Alma Mater Studiorum
Università di Bologna

School of Engineering and Architecture
Information Technology Department
— Science and Engineering —

Degree in Computer Engineering

Degree Thesis
in
Calcolatori Elettronici T

Arduino projects for educational purposes

candidate
Riccardo Muggiasca

supervisor
prof. ing. Stefano Mattoccia

Academic Year 2017/2018
Session VI – March 14th, 2019

Abstract

The thesis is about the necessity of implementing a new array of simple and yet stimulating experiments with the use of the widely known **Arduino Uno** microcontroller board. These exercises will be inserted into the course programme of **Calcolatori Elettronici T** (Electronic Calculators – First-Cycle Degree) hosted by professor **Stefano Mattoccia** at the *School of Engineering and Architecture*, **Bologna University (Italy)**.

The work has been organized in the form of *ready-to-action* laboratory exercises, complete with methodology, hardware and software components. These elements will prove useful for studying the basic principles of computing hardware design and programming, central themes of the professor's course, in the hope to captivate the interest of students and to make them become more focused and passionate about the subject itself.

Contents

Chapter 1

Motivations

1.1 Introduction

Well known for its era, the ATmega 328P[1] CPU by Atmel[2] was invented in the 1990s and broadly used in the electronics industry ever since, to take advantage of its speed and versatility. In recent years, that single 8-bit RISC[3] microprocessor has been implanted onto the Arduino Uno[4] board and it is allowing all kinds of experiments, made by millions of interested students and enthusiasts, doing it for work, research or simply leisure.

When these classes of people intersecate, it might mean you are facing yourself with a Computer Science or Computer Engineering student. This short lecture has been written specifically for this kind of audience, but every other passionate subject might find some benefit from this relatively short, but nonetheless dense lecture, with a little bit of extra work and self documentation.

Whether he or she will get a better orientation on the matter of tinkering with the Arduino by doing some "simple" experiments, is left to the reader's own consideration.

1.2 What are we going to talk about?

Proceeding step by step in configuration and method, we are going to start from the hardware level, with a fundamental exercise regarding assembly language[5] and how it handles hardware resources and data, to actually execute programmed instructions.

Then, we will proceed further by building a couple of practical examples of real world applications, such as a LED[6] scoreboard lookalike and a sonar proximity sensor.

The latter of these experiments also features an interrupt[7] procedure calling scenario, that will show us how firmware programs can handle an interrupt signal situation and how we can set them in doing so.

Chapter 2

The AVR Development Environment

2.1 Definitions

To put our laboratory experiments into practice, we need dedicated pieces of hardware and software, which are all properly described inside their own dedicated chapters.

Every piece of software has been written in pure **C language**[8], without any of the comfort libraries naturally offered by the official Arduino project[9], in order to better interact with all the hardware elements of the Arduino board.

In this case, the open-source **AVR programming environment**[10] comes to fruition, a set of C program libraries designed to directly interact with the ATmega 328P microprocessor, placed alongside an extract of the **C standard library**[11, 12]. We use just a small subset of those libraries, the ones strictly needed to allow us to put our ideas into practice.

The computing environment in which we operate is **Ubuntu Linux**[13], which is in turn based on the **Debian/GNU Linux**[14] operating system; any Debian-derived distribution released in recent years fully fits the purpose of what we do.

The only other mandatory requirement to fulfil is to have a standard USB 2.0 (or newer/faster) port available, in order to properly connect the Arduino board to the PC, to power it and to be able to program its internal firmware to suit our needs.

2.2 Installation

Once we have our Ubuntu Linux PC up and running, we need to prepare our development environment. Open a terminal window[15], whether it is a simple terminal emulator or a secondary *tty* interface and install the necessary software components, along with all of their packets' dependencies[16]:

```
sudo apt-get update
sudo apt-get upgrade
sudo apt-get install gcc-avr gdb-avr binutils-avr avr-libc avrdude
```

A little explanation of what these commands mean:

- `sudo` is the directive that allows us to gain superuser privileges. If enabled to do so[1], we are asked for our user's password in order to be temporarily granted administrator privileges and install the software we need;

[1] Please refer to proper Linux administration guidelines to get help on this topic.

- `apt-get` is the easiest Ubuntu's package manager tool we can interact with;
- `update` checks for updates on every current software installed on our system;
- `upgrade` installs updates to every software packages the `update` command found to be, in fact, updated;
- `install <packet_list>` installs the specified software packets and, by default, all of their own dependencies, which are other relevant software the declared packets use to function properly; more in detail we have to install[17]:
  - `gcc-avr`
    the customized gcc compiler able to interact with the AVR program libraries;
  - `gdb-avr`
    the gcc AVR debugger, which we do not use, but still needed because of some dependencies' contraints;
  - `binutils-avr`
    the AVR binary utilities, such as linker (`avr-ld`), object file disassembler (`avr-objdump`) and extractor (`avr-objcopy`). We are going to indirectly use just the linker (`avr-ld`) through the compiler (`avr-gcc`), the other tools are still needed because of dependencies' contraints;
  - `avr-libc`
    the AVR C program library set, essential to compile our firmware;
  - `avrdude`
    firmware loader utility program, essential to load our firmware onto the Arduino board through the USB interface;

We also need an advanced text editor to write C program files. I recommend choosing any text editor which supports syntax highlighting, so as not to get lost among the lines of code we have to deal with. Many of the built-in text editors of Ubuntu are fully compatible with general programming writing tasks.

2.3 Compiling and Loading our Firmware

Every laboratory experiment requires the selected hardware to be properly assembled together and the programmed firmware to be loaded onto the Arduino board to make everything work.

To prepare our custom software to be loaded and executed we have to make use of the development tools installed by the procedure just described in section 2.2 by using the following commands[16, 18, 19, 21]:

```
sudo chmod a+rw /dev/ttyACM0
avr-gcc -Os -mmcu=atmega328p -DF_CPU=16000000UL -o prog.elf prog.c aux.c
avrdude -F -V -c arduino -p ATmega328P -P /dev/ttyACM0 -b 115200
        -U flash:w:prog.elf:e
```

Here follows the detailed explanation of these commands:

- `sudo chmod a+rw /dev/ttyACM0`[18]
  Enables the `/dev` Arduino (device) interface to be read and written by anyone in the system, removing the need to prepone `sudo` to any other command regarding the Arduino device. The setting is applied only until the Arduino remains connected to the PC; if disconnected, the command has to be executed again.
  NOTE: verify if your Arduino's interface name is one of the following: `/dev/ttyACM0`, `/dev/ttyUSB0`, `/dev/ppi0` or `/dev/cuaa0` so it has to be found in the `/dev` system directory and specified accordingly.

- `avr-gcc` is our firmware's compiler[19];
  `-Os` makes sure to apply all the second-level compiler optimisations which do not inflate the final binary code, because our firmwares are simple program routines and must fit inside the 32 KB of the Arduino's flash memory[20];
  `-mmcu=atmega328p` specifies which architecture to compile our firmware for;
  `-DF_CPU=16000000UL` sets the ATmega 328P's clock frequency to 16 MHz for the program code to be compiled and executed. The letters `UL` at the end are indicative of the `unsigned long` numeric data type;
  `-o prog.elf` declares the name of the compiled final object file. The extension is arbitrary, but the `avr-gcc` compiler outputs an ELF-compatible file (in Executable and Linkable Format) so the `.elf` extension is suitable;
  `prog.c aux.c` is just a list of C program files that have to be compiled in one executable module, hence their arbitrary name. Necessary header files are automatically included only if proper `#include` directives are written inside the `.c` files;

- `avrdude` is our firmware's loader[21];
  `-F` overrides check of device signature, so we can use the command with every Arduino on earth without getting non-relevant error messages;
  `-V` disables integrity checks for blocks of data when uploading to Arduino;
  `-c arduino` specifies which method of programming (loading) has to be used, that is heavily hardware dependent;
  `-p ATmega328P` declares the type of MCU (Micro Controller Unit) present on the Arduino board connected to the PC;
  `-P /dev/ttyACM0` the device to which the programmer is attached to the PC. Same **NOTE** of the first command also applies here;
  `-b 115200` sets an explicit upper boundary for the baud rate to be applied. Default value is unknown, so we normally use 115200 because it is specified in the official Arduino documentation[22];
  `-U flash:w:prog.elf:e` can be seen as a specification for:
  `-U memory-type:write-directive:file-to-load:file-format-specifier`
  In our case everything is referred to the ELF file format, which is the most common compiled binary program format for UNIX-derived systems and the like[23];

Chapter 3

The AVR Arduino custom library

To write well done and efficient programs always requires effort. We could write everything on our own, or we could choose to get some help to solve the issues that eventually arise.

Apart from the supporting/default/standard C libraries we use for our laboratory experiments, there is also a custom-made library to employ, which simplifies the aspects of machine control and data register operations (useful when communicating with secondary hardware devices).

Here is its detailed explanation, worth of a deep look, before moving on with the actual experiments.

3.1 How it is made

`avr_arduino` is a library specifically designed to realize the fundamental and common pieces of code found inside the laboratory experiments described in this thesis.

It is not delivered as a *ready-to-use* pre-compiled black box, because of the divulgative reasons that accompany the experiments with Arduino we are going to discuss.

This simple C program library is in fact composed by the files:

- `avr_arduino.h`
  the library's header file, to be included in all of our laboratory projects;
- `avr_arduino.c`
  the library's program file, which implements and makes use of the constant and function declarations found inside the header file;

We now introduce these two files in more detail, starting on the next page, by looking at their source code and commenting it where necessary.

3.2 avr_arduino.h

```
// our library definition
#ifndef AVR_ARDUINO
#define AVR_ARDUINO

// inclusion of fundamental libraries
#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/delay.h>

// macro definitions
#define BYTE unsigned char
#define IN 0
#define OUT 1
#define LOW 0
#define HIGH 1

// single-bit port value extraction
BYTE port_bit(BYTE port,BYTE bit_index);

// single-bit port register assignment
BYTE port_setup(BYTE port,BYTE bit_index,BYTE value_direction);

// full-byte (with exception) port register assignment
BYTE port_setup_full(BYTE port,BYTE value_direction);

#endif
```

3.2.1 C preprocessor directives

Each declaration starting with `#` is a C source preprocessor directive[24], which helps the compiler to get the right amount of source code to verify and build.

`#ifndef` is a preprocessor directive made to verify if the subsequent amount of code has already been included in the compiling process (or `#define`d, as the following directive says) by giving it a unique name.

`#endif` at the end closes the conditional source inclusion block. These directive structures are commonly called as *include guards*[25]. A double occurrence of the same entity always causes a fatal error to the C compiler.

3.2.2 Inclusion of external libraries

`#include` is the directive dedicated to actually "include" other pieces of code inside the current source file, to get a full reference on which external programming functionalities are available and how to use them. They will later be compiled and linked together with our own program's code.

Standard libraries contained inside the C compiler include paths are to be written as a relative path inside the < and > symbols; custom-made libraries (therefore located outside such inclusion paths) have to be specified with their absolute path instead, written inside the "" symbols.

To make our custom library work, we have to include the following source components[12]:

- `avr/io.h`
  Contains every low-level machine reference on registers and input/output interfaces. The target microcontroller hardware is always specified at compile time, as seen in section 2.3.

- `avr/interrupt.h`
  It offers various functionalities that are dedicated to interrupt signals' handling.

- `util/delay.h`
  As its name suggests, contains execution delay functions, useful for timed instruction execution, as we will be able to see during the lab experiments.

3.2.3 Macro definitions

In C programming, macros are basically small sections of code that are rewritten in a more synthetic and comprehensible way, to be substituted with their original content by the C preprocessor at compile time[24]. Taking by example the first of them we have:

```
#define BYTE unsigned char
```

This directive defines the `unsigned char` data type as being named simply `BYTE` (all in capital letters). This is not a new data type[1], but instead it is just a simpler way to write the data type specification "`unsigned char`" in a more compact and easy format, inside every source file that might include this `avr_arduino.h` header file in particular.

The rest of them are just alternative names for the numbers 0 (`IN,LOW`) and 1 (`OUT,HIGH`) and will prove useful when writing calls to the functions we are about to introduce.

3.2.4 Function declarations

Last but not least, every function in a C program which is not identified as `main()` (the function in which the program execution starts) needs to be declared first and then defined second[26], especially if it is located in a different `.c` file (or module) than the `main()` function.

The `avr_arduino` library contains the following functions:

- `BYTE port_bit(BYTE port,BYTE bit_index);`
  It is capable of extracting the single bit value contained inside the `port` variable at position `bit_index`, the latter ranging from 0 to 7 (eight bits in total).

- `BYTE port_setup(BYTE port,BYTE bit_index,BYTE value_direction);`
  It sets the value expressed by `value_direction` of the bit indicated by `bit_index` inside the 8 bit word identified by `port`. It then returns the updated value of the same `port` passed as input, to be assigned to the correct port selected by the programmer.

[1] See `typedef` directive in the C language specification.

- `BYTE port_setup_full(BYTE port,BYTE value_direction);`
  Same behaviour as the first function, but missing the `bit_index` parameter. This means the "`full`" function replicates the `value_direction` bit for each and every bit inside the selected `port` register.

Let us now have a look to the minimal peculiarities that are common to each of them.

3.2.4.1 Ports

The name "port" has been chosen to reflect the actual communication ports of the ATmega 328P microprocessor and, by extension, the ones available on the Arduino board[27]. Those are identified as `PORTB`, `PORTC` and `PORTD`.

`PORTB`, and `PORTD` have an 8 bit resolution, while `PORTC` has just a 7 bit resolution, because of the technical design and package characteristics of the ATmega 328P itself[28].

Same properties reflect to the Data Direction Registers, which are basically the setup registers for the pins of those three ports we mentioned. Those are `DDRB`, `DDRC` and `DDRD` and their resolution is the same as their `PORT` counterparts.

3.2.4.2 Bit indexes

Bits are of course numbered from 0 to 7 and correspond to a set of keywords widely used in AVR low-level programming.

For instance, considering `PORTB`, we have {`PORTB0`,`PORTB1`,...,`PORTB7`} and the same goes for the other two ports (remember that `PORTC` has only 7 bits to operate on).

