=1;
            __delay_us(HCSR04_T_TRIGGER);
            HCSR04_TRIGGER4=0;
            break;
    }
}

//return the distance in centimeters having a rising and falling edge times of a 16 bit
    ↪ timer
//only one wrapping of timer is allawed
//if target is out of range return -1
float HCSR04_get_distance(unsigned T_start, unsigned T_stop){
    float time, distance;   //time in seconds and distance in centimeters
    unsigned T_delta;   //number of timer ticks
    T_delta=T_stop-T_start;
    time=((float)T_delta)*HCSR04_TIMER_PERIOD;
    if(time>=0.038F)    //out of range
        distance=HCSR04_OUT_OF_RANGE;   //distance=0
    else
        distance=((time*hcsr04_sound_speed)/2);
    return distance;
}

//set the sound speed at a given temperature
void HCSR04_set_temperature(float temperature){
    hcsr04_sound_speed=33145.0F+(62.0F*temperature);
}
```

Listing 4.5: HCSR04.h

```
/************************************/
/*                                  */
/*  file:       hcsr04.h            */
/*  author:     Nicola Vianello     */
/*  last edit:  Venice 27/12/2018   */
/*                                  */
/************************************/

#include <xc.h>
#include "delay.h"

#define HCSR04_TRIGGER1 _LATC3
#define HCSR04_TRIGGER2 _LATB3
#define HCSR04_TRIGGER3 _LATB2
#define HCSR04_TRIGGER4 _LATC8
#define HCSR04_ECHO1    _RC7
#define HCSR04_ECHO2    _RB10
#define HCSR04_ECHO3    _RB11
#define HCSR04_ECHO4    _RC9

#define HCSR04_TRIGGER1_DIRECTION   _TRISC3
#define HCSR04_TRIGGER2_DIRECTION   _TRISB3
#define HCSR04_TRIGGER3_DIRECTION   _TRISB2
#define HCSR04_TRIGGER4_DIRECTION   _TRISC8
#define HCSR04_ECHO1_DIRECTION  _TRISC7
#define HCSR04_ECHO2_DIRECTION  _TRISB10
#define HCSR04_ECHO3_DIRECTION  _TRISB11
#define HCSR04_ECHO4_DIRECTION  _TRISC9

#define HCSR04_TIMER_PRESCALER 64   //prescaler 1:64

#define HCSR04_TIMER_PERIOD ((float)HCSR04_TIMER_PRESCALER/((float)FCY))    //timer period
    ↪ in seconds

#define HCSR04_T_TRIGGER 10 //us

#define HCSR04_OUT_OF_RANGE -1.0F

//initialize pins
void HCSR04_init();

//pulse on trigger
void HCSR04_trigger_strobe(unsigned char channel);

//return the distance having a timer start and stop time; only one wrapping of timer; if
    ↪ distance is out of range then return 0.0F
float HCSR04_get_distance(unsigned T_start, unsigned T_stop);

//set the speed of sound at a given temperature
void HCSR04_set_temperature(float temperature);
```

Listing 4.6: lcd.c

```
/************************************/
/*                                  */
/*  file:       lcd.c               */
/*  author:     Nicola Vianello     */
/*  last edit:  Venice 26/12/2018   */
/*                                  */
/************************************/

#include "lcd.h"

//send a pulse on enable
void lcd_strobe(){
    LCD_EN=1;
```

```
    __delay_us(LCD_T_STROBE);
    LCD_EN=0;
}

//write a byte to the LCD
void lcd_write(unsigned char c){
    lcd_byte input;
    input.byte=c;
    #if LCD_BUS_WIDTH_MODE==4
        LCD_DATA7=input.bit7;
        LCD_DATA6=input.bit6;
        LCD_DATA5=input.bit5;
        LCD_DATA4=input.bit4;
        lcd_strobe();
        __delay_us(LCD_T_NIBBLE);
        LCD_DATA7=input.bit3;
        LCD_DATA6=input.bit2;
        LCD_DATA5=input.bit1;
        LCD_DATA4=input.bit0;
        lcd_strobe();
        __delay_us(LCD_T_NIBBLE);
    #elif LCD_BUS_WIDTH_MODE==8
        LCD_DATA7=input.bit7;
        LCD_DATA6=input.bit6;
        LCD_DATA5=input.bit5;
        LCD_DATA4=input.bit4;
        LCD_DATA3=input.bit3;
        LCD_DATA2=input.bit2;
        LCD_DATA1=input.bit1;
        LCD_DATA0=input.bit0;
        lcd_strobe();
        __delay_us(LCD_T_NIBBLE);
    #else
        #error lcd.h: LCD_BUS_WIDTH_MODE must be 4 or 8
    #endif
}

//initialise the LCD
void lcd_init(){
    //init sequence

    //set port as output
    LCD_RS_DIRECTION=0;
    LCD_EN_DIRECTION=0;
    LCD_DATA7_DIRECTION=0;
    LCD_DATA6_DIRECTION=0;
    LCD_DATA5_DIRECTION=0;
    LCD_DATA4_DIRECTION=0;

    //clear control and data
    LCD_RS = 0;
    LCD_EN = 0;
    LCD_DATA7=0;
    LCD_DATA6=0;
    LCD_DATA5=0;
    LCD_DATA4=0;

    //wait after power applied,
    __delay_ms(LCD_T_POWERUP);

    //init data 0011
    LCD_DATA5=1;
    LCD_DATA4=1;

    //wait
    __delay_ms(LCD_T_NIBBLE);

    //strobe
    lcd_strobe();

    //wait
    __delay_ms(LCD_T_SLOW_CMD);

    //strobe
```

```
    lcd_strobe();

    //wait
    __delay_ms(LCD_T_NIBBLE);

    //strobe
    lcd_strobe();

    //wait
    __delay_ms(LCD_T_CMD);

    //now display is started and it is in 8bit mode

    #if LCD_BUS_WIDTH_MODE==4
        LCD_DATA7=0;
        LCD_DATA6=0;
        LCD_DATA5=1;
        LCD_DATA4=0;
        __delay_us(LCD_T_NIBBLE);
        lcd_strobe();
        __delay_us(LCD_T_CMD);
        //now display is in a 4bit mode
    #endif

    //init settings
    #if LCD_BUS_WIDTH_MODE==4
        #if LCD_LINES==1
            lcd_command(FUNCTION_SET | DATA_LENGTH_4BIT | DISPLAY_LINE_1 | FONT_5x8);
        #elif (LCD_LINES==2 || LCD_LINES==4)
            lcd_command(FUNCTION_SET | DATA_LENGTH_4BIT | DISPLAY_LINE_2 | FONT_5x8);
        #else
            #error lcd.h: LCD_LINES must be 1, 2 or 4
        #endif
    #elif LCD_BUS_WIDTH_MODE==8
        #if LCD_LINES==1
            lcd_command(FUNCTION_SET | DATA_LENGTH_8BIT | DISPLAY_LINE_1 | FONT_5x8);
        #elif (LCD_LINES==2 || LCD_LINES==4)
            lcd_command(FUNCTION_SET | DATA_LENGTH_8BIT | DISPLAY_LINE_2 | FONT_5x8);
        #else
            #error lcd.h: LCD_LINES must be 1, 2 or 4
        #endif
    #else
        #error lcd.h: LCD_BUS_WIDTH_MODE must be 4 or 8
    #endif