Same rule is applied to the Data Direction Register bits; considering `DDRB`, we have {`DDB0`,`DDB1`,...,`DDB7`} and the format is equally applied to the other two registers.

3.2.4.3 Value or direction?

Here is why those `#define` directives of alternative names for the numbers 0 and 1 happen to be useful, as described in section 3.2.3.

When handling ports, it comes natural to think about the logical level, or value, that pin or port should assume, so we could say `LOW` or `HIGH` respectively.

Instead, when thinking about data direction settings, we are more naturally inclined towards the keywords `IN` and `OUT`.

In reality, there is no difference between the values we use, but in this way these functionalities become easier to understand and code.

3.2.4.4 Data types

Each data type used by those functions is denoted as `BYTE`, the same data type defined earlier in section 3.2.3. Having to dialogue directly with the 8 bit registers of the ATmega 328P microprocessor, a `BYTE` is in fact the only data type we care about and it is just a binary number, independent of the meaning we might associate to it.

In this way, we build a *weakly typed* programming environment in which the meaning of data is directly handled by us; this gives us the possibility to fully understand what we are doing, which is one of the main goals we want to achieve.

3.3 avr_arduino.c

```
// inclusion of this library's header file
#include "avr_arduino.h"

// single-bit port value extraction
BYTE port_bit(BYTE port,BYTE bit_index)
{
  return port & (1 << bit_index);
}

// single-bit port register assignment
BYTE port_setup(BYTE port,BYTE bit_index,BYTE value_direction)
{
  if(value_direction)
    port |= (1 << bit_index);
  else
    port &= ~(1 << bit_index);
  return port;
}

// full-byte port register assignment
BYTE port_setup_full(BYTE port,BYTE value_direction)
{
  // PORTC resolution is 7 bits instead of 8
  // because Arduino and ATmega328P are built this way
  if(port == PORTC)
  {
    if(value_direction)
      port = 0b1111111;
    else
      port = 0b0000000;
  }
  else
  {
    if(value_direction)
      port = 0b11111111;
    else
      port = 0b00000000;
  }
  return port;
}
```

3.3.1 Inclusions and definitions

The `avr_arduino.c` implementation file starts with the inclusion of its own header, containing all the macro and function declarations we have seen earlier, so as to employ them properly inside this C module.

3.3.1.1 `port_pin()` function

```
BYTE port_bit(BYTE port,BYTE bit_index)
{
  return port & (1 << bit_index);
}
```

3.3.1.1.1 Behaviour The function's body is made by just a single line of code: it returns the bitwise AND logical operation's result between the parameter `port` and the constant value 1, logically shifted by `bit_index` positions.

3.3.1.2 `port_setup()` function

```
BYTE port_setup(BYTE port,BYTE bit_index,BYTE value_direction)
{
  if(value_direction)
    port |= (1 << bit_index);
  else
    port &= ~(1 << bit_index);
  return port;
}
```

3.3.1.2.1 Behaviour It realizes the behaviour described in 3.2.4 by evaluating if the least significant bit of `value_direction` is different from 0.

If so, it shifts the number 1 (00000001 in binary) for `bit_index` positions and puts it in logical OR with the 8 bit word given by `port`, thus setting the bit found in position `bit_index` to 1 if it was originally 0, or leaving it to 1 if it was already of that value.

If the least significant bit of `value_direction` is actually 0, then the function shifts the number 1 (00000001 in binary) for `bit_index` positions, performs a bitwise NOT (thanks to the *tilde* operator) and puts it in logical AND with the 8 bit word given by `port`, thus setting the bit found in position `bit_index` to 0, independently of its original value.

The final result of each operation is set into the `port` variable and returned to the function caller at the end.

3.3.1.3 `port_setup_full()` function

```
BYTE port_setup_full(BYTE port,BYTE value_direction)
{
  // PORTC resolution is 7 bits instead of 8
  // because Arduino and ATmega328P are built this way
  if(port == PORTC)
  {
    if(value_direction)
      port = 0b1111111;
    else
      port = 0b0000000;
  }
```

```
  else
  {
    if(value_direction)
      port = 0b11111111;
    else
      port = 0b00000000;
  }
  return port;
}
```

3.3.1.3.1 Behaviour At first, it verifies if the `port` variable's value is equivalent to the one of `PORTC`. If so, the assignable word to the `port` variable will be of 7 bits of resolution instead of 8.

Finally, the function operates a full assignment of a zero-word or a one-word to the `port` variable and returns its value to the function caller at the end.

Chapter 4

Assembler

The first laboratory experiment we tinker with is the assembler programming exercise, that stands out a little from the others that follow. To be able to handle the computer machine properly and to program it in the best and efficient way, any knowledge on the underlying hardware architecture is always welcomed and essential.

4.1 Why we do this

When in need to stay in close proximity to the hardware level, a number of considerations must take place. We don't just need to translate our program logic into code, we also have to be careful about the physical architecture our microprocessor has been designed on, by taking into consideration its functioning and efficiency issues.

Chapter 3 introduced not only two useful libraries we are going to use for our experiments, but also some hardware notions that justified their development. This is our simplified way to have a look inside the ATmega 328P microprocessor, to see what happens with the CPU registers and to be able to directly pilot data flow and instruction execution.

4.2 Differences with DLX

This example can be put in parallel with what one might have studied about DLX (DeLuXe) architecture scheme and instruction set[29]. The ATmega 328P is based on a very similar model, but astrays itself from it, because of its proprietary origin and design.

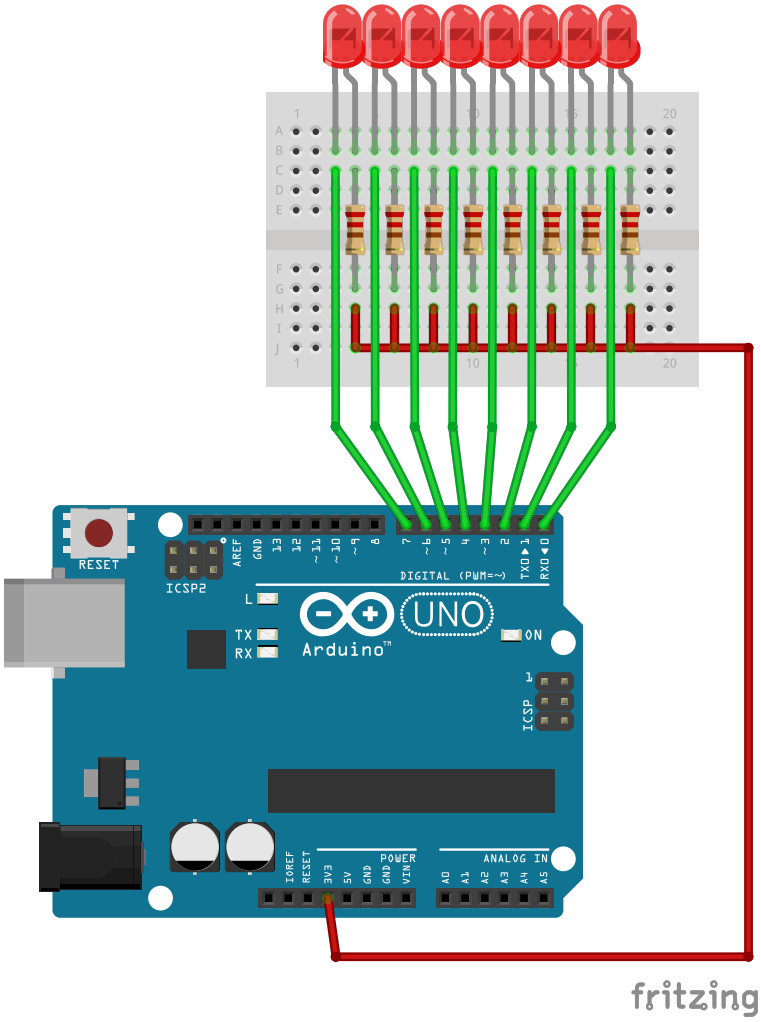
Any comment on the existing differences between the two assembly language versions is denoted inside the respective sections that follow.

4.3 What we are going to do

In this laboratory experiment we will be able to realize some simple math and logic operations with two 8 bit data words (named `op1` and `op2`) and to have them directly executed by sections of explicitly written assembler code, instead of by the usual pure C language math operators.

The final result of each of these operations will be shown in binary format by an 8 bit led row, properly activated by our code.

4.4 Schematic



4.4.1 Technical notes

Standard Red Led supply[20]	
Voltage	2.0 V
Current	20 mA

Arduino PIN supply[31]	
Voltage	5.0 V or 3.3 V
Current	40 mA or 20 mA

Every standard red led canonically functions with a supply voltage of 2.0 V and a current of 20 mA, while the Arduino can power up small devices in configurations of 5.0 V and 3.3 V, with each data pin outputting about 40 mA or 20 mA in current, depending on the Arduino's hardware revision.

A supply current of 40 mA can be too much for our leds, which in almost every hardware application need a proper resistor to pull down the supply current a bit, so as not to break them.

The component used in this example is a pre-built 8 bit led row, already equipped with parallel resistors for each one of the leds. If we are not able to use such a component, 8 red leds with a pull-up resistor of about 220 Ω each (as shown above in the schematic) should work fine.

4.5 What we need

The components for this experiment are:

- an Arduino Uno compatible board
- a USB printer cable
- a 8 red led row
- the source project files, consisting of:
  - the assembler header (`assembly.h`) and instructions (`assembly.sx`) files
  - the utility "led row" source files, header (`led8.h`) and implementation (`led8.c`)
  - the main program module (`assembly.c`)

Here follows the detailed explanation for each source file in this order. The assembly implementation file is of extension `.sx` instead of `.c`, it is not a mistake, but this aspect is going to be explained in a later section.

4.6 assembly.h

```
// global register variables declaration
// the name the compiler defines when handling assembler code
#ifdef __ASSEMBLER__

#define op1 r2
#define op2 r4
#define res r6

#else // not in assembler compiling context
      // so we compile C language instead

register unsigned char op1 asm("r2");
register unsigned char op2 asm("r4");
register unsigned char res asm("r6");

#endif
```

4.6.1 Preprocessor directives

Conditional definition of `__ASSEMBLER__` context, which is a standard name for the `avr-gcc` compiler to refer to if dealing with assembler code. The file is in fact organized to be handled both by the C compiler and/or the Assembler compiler[32].

4.6.2 Macro definitions

The key point of this header file is to define names that differ from the explicit internal general purpose registers of the ATmega microprocessor, in order to have a better comprehension of which data we are dealing with.

Both sections in the header file define alternative names to be used for registers `r2,r4,r6` by mapping them respectively to names `op1,op2,res` which mean: operand one (`op1`), operand two (`op2`), final result (`res`).

4.6.3 `register` variables

More specifically, the second section of the header file (after the `#else` directive) defines three `register` variables[33], by the same name-register association described earlier.

Those variables will be directly mapped onto the specified registers in the assembly (`asm`) context and operations that involve their use will apply their effect on the mapped internal registers of the ATmega 328P microprocessor.

4.7 assembly.sx

```
#include "assembly.h"

; simple operational math
; and logic functions
; taken from the official
; ATmega 328P datasheet

; arithmetic operations
; push/pop methodology

.global asm_add
asm_add:
  push op1
  add op1,op2
  mov res,op1
  pop op1
  ret

.global asm_sub
asm_sub:
  push op1
  sub op1,op2
  mov res,op1
  pop op1
  ret

; MUL result is placed in R1:R0
.global asm_mul
asm_mul:
  push op1
  mul op1,op2
  mov res,r0
  pop op1
  ret

; logical operations
; mov to a register methodology

.global asm_and
asm_and:
  mov r12,op1
  and r12,op2
  mov res,r12
  ret

.global asm_or
asm_or:
  mov r12,op1
  or r12,op2
  mov res,r12
  ret

.global asm_xor
asm_xor:
  mov r12,op1
  eor r12,op2
  mov res,r12
  ret

; we just shift the previous result
; we had from a past operation

.global asm_lsl
asm_lsl:
  lsl res
  ret
```

```
.global asm_lsr
asm_lsr:
  lsr res
  ret
```

4.7.1 Why `.sx` and not `.c`?

The reason why this assembly program module brings the `.sx` extension is simple; according to the `avr-gcc` specifications, it is possible to actually mix C and Assembly language inside the same firmware project[32].

More than that, we are able to write "special" source files, in which C and Assembler code can coexist and interact without special declarations, but one: they must have the `.sx` extension, in order to be properly recognised by the compiler.

Afterwards, the C and the Assembly compiler are automatically called whenever necessary and the program code gets compiled in its entirety.

4.7.2 Inclusions and definitions

The `assembly.sx` program module includes the `assembly.h` header, containing all the macro declarations we have seen earlier.

It then defines the math and logical functions, which realize the following operations:

Operation Type	Description	Function Name
arithmetical	sum	`asm_sum()`
	subtraction	`asm_sub()`
	multiplication	`asm_mul()`
logical	and	`asm_and()`
	or	`asm_or()`
	exclusive or	`asm_xor()`
	logical shift left	`asm_lsl()`
	logical shift right	`asm_lsr()`

We now take some of the functions and examine them in more detail, as an example of how the code has been written[34].

4.7.2.1 `asm_add()` function

```
.global asm_add
asm_add:
  push op1
  add op1,op2
  mov res,op1
  pop op1
  ret
```

4.7.2.1.1 Labels `.global` is the assembly directive which declares the code label `asm_add` as a callable reference in the whole program, not just inside the `assembly.sx` module[35].

When programming directly with assembly language, we have in fact to reason not by "functions", but by relative memory addresses in which to find the next program

instruction to execute. Labels such as `asm_add` allow us to directly jump the execution to the desired section of our program.

4.7.2.1.2 Behaviour This sequence of instructions simply "pushes" to the stack[36] the contents of the register identified by `op1`; it then adds the content of `op1` with `op2` and puts the operation's result inside the `op1` register.

This final result is copied (`mov`ed) inside the `res`(ult) register and finally the original `op1` value is "popped" off the stack and put inside the actual register `op1`, restored to be used in a possible next operation.

The `ret` instruction virtually "closes" the function, by terminating the current instruction sequence and returns to the caller.