    //hide powerup garbage
    lcd_command(DISPLAY_ON_OFF_CONTROL | DISPLAY_OFF | CURSOR_OFF | BLINK_OFF);
    __delay_us(LCD_T_CMD);

    //clear powerup garbage
    lcd_clear();

    //entry mode set
    lcd_command(ENTRY_MODE_SET | INCREMENT | SHIFT_OFF);
    __delay_us(LCD_T_CMD);

    //display on
    lcd_command(DISPLAY_ON_OFF_CONTROL | DISPLAY_ON | CURSOR_OFF | BLINK_OFF);
    __delay_us(LCD_T_CMD);
}

//write one character to the LCD
void lcd_print_char(char c){
    LCD_RS=1;
    lcd_write(c);
}

//write a string of chars to the LCD
void lcd_print_string(const char *string){
    LCD_RS=1;
    while((*string)!=0){
        lcd_write(*string);
        string++;
    }
```

```
}

//send a command to the LCD
void lcd_command(unsigned char command){
    LCD_RS=0;
    lcd_write(command);
}

//clear and home the LCD
void lcd_clear(void){
    lcd_command(CLEAR_DISPLAY);
    __delay_ms(LCD_T_SLOW_CMD);
}

//  Go to the specified position
void lcd_goto(unsigned char x, unsigned char y){
    unsigned char lcd_ddram_address;
    #if LCD_LINES==1
        lcd_ddram_address=x;
    #elif LCD_LINES==2
        switch(y){
            case 0:
                lcd_ddram_address=x;
                break;
            default:    //y=1
                lcd_ddram_address=0x40+x;
                break;
        }
    #elif LCD_LINES==4
        switch(y){
            case 0:
                lcd_ddram_address=x;
                break;
            case 1:
                lcd_ddram_address=0x40+x;
                break;
            case 2:
                lcd_ddram_address=0x14+x;
                break;
            default:    //y=3
                lcd_ddram_address=0x54+x;
                break;
        }
    #else
        #error lcd.h: LCD_LINES must be 1, 2 or 4
    #endif
    lcd_command(SET_DDRAM_ADDRESS | lcd_ddram_address);
    __delay_us(LCD_T_CMD);
}

//print "string" for "T_on" times that this function is called and after print a blank
    ↪ string for "T_off" times this function is called
void lcd_blink_print_string(const char* string, unsigned T_on, unsigned T_off){
    static unsigned count=0;
    static unsigned char blink_state=1;
    if(blink_state){
        lcd_print_string(string);
    }
    else{
        while((*string)!=0){
            lcd_print_char(' ');
            string++;
        }
    }
    count++;
    if(blink_state==1 && count>=T_on){
        blink_state=0;
        count=0;
    }
    else if(blink_state==0 && count>T_off){
        blink_state=1;
        count=0;
    }
}
```

```
//print "c" for "T_on" times that this function is called and after print a blank string
    ↪ for "T_off" times this function is called
void lcd_blink_print_char(char c, unsigned T_on, unsigned T_off){
    static unsigned count=0;
    static unsigned char blink_state=1;
    if(blink_state){
        lcd_print_char(c);
    }
    else{
        lcd_print_char(' ');
    }
    count++;
    if(blink_state==1 && count>=T_on){
        blink_state=0;
        count=0;
    }
    else if(blink_state==0 && count>T_off){
        blink_state=1;
        count=0;
    }
}
```

Listing 4.7: lcd.h

```
/***********************************/
/*                                 */
/*  file:       lcd.h              */
/*  author:     Nicola Vianello    */
/*  last edit:  Turin 04/03/2019   */
/*                                 */
/***********************************/

#include <xc.h>
#include "delay.h"

#define LCD_LINES 4     //must be 1,2 or 4
#define LCD_BUS_WIDTH_MODE 4    //must be 4 or 8

//pin
#define LCD_EN          _LATB15 //enable
#define LCD_RS          _LATB14 //register select
#define LCD_DATA0               //only for 8bit mode
#define LCD_DATA1               //only for 8bit mode
#define LCD_DATA2               //only for 8bit mode
#define LCD_DATA3               //only for 8bit mode
#define LCD_DATA4       _LATA0
#define LCD_DATA5       _LATA1
#define LCD_DATA6       _LATB0
#define LCD_DATA7       _LATB1

//direction
#define LCD_EN_DIRECTION _TRISB15       //data direction register
#define LCD_RS_DIRECTION _TRISB14
#define LCD_DATA0_DIRECTION //only for 8bit mode
#define LCD_DATA1_DIRECTION //only for 8bit mode
#define LCD_DATA2_DIRECTION //only for 8bit mode
#define LCD_DATA3_DIRECTION //only for 8bit mode
#define LCD_DATA4_DIRECTION _TRISA0
#define LCD_DATA5_DIRECTION _TRISA1
#define LCD_DATA6_DIRECTION _TRISB0
#define LCD_DATA7_DIRECTION _TRISB1

//time
#define LCD_T_POWERUP 10 //ms
#define LCD_T_NIBBLE 6 //us
#define LCD_T_STROBE 25 //us
#define LCD_T_SLOW_CMD 2 //ms
#define LCD_T_CMD 40 //us
```

```
//command
#define CLEAR_DISPLAY           0b00000001
#define RETURN_HOME             0b00000010
#define ENTRY_MODE_SET      0b00000100
    #define INCREMENT               0b00000010
    #define DECREMENT               0b00000000
    #define SHIFT_ON                0b00000001
    #define SHIFT_OFF               0b00000000
#define DISPLAY_ON_OFF_CONTROL 0b00001000
    #define DISPLAY_ON          0b00000100
    #define DISPLAY_OFF             0b00000000
    #define CURSOR_ON               0b00000010
    #define CURSOR_OFF          0b00000000
    #define BLINK_ON                0b00000001
    #define BLINK_OFF               0b00000000
#define CURSOR_OR_DISPLAY_SHIFT 0b00010000
    #define DISPLAY_SHIFT           0b00001000
    #define CURSOR_SHIFT            0b00000000
    #define SHIFT_RIGHT         0b00000100
    #define SHIFT_LEFT              0b00000000
#define FUNCTION_SET            0b00100000
    #define DATA_LENGTH_8BIT        0b00010000
    #define DATA_LENGTH_4BIT        0b00000000
    #define DISPLAY_LINE_2          0b00001000
    #define DISPLAY_LINE_1          0b00000000
    #define FONT_5x11               0b00000100
    #define FONT_5x8                0b00000000
#define SET_CGRAM_ADDRESS       0b01000000
#define SET_DDRAM_ADDRESS       0b10000000

//special character
#define LCD_DEGREE_SYMBOL   0b11011111

typedef union {
    unsigned char byte;
    struct {
        unsigned bit0:1;
        unsigned bit1:1;
        unsigned bit2:1;
        unsigned bit3:1;
        unsigned bit4:1;
        unsigned bit5:1;
        unsigned bit6:1;
        unsigned bit7:1;
    };
} lcd_byte;

//pulse on enable
void lcd_strobe();

//send a byte to lcd controller
void lcd_write(unsigned char c);

//initialise the lcd
void lcd_init();

//writes a character in the lcd
void lcd_print_char(char c);

//writes a string in the lcd
void lcd_print_string(const char* string);

//write a command in the lcd controller
void lcd_command(unsigned char command);

//clear lcd
void lcd_clear(void);

//set position to write in the lcd
void lcd_goto(unsigned char x, unsigned char y);

//print "string" for "T_on" times that this function is called and after print a blank
    ↪ string for "T_off" times this function is called
void lcd_blink_print_string(const char* string, unsigned T_on, unsigned T_off);
```

```
//print "c" for "T_on" times that this function is called and after print a blank string
    ↪ for "T_off" times this function is called
void lcd_blink_print_char(char c, unsigned T_on, unsigned T_off);
```

Listing 4.8: buzzer.c

```
/***********************************/
/*                                 */
/*  file:       buzzer.c           */
/*  author:     Nicola Vianello    */
/*  last edit:  Venice 27/12/2018  */
/*                                 */
/***********************************/

#include "buzzer.h"

//enable buzzer for "T_on" times and after disable buzzer for "T_off" times; T_on and T_off
    ↪  are the number of times the function is called up
void buzzer_blink(unsigned T_on, unsigned T_off){
    static unsigned count=0;
    static unsigned char blink_state=0;
    BUZZER=blink_state;
    count++;
    if(blink_state==0 && count>=T_off){
        blink_state=1;
        count=0;
    }
    else if(blink_state==1 && count>=T_on){
        blink_state=0;
        count=0;
    }
}

//initialize pin
void buzzer_init(){
    BUZZER_DIRECTION=0;
    BUZZER=0;
}
```

Listing 4.9: buzzer.h

```
/***********************************/
/*                                 */
/*  file:       buzzer.h           */
/*  author:     Nicola Vianello    */
/*  last edit:  Venice 27/12/2018  */
/*                                 */
/***********************************/

#include <xc.h>

#define BUZZER _LATB13
#define BUZZER_DIRECTION _TRISB13

//buzzer stay enabled until this function is called T_on times, and after buzzer stay
    ↪ disabled for T_off times
void buzzer_blink(unsigned T_on, unsigned T_off);

//initialize pin
void buzzer_init();
```

Listing 4.10: ADC080x.c

```
/************************************/
/*                                  */
/*  file:       ADC080x.c           */
/*  author:     Nicola Vianello     */
/*  last edit:  Venice 26/12/2018   */
/*                                  */
/************************************/

#include "ADC080x.h"

void ADC080x_init(){
ADC080x_DATA7_DIRECTION=1;
ADC080x_DATA6_DIRECTION=1;
ADC080x_DATA5_DIRECTION=1;
ADC080x_DATA4_DIRECTION=1;
ADC080x_DATA3_DIRECTION=1;
ADC080x_DATA2_DIRECTION=1;
ADC080x_DATA1_DIRECTION=1;
ADC080x_DATA0_DIRECTION=1;
ADC080x_READY_DIRECTION=1;
ADC080x_START_DIRECTION=0;

ADC080x_START=1;
}

void ADC080x_start_conversion(){
    ADC080x_START=0;
    __delay_us(1);  //at least 100ns
    ADC080x_START=1;
}

unsigned char ADC080x_read(){
    ADC080x_byte data;
    data.byte=0;
    if(ADC080x_DATA0)
        data.bit0=1;
    if(ADC080x_DATA1)
        data.bit1=1;
    if(ADC080x_DATA2)
        data.bit2=1;
    if(ADC080x_DATA3)
        data.bit3=1;
    if(ADC080x_DATA4)
        data.bit4=1;
    if(ADC080x_DATA5)
        data.bit5=1;
    if(ADC080x_DATA6)
        data.bit6=1;
    if(ADC080x_DATA7)
        data.bit7=1;
    return data.byte;
}
```

Listing 4.11: ADC080x.h

```
/************************************/
/*                                  */
/*  file:       ADC080x.h           */
/*  author:     Nicola Vianello     */
/*  last edit:  Venice 26/12/2018   */
/*                                  */
/************************************/

#include <xc.h>
#include "delay.h"
```

```
#define ADC080x_DATA7   _RB9
#define ADC080x_DATA6   _RB8
#define ADC080x_DATA5   _RB7
#define ADC080x_DATA4   _RB5
#define ADC080x_DATA3   _RC5
#define ADC080x_DATA2   _RC4
#define ADC080x_DATA1   _RA9
#define ADC080x_DATA0   _RA8
#define ADC080x_START   _LATC2
#define ADC080x_READY   _RC6

#define ADC080x_DATA7_DIRECTION   _TRISB9
#define ADC080x_DATA6_DIRECTION   _TRISB8
#define ADC080x_DATA5_DIRECTION   _TRISB7
#define ADC080x_DATA4_DIRECTION   _TRISB5
#define ADC080x_DATA3_DIRECTION   _TRISC5
#define ADC080x_DATA2_DIRECTION   _TRISC4
#define ADC080x_DATA1_DIRECTION   _TRISA9
#define ADC080x_DATA0_DIRECTION   _TRISA8
#define ADC080x_START_DIRECTION   _TRISC2
#define ADC080x_READY_DIRECTION   _TRISC6

typedef union {
    unsigned char byte;
    struct {
        unsigned bit0:1;
        unsigned bit1:1;
        unsigned bit2:1;
        unsigned bit3:1;
        unsigned bit4:1;
        unsigned bit5:1;
        unsigned bit6:1;
        unsigned bit7:1;
    };
} ADC080x_byte;

void ADC080x_start_conversion();

void ADC080x_init();

unsigned char ADC080x_read();
```

Listing 4.12: temperature.c

```
/***********************************/
/*                                 */
/*  file:       temperature.c      */
/*  author:     Nicola Vianello    */
/*  last edit:  Venice 26/12/2018  */
/*                                 */
/***********************************/

#include "temperature.h"

//return temperature in degree Celsius having a value of ADC conversion
float temperature_get_celsius(unsigned conversion){
    float sensor_out=((float)conversion)*ADC_GAP;   //out in volts
    float celsius=(sensor_out-TEMPERATURE_OFFSET)*(1.0F/TEMPERATURE_GAIN);
    return celsius;
}
```

Listing 4.13: temperature.h

```
/***********************************/
/*                                 */
/*  file:       temperature.h      */
/*  author:     Nicola Vianello    */
/*  last edit:  Venice 21/02/2019  */
/*                                 */
/***********************************/

#define ADC_BIT 8               //ADC bits number
#define ADC_REFERENCE   5.1F    //ADC voltage reference

#define SENSOR_GAIN         0.01F   //sensor V/Celsius
#define SENSOR_OFFSET       0.0F    //sensor output voltage when temperature is zero
    ↪ Celsius degree
#define CONDITIONING_GAIN   9.15F   //gain of conditioning circuit

#define TEMPERATURE_GAIN    (SENSOR_GAIN*CONDITIONING_GAIN)     //overall V/Celsius
#define TEMPERATURE_OFFSET  (SENSOR_OFFSET*CONDITIONING_GAIN)   //conditioning circuit
    ↪ output voltage when temperature is zero Celsius degree

#define ADC_GAP (ADC_REFERENCE/((float)(1U<<ADC_BIT)))      //voltage of an ADC gap