4.7.2.2 `asm_xor()` function

```
.global asm_xor
asm_xor:
  mov r12,op1
  eor r12,op2
  mov res,r12
  ret
```

4.7.2.2.1 Behaviour This sequence of instructions simply copies to the register `r12` (arbitrarily chosen by the programmer) the contents of the register identified by `op1`.

It then executes and exclusive OR logical operation between `r12` and `op2` and puts the result inside the `r12` register.

From there, the result is copied into the `res` register and the sequence of instructions gets terminated by the ret directive.

4.7.2.3 `asm_lsl()` function

```
.global asm_lsl
asm_lsl:
  lsl res
  ret
```

4.7.2.3.1 Behaviour In this case the instruction sequence is plainly simple: the content of `res` register is shifted to the left by one bit and the block terminates right after with the `ret` instruction.

4.7.2.3.2 Proper use This function needs preparation by the program section which calls it. It is needed to have a significant value inside the `res` register *before* calling this procedure, because it only shifts a register's content without making any assignment first. This is something to be handled externally[34].

4.8 led8.h

```
#ifndef LED8
#define LED8

#define ONE_SEC 1000
#define TWO_SEC 2000
#define BLK_SEC 250

void led8_init();
void led8_start();
void led8_operands();
void led8_math();
void led8_logic();
void led8_shift();

#endif
```

4.8.1 Hardware compatibility disclaimer

Please keep in mind that it is not necessary to have exactly the same led hardware component that has been used in this project.

Instead of a pre-built 8 led row it is possible to make use of 8 single red leds, connected to the Arduino in the same way the schematic described in section 4.4 and they remain fully compatible with the illustrated program code.

4.8.2 Preprocessor directives

Conditional definition of `LED8` section name, containing a block of constants and function declarations related to the 8 led row used in the experiment. As already described in section 3.2.1, this is done to prevent multiple inclusions of a header file.

4.8.3 Macro definitions

Only three macros are defined in this header file and they are all referred to some standard milliseconds amounts to be used in the implementation file. `ONE_SEC` and `TWO_SEC` can be considered self-explanatory, `BLK_SEC` instead refers to a "blink" seconds time interval, which in fact lasts just a quarter of a second.

Those are employed to delay code execution a bit, to give the user a small time to properly read the leds' output configuration.

4.8.4 Function declarations

Six functions are declared, each of them with no explicit parameter, no return data and with the "`led8`" prefix, to be easily recognisable in the code.

- `led8_init()`
  Initialises the ATmega 328P communication port to which the 8 leds are attached;
- `led8_start()`
  Generates an output start sequence for the 8 leds, by making them blink a number of times (this is where `BLK_SEC` is used);
- `led8_operands()`
  Shows the binary output of the two numeric operands used for the math and logic operations;
- `led8_math()`
  Executes the math operations explained in section 4.7.2 and outputs their result on the 8 leds;
- `led8_logic()`
  Executes the logic operations explained in section 4.7.2 and outputs their result on the 8 leds;
- `led8_shift()`
  Creates a sample result and shifts it left and right on the 8 bits, as if it is "moving";

4.9 led8.c

```
#include "led8.h"
#include "avr_arduino.h"
#include "assembly.h"

// port D register output initialization for led row
void led8_init()
{
  PORTD = port_setup_full(PORTD,HIGH);
  DDRD = port_setup_full(DDRD,OUT);
}

// three 8-bit led blinks, acting as a visual start signal
void led8_start()
{
  PORTD = port_setup_full(PORTD,LOW);
  _delay_ms(BLK_SEC);
  PORTD = port_setup_full(PORTD,HIGH);
  _delay_ms(BLK_SEC);
  PORTD = port_setup_full(PORTD,LOW);
  _delay_ms(BLK_SEC);
  PORTD = port_setup_full(PORTD,HIGH);
```

```
  _delay_ms(BLK_SEC);
  PORTD = port_setup_full(PORTD,LOW);
  _delay_ms(BLK_SEC);
  PORTD = port_setup_full(PORTD,HIGH);
  _delay_ms(BLK_SEC);
}

// led output blinks of the two operands
void led8_operands()
{
  PORTD = ~(op1);
  _delay_ms(TWO_SEC);
  PORTD = ~(op2);
  _delay_ms(TWO_SEC);
  PORTD = port_setup_full(PORTD,HIGH);
  _delay_ms(ONE_SEC);
}

// execution of some math operations (witten in assembly language)
// with led output blink and some waiting time for human reading
void led8_math()
{
  asm_add();
  PORTD = ~(res);
  _delay_ms(TWO_SEC);

  asm_sub();
  PORTD = ~(res);
  _delay_ms(TWO_SEC);

  asm_mul();
  PORTD = ~(res);
  _delay_ms(TWO_SEC);
}

// execution of some logical operations (written in assembly language)
// with led output blink and some waiting time for human reading
void led8_logic()
{
  asm_and();
  PORTD = ~(res);
  _delay_ms(TWO_SEC);

  asm_or();
  PORTD = ~(res);
  _delay_ms(TWO_SEC);
```

```
  asm_xor();
  PORTD = ~(res);
  _delay_ms(TWO_SEC);
}

// execution of some shift operations (written in assembly language)
// with led output blink and some waiting time for human reading
void led8_shift()
{
  // calculating an example result to shift around
  asm_add();

  asm_lsl();
  PORTD = ~(res);
  _delay_ms(ONE_SEC);

  asm_lsl();
  PORTD = ~(res);
  _delay_ms(ONE_SEC);

  asm_lsr();
  PORTD = ~(res);
  _delay_ms(ONE_SEC);

  asm_lsr();
  PORTD = ~(res);
  _delay_ms(ONE_SEC);
}
```

4.9.1 Inclusions and definitions

```
#include "led8.h"
#include "avr_arduino.h"
#include "assembly.h"
```

The `led8.c` program module starts with the inclusion of three header files:

- `led8.h` for its local function definitions we saw earlier in section 4.8
- `avr_arduino.h` to be able to make use of its port and pin manipulation functions, namely `port_setup()` and `port_setup_full()`
- `assembly.h` to make direct use of some ATmega 328P internal registers and their name aliases, namely `op1`, `op2`, `res`

4.9.1.1 `led8_init()` function

```
void led8_init()
{
  PORTD = port_setup_full(PORTD,HIGH);
  DDRD = port_setup_full(DDRD,OUT);
}
```

4.9.1.1.1 Behaviour This function initialises the `PORTD` communication port, where the 8 led cathode pins are connected.

First, it sets all 8 port pins to logical level `HIGH`, it then enables the output direction again for all 8 port pins. Reversing the execution of those two instructions produces the same result.

4.9.1.1.2 LEDs with negative logic The 8 leds stay off during the initialisation because the 8 bit led row used in this example experiment works in negative logic; in fact, each of the leds is attached to the Arduino/ATmega 328P's `PORTD` pins by the diode's cathode terminal.

It must be remembered that the electrical current inside a LED always flows in by the anode (+) and then flows out of the cathode (-), it is not possible to make it in reverse; diodes are specifically designed to be powered in one direction only[37].

4.9.1.2 `led8_start()` function

```
void led8_start()
{
  PORTD = port_setup_full(PORTD,LOW);
  _delay_ms(BLK_SEC);
  PORTD = port_setup_full(PORTD,HIGH);
  _delay_ms(BLK_SEC);
  ... // repeated three times
}
```

4.9.1.2.1 Behaviour This function turns on and off all of the 8 leds at once for three times; it does that without a `for` cycle, not wanting to allocate another variable and more operative code needed for such a simple task. In addition, the repeated code gives a clearer idea of what actually happens to the `PORTD` pins.

4.9.1.3 `led8_operands()` function

```
void led8_operands()
{
  PORTD = ~(op1);
  _delay_ms(TWO_SEC);
  PORTD = ~(op2);
  _delay_ms(TWO_SEC);
  PORTD = port_setup_full(PORTD,HIGH);
  _delay_ms(ONE_SEC);
}
```

4.9.1.3.1 Behaviour This operands-dedicated function introduces a shortcut in port bit activation. To visualise the two operands, it shows them one after the other every two seconds and then turns off all leds for just one second, before returning the execution to the caller.

4.9.1.3.2 A NOT assignment The single operands are shown on the 8 led row simply by assigning their `BYTE` value to `PORTD`. Before assigning though, the tilde (∼) operator executes a bitwise NOT (one's complement) to all of the 8 bits of the `BYTE` data, because the 8 bit led row used in our example works in negative logic, as already stated in section 4.9.1.1.2.

4.9.1.4 `led8_math()` function

```
void led8_math()
{
  asm_add();
  PORTD = ~(res);
  _delay_ms(TWO_SEC);

  asm_sub();
  PORTD = ~(res);
  _delay_ms(TWO_SEC);

  asm_mul();
  PORTD = ~(res);
  _delay_ms(TWO_SEC);
}
```

4.9.1.4.1 Behaviour The function simply calls every math functionality that has been written in assembly language (as seen in section 4.7) and visualises their result on the 8 bit led row by assigning the `res` value to `PORTD` (as explained earlier in 4.9.1.3.2). It then waits two seconds before continuing with the next instruction.

4.9.1.5 `led8_logic()` function

```
void led8_logic()
{
  asm_and();
  PORTD = ~(res);
  _delay_ms(TWO_SEC);

  asm_or();
  PORTD = ~(res);
  _delay_ms(TWO_SEC);

  asm_xor();
  PORTD = ~(res);
  _delay_ms(TWO_SEC);
}
```

4.9.1.5.1 Behaviour The function is a direct mirror of the `led8_math()` function, described in the previous section 4.9.1.4 but, instead of math functions, it simply calls every logical functionality that has been written in assembly language (as seen in section 4.7). It then visualises their result on the 8 bit led row and waits two seconds before moving to the next instruction.

4.9.1.6 `led8_shift()` function

```
void led8_shift()
{
  // calculating an example result to shift around
  asm_add();

  asm_lsl();
  PORTD = ~(res);
  _delay_ms(ONE_SEC);

  asm_lsl();
  PORTD = ~(res);
  _delay_ms(ONE_SEC);

  asm_lsr();
  PORTD = ~(res);
  _delay_ms(ONE_SEC);

  asm_lsr();
  PORTD = ~(res);
  _delay_ms(ONE_SEC);
}
```

4.9.1.6.1 Behaviour The shift function simply calculates a sample result, to be placed in the variable/register recognisable as `res`, and then shifts it two times to the left (towards the *most* significant bit) and two other times on the right (towards the *least* significant bit).

4.9.1.6.2 Functional details The sample result is given by the execution of the `asm_add()` function, that updates the value of `res` variable/register.

For each operation, the result is shown on the 8 bit led row and there is a waiting time of just one second, to make the four shifts happen as a sort of continuous animation.

4.10 assembly.c

```
#include "assembly.h"
#include "led8.h"

void main(void)
{
  // operands initialization (example values)
  // 14 = 1110 and 7 = 0111
  op1=14,op2=7,res=0;

  // 8 led row initialization
  led8_init();

  // endless execution loop of math and logical operations
  // each of them is shown with led blink outputs
  while(1)
  {
    led8_start();
    led8_operands();
    led8_math();
    led8_logic();
    led8_shift();
  }
}
```

The `assembly.c` program file is the main file of this little project, in which the `main()` function of our firmware is found.

4.10.1 Inclusions and definitions

```
#include "assembly.h"
#include "led8.h"
```

The `led8.c` program module includes just a couple of header files:

- `assembly.h` to be able to configure the preferred ATmega 328P internal registers `op1`, `op2` and `res`
- `led8.h` to make use of the `led8` function family

`avr_arduino.h` is not directly included here, because its functionalities are not needed inside this C module.

4.10.1.1 `main()` function

```
void main(void)
{
  op1=14,op2=7,res=0;
  led8_init();

  while(1)
  {
    led8_start();
    led8_operands();
    led8_math();
    led8_logic();
    led8_shift();
  }
}
```

4.10.1.1.1 Behaviour This `main()` function doesn't have a return value nor any input parameters, thus they are ignored if somewhat provided.

It assigns exemplary numerical values to the variables/registers, specifically chosen to create some variety of bit configurations when recalculated by the math and logic operations already examined:

- `op1 = 14 = 00001110`
- `op1 =  7 = 00000111`

Right after that, the `main()` function initialises the dedicated port communication pins to be used in output mode, but every `avr_arduino` library function usage is hidden behind the `led8_init()` function call.

4.10.1.1.2 Endless loop Lastly, we have an endless loop cycle composed by a `while(1)` construct; the "repeat condition" inside the brackets is always *true*, because the constant number 1 always stays different from zero, which would be *false*[38].

Inside the loop, we have those functions which actually calculate the result and output it by activating the 8 bit led row, called in this specific order:

1. `led8_start()`
2. `led8_operands()`
3. `led8_math()`
4. `led8_logic()`
5. `led8_shift()`

This sequence of five function calls is then repeated an unlimited number of times, because of the endless loop construct.

This is not something of concern, because it is typical of firmware programs to have a cyclic program structure, which enables them to keep doing their own target actions, such as waiting for input, sensor monitoring, etc.

Chapter 5

Scoreboard

The second laboratory experiment we are going to discuss about is aimed to focus on a more complex device and on its dedicated way to communicate to it.

When tinkering with leds, as we have seen in the previous experiment, it is just about turning on and off some virtual switches, in the form of digital output pin levels, HIGH and LOW.

In this experiment, we have to properly communicate with a device that comes with an ordered and predefined set of leds installed and not directly available to us. In fact, we have to pass through a pre-built logical interface to indirectly "pilot" the leds' activation, and so we are bound to know in detail how to "talk" to this device.

We now give a proper introduction to these new project elements, before moving on with the actual experiment.

5.1 The SPI Interface

The acronym SPI stands for Serial Peripheral Interface and, as the name suggests, it allows a fixed and easy way of communicating with peripheral devices.

In most implementations, input and output data flows are in parallel and a device (such as the Arduino) can send and receive serial information (one bit after the other) at the same time, giving it the form of a full-duplex data channel[39].

5.1.1 Master and Slave

Communication between two devices is realised by a master-slave model. No standard protocol is used to negotiate these roles, it all depends on the specific behaviour of their own interfaces.