//return temperature in degree Celsius having a value of ADC conversion
float temperature_get_celsius(unsigned conversion);
```

4.2 HDL

All the hardware description code is developed in VHDL using Quartus IDE. The connection between the FPGA Pin and the inputs and outputs of the top level file are listed in Table 4.1.

The high level general functioning of the implemented hardware is already explained in section 2.9. Going into a lower level description, once the data transmission is requested, the data pointed by the RAM memory pointer, that is the last written data, is loaded into a 10 bit shift register concatenating a '0' as least significant bit (for the UART start) and a '1' as most significant bit (for the UART stop), after which, the UART transmission line, which is normally fixed to '1', is connected to the output of this shift register, which corresponds to its least significant bit. When the data transmission is so initialized, it proceeds using three nested counters: the innermost one counts the number of clock cycles and after a number of clock cycles corresponding to a bit time (the baud rate can be quickly changed by changing a constant in Listing 4.16) its count is cleared and the contents of the shift register are shifted to the right of one position. The second counter keeps count of the number of transmitted bits for each data, and, after all ten bits have been transmitted, the value of the third counter is increased, which counts the number of transmitted data, and the RAM memory pointer is decremented, pointing now to the data written

immediately before the one just transmitted. This data will be loaded in the shift register, restarting the loop. Once all the data has been transmitted, in other words the third counter is equal to 1023, the transmission is completed and the RAM memory pointer is at the exact same value it was before starting the transmission.

Another low level consideration is about the process used inside the top_level module: only after the circuit was programmed in the FPGA it has been noticed that the ADC done signal is long enough to restart the memory writing of the same data several times, so this process has been added that acts as a monostable multivibrator: indifferently from how long the done pulse lasts, it sends to the control unit a pulse of a single clock cycle.

Table 4.1: FPGA pin planner.

Signal	Pin
start_writing	Y4
UART_tx	U11
UART_rx	Y5
in_data_SSRAM[0]	T12
in_data_SSRAM[1]	AH6
in_data_SSRAM[2]	AG5
in_data_SSRAM[3]	AH2
in_data_SSRAM[4]	AG6
in_data_SSRAM[5]	AE4
in_data_SSRAM[6]	T11
in_data_SSRAM[7]	AF6
clk	V11
rstn	AH17

Listing 4.14: top_level.vhd

```
---------------------------------------
--                                   --
--  file:       top_level.vhd        --
--  author:     Nicola Vianello      --
--  last edit:  Turin 04/03/2019     --
--                                   --
---------------------------------------

library ieee;
use ieee.std_logic_1164.all;

entity top_level is
port(   start_writing, UART_rx, clk, rstn : in std_logic;
        in_data_ssram : in std_logic_vector(7 downto 0);
        UART_tx : out std_logic);
end entity top_level;
```

```
architecture struct of top_level is

component control_unit is
port( start_writing, clk, rstn, UART_rx, tc_10_bit_cnt, tc_50_clk_cnt, tc_bittime_clk_cnt,
     ↪ tc_memsize_data_cnt : in std_logic;
       rdn_wr_data_ssram, en_addr_cnt, upn_down_addr_cnt, PISO_load, PISO_shift,
            ↪ uart_tx_mux_sel, rstn_bit_cnt, en_bit_cnt, rstn_clk_cnt, en_clk_cnt,
            ↪ rstn_data_cnt, en_data_cnt : out std_logic);
end component control_unit;

component datapath is
    port(   clk, upn_down_addr_cnt, en_addr_cnt, PISO_load, PISO_shift, UART_tx_mux_sel,
         ↪ rstn_bit_cnt, en_bit_cnt, rstn_clk_cnt, en_clk_cnt, rstn_data_cnt, en_data_cnt
         ↪ : in std_logic;
           UART_tx, tc_10_bit_cnt, tc_50_clk_cnt, tc_bittime_clk_cnt, tc_memsize_data_cnt:
                ↪  out std_logic;
           addr_data_ssram: out std_logic_vector(9 downto 0);
           out_data_ssram: in std_logic_vector(7 downto 0));
end component datapath;

component SSRAM_dual_port is
    generic(    N_data_bit : natural := 8;
                N_address_bit : natural := 10);
    port( data_in : in std_logic_vector(N_data_bit-1 downto 0);
          data_out : out std_logic_vector(N_data_bit-1 downto 0);
          address : in std_logic_vector(N_address_bit-1 downto 0);
          rdn_wr, cs, clk : in std_logic);
end component SSRAM_dual_port;

signal addr_data_ssram: std_logic_vector(9 downto 0);
signal out_data_ssram: std_logic_vector(7 downto 0);
signal rdn_wr_data_ssram, en_addr_cnt, upn_down_addr_cnt, PISO_load, PISO_shift,
     ↪ UART_tx_mux_sel, rstn_bit_cnt, en_bit_cnt, tc_10_bit_cnt, rstn_clk_cnt,
     ↪ tc_50_clk_cnt, tc_bittime_clk_cnt, en_clk_cnt, tc_memsize_data_cnt, en_data_cnt,
     ↪ rstn_data_cnt: std_logic;

signal start_writing_conditioned: std_logic;

begin

    control_unit_inst: control_unit
        port map(start_writing=>start_writing_conditioned, clk=>clk, rstn=>rstn, UART_rx=>
             ↪ UART_rx, tc_10_bit_cnt=>tc_10_bit_cnt, tc_50_clk_cnt=>tc_50_clk_cnt,
             ↪ tc_bittime_clk_cnt=>tc_bittime_clk_cnt, tc_memsize_data_cnt=>
             ↪ tc_memsize_data_cnt, rdn_wr_data_ssram=>rdn_wr_data_ssram, en_addr_cnt=>
             ↪ en_addr_cnt, upn_down_addr_cnt=>upn_down_addr_cnt, PISO_load=>PISO_load,
             ↪ PISO_shift=>PISO_shift, uart_tx_mux_sel=>uart_tx_mux_sel, rstn_bit_cnt=>
             ↪ rstn_bit_cnt, en_bit_cnt=>en_bit_cnt, rstn_clk_cnt=>rstn_clk_cnt,
             ↪ en_clk_cnt=>en_clk_cnt, rstn_data_cnt=>rstn_data_cnt, en_data_cnt=>
             ↪ en_data_cnt);

    datapath_inst: datapath
        port map(clk=>clk, upn_down_addr_cnt=>upn_down_addr_cnt, en_addr_cnt=>en_addr_cnt,
             ↪ PISO_load=>PISO_load, PISO_shift=>PISO_shift, UART_tx_mux_sel=>
             ↪ UART_tx_mux_sel, rstn_bit_cnt=>rstn_bit_cnt, en_bit_cnt=>en_bit_cnt,
             ↪ rstn_clk_cnt=>rstn_clk_cnt, en_clk_cnt=>en_clk_cnt, rstn_data_cnt=>
             ↪ rstn_data_cnt, en_data_cnt=>en_data_cnt, UART_tx=>UART_tx, tc_10_bit_cnt=>
             ↪ tc_10_bit_cnt, tc_50_clk_cnt=>tc_50_clk_cnt, tc_bittime_clk_cnt=>
             ↪ tc_bittime_clk_cnt, tc_memsize_data_cnt=>tc_memsize_data_cnt,
             ↪ addr_data_ssram=>addr_data_ssram, out_data_ssram=>out_data_ssram);

    data_SSRAM: SSRAM_dual_port
        generic map(N_data_bit=>8, N_address_bit=>10)
        port map(data_in=>in_data_ssram,    data_out=>out_data_ssram, address=>
             ↪ addr_data_ssram, rdn_wr=>rdn_wr_data_ssram, cs=>'1', clk=>clk);

    start_writing_conditioning: process(clk)
    variable previous_value: std_logic;
    begin
        if rising_edge(clk) then
            if(start_writing='0' and previous_value='1') then
                start_writing_conditioned<='0';
            else
```