Different pins are usually employed to transfer data: the MOSI pin (Master Out Slave In) and the MISO pin (Master In Slave Out). Having two different pins allows a bidirectional data transfer if it is clear which device acts as master and which is dependent by it[40].

Our case is much more simpler to handle; we are going to design a single master SPI interface (implemented by the Arduino) and treat the target device as a slave, by using the MOSI pin only and fully respecting the device's use specifications.

5.2 The 1088AS 8x8 LED Matrix

5.2.1 The single component

This device is formed by an 8x8 led matrix with 16 pins. It is most often assembled together with a MAX7219 8-digit led display driver, which offers a serial interface to the outside world.

Originally designed for 7-segments number led displays, the MAX7219 decoder-driver can properly remap and pilot a 8x8 led matrix. According to its own datasheet, the MAX7219 is not fully compatible with the SPI[41], so we are going to emulate its "data sending algorithm" ourselves in the firmware code.

5.2.2 How it works

5.2.2.1 Connector pins

The 8x8 led matrix's pins are a total of five:

- Vcc (5 V is the recommended input voltage)
- GND
- DIN (Data Input pin)
- CS (Chip Selector pin)
- CLK (Clock Pulse pin)

5.2.2.2 Data transfer procedure

To send data from the Arduino to the matrix device, of course the device itself needs to be powered up. Then, a very specific series of operations is necessary for the device to properly work[41, 42]:

1. the CS pin is brought to logical level LOW
2. the DIN pin is updated with the logical level of interest (HIGH or LOW)
3. the CLK pin is brought to logical level HIGH (positive edge trigger), to have the bit acquired by the decoder-driver
4. the CLK pin is brought to logical level LOW
5. the CS pin is brought to logical level HIGH, applying the modifications set by the received information by adjusting led on/off states

5.2.2.3 Data packet format

The step sequence of phases 2,3,4 has to be executed 16 times, because of the data packet format accepted by the MAX7219 decoder-driver, described below[41]:

D11	**D10**	**D9**	**D8**	**D7**	**D6**	**D5**	**D4**	**D3**	**D2**	**D1**	**D0**
address				data							

The four MSBs [**D15...D12**] are sent first and then immediately ignored, but must always be sent either way, because the packet format cannot be changed.

Then, the other four MSBs [**D11...D8**] are sent and represent the matrix's target column address we want to send our 8-bit data word into, to (de)activate its leds in the proper way.

Finally the remaining 8 LSBs [**D7...D0**] contain the activation data for the selected column's leds, with each led being set by the state of the single bit that is found in the same position.

5.2.2.4 Big Endianness

Every 16-bits data packet has to be prepared in Big Endian mode[41]. This means that the necessary information have to be sent in order and from the most significant bit (MSB) to the least one (LSB).

As seen above in 5.2.2.3, all of the *ignored*, *address* and *data* "subpackets" have to be sent in this described order and in Big Endian format.

5.2.2.5 Positive edge trigger and shifting

At every positive clock edge, not just the selected one, but every internal data register is shifted one bit forward, to make room for the newly acquired bit.

If set in a specific configuration, each led column preserves its internal state even after poweroff and is able to restore it when turning the matrix on again.

5.2.3 A LED matrix multiplied by 4

The actual component that needs to be used in this experiment is the 4 matrixes version of the single component that has just been described.

Its way of functioning remains the same, with just one difference: each 16-bit word overflows into the flollowing matrix one bit at a time for each clock pulse (positive edge triggered).

This can be regarded as the device's *data overflow mechanism*, permitted by the *Daisy Chain* configuration of the led matrixes, which are placed one after the other and connected by their respective SPI pins as if they were forming a chain[43].

5.2.3.1 Led and bit mapping

When preparing data to be sent to the led matrix array, it must be noted that, by keeping the connector pins on the right and according to the MAX7219 datasheet[41]:

- the numbering of led columns (or lines, given this new perspective) goes from top to bottom, with the indexes 1 to 8 (not from 0 to 7 !)
- single leds are numbered from 0 to 7 moving from right to left

To sum up, the bit and line numbering starts from the upper right corner of the four matrixes block, if keeping the connector pins on your right.

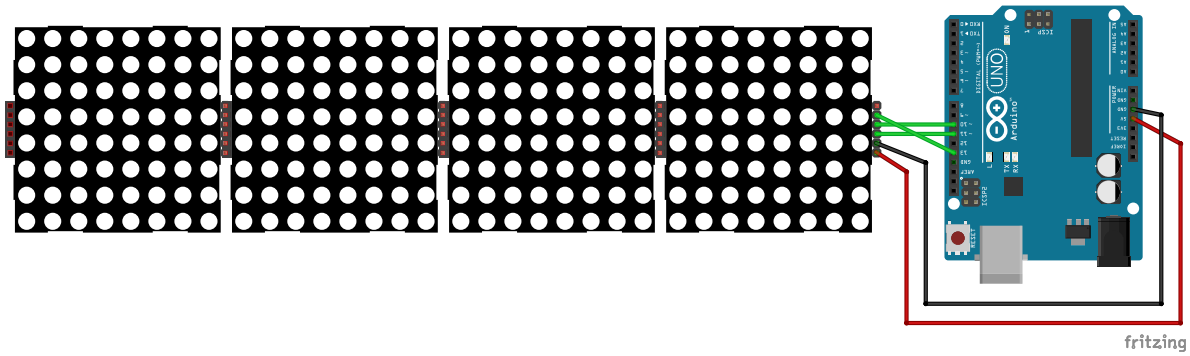
5.3 What we are going to do

For the sake of simplicity, instead of realising an actual scoreboard to be used, for example, in a basketball game, we concentrate on how this scoreboard actually works and what we can do with it.

So, in this project we are going to:

1. alternatively turn on and off the four matrixes by using their test mode (explained later)
2. write the universally famous "LOVE" word, letter by letter on each of the four matrixes
3. fill the four matrixes with randomly generated on/off configurations of leds which change every half a second

5.4 Schematic



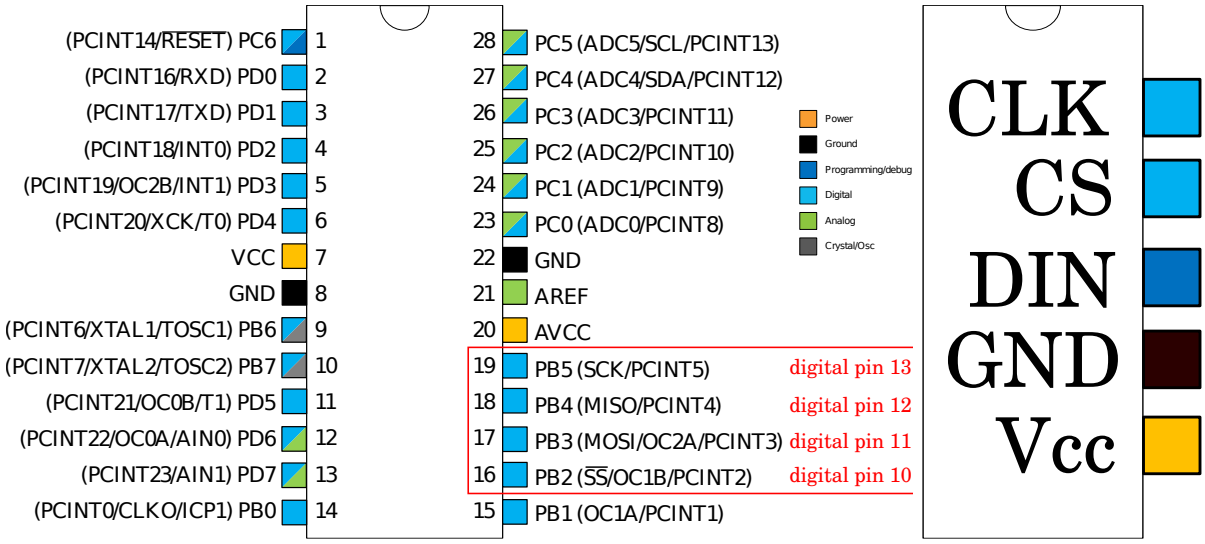
5.4.1 Arduino SPI pins

Even though the SPI is going to be just emulated, connecting to the dedicated interface pins on the Arduino is still a good choice, to get to know better a widespread communication interface, that is found on many other devices too.

The figure that can be found on the next page shows:

- which digital pins on the Arduino (the ones of the ATmega 328P) are dedicated to the SPI pins, described in section 5.1.1
- the actual order in which the 4 led matrixes component are found and connected to the Arduino

In the previous section's schematic there is an extra pin, which has to be ignored, because it is not present on the actual hardware component.



5.5 What we need

A few components are necessary to build up what we described, most of the logic is implemented via software. More specifically, we need:

- an Arduino Uno compatible board
- a USB printer
- a 4 led matrixes component, equipped with a SPI interface made by the five pins described in 5.2.2.1
- the source project files, consisting of:
  - the avr_arduino library source files, header (`avr_arduino.h`) and implementation (`avr_arduino.c`)
  - the SPI emulator header (`spi_master.h`) and implementation (`spi_master.c`)
  - the led matrix utilities, header (`ledmatrix.h`) and implementation (`ledmatrix.c`)
  - the main program module (`scoreboard.c`)

Here follows the detailed explaination for each source file in this order.

5.6 spi_master.h

```
#ifndef SPI_MASTER
#define SPI_MASTER

#include "avr_arduino.h"

void spi_init(void);
void spi_send(BYTE address,BYTE data);

#endif
```

5.6.1 Inclusions and definitions

After the usual "include guard" preprocessor directives[25], the `avr_arduino.h` header file is included first, to make use of the bit manipulation functions as well as the most generic and custom-defined `BYTE` data type.

5.6.2 Function declarations

`void spi_init(void);`
As its name suggests, it initialises the port pins used by the Arduino to connect to the 4 led matrixes component. A necessary operation, that needs to be executed only once.

`void spi_send(BYTE address,BYTE data);`
It implements the actual SPI data sending algorithm, carefully respecting those specifications already discussed in section 5.2.2.2. It is called each and every time an 8-bit data word needs to be written to the 4 matrixes' device in one of the eight available columns.

5.7 spi_master.c

```
#include "spi_master.h"

void spi_init(void)
{
  // PORTB bits direction setup for use as SPI interface
  DDRB = port_setup_full(DDRB,IN);  // everything in
  DDRB = port_setup(DDRB,DDB5,OUT); // SCK out
  DDRB = port_setup(DDRB,DDB3,OUT); // MOSI out
  DDRB = port_setup(DDRB,DDB2,OUT); // ~SS out
}

void spi_send(BYTE address,BYTE data)
{
  BYTE word,bit,high_low;

  PORTB = port_setup(PORTB,PB2,LOW);        // ~SS = ~CS = active low

  // we now send the two address and data words one after the other
  // as a 16-bit data packet the MAX7219 can approach correctly
  for(word=0;word<2;word++)
  {
    for(bit=0;bit<8;bit++)  // 1 byte = 8 bits
    {
      // led matrixes packet data format is Big Endian
      if(!word) // address word is sent first
      {
        high_low = address & 0b10000000;    // bitwise AND
        address <<= 1;                      // left shift
      }
```

```
      else      // data word is sent second
      {
        high_low = data & 0b10000000;       // bitwise AND
        data <<= 1;                         // left shift
      }

      // send the single bit, not a full word if not zero
      if(high_low)
        PORTB = port_setup(PORTB,PB3,HIGH); // MOSI data output 1
      else
        PORTB = port_setup(PORTB,PB3,LOW);  // MOSI data output 0

      // manual SPI clock signal handling
      PORTB = port_setup(PORTB,PB5,HIGH);   // SCK clock input pin up
      PORTB = port_setup(PORTB,PB5,LOW);    // SCK clock input pin down
    }
  }
  PORTB = port_setup(PORTB,PB2,HIGH);       // ~SS = ~CS = inactive high
}
```

5.7.1 Inclusions and definitions

The `spi_master.c` program module only includes its respective header file `spi_master.h`, which already contains every necessary resource to handle. It then follows in defining the body of the two functions previously declared.

5.7.1.1 `spi_init()` function

```
void spi_init(void)
{
  // PORTB bits direction setup for use as SPI interface
  DDRB = port_setup_full(DDRB,IN);  // everything in
  DDRB = port_setup(DDRB,DDB5,OUT); // SCK out
  DDRB = port_setup(DDRB,DDB3,OUT); // MOSI out
  DDRB = port_setup(DDRB,DDB2,OUT); // ~SS out
}
```

5.7.1.1.1 Behaviour The function makes use of the bit manipulation utilities proper of the avr_arduino custom library. `PORTB` has each of its pins set to input mode (as if closing them from the outside) and then only the pins 2, 3 and 5 are "opened" to be able to output our data of interest.

5.7.1.1.2 Chip Select's negative logic Every setting is applied through the Data Direction Register of `PORTB`. Moreover, the bit `PORTB2 = DDB2 = ~SS = ~CS` corresponds to the device's Chip Select pin, but it works in negative logic (remember that the $\sim$ operator is a bitwise NOT), as described in the MAX7219 datasheet[41].

5.7.1.2 `spi_send()` function

We are going to analyse its source code by separate sections, so as not to get too confused!

5.7.1.2.1 Variables and initialisation

```
void spi_send(BYTE address,BYTE data)
{
  BYTE word,bit,high_low;

  PORTB = port_setup(PORTB,PB2,LOW);        // ~SS = ~CS = active low
```

Three `BYTE` variables are declared and then used in the following sections, alogside with the `address` and `data` `BYTE` variables, both of which come as function parameters.

Communication with the slave component is then initialised, by setting the Chip Select drive bit to LOW, therefore activating the "receptive" mode of the 4 led matrixes input registers.

5.7.1.2.2 Loop cycles

```
  // we now send the two address and data words one after the other
  // as a 16-bit data packet the MAX7219 can approach correctly
  for(word=0;word<2;word++)
  {
    for(bit=0;bit<8;bit++)  // 1 byte = 8 bits
    {
```

The `word` and `bit` `BYTE` variables are employed to regulate two `for` cycles: the *outer* one cycles between the `address` and `data` 8 bit words and the *inner* one cycles between the 8 bits of which every word to send is made of.

5.7.1.2.3 Bit discrimination

```
      // led matrixes packet data format is Big Endian
      if(!word) // address word is sent first
      {
        high_low = address & 0b10000000;    // bitwise AND
        address <<= 1;                      // left shift
      }
      else      // data word is sent second
      {
        high_low = data & 0b10000000;       // bitwise AND
        data <<= 1;                         // left shift
      }
```

Depending on which of the two iterations the program is, it checks if the MSB of the `address` or `data` word is 0 or 1 through a bitwise AND operation and sets the `high_low` variable accordingly. Then, the other variable is shifted to the left by one bit, so as to examine the next MSB on the future iteration of the `for` cycle.

5.7.1.2.4 Address/Data bit output

```
      // send the single bit, not a full word if not zero
      if(high_low)
        PORTB = port_setup(PORTB,PB3,HIGH); // MOSI data output 1
      else
        PORTB = port_setup(PORTB,PB3,LOW);  // MOSI data output 0
```

After having calculated if the MSB was 0 or 1, the function makes use of the `port_setup()` library function, to send the LSB of `high_low` variable to the 4 led matrixes' device. The bit is transferred from the Arduino MOSI pin to the slave device's DIN pin.

5.7.1.2.5 Clock positive edge trigger

```
      // manual SPI clock signal handling
      PORTB = port_setup(PORTB,PB5,HIGH);   // SCK clock input pin up
      PORTB = port_setup(PORTB,PB5,LOW);    // SCK clock input pin down
    }
  }
```

A clock rising edge has to be provided to the 4 led matrixes' device, in order to have the installed MAX7219 decoder-driver to acquire the newly prepared bit of information, whether it belongs to the `address` or `data` words. Then, the clock's logical level has to be restored to its original state and the body of both `for` cycles terminate here.

5.7.1.2.6 Data transfer ending

```
  PORTB = port_setup(PORTB,PB2,HIGH);       // ~SS = ~CS = inactive high
}
```

The last instruction of the `spi_send()` function is to reset the Chip Select signal to an inactive state (HIGH, because of its negative logic, as specified in section 5.7.1.1.2), to end the data insertion process and to have the changes automatically applied on the led matrixes' panels.

5.8 ledmatrix.h

```
#ifndef LEDMATRIX
#define LEDMATRIX

#define REG_DECODE   0b00001001
#define REG_BRIGHT   0b00001010
#define REG_SCAN     0b00001011
#define REG_OPERATE  0b00001100
#define REG_TEST     0b00001111

#define LEDS_COL1 0x01
#define LEDS_COL2 0x02
#define LEDS_COL3 0x03
#define LEDS_COL4 0x04
```

```
#define LEDS_COL5 0x05
#define LEDS_COL6 0x06
#define LEDS_COL7 0x07
#define LEDS_COL8 0x08

#define ALL_LEDS 7
#define OFF 0
#define ON  1

#include "avr_arduino.h"

void ledmatrix_setup(BYTE brightness);
void ledmatrix_operate(BYTE reg_addr,BYTE on_off);
void ledmatrix_zero(BYTE matrix_num);
void ledmatrix_example(BYTE cycles,BYTE matrix_num);
void ledmatrix_random(void);

#endif
```

5.8.1 Inclusions and definitions

The file starts with the conditional block definition preprocessor directive and continues by defining a set of useful constants and function prototypes, aimed at building a simple interface to make a comfortable use of the multi led matrix component we are going to utilise.

5.8.1.1 REG constants

```
#define REG_DECODE   0b00001001
#define REG_BRIGHT   0b00001010
#define REG_SCAN     0b00001011
#define REG_OPERATE  0b00001100
#define REG_TEST     0b00001111
```

These constants are a simple literal representation of the binary numbers identified as the register addresses for the functionalities described below, all of which being part of the MAX7219 decoder-driver's capabilities. Their description and use have been extracted from the MAX7219 datasheet documentation[41]:

- `REG_DECODE`
  decoder register – enables or disables the BCD (Binary Code Decimal) decoding capability of MAX7219 – we are not going to use that.

- `REG_BRIGHT`
  brightness register – sets the brightness level of the matrixes' leds on a scale that goes from 1 to 15 (0 would mean leaving the leds off). For most uses, it is recommended not to go above level 5, so that it results bright enough without hurting the eyes too much.

- `REG_SCAN`
  scanner register – it defines how many led columns are allowed to be lit up, according to their positional index $[0 \dots 7]$ – we are setting it to its maximum value.

- `REG_OPERATE`
  operational register – also known as the shutdown register, according to the MAX7219 datasheet[41] – it turns on or off the set of led matrixes – we are of course going to use it in the firmware program.

- `REG_TEST`
  test mode register – when enabled, the led matrix ignores its current state register settings and stays on with every led at full brightness, until a "test off" command is issued – we use this only at the beginning of our demo program, see later.

5.8.1.2 LEDS constants

```
#define LEDS_COL1 0x01
#define LEDS_COL2 0x02
#define LEDS_COL3 0x03
#define LEDS_COL4 0x04
#define LEDS_COL5 0x05
#define LEDS_COL6 0x06
#define LEDS_COL7 0x07
#define LEDS_COL8 0x08
```

These constants are a simple literal representation of the hexadecimal numbers (for the sake of variety) identified as the register addresses of the eight led columns (or lines, if we look at the component horizontally) in which the eight bit data word is to be specified to (de)activate the single leds.

5.8.1.3 Generic constants

```
#define ALL_LEDS 7
#define OFF 0
#define ON  1
```

These constants have been defined for a better reading and comprehension of some portions of code, in which we have to enable single leds or send specific commands to the 4 led matrixes' device.

5.8.2 Function declarations

```
#include "avr_arduino.h"

void ledmatrix_setup(BYTE brightness);
void ledmatrix_operate(BYTE reg_addr,BYTE on_off);
void ledmatrix_zero(BYTE matrix_num);
void ledmatrix_example(BYTE cycles,BYTE matrix_num);
void ledmatrix_random(void);
```

After having included the `avr_arduino.h` header library file, the `ledmatrix.h` "interface" declares five utility functions:

- `void ledmatrix_setup(BYTE brightness);`
  dedicated to the initial setup of all the led matrix device's characteristics described in section 5.8.1.1. Brightness level is a mandatory parameter.

- `void ledmatrix_operate(BYTE reg_addr,BYTE on_off);`
  generic setup function that can accept any valid setup register belonging to the 4 led matrixes' device and set any value to it. This function is in fact an `spi_send()` wrapper, mainly used to send on/off and test instructions to the slave device.

- `void ledmatrix_zero(BYTE matrix_num);`
  the function is capable of turning off all leds of the specified number of matrixes passed as a parameter. If there are some more matrixes than specified, they might not be fully turned off, because of the register shifting mechanism described earlier in section 5.2.2.5.

- `void ledmatrix_example(BYTE cycles,BYTE matrix_num);`
  as its name suggests, this example function actually realises the on-screen writing of the single word "LOVE", by sending specific activation commands to all of the eight columns (or lines) of how many led matrixes are specified.

- `void ledmatrix_random(void);`
  it creates a random generation of column (line) number and led activation pattern to be repeatedly sent to the 4 matrixes device, so as to create random led activation sequences that can clearly show how the data overflow mechanism works.

5.9 ledmatrix.c

```
#include <stdlib.h>
#include "spi_master.h"
#include "ledmatrix.h"

void ledmatrix_setup(BYTE brightness)
{
  // SPI interface initialization
  spi_init();

  // decode mode = 0 (no BCD decoding)
  spi_send(REG_DECODE,OFF);

  // intensity (brightness) level = [0...15]
  spi_send(REG_BRIGHT,brightness);

  // how many columns will light up according to their index [0...7]
  spi_send(REG_SCAN,ALL_LEDS);
}
```

```
void ledmatrix_operate(BYTE reg_addr,BYTE on_off)
{
  /*
   * display test OR matrixes on/off function.
   * the TEST commands stack up onto each
   * other and overflow to the next matrixes.
   */
  spi_send(reg_addr,on_off);
}

void ledmatrix_zero(BYTE matrix_num)
{
  BYTE matrix,led_column;

  spi_send(REG_OPERATE,ON);

  // we will cycle for the minimum number of times to set everything
  // to zero, because of how the matrixes overflow data to each other
  for(matrix=1;matrix<=(8+matrix_num-1);matrix++)
    for(led_column=LEDS_COL1;led_column<=LEDS_COL8;led_column++)
      spi_send(led_column,0b00000000);

  spi_send(REG_OPERATE,OFF);
}

// activation of leds to form the word LOVE letter by letter
// it shows how data overflows through the defined number
// of matrixes, thus generalising the function
void ledmatrix_example(BYTE cycles,BYTE matrix_num)
{
  BYTE idx,matrix;
  for(idx=0;idx<cycles;idx++)
  {
    // L
    spi_send(LEDS_COL1,0xC0);
    spi_send(LEDS_COL2,0xC0);
    spi_send(LEDS_COL3,0xC0);
    spi_send(LEDS_COL4,0xC0);
    spi_send(LEDS_COL5,0xC0);
    spi_send(LEDS_COL6,0xC0);
    spi_send(LEDS_COL7,0xFF);
    for(matrix=0;matrix<matrix_num;matrix++)
      spi_send(LEDS_COL8,0xFF);

    _delay_ms(500);
```

```
    // O
    spi_send(LEDS_COL1,0x3C);
    spi_send(LEDS_COL2,0x7E);
    spi_send(LEDS_COL3,0xE7);
    spi_send(LEDS_COL4,0xC3);
    spi_send(LEDS_COL5,0xC3);
    spi_send(LEDS_COL6,0xE7);
    spi_send(LEDS_COL7,0x7E);
    for(matrix=0;matrix<matrix_num;matrix++)
      spi_send(LEDS_COL8,0x3C);

    _delay_ms(500);

    // V
    spi_send(LEDS_COL1,0b11000011);
    spi_send(LEDS_COL2,0b11000011);
    spi_send(LEDS_COL3,0b11000011);
    spi_send(LEDS_COL4,0b01100110);
    spi_send(LEDS_COL5,0b01100110);
    spi_send(LEDS_COL6,0b01111110);
    spi_send(LEDS_COL7,0b00011000);
    for(matrix=0;matrix<matrix_num;matrix++)
      spi_send(LEDS_COL8,0b00011000);

    _delay_ms(500);

    // E
    spi_send(LEDS_COL1,0xFF);
    spi_send(LEDS_COL2,0xFF);
    spi_send(LEDS_COL3,0xC0);
    spi_send(LEDS_COL4,0xFF);
    spi_send(LEDS_COL5,0xFF);
    spi_send(LEDS_COL6,0xC0);
    spi_send(LEDS_COL7,0xFF);
    for(matrix=0;matrix<matrix_num;matrix++)
      spi_send(LEDS_COL8,0xFF);

    _delay_ms(500);
  }
}


// this function is to be called last
// because of the endless loop within
void ledmatrix_random(void)
{
  int led_col,led_num;
```

```
  // random seed generation setting
  // 8072 is just a value of preference
  srand(8072);

  // endless loop
  while(1)
  {
    /*
     * random generation of:
     * led_col = which column to select
     * led_num = which leds to enable in that column
     *           depending on the binary number
     * +1 is to be sure we have at least one active bit
     * in our 8-bits word to send, module zero is useless
     */
    led_col = rand() % 8 + 1;
    led_num = rand() % 256 + 1;

    // if generated numbers are in range we send
    // the numeric information to the matrixes
    if(led_col <= 8 && led_num <= 256)
      // only the 8 least significant bits of int
      // will be considered and for us it's fine
      spi_send(led_col,led_num);

    _delay_ms(500);
  }
}
```

5.9.1 Inclusions and definitions

```
#include <stdlib.h>
#include "spi_master.h"
#include "ledmatrix.h"
```

There are only three inclusions inside this `ledmatrix.c` program module:

1. `stdlib.h` comes from the standard C library and it is necessary to make use of the `rand()` function, for pseudo-random number generation;
2. `spi_master.h` is needed to utilise the initialisation and sending functionalities;
3. `ledmatrix.h` defines the constants and function prototypes used and defined inside this very module;

The `ledmatrix.c` source file continues with its header's function definitions.

5.9.1.1 `ledmatrix_setup()` function

```
void ledmatrix_setup(BYTE brightness)
{
  // SPI interface initialization
  spi_init();

  // decode mode = 0 (no BCD decoding)
  spi_send(REG_DECODE,OFF);

  // intensity (brightness) level = [0...15]
  spi_send(REG_BRIGHT,brightness);

  // how many columns will light up according to their index [0...7]
  spi_send(REG_SCAN,ALL_LEDS);
}
```

5.9.1.1.1 Behaviour The function executes some calls among the SPI functionalities. It first enables the "emulated" SPI communication pins by calling their initialiser function and then defines the settings about BCD encoding, led brightness and led column (line) activation.

Everything is realised by calling the SPI functionalities and using the proper constants declared in `ledmatrix.h`, allowing a better understanding of the code.

5.9.1.2 `ledmatrix_operate()` function

```
void ledmatrix_operate(BYTE reg_addr,BYTE on_off)
{
  /*
   * display test OR matrixes on/off function.
   * the TEST commands stack up onto each
   * other and overflow to the next matrixes.
   */
  spi_send(reg_addr,on_off);
}
```

5.9.1.2.1 Behaviour As already described, this function has been created to send specific commands of *power* on/off and *test* on/off to the 4 led matrixes device.

Being in fact just a wrapper function of the `spi_send()` functionality, it is employable by any user of the `ledmatrix` interface to forward specific configuration instructions to the slave device.

5.9.1.2.2 Device-specific functional note The multi-line comment included inside the function states an interesting fact: the *test* command, if sent multiple times, overflows to the next matrix in the line. The *power* command, instead, does not overflow.

5.9.1.3 `ledmatrix_zero()` function

```
void ledmatrix_zero(BYTE matrix_num)
{
  BYTE matrix,led_column;

  spi_send(REG_OPERATE,ON);

  // we will cycle for the minimum number of times to set everything
  // to zero, because of how the matrixes overflow data to each other
  for(matrix=1;matrix<=(8+matrix_num-1);matrix++)
    for(led_column=LEDS_COL1;led_column<=LEDS_COL8;led_column++)
      spi_send(led_column,0b00000000);

  spi_send(REG_OPERATE,OFF);
}
```

5.9.1.3.1 Behaviour In the absence of a proper reset command, this function turns on the matrix set (to be sure they are so).