```
                start_writing_conditioned<='1';
            end if;
            previous_value:= start_writing;
        end if;
    end process start_writing_conditioning;

end architecture struct;
```

Listing 4.15: control_unit.vhd

```
-------------------------------------
--                                 --
--  file:       control_unit.vhd   --
--  author:     Nicola Vianello    --
--  last edit:  Turin 04/03/2019   --
--                                 --
-------------------------------------

library ieee;
use ieee.std_logic_1164.all;

entity control_unit is
port( start_writing, clk, rstn, UART_rx, tc_10_bit_cnt, tc_50_clk_cnt, tc_bittime_clk_cnt,
     ↪ tc_memsize_data_cnt : in std_logic;
        rdn_wr_data_ssram, en_addr_cnt, upn_down_addr_cnt, PISO_load, PISO_shift,
             ↪ uart_tx_mux_sel, rstn_bit_cnt, en_bit_cnt, rstn_clk_cnt, en_clk_cnt,
             ↪ rstn_data_cnt, en_data_cnt : out std_logic);
end entity control_unit;

architecture behavioral of control_unit is

type state_type is (Idle, Wait_Data, Write_Memory, Read_Memory, Load_PISO, Wait_Bit_Time,
     ↪ Shift_Data_out);
signal future_state, present_state : state_type;

begin

    future_state_evaluation: process(present_state, start_writing, clk, rstn, UART_rx,
         ↪ tc_10_bit_cnt, tc_50_clk_cnt, tc_bittime_clk_cnt, tc_memsize_data_cnt)
    begin
        if rstn='0' then
            future_state<=Idle;
        else
            future_state<=Idle; --default state
            case present_state is
                when Idle => case UART_rx is
                    when '0' => future_state<=Read_Memory;
                    when others => case start_writing is
                        when '0' => future_state<=Wait_Data;
                        when others => future_state<=Idle;
                    end case;
                end case;
                when Wait_Data => case tc_50_clk_cnt is
                    when '1' => future_state<=Write_Memory;
                    when others => future_state<=Wait_Data;
                end case;
                when Write_Memory => future_state<=Idle;
                when Read_Memory => future_state<=Load_PISO;
                when Load_PISO => future_state<=Wait_Bit_Time;
                when Wait_Bit_Time => case tc_bittime_clk_cnt is
                    when '1' => future_state<=Shift_Data_Out;
                    when others => future_state<=Wait_Bit_Time;
                end case;
                when Shift_Data_Out => case tc_10_bit_cnt is
                    when '0' => future_state<=Wait_Bit_Time;
                    when others => case tc_memsize_data_cnt is
                        when '0' => future_state<=Read_memory;
                        when others => future_state<=Idle;
                    end case;
                end case;
```

```
            end case;
        end if;
    end process future_state_evaluation;

    state_transition: process(clk)
    begin
        if rising_edge(clk) then
            present_state<=future_state;
        end if;
    end process state_transition;

    output_decode: process(present_state, UART_rx, start_writing, tc_10_bit_cnt)
    begin
        --default output
        en_addr_cnt<='0';
        upn_down_addr_cnt<='0';
        PISO_load<='0';
        PISO_shift<='0';
        rstn_bit_cnt<='1';
        en_bit_cnt<='0';
        rstn_clk_cnt<='1';
        en_clk_cnt<='0';
        rstn_data_cnt<='1';
        en_data_cnt<='0';
        rdn_wr_data_ssram<='0';
        UART_tx_mux_sel<='0';
        case present_state is
            when Idle =>
                rstn_clk_cnt<='0';
                rstn_data_cnt<='0';
                if UART_rx='1' then
                    if start_writing='0' then
                        en_addr_cnt<='1';
                    end if;
                end if;
            when Wait_Data=>
                en_clk_cnt<='1';
            when Write_Memory=>
                rdn_wr_data_ssram<='1';
            when Read_Memory=>
                en_addr_cnt<='1';
                upn_down_addr_cnt<='1';
                rstn_clk_cnt<='0';
                rstn_bit_cnt<='0';
            when Load_PISO=>
                PISO_load<='1';
            when Wait_Bit_Time=>
                en_clk_cnt<='1';
                UART_tx_mux_sel<='1';
            when Shift_Data_out=>
                PISO_shift<='1';
                en_bit_cnt<='1';
                rstn_clk_cnt<='0';
                UART_tx_mux_sel<='1';
                if tc_10_bit_cnt='1' then
                    en_data_cnt<='1';
                end if;
        end case;
    end process output_decode;

end architecture behavioral;
```

Listing 4.16: datapath.vhd

```
-------------------------------------
--                                 --
--  file:       datapath.vhd       --
--  author:     Nicola Vianello    --
--  last edit:  Venice 25/02/2019  --
--                                 --
```

```
------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity datapath is
    port(   clk, upn_down_addr_cnt, en_addr_cnt, PISO_load, PISO_shift, UART_tx_mux_sel,
         ↪ rstn_bit_cnt, en_bit_cnt, rstn_clk_cnt, en_clk_cnt, rstn_data_cnt, en_data_cnt
         ↪ : in std_logic;
           UART_tx, tc_10_bit_cnt, tc_50_clk_cnt, tc_bittime_clk_cnt, tc_memsize_data_cnt:
                ↪  out std_logic;
           addr_data_ssram: out std_logic_vector(9 downto 0);
           out_data_ssram: in std_logic_vector(7 downto 0));
end entity datapath;

architecture struct of datapath is

component counter is
    generic(Nbits: natural := 8);
    port(   en, clk, rstn, upn_down: in std_logic;
            cnt: out std_logic_vector(Nbits-1 downto 0));
end component counter;

component PISO_register is
    generic(Nbits: natural:=8);
    port(   clk, load, shift_enable: in std_logic;
            data_in: in std_logic_vector(Nbits-1 downto 0);
            data_out: out std_logic);
end component PISO_register;

component mux is
    generic(    Nbits: natural := 1;
                Nsel: natural := 1);
    port(       inputs: in std_logic_vector((2**Nsel)*Nbits-1 downto 0);
                sel: in std_logic_vector(Nsel-1 downto 0);
                output: out std_logic_vector(Nbits-1 downto 0));
end component mux;

signal out_data_ssram_ext, data_cnt: std_logic_vector(9 downto 0);
signal clk_cnt: std_logic_vector(12 downto 0);
signal bit_cnt: std_logic_vector(3 downto 0);
signal PISO_out: std_logic;

constant baud_rate: integer := 1280000;
constant clock_frequency: integer := 50000000;
constant clock_cycles_per_bit_time: integer:= clock_frequency/baud_rate;

begin

    address_counter: counter
        generic map(Nbits=>10)
        port map(en=>en_addr_cnt, clk=>clk, rstn=>'1', upn_down=>upn_down_addr_cnt, cnt=>
             ↪ addr_data_ssram);

    out_data_ssram_ext(9)<='1';
    out_data_ssram_ext(8 downto 1)<=out_data_ssram(7 downto 0);
    out_data_ssram_ext(0)<='0';

    UART_tx_PISO_register: PISO_register
        generic map(Nbits=>10)
        port map(clk=>clk, load=>PISO_load, shift_enable=>PISO_shift, data_in=>
             ↪ out_data_ssram_ext, data_out=>PISO_out);

    UART_tx_mux: mux
        generic map(Nbits=>1, Nsel=>1)
        port map(inputs(0)=>'1', inputs(1)=>PISO_out, sel(0)=>UART_tx_mux_sel, output(0)=>
             ↪ UART_tx);

    bit_counter: counter
        generic map(Nbits=>4)
        port map(en=>en_bit_cnt, clk=>clk, rstn=>rstn_bit_cnt, upn_down=>'0', cnt=>bit_cnt)
             ↪ ;
    tc_10_bit_cnt<='1' when bit_cnt=std_logic_vector(to_unsigned(9, bit_cnt'length)) else
```

```
        ↪ '0';

    clock_counter: counter
        generic map(Nbits=>13)
        port map(en=>en_clk_cnt, clk=>clk, rstn=>rstn_clk_cnt, upn_down=>'0', cnt=>clk_cnt)
            ↪ ;
    tc_50_clk_cnt<='1' when clk_cnt=std_logic_vector(to_unsigned(49, clk_cnt'length)) else
        ↪ '0';
    tc_bittime_clk_cnt<='1' when clk_cnt=std_logic_vector(to_unsigned(
        ↪ clock_cycles_per_bit_time-2, clk_cnt'length)) else '0';

    data_counter: counter
        generic map(Nbits=>10)
        port map(en=>en_data_cnt, clk=>clk, rstn=>rstn_data_cnt, upn_down=>'0', cnt=>
            ↪ data_cnt);
    tc_memsize_data_cnt<='1' when data_cnt=std_logic_vector(to_unsigned(1023, data_cnt'
        ↪ length)) else '0';

end architecture struct;
```

Listing 4.17: SSRAM_dual_port.vhd

```
----------------------------------
--                              --
--  file:       SSRAM_dual_port.vhd --
--  author:     Nicola Vianello     --
--  last edit:  Turin 19/12/2018    --
--                              --
----------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity SSRAM_dual_port is
    generic(    N_data_bit : natural := 8;
                N_address_bit : natural := 10);
    port( data_in : in std_logic_vector(N_data_bit-1 downto 0);
            data_out : out std_logic_vector(N_data_bit-1 downto 0);
            address : in std_logic_vector(N_address_bit-1 downto 0);
            rdn_wr, cs, clk : in std_logic);
end entity SSRAM_dual_port;

architecture behavioral of SSRAM_dual_port is
    type mem_type is array(0 to 2**N_address_bit-1) of std_logic_vector(N_data_bit-1 downto
        ↪  0);
    signal mem : mem_type ;
    begin
        process(clk)
        begin
            if rising_edge(clk) then
                if cs='1' then
                    if rdn_wr='0' then  --read
                        data_out<=mem(to_integer(unsigned(address)));
                    else    --write
                        mem(to_integer(unsigned(address)))<=data_in;
                    end if;
                end if;
            end if;
        end process;
end architecture behavioral;
```

Listing 4.18: counter.vhd

```
--------------------------------------
--                                  --
--  file:       SSRAM_dual_port.vhd --
--  author:     Nicola Vianello     --
--  last edit:  Turin 02/01/2019    --
--                                  --
--------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity counter is
    generic(Nbits: natural := 8);
    port(   en, clk, rstn, upn_down: in std_logic;
            cnt: out std_logic_vector(Nbits-1 downto 0));
end entity counter;

architecture behavioral of counter is
begin
    process(clk)
    variable cnt_var : std_logic_vector(cnt'range) :=(others=>'0');
    begin
        if rising_edge(clk) then
            if rstn='0' then
                cnt_var:=(others=>'0');
            else
                if en='1' then
                    if upn_down='0' then
                        cnt_var:=cnt_var+1;
                    else
                        cnt_var:=cnt_var-1;
                    end if;
                end if;
            end if;
        end if;
    cnt<=cnt_var;
    end process;
end architecture behavioral;
```

Listing 4.19: mux.vhd

```
--------------------------------------
--                                  --
--  file:       mux.vhd             --
--  author:     Nicola Vianello     --
--  last edit:  Turin 29/12/2018    --
--                                  --
--------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity mux is
    generic(    Nbits: natural := 1;
                Nsel: natural := 1);
    port(       inputs: in std_logic_vector((2**Nsel)*Nbits-1 downto 0);
                sel: in std_logic_vector(Nsel-1 downto 0);
                output: out std_logic_vector(Nbits-1 downto 0));
end entity mux;

architecture behavioural of mux is
begin
        output <= inputs((to_integer(unsigned(sel))+1)*Nbits-1 downto to_integer(unsigned(
            ↪ sel))*Nbits);
```

```
end architecture behavioural;
```

Listing 4.20: PISO_register.vhd

```
--------------------------------------
--                                  --
--  file:       PISO_register.vhd   --
--  author:     Nicola Vianello     --
--  last edit:  Turin 22/12/2018    --
--                                  --
--------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity mux is
    generic(    Nbits: natural := 1;
                Nsel: natural := 1);
    port(       inputs: in std_logic_vector((2**Nsel)*Nbits-1 downto 0);
                sel: in std_logic_vector(Nsel-1 downto 0);
                output: out std_logic_vector(Nbits-1 downto 0));
end entity mux;

architecture behavioural of mux is
begin
        output <= inputs((to_integer(unsigned(sel))+1)*Nbits-1 downto to_integer(unsigned(
            ↪ sel))*Nbits);
end architecture behavioural;

library ieee;
use ieee.std_logic_1164.all;

entity PISO_register is
    generic(Nbits: natural:=8);
    port(   clk, load, shift_enable: in std_logic;
            data_in: in std_logic_vector(Nbits-1 downto 0);
            data_out: out std_logic);
end entity PISO_register;

architecture behavioral of PISO_register is

signal data: std_logic_vector(data_in'range);

begin
    process(clk)
    begin
        if rising_edge(clk) then
            if load='1' then
                data<=data_in;
            elsif shift_enable='1' then
                data(Nbits-2 downto 0)<=data(Nbits-1 downto 1);
                data(Nbits-1)<='0';
            end if;
        end if;
    end process;

    data_out<=data(0);