Then, for each of the matrixes indicated by the parameter, sends eight `BYTE` values equal to zero, one for each column (or line).

Lastly, it turns off the matrixes device, leaving it ready to be used again by another utility function.

5.9.1.3.2 The influence of data overflow The total matrixes iteration number specified in the iteration condition of the outer `for` cycle is altered, to be equal to the minimum number of iterations that take the data overflow mechanism into account, that has been specified previously in section 5.2.3.

5.9.1.4 `ledmatrix_example()` function

```
void ledmatrix_example(BYTE cycles,BYTE matrix_num)
{
  BYTE idx,matrix;
  for(idx=0;idx<cycles;idx++)
  {
   ...
    // V
    spi_send(LEDS_COL1,0b11000011);
    spi_send(LEDS_COL2,0b11000011);
    spi_send(LEDS_COL3,0b11000011);
    spi_send(LEDS_COL4,0b01100110);
    spi_send(LEDS_COL5,0b01100110);
    spi_send(LEDS_COL6,0b01111110);
    spi_send(LEDS_COL7,0b00011000);
    for(matrix=0;matrix<matrix_num;matrix++)
      spi_send(LEDS_COL8,0b00011000);
    ...
}}
```

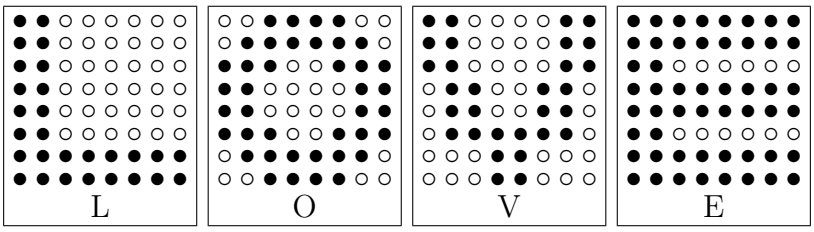
5.9.1.4.1 Behaviour The shortened version here depicted of the `ledmatrix_example()` function shows how every letter of the word "LOVE" is written to the matrixes, one after the other. Between each letter there is a waiting time of half a second, to have it readable by the user before changing it again.

The whole "LOVE" word is written, letter by letter, a number of `cycles` times, that is what the external `for` cycle is for. Only the "V" letter is written with binary number constants, the other three are written with hexadecimal number constants, just for the sake of variety and simplicity.

5.9.1.4.2 Letter data shifting Each line is assigned a specific `BYTE` value to visualise the letter and every `spi_send()` call shifts every line by eight bits, copying their contents to the next matrix, according to the data overflow mechanism described inside section 5.2.3.

This is why the last line is written `matrix_num` times instead of just one, because the last line has to fill in all matrixes, allowing the lines before to fill every matrix too.

5.9.1.4.3 The four letters on display Every letter is displayed equally on all of the matrixes. This is how each of the four letters looks like on a 8x8 led square:



5.9.1.5 `ledmatrix_random()` function

```
void ledmatrix_random(void)
{
  int led_col,led_num;

  // random seed generation setting
  // 8072 is just a value of preference
  srand(8072);

  // endless loop
  while(1)
  {
    led_col = rand() % 8 + 1;
    led_num = rand() % 256 + 1;

    if(led_col <= 8 && led_num <= 256)
      spi_send(led_col,led_num);

    _delay_ms(500);
  }
}
```

5.9.1.5.1 Behaviour This function declares two integer values that receive the pseudo-randomly generated value from the `rand()` function, properly calibrated by the `srand()` (randomizer seed) utility, of which the 8072 is a completely arbitrary value.

`led_col` and `led_num` are generated with a maximum value of 8 and 256 respectively, but sometimes this might not be true; that is why a simple value check is placed right afterwards.

Then, the generated value is sent to the generated led column (or line) of the 4 matrixes' device and the `while` loop restarts.

5.9.1.5.2 Alphanumeric note A direct correspondence between every letter and its `BYTE` values can be found when compiling the numeric binary codes and considering the MSB on the left and `LEDS_COL1` as the top line.

5.10 scoreboard.c

```
#include "ledmatrix.h"

void main(void)
{
  ledmatrix_setup(1);

  ledmatrix_zero(4);
  ledmatrix_operate(REG_OPERATE,ON);

  ledmatrix_example(4,4);

  ledmatrix_zero(4);
  ledmatrix_operate(REG_OPERATE,ON);

  ledmatrix_random();
}
```

5.10.1 Inclusions and definitions

5.10.1.1 `main()` function

After the `ledmatrix.h` header file inclusion, the `main()` function calls the `ledmatrix_setup()` utility, to begin using the 4 led matrixes' device connected to the Arduino.

A reset-to-zero operation (of four matrixes) is performed, to clear any previous or randomly generated activation state; right after that, all of the matrixes are turned on, ready to accept instructions.

The "LOVE" writing example is executed, realising 4 cycles with 4 matrixes to write onto; then another reset-to-zero and activation procedure takes place, leading to the execution of the matrixes' random-filling functionality.

`ledmatrix_random()` has to be called last, because the inherent loop cycle never exits nor terminates its execution.

Chapter 6

Sonar

The last laboratory experiment we face is a perfect excuse to make use of an LCD (Liquid Crystal Display) and a ultrasonic distance proximity sensor. Independently of how intriguing these components may appear, we have to think even more carefully to what we have and can do with them.

In order to execute the proposed exercise, it is (as always) fundamental to get to know these two components in detail, by understanding how they work, so as to be able to use them properly, a factor that especially this time must not be underestimated.

6.1 The 1602A v2.0 16x2 LCD display

6.1.1 How it works

This display comes in various layouts, black digits on a green background or white digits on a blue background, but for this experiment either of them are fine. They are usually sold without proper breadboard pins to attach our cables to, but those pins can be welded up to the component's board with proper tools.

The LCD is divided in two rows (or lines) of 16 characters, each of them composed by a small 5x8 dot matrix, the activation of which is handled by the screen's internal circuitry. Our very model comes with a SPLC780D controller attached, that defines the serial communication protocol to use and handles the data to display onto the screen, as its device datasheet describes[44].

6.1.1.1 Connector pins

If facing the LCD display correctly, by having the connector pins on the top side, it is possible to notice that they are a total of 16; starting from the leftmost they are the following:

1. **VSS**
   Display's main ground line, to be connected to GND.

2. **VDD**
   Display's main power source, its datasheet suggests to have it between 4.5 V and 5.5 V, so the 5 V Arduino plug is fine.

3. **V0**
   Differential voltage supply, connectible to a potentiometer (variable resistor), in order to adjust the leds' display contrast on the screen; in our implementation it has been connected to GND for maximum output contrast.

4. **RS**
   Register Select pin;
   RS = 0 : data is considered to be a protocol-specific command.
   RS = 1 : data is an ASCII-compatible[45] character code to display on screen.

5. **RW**
   Read (RW = 1) or Write (RW = 0) pin; we just need to write to the device, so it is going to be connected to GND.

6. **E**
   Enable pin;
   Information sent through the eight data lines is acquired, stored and displayed by the LCD device's controller only on the falling edge of signal E.

7. **D0 . . . D7**
   Data transmission lines. If it is a device's command or a character symbol, it can be distinguished by the RS signal's activation state.

8. **A**
   Anode input current for the backlight led illumination circuit. An external pull-up resistor (from 110 Ω impedence level onwards) and a 3.3 V or 5 V power source are required to properly turn the backlight on during operational time.

9. **K**
   Cathode output current for the backlight led illumination circuit; it must be always connected to GND.

6.2 The HC-SR04 ultrasonic distance sensor

6.2.1 How it works

The ultrasonic sensor is composed of two rounded speaker-like blocks, a transmitter T and a receiver R, as it is possible to see printed on the board itself. The circuitry installed on the sensor serves to redirect signals to and from the ultrasonic speakers.

The T block is capable of generating a short high frequency signal that, being reflected onto objects, is then received by the R block and lasts for a time period that is directly proportional to the distance the same pulse has just travelled.

6.2.1.1 Pulse format

More specifically, once the device is triggered by a HIGH logical level signal, which is sent on the Trigger pin and lasting at least 10 μs, the ultrasonic sensor emits a 8 square wave oscillation burst, at a frequency of about 40 kHz (a period of 25 μs)[46].

Right after that, the device raises its Echo signal and keeps it on HIGH until it senses the same ultrasonic pulse bouncing back from the object to which it was aimed at. That

specific amount of time has to be measured and taken into consideration to calculate the distance travelled by the ultrasonic pulse, as explained later.

6.2.1.2 Connector pins

This sonar sensor device presents just 4 pins, some of which have already been indirectly described in the prevoius paragraph:

- **Vcc**
  Power tension; 5V is the recommended input voltage for the device.
- **Trigger**
  A short logical level HIGH signal causes the emission of the 8 cycle 40 kHz pulses.
- **Echo**
  Signal pin that stays on logical level HIGH until the ultrasonic impulse is received back.
- **GND**
  The ground power terminal.

6.2.2 The speed of sound within the Arduino Uno

6.2.2.1 From meters to millimeters

Scientifically, the velocity in which sound waves can travel through air is strongly dependent on temperature. At 20°C the speed of sound is about 343 m/s (meters per second), but on the datasheet they say 340 m/s, to simplify and obtain an average value.

According to the HC-SR04 datasheet[46], minimum measurable distance is about 20 millimeters and maximum distance is around 4 meters. If we consider:

$$v_s = 343 \text{ m/s} = 343 \cdot 10^3 \text{ mm/s}$$
$$\Downarrow$$
$$1 \text{ mm in } \frac{1}{343 \cdot 10^3} \ s = 1 \text{ mm in } 3\mu s$$
$$\Downarrow$$
$$20 \text{ mm in } 3\mu s \cdot 20 = 60\mu s$$

It takes 60 μs to travel 20 mm through air in ordinary conditions. So our counter needs to be able to count in microseconds (μs), to be as precise as possible. It turns out we are already very well equipped for this task.

6.2.2.2 ATmega's counter recalibration

The ATmega 328P microprocessor mounted on the Arduino Uno we use is capable of running at a maximum clock frequency of 16 MHz[20], which is *16 times faster* than the 1 MHz clock speed we need. Moreover, each firmware we use that comes from this very report is always explicitly compiled for the ATmega 328P at a clock speed of 16 MHz. That is the definitive reason why *we need to adjust the coefficient* suggested by the ultrasonic sensor's datasheet.

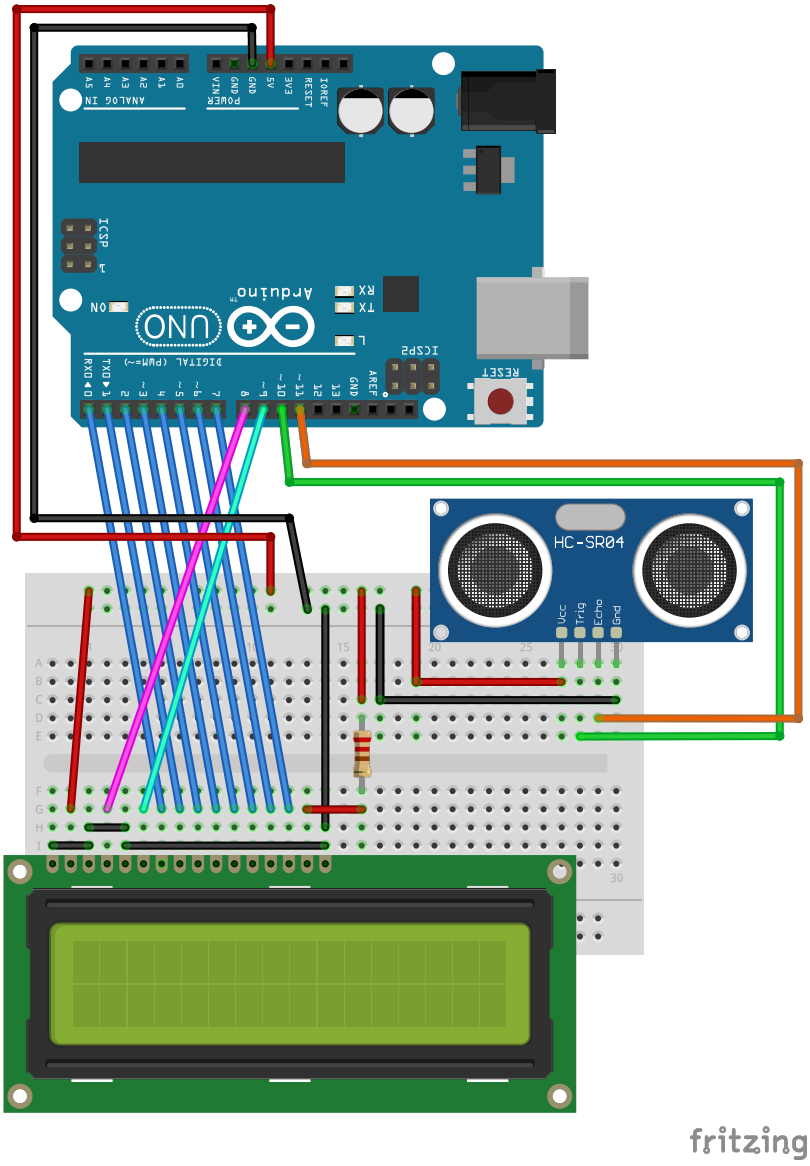
Instead of the 58 μs value, we can have it rounded to 60 μs and multiply that by 16. This gives us a final value of 960 μs, which will divide the actual value counted by the ATmega microprocessor. Practical explanation of this aspect will be given when examining the project's source code.

6.3 What we are going to do

After having talked about what is necessary to know and understand as a prerequisite, in this last experiment we are going to realise what may seem obvious to think about, but not as much as actually doing it.

In practical terms, we are simply going to write on the LCD display the millimetric distance measured by the ultrasonic sensor. Its value will be updated in real time, because the sensor's activation is going to be continuously triggered.