end architecture behavioral;
```

4.2.1 Testbench and simulation

Before programming the FPGA, the written VHDL code was tested using ModelSim and the testbench in Listing 4.21. First it sends a reset signal to the control unit, after which, it writes three data into memory and then requests the transmission.

In Figure 4.2 is shown the beginning of the simulation, where there are the reset signal and the writing of the first data. In this simulation the writing begins at address 1 because the pointer is initialized to 0 and the address pointed by it is the last written, so the writing starts at the next address. Actually the pointer, which is a register, will be initialized to a random value but this does not matter because the memory will be used as a FIFO list.

Figure 4.3 shows the transmission of the first data and Figure 4.4 shows the end of the transmission, we can see that the pointer has the same value that it had before starting transmission.

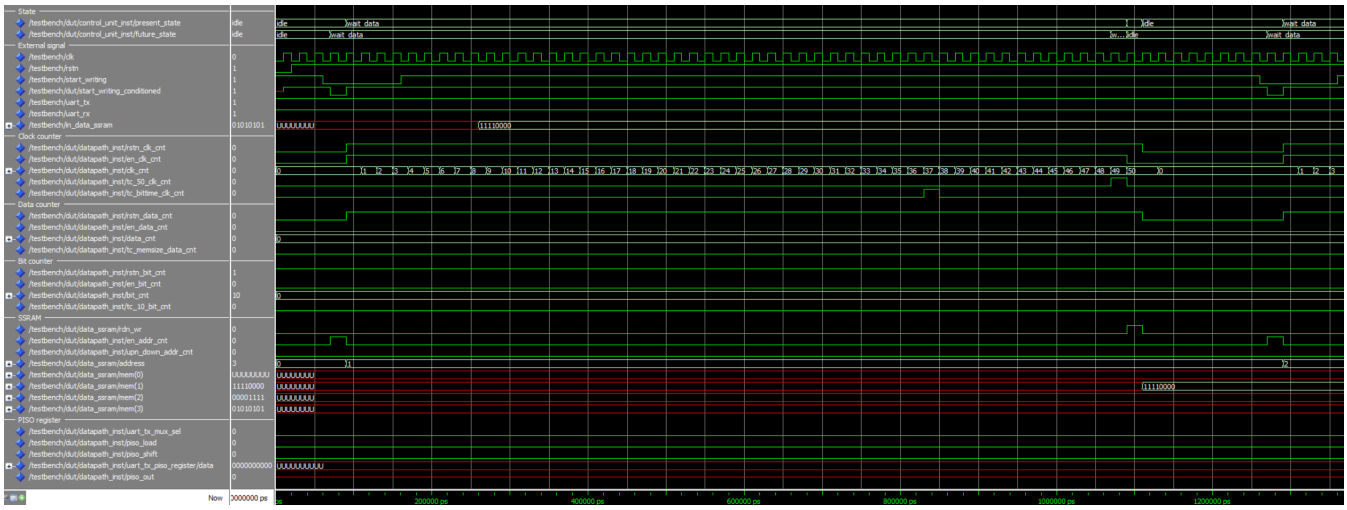


Figure 4.2: Simulation of the first data writing.

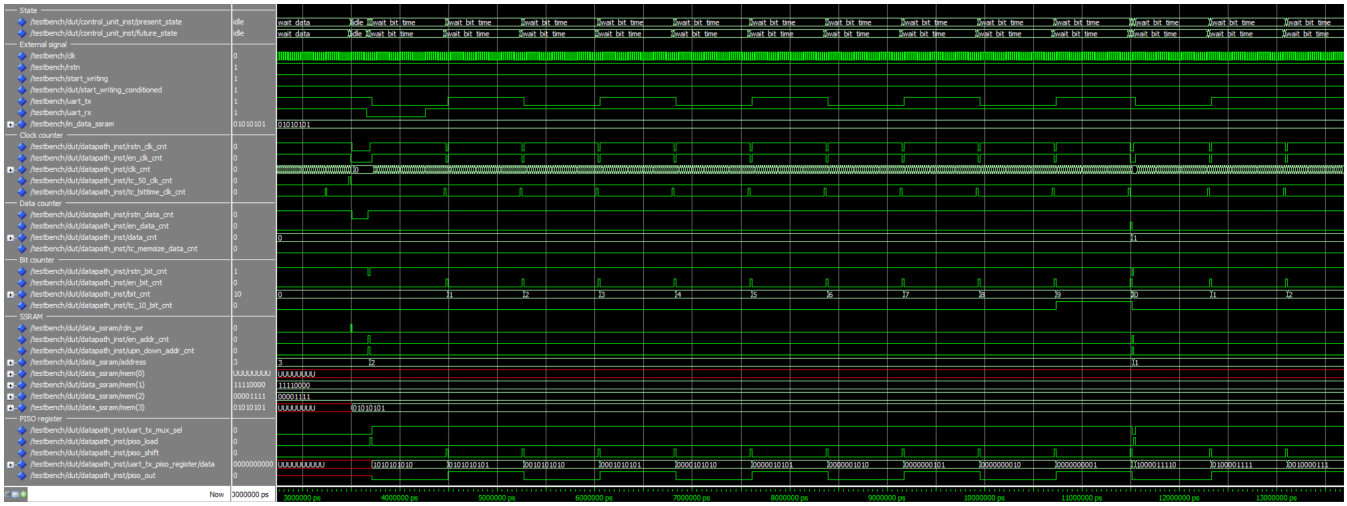


Figure 4.3: Simulation of the first data transmission.

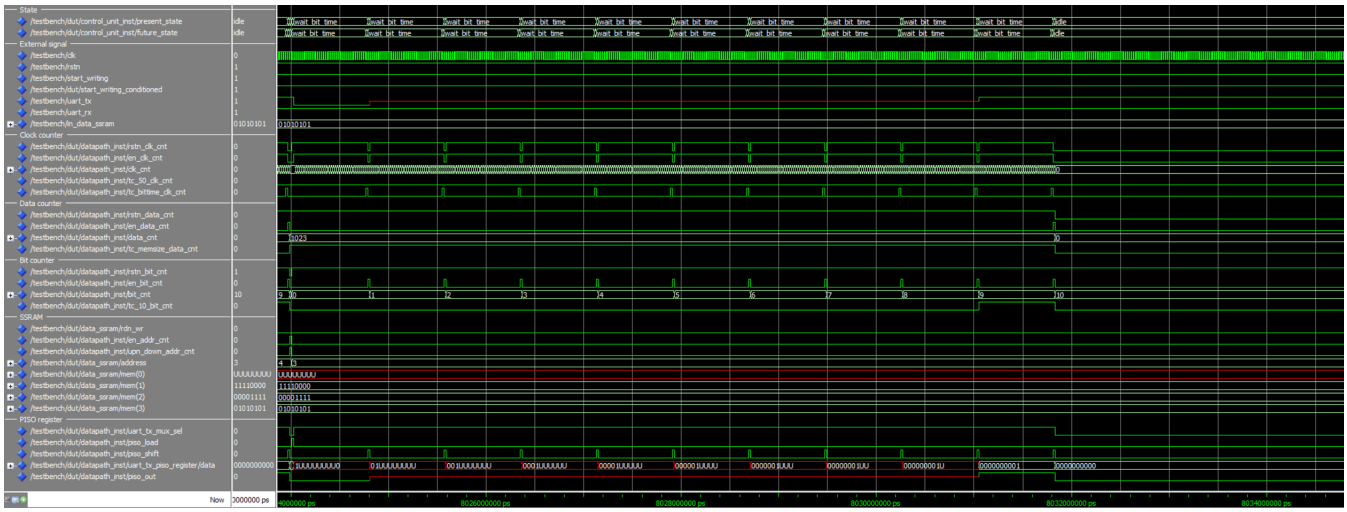


Figure 4.4: Simulation of the transmission end.

Listing 4.21: testbench.vhd