6.4 Schematic



6.4.1 Relevant notes

Each component is connected exactly as explained during their individual analysis. As already mentioned, the LCD's backlight input current is reduced by a 220 Ω resistor; the ultrasonic sensor's operative current is around 15 mA, so the 20 mA coming out of the digital and 5 V power pins is suitable, as already discussed in section 4.4.1.

6.5 What we need

Let us regroup and make a list of what we actually need to finally be able to build up this deeply intriguing experiment:

- an Arduino Uno compatible board
- a USB printer cable
- a middle-sized (or larger) breadboard
- at least 24 breadboard connecting cables
- an LCD 1602A 16x2 display, no matter the colour
- a 110 Ω (or above) resistor
- an HC-SR04 ultrasonic sensor
- the project source files, including:
  - the avr_arduino library source files, header (`avr_arduino.h`) and implementation (`avr_arduino.c`)
  - the LCD display dedicated header (`lcd.h`) and program module (`lcd.c`)
  - the ultrasonic sensor dedicated header (`sensor.h`) and program module (`sensor.c`)
  - the main program module (`sonar.c`)

6.6 lcd.h

```
#ifndef LCD_DISPLAY
#define LCD_DISPLAY

#include "avr_arduino.h"

// internal instruction specification flags
#define CMD 0
#define DAT 1

// commands taken from the SPLC780D controller datasheet
#define LCD_ON       0b00001100
#define LCD_OFF      0b00001000
```

```
#define LCD_FUNCTION 0b00111000
#define LCD_RESET    0b00110000
#define LCD_ENTRY    0b00000110
#define LCD_CLEAR    0b00000001

#define LCD_LINE1    0b10000000
#define LCD_LINE2    0b11000000

// public led display utility function declarations
void lcd_ports(void);
void lcd_setup(void);
void lcd_string(BYTE string[]);
void lcd_line(BYTE line_number);

#endif
```

6.6.1 Inclusions and definitions

The `avr_arduino.h` header file is included as a first thing, then two aliases of 0 (`CMD`) and 1 (`DAT`) are defined, because they have something to do with the ability to distinguish between command data and display data that is sent to the LCD, as we are about to see.

Those constants are followed by another series of binary constants, which represent the useful command data we need to send to our LCD in order to use it properly. As the code comment says, they have been directly extracted from the 1602A's controller datasheet, the SPLC780D circuit[44]. A few words on the less obvious:

- `LCD_FUNCTION` regulates the data transmission format, how many lines can be used among the ones available and the character font, of resolution 5x8 or 5x10 pixels (the latter would force the single-line mode because of space issues).
- `LCD_RESET` is a special command word to be issued only during the screen's initialisation procedure (explained later).
- `LCD_ENTRY` defines whether the display gets written from left to right or vice versa and if it shifts when data goes beyond the first 16 characters, because each line can hold up to 80 symbols to memory.
- `LCD_LINEx` places the writing cursor to the beginning of line number `x`.

6.6.2 Function declarations

Please note that in the header file have been listed only those functions that appear useful to other program modules. There are some other function definitions, which appear only inside the `lcd.c` module, that are going to be explained in the next section.

- `void lcd_ports(void);`
  Dedicated to apply the correct port bit direction settings on the Arduino.
- `void lcd_setup(void);`
  The proper LCD's initialisation procedure, to be executed right after power on.

- `void lcd_string(BYTE string[]);`
  Function capable of sending a string of characters `BYTE = unsigned char` to be written to the LCD.

- `void lcd_line(BYTE line_number);`
  Line-changer function, it also resets the cursor's position to the first symbol on the selected line.

6.7 lcd.c

```
#include "lcd.h"

// data and control bit port setup function
void lcd_ports(void)
{
  // eight PORTD data lines out
  DDRD = port_setup(DDRD,DDD7,OUT);
  DDRD = port_setup(DDRD,DDD6,OUT);
  DDRD = port_setup(DDRD,DDD5,OUT);
  DDRD = port_setup(DDRD,DDD4,OUT);
  DDRD = port_setup(DDRD,DDD3,OUT);
  DDRD = port_setup(DDRD,DDD2,OUT);
  DDRD = port_setup(DDRD,DDD1,OUT);
  DDRD = port_setup(DDRD,DDD0,OUT);

  DDRB = port_setup(DDRB,DDD1,OUT); // E = Enable
  DDRB = port_setup(DDRB,DDD0,OUT); // RS = Register Select
}

// data bits value reassignment procedure
void lcd_data(BYTE info)
{
  BYTE bit;

  // eight PORTD data lines reset
  bit = port_bit(info,PORTD7);
  PORTD = port_setup(PORTD,PORTD7,bit);
  bit = port_bit(info,PORTD6);
  PORTD = port_setup(PORTD,PORTD6,bit);
  bit = port_bit(info,PORTD5);
  PORTD = port_setup(PORTD,PORTD5,bit);
  bit = port_bit(info,PORTD4);
  PORTD = port_setup(PORTD,PORTD4,bit);
  bit = port_bit(info,PORTD3);
  PORTD = port_setup(PORTD,PORTD3,bit);
  bit = port_bit(info,PORTD2);
  PORTD = port_setup(PORTD,PORTD2,bit);
  bit = port_bit(info,PORTD1);
```

```
  PORTD = port_setup(PORTD,PORTD1,bit);
  bit = port_bit(info,PORTD0);
  PORTD = port_setup(PORTD,PORTD0,bit);
}


// data or command write instruction sequence
void lcd_write(BYTE info,BYTE is_data)
{
  // proper setup of signal RS
  PORTB = port_setup(PORTB,PORTB0,is_data);

  // reset of signal E
  PORTB = port_setup(PORTB,PORTB1,LOW);

  // info assignment to data output port D
  lcd_data(info);

  // falling edge of signal E
  PORTB = port_setup(PORTB,PORTB1,HIGH);
  _delay_us(1);
  PORTB = port_setup(PORTB,PORTB1,LOW);
  _delay_us(1);
}

// lcd display initial setup procedure
void lcd_setup(void)
{
  // power-up delay
  _delay_ms(100);

  // three reset command cycle
  // with dedicated delay times
  lcd_write(LCD_RESET,CMD);
  _delay_ms(10);
  lcd_write(LCD_RESET,CMD);
  _delay_us(200);
  lcd_write(LCD_RESET,CMD);
  _delay_us(200);

  // mode, lines and font setup
  lcd_write(LCD_FUNCTION,CMD);
  _delay_us(80);

  // display off command
  lcd_write(LCD_OFF,CMD);
  _delay_us(80);
```

```
  // display clear directive
  lcd_write(LCD_CLEAR,CMD);
  _delay_ms(4);

  // display shifting mode setup
  lcd_write(LCD_ENTRY,CMD);
  _delay_us(80);

  // display on command
  lcd_write(LCD_ON,CMD);
  _delay_us(80);
}

// writes a well-formed string on the lcd display
void lcd_string(BYTE string[])
{
  BYTE i = 0;
  for(i=0;string[i]!=0;i++)
  {
    lcd_write(string[i],DAT);
    _delay_us(80);
  }
}

// changes the line in which to write
// 1 = first line | 2 = second line
void lcd_line(BYTE line_number)
{
  if(line_number == 2)
    lcd_write(LCD_LINE2,CMD);
  else
    lcd_write(LCD_LINE1,CMD);

  _delay_us(80);
}
```

6.7.1 Inclusions and definitions

The `lcd.c` program module includes its respective header file `lcd.h` and defines all of the already mentioned functions, plus some other ones, available for internal use only.

6.7.1.1 The firmware programmer's extensive note

When developing the functions that follow, the LCD 1602A and SPLC780D controller datasheet[44] has been heavily used to learn about what to do and to translate it into code. During this process, a series of unexpected incompatibilities arose, even though the display was well known and handled inside more common Arduino projects.

The SPLC780D controller's main reference document has proved itself to be incomplete and lacking in detail. Furthermore, a thorough research on the Internet solved the issue,

by illustrating the proper instruction sequence and timing[47], which were just partially drafted on the main reference document.

For these reasons, all functions contained in the `lcd.c` program module do not make use of `for` or `while` cycles when changing the single port pins' status with the `port_setup()` functionality, while the `port_setup_full()` capability has not been used at all. Too much rapid and sudden pin value changes are of nothing but disturbance to the LCD's SPLC780D controller and that has been verified mainly through experimental trial.

6.7.1.2 `lcd_ports()` function

```
void lcd_ports(void)
{
  // eight PORTD data lines out
  DDRD = port_setup(DDRD,DDD7,OUT);
  DDRD = port_setup(DDRD,DDD6,OUT);
  DDRD = port_setup(DDRD,DDD5,OUT);
  DDRD = port_setup(DDRD,DDD4,OUT);
  DDRD = port_setup(DDRD,DDD3,OUT);
  DDRD = port_setup(DDRD,DDD2,OUT);
  DDRD = port_setup(DDRD,DDD1,OUT);
  DDRD = port_setup(DDRD,DDD0,OUT);

  DDRB = port_setup(DDRB,DDD1,OUT); // E = Enable
  DDRB = port_setup(DDRB,DDD0,OUT); // RS = Register Select
}
```

6.7.1.2.1 Behaviour This function employs the `port_setup()` functionality to sequentially activate all bits of `PORTD` and the first couple bits of `PORTB` in output mode. They are all done from the MSB to the LSB, but this order is not mandatory. Note the Enable and Register Select signals, while the Read/Write signal is not present, because it gets statically connected to GND.

6.7.1.3 `lcd_data()` function

```
void lcd_data(BYTE info)
{
  BYTE bit;

  // eight PORTD data lines reset
  bit = port_bit(info,PORTD7);
  PORTD = port_setup(PORTD,PORTD7,bit);
  bit = port_bit(info,PORTD6);
  PORTD = port_setup(PORTD,PORTD6,bit);
  bit = port_bit(info,PORTD5);
  PORTD = port_setup(PORTD,PORTD5,bit);
  bit = port_bit(info,PORTD4);
  PORTD = port_setup(PORTD,PORTD4,bit);
  bit = port_bit(info,PORTD3);
  PORTD = port_setup(PORTD,PORTD3,bit);
```

```
  bit = port_bit(info,PORTD2);
  PORTD = port_setup(PORTD,PORTD2,bit);
  bit = port_bit(info,PORTD1);
  PORTD = port_setup(PORTD,PORTD1,bit);
  bit = port_bit(info,PORTD0);
  PORTD = port_setup(PORTD,PORTD0,bit);
}
```

6.7.1.3.1 Behaviour Similarly to the previous one, this function makes use of the `port_setup()` feature to sequentially assign a new value to `PORTD` one bit at a time. The `BYTE` variable `bit` is declared to temporarily host the extracted bit value from the 8 bit word `info`.

6.7.1.3.2 Visibility This function is the first among those which are available for internal program module use only and separates the actual process of data assignment from the writing procedure that follows.

6.7.1.4 `lcd_write()` function

```
void lcd_write(BYTE info,BYTE is_data)
{
  // proper setup of signal RS
  PORTB = port_setup(PORTB,PORTB0,is_data);

  // reset of signal E
  PORTB = port_setup(PORTB,PORTB1,LOW);

  // info assignment to data output port D
  lcd_data(info);

  // falling edge of signal E
  PORTB = port_setup(PORTB,PORTB1,HIGH);
  _delay_us(1);
  PORTB = port_setup(PORTB,PORTB1,LOW);
  _delay_us(1);
}
```

6.7.1.4.1 Behaviour In order to properly write information to the LCD, the function configures the RS signal depending on the type of information identified by the caller and the E signal, to be at logical level LOW.

Right after that, it calls the `lcd_data()` function shown earlier to setup the 8-bit bus data (represented by `PORTD`). Lastly, it generates a rising and a falling edge of the E signal, which is the actual latch command that allows the LCD's controller to acquire and store the 8-bit word of `PORTD` into memory.

There is a one microsecond delay between the rising and the falling edge of signal E, so the SPLC780D controller can receive the descending front with the right timing.

6.7.1.4.2 Visibility This is the second function for internal use only, because it is called only by those functions which need to write some information to the LCD and specify if it is all about *command* data or *symbol* data.

6.7.1.5 `lcd_setup()` function

The LCD setup procedure always needs to be called right after the PORTx direction setting, so as to properly initialise the LCD for data output.

The different steps in which it has been subdivided are described inside the SPLC780D controller datasheet[44] and have been transcripted into code following those indications. Everything stated in the programmer's extensive note (section 6.7.1.1) also applies here.

Inside this function, there are multiple calls to the `lcd_write()` procedure with the proper command to send and the `CMD` parameter, specifying to the LCD's controller that the incoming data has to be treated as a directive and not as a symbol to be displayed.

6.7.1.5.1 Power-up delay

```
void lcd_setup(void)
{
  // power-up delay
  _delay_ms(100);
  ...
```

Right at the start there is a delay construct that suspends program execution for about 100 milliseconds, the time needed to the LCD's controller to properly configure its internal logic.

6.7.1.5.2 Device reset procedure

```
  ...
  // three reset command cycle
  // with dedicated delay times
  lcd_write(LCD_RESET,CMD);
  _delay_ms(10);
  lcd_write(LCD_RESET,CMD);
  _delay_us(200);
  lcd_write(LCD_RESET,CMD);
  _delay_us(200);

  // mode, lines and font setup
  lcd_write(LCD_FUNCTION,CMD);
  _delay_us(80);
  ...
```

The execution continues by sending the `LCD_RESET` command to the LCD for three times, by waiting 10 milliseconds the first time and 200 microseconds the second and third time. Then, the `LCD_FUNCTION` instruction is sent to the LCD, setting up its main operation mode and this time a 80 microseconds delay is applied.

6.7.1.5.3 Final touches

```
  ...
  // display off command
  lcd_write(LCD_OFF,CMD);
  _delay_us(80);

  // display clear directive
  lcd_write(LCD_CLEAR,CMD);
  _delay_ms(4);

  // display shifting mode setup
  lcd_write(LCD_ENTRY,CMD);
  _delay_us(80);

  // display on command
  lcd_write(LCD_ON,CMD);
  _delay_us(80);
}
```

The function ends by turning off and on the LCD itself; in the middle of the power commands, the display is cleared from previous or incorrect data and the proper display shifting mode is selected.

In our very case, the display does not show its character cursor and does not shift when text overflows the sixteenth character on a line.

6.7.1.6 Note on `lcd_write()` calling

From now on, all calls to the function `lcd_write()` require a subsequent time delay of at least 40 microseconds, because of the inherent design of the SPLC780D controller[44] (with the exception of the `LCD_CLEAR` command, which takes more time for the LCD's controller logic to complete).

As a safe choice, the minimum waiting time has been doubled and needs to be applied each time the execution returns from the `lcd_write()` call.

6.7.1.7 `lcd_string()` function

```
void lcd_string(BYTE string[])
{
  BYTE i = 0;
  for(i=0;string[i]!=0;i++)
  {
    lcd_write(string[i],DAT);
    _delay_us(80);
  }
}
```

6.7.1.7.1 Behaviour The function accepts a well-formed string of (`BYTE`), so an `unsigned char` array that holds a null (`'\0'`) character at the end.

It cycles through the parameter string and sends one `BYTE` at a time to the LCD by calling the `lcd_write()` function, specifying that in this case we are sending display data (`DAT`) and not a controller's command (`CMD`).

6.7.1.8 `lcd_line()` function

```
void lcd_line(BYTE line_number)
{
  if(line_number == 2)
    lcd_write(LCD_LINE2,CMD);
  else
    lcd_write(LCD_LINE1,CMD);

  _delay_us(80);
}
```

6.7.1.8.1 Behaviour This function accepts a `BYTE` parameter, considered to be numerical, which is used to choose to which LCD line the writing cursor has to be reset, enabling the screen's controller to write something new, by starting from the first symbol on the selected line.

6.8 sensor.h

```
#ifndef SONIC_SENSOR
#define SONIC_SENSOR

#include <stdint.h>
#include <stdio.h>
#include "avr_arduino.h"

// volatile unsigned integer variables
// to host temporary numeric results
volatile uint16_t distance_mm,echo,pulse_time;

// function declarations
void sensor_ports(void);
void sensor_setup(void);
void sensor_trigger(void);

#endif
```

6.8.1 Inclusions and definitions

The sensor header file contains the usual definition directives and then includes three header files:

- `stdint.h` for the availability of the `uint16_t` data type, defined as `unsigned int`.
- `stdio.h` to make use of the `snprintf()` function (used inside the `main()` function, see later).
- `avr_arduino.h` our well-known custom library that offers the `port` functions' family.

Before the functions declaration section, three `volatile` variables are declared:

- `distance_mm` is going to host the final millimetric distance value recorded by the ultrasonic sensor.
- `echo` is representative of the Echo signal's current value and serves to understand in which measure phase we find ourselves in during the program's execution.
- `pulse_time` is indicative of the cycles that the ATmega's internal counter has recorded while waiting for the Echo signal to change its state (that is how time gets measured with sufficient precision, see later).

Lastly, we have another triple of function declaration directives:

- `void sensor_ports(void);`
  Enables up the proper Trigger and Echo bits on `PORTB`.
- `void sensor_setup(void);`
  Sets up the interrupt condition on the Echo bit of `PORTB` and enables all hardware interrupts.
- `void sensor_trigger(void);`
  Causes the Trigger bit of `PORTB` to go at logical level HIGH for at least 10 microseconds and to return in LOW state afterwards.

6.8.1.1 `volatile` variables

Those are variables that, independently by the scope in which they are collocated, can be subjected to an "unexpected" value change, caused by a program execution flow that is external to the one identified with the `main()` function and its secondary procedure calls[33].

The perfect example for this kind of situation is the *interrupt handler* function, which is never called by the regular program, but directly by the hardware microprocessor when the corresponding interrupt signal condition is met. We are going to make use of all this.

6.9 sensor.c

```
#include "sensor.h"

// echo and trigger PORTB pins direction setup function
void sensor_ports(void)
{
  // PORTB sensor dedicated bits
  DDRB = port_setup(DDRB,DDB3,IN);  // echo in
  DDRB = port_setup(DDRB,DDB2,OUT); // trigger out
}

// ultrasonic sensor bit interrupt setup function
void sensor_setup(void)
{
  // echo signal state indicator
  echo = 0;

  // disable all interrupts
  cli();
  // Pin Change Interrupt Control Register, where PCIE0 refers to PORTB
  PCICR = port_setup(PCICR,PCIE0,HIGH);
  // Pin Change MaSK of PORTB, that enables catching of bit-specific interrupts
  PCMSK0 = port_setup(PCMSK0,PCINT3,HIGH);
  // enable all interrupts
  sei();
}

// utility ultrasonic trigger signal emission function
void sensor_trigger(void)
{
  // trigger signal assignment
  PORTB = port_setup(PORTB,PORTB2,HIGH);
  _delay_us(20); // microseconds's waiting time (at least 10 is needed)
  PORTB = port_setup(PORTB,PORTB2,LOW);
}

// Interrupt Service Routine for this sensor
// operating on the PORTB interrupt vector
ISR(PCINT0_vect)
{
  // if echo signal is HIGH
  if(echo)
  {
    // TC1 counter deactivation
    TCCR1B = port_setup_full(TCCR1B,LOW);
    // pulse time value assignment
    pulse_time = TCNT1;
```

```
    // TC1 internal value reset
    TCNT1 = port_setup_full(TCNT1,LOW);
    // echo signal state indicator toggle
    echo = 0;
    // distance in mm from pulse time calculus
    distance_mm = pulse_time / 960;
  }
  // else if echo signal is LOW
  else
  {
    // TC1 counter activation
    // TCCR1B = Timer Counter 1 Control Register B
    // CS10 = clock source selector (no prescaling)
    TCCR1B = port_setup(TCCR1B,CS10,HIGH);
    // echo signal state indicator toggle
    echo = 1;
  }
}
```

6.9.1 Inclusions and definitions

The `sensor.c` program module starts with the inclusion of its own header file. It then proceeds in defining all of the header-declared functions plus the interrupt handler necessary to the counting process.

6.9.1.1 `sensor_ports()` function

```
void sensor_ports(void)
{
  // PORTB sensor dedicated bits
  DDRB = port_setup(DDRB,DDB3,IN);  // echo in
  DDRB = port_setup(DDRB,DDB2,OUT); // trigger out
}
```

6.9.1.1.1 Behaviour This function simply enables the Echo input and Trigger output signals on the Arduino/ATmega 328P's `PORTB`, by making use of the `port_setup()` function.

6.9.1.2 `sensor_setup()` function

```
void sensor_setup(void)
{
  // echo signal state indicator
  echo = 0;

  // disable all interrupts
  cli();
  // Pin Change Interrupt Control Register, where PCIE0 refers to PORTB
  PCICR = port_setup(PCICR,PCIE0,HIGH);
```

```
  // Pin Change MaSK of PORTB, that enables catching of bit-specific interrupts
  PCMSK0 = port_setup(PCMSK0,PCINT3,HIGH);
  // enable all interrupts
  sei();
}
```

6.9.1.2.1 Behaviour The `sensor_setup()` function is fully dedicated to establish an interrupt trigger onto the Echo bit of `PORTB` with the help of five instructions:

1. the `echo` reference variable gets initialised to zero, because the sensor has not been triggered yet; instead of catching the actual value and generate another nested interrupt, a local value reference is still suitable for this purpose.
2. every interrupt is disabled by the `cli()` (clear interrupt) instruction.
3. the Pin Change Interrupt Control Register (`PCICR` for short) value is altered by putting to logical level HIGH the bit in the `PCIE0` position, which guarantees the `PORTB`'s interrupt vector trigger.
4. the Pin Change Mask of `PORTB` (`PCMSK0`) value is altered by setting to logical level HIGH the bit in the `PCINT3` position, that corresponds to the fourth bit of `PORTB`, the one to which the Echo signal is mapped on.
5. every interrupt is re-enabled by the `sei()` (set interrupt) instruction.

In this way, an interrupt signal is generated each time the Echo signal changes its logical level, from LOW to HIGH or from HIGH to LOW.

6.9.1.3 `sensor_trigger()` function

```
void sensor_trigger(void)
{
  // trigger signal assignment
  PORTB = port_setup(PORTB,PORTB2,HIGH);
  _delay_us(20); // microseconds waiting time (at least 10 is needed)
  PORTB = port_setup(PORTB,PORTB2,LOW);
}
```

6.9.1.3.1 Behaviour The function is called each time the firmware code wants to trigger the ultrasonic sensor's activation. It makes use of the `port_setup()` function to shift the Trigger pin of `PORTB` to logical value HIGH and then resetting it again to LOW after 20 microseconds. By doing so, the function fully accomplishes the correct trigger procedure as described in the ultrasonic sensor's datasheet (where it is written that the trigger time should be at least 10 microseconds long)[46].

6.9.1.4 `ISR(PCINT0_vect)` interrupt handler

```
ISR(PCINT0_vect)
{
  // if echo signal is HIGH
  if(echo)
  {
    // TC1 counter deactivation
    TCCR1B = port_setup_full(TCCR1B,LOW);
    // pulse time value assignment
    pulse_time = TCNT1;
    // TC1 internal value reset
    TCNT1 = port_setup_full(TCNT1,LOW);
    // echo signal state indicator toggle
    echo = 0;
    // distance in mm from pulse time calculus
    distance_mm = pulse_time / 960;
  }
  // else if echo signal is LOW
  else
  {
    // TC1 counter activation
    // TCCR1B = Timer Counter 1 Control Register B
    // CS10 = clock source selector (no prescaling)
    TCCR1B = port_setup(TCCR1B,CS10,HIGH);
    // echo signal state indicator toggle
    echo = 1;
  }
}
```

`ISR(vector)` is in fact an AVR library C language *macro*, hiding a series of directives and a generic function pointer, all of which is not going to be discussed.

6.9.1.4.1 Behaviour This interrupt handler is called due to the interrupt setting realised inside the `sensor_setup()` function; each time the sensor's Echo signal changes its logical level state, this interrupt procedure distinguishes two cases of operation:

1. if the `echo` reference signal's variable is HIGH (or likewise, different from zero):
    - the ATmega microprocessor's internal counter is deactivated by assigning all bits of `TCCR1B` (Timer Counter 1 Control Register B) to zero
    - the `pulse_time` integer variable is assigned the value contained in `TCNT1` (Timer Counter 1) register so as to temporarily store it for processing
    - the value contained in `TCNT1` (Timer Counter 1) register is reset to zero, along with the `echo` variable's value
    - the millimetric distance is calculated by dividing the `pulse_time` value for the 960 coefficient, which has been thoroughly explained in section 6.2.2.2

2. if the `echo` reference signal's variable is LOW (or likewise, equal to zero):

   - the ATmega microprocessor's internal Timer Counter 1 is activated, by setting the `CS10` (Clock Select) bit of `TCCR1B` (Timer Counter 1 Control Register B) to logical level HIGH; this enables the background counting process by selecting the internal clock source with no prescaling applied (timer frequency is equal to clock frequency)
   - the `echo` variable's value is set to one, because the Echo signal has just arrived and is now expected to end "soon"

6.10 sonar.c

```
#include "lcd.h"
#include "sensor.h"

#define DIGITS 8

void main(void)
{
  // preallocation of the numeric distance's string equivalent
  BYTE distance_string[DIGITS];

  // LCD display initialisation
  lcd_ports();
  lcd_setup();
  lcd_string("- Sonic Sensor -");

  // ultrasonic sensor initialisation
  sensor_ports();
  sensor_setup();

  // endless execution loop
  while(1)
  {
    sensor_trigger();

    // LCD display second line selection
    lcd_line(2);

    // conversion and writing of the measured distance
    lcd_string("dist: ");
    snprintf(distance_string,DIGITS,"%d",distance_mm);
    lcd_string(distance_string);
    // trailing whitespace is to prevent character overlapping
    lcd_string(" mm      ");
  }
}
```

6.10.1 Inclusions and definitions

On the main program module, both `lcd.h` and `sensor.h` header files are included; it also appears the definition of the `DIGITS` constant, made simply to ease the repeated writing of how many bytes at maximum have to be allocated for the `distance_string` that will hold the string*ified* version of the `distance_mm` integer value.

6.10.1.1 `main()` function

The `main()` function's behaviour is plain and simple. It calls the `_ports` and `_setup` functions of both the LCD (first) and the ultrasonic sensor (second). It then enters the `while(1)` endless execution loop, where it cyclically triggers the ultrasonic sensor and fully rewrites the second line of the LCD.

The `snprintf()` function acquires the `distance_mm` integer value and converts it into a well-formed string, ready to be sent to the display by the `lcd_string()` functionality. The second line terminates with a series of seemengly useless spaces, which are instead very useful to overlap and delete those characters that might not be rewritten on the line in the case of a `distance_string` composed by less than four digits.

Chapter 7

Conclusions

The three practical experiments with Arduino we just envisioned gave us a pretty good insight on what is definitely possible doing with such a flexible development platform. Arduino has spread so much among passionate electronics tinkerers because it is relatively easy to use, but people often employ this board's capacities only to make what they are truly interested in, whereas this thesis tried to furnish proof on how the underlying hardware is designed and actually functions.

7.1 Arduino, but without its IDE

Anyone approaching the Arduino board for the first time is invited to program it by using its official Integrated Development Environment (IDE), made available by the platform's creators. This programming methodology allows a very comfortable use of the machine and detaches the user little enough from the hardware level so as to render their experience enjoyable, or in some cases even fun.

Our interests diverged from this very welcoming approach. With the Arduino IDE it is not possible to actually learn how the hardware components work, all you can do is just to use them. Our manual programming method and command line usage is not very popular, because it is slower, more difficult to understand and organise, but it nonetheless allowed us to build functioning projects from the very ground up.

7.2 The AVR Arduino library's purpose

Starting from the barebone AVR programming library, it has been possible to "touch" the machine and understand how it actually worked, removing the usual abstraction layers and allowing us to handle pure byte data and binary constants.

The `avr_arduino` library had been created before the experiments to ease code writing a bit in its form, to basically handle bitwise operations by confining them to a proper space where they could be understood better. In the end, this created a useful set of functions, which can always be expanded depending on the programmer's necessities.

7.3 The three experiments

What kind of experience have we gained from the three laboratory exercises demonstrated in this thesis?

The **Assembler** experiment brought us in direct contact with the Arduino machine and the ATmega 328P microprocessor installed on it and taught us how to realise detailed