```
----------------------------------------
--                                    --
--  file:       testbench.vhd         --
--  author:     Nicola Vianello       --
--  last edit:  Turin 04/01/2019      --
--                                    --
```

```
--------------------------------------

library ieee;
use ieee.std_logic_1164.all;

entity testbench is
end entity testbench;

architecture tb of testbench is

constant clk_period: time :=20 ns;

component top_level is
port(   start_writing, UART_rx, clk, rstn : in std_logic;
        in_data_ssram : in std_logic_vector(7 downto 0);
        UART_tx : out std_logic
);
end component top_level;

signal start_writing, UART_rx, UART_tx, clk, rstn : std_logic;
signal in_data_ssram : std_logic_vector(7 downto 0);

begin

    dut: top_level port map(start_writing=>start_writing, UART_rx=>UART_rx, clk=>clk, rstn
         ↪ =>rstn, in_data_ssram=>in_data_ssram, UART_tx=>UART_tx);

    clk_gen: process is
    begin
        clk <= '0';
        wait for clk_period/2;
        clk <= '1';
        wait for clk_period/2;
    end process clk_gen;

    stim_gen: process is
    begin

        --init
        rstn<='0';
        UART_rx<='1';
        start_writing<='1';
        wait for clk_period;
        rstn<='1';
        wait for 2*clk_period;

        --write data 1
        start_writing<='0';
        wait for 5*clk_period;
        start_writing<='1';
        wait for 5*clk_period;
        in_data_ssram<="11110000";
        wait for 50*clk_period;

        --write data 2
        start_writing<='0';
        wait for 5*clk_period;
        start_writing<='1';
        wait for 5*clk_period;
        in_data_ssram<="00001111";
        wait for 50*clk_period;

        --write data 3
        start_writing<='0';
        wait for 5*clk_period;
        start_writing<='1';
        wait for 5*clk_period;
        in_data_ssram<="01010101";
        wait for 50*clk_period;

        --outuput request
        uart_rx<='0';
        wait for 30*clk_period;
        uart_rx<='1';
```

```
        wait;
    end process stim_gen;

end architecture tb;
```

4.3 PC interfacing program

The PC interfacing script is written in python using the pySerial module, the functioning is very simple: once it is connected to the serial port it enters into an infinite loop in which each time it asks the user if he want to finish the program execution or he want to request data, if the data are requested, the file that will be written is cleared and the NULL character (which corresponds to eight zeros) is transmitted as start signal for the data transmission. Subsequently the received data are taken one by one, and for each data a string is made formed by the data sampling time, the real data, and the corresponding temperature. To obtain the data sampling time, the current time is assigned to the first received data, and then one second is subtracted for each incoming data, so there will be a maximum error of 1 s. To obtain the temperature the same formula also employed by the microcontroller is used. These strings will be both added to the file and printed in the terminal.

Listing 4.22: temperature_monitor.py

```
########################################
#                                      #
#   file:       temperature_monitor.py #
#   author:     Nicola Vianello        #
#   last edit:  Venice 25/02/2019      #
#                                      #
########################################

import time
import serial

ADC_REF=5.1                 #ADC voltage reference
ADC_BIT=8                   #ADC bit number
SENSOR_GAIN=0.01            #sensor gain in V/Celsius
CONDITIONING_GAIN=9.15      #signal conditioning circuit gain

TEMPERATURE_GAIN=SENSOR_GAIN*CONDITIONING_GAIN
ADC_GAP=ADC_REF/(1<<ADC_BIT)

mySerial = serial.Serial(
    port='COM7',
    baudrate=1280000,
    parity=serial.PARITY_NONE,
    stopbits=serial.STOPBITS_ONE,
    bytesize=serial.EIGHTBITS
)
```

```
mySerial.isOpen()

while 1 :
    print ('Insert "exit" to leave the program.\r\nInsert "start" to request data.')
    command = input()
    if command == 'exit':
        mySerial.close()
        exit()
    elif command == 'start':
        file = open("output.txt", 'w')
        file.write('')
        file.close()
        file = open("output.txt", 'a')
        mySerial.write(b'\0')
        for n in range(1024):
            currentTime = time.localtime(time.time()-n)
            while mySerial.inWaiting() == 0:
                pass
            data = int.from_bytes(mySerial.read(1), byteorder='big', signed=False)
            temperature = data*ADC_GAP/TEMPERATURE_GAIN
            line = '@ ' + str(currentTime.tm_hour) + ':' + str(currentTime.tm_min) + ':' +
                 ↪ str(currentTime.tm_sec) + '\tADC output: ' + str(data) + '\t\
                 ↪ tTemperature: ' + str(temperature) + chr(176) + 'C'
            print (line)
            file.write(line + '\n')
        file.close()
```

Bibliography

[1] Altera, *Cyclone V Device Datasheet*, October 2011 - revised 12/06/2015.

[2] Altera, *DE0-Nano-Soc, User Manual*, V1.7, 11/04/2018.

[3] Diodes Incorporated, *1N5817-1N5819, 1.0A SCHOTTKY BARRIER RECTIFIER*, DS23001, Rev. 8-2.

[4] Elec Freaks, *"Ultrasonic Ranging Module HC-SR04"*.

[5] Hitachi Ltd., *"HD44780U (LCD-II)"*, ADE-207-272(Z), Rev. 0.0, 1998.

[6] Mazzoldi P., Nigro M., Voci C., *"Fisica Volume 1"*, 2° ed., EdiSES, 24/01/2001, ISBN 8879591371.

[7] Microchip Technology Inc., *"Compiled Tips 'N Tricks Guide"*, DS01146B, 26/03/2009.

[8] Microchip Technology Inc., *"PIC24FJ64GB004 Family Data Sheet"*, 39940D, ISBN: 978-1-60932-439-1, 15/07/2010.

[9] ON Semiconductor, *"2N3903, 2N3904 General Purpose Transistors"*, 2N3903/D, Rev.8, August 2012.

[10] Passerone Claudio, materials for the course: *"Electronics for Embedded Systems"*, 2018-2019.

[11] Shenzen Eone Electronics Co. Ltd., *"Specification For LCD Module 2004A"*, Ver1.0, 21/09/2007.

[12] STMicroelectronics, *L4960, 2.5A POWER SWITCHING REGULATOR*, June 200.

[13] STMicroelectronics, *LD1117 SERIES, LOW DROP FIXED AND ADJUSTABLE POSITIVE VOLTAGE REGULATORS*, Rev.19, December 2005.

[14] Texas Instruments, *"ADC080x 8-Bit, μP-Compatible, Analog-to-Digital Converters"*, SNOSBI1C, November 2009 - revised June 2015.

[15] Texas Instruments, *"LM35 Precision Centigrade Temperature Sensors"*, SNIS159G, August 2016 - revised December 2017.

[16] Texas Instruments, *"OPA347 OPA2347 OPA4347, microPower, Rail-to-Rail Operational Amplifiers"*, SBOS167D, November 2000 - revised July 2007.

[17] Vishay Semiconductors, *1N4148, Small Signal Fast Switching Diodes*, 81857, Rev. 1.4, 06/07/2017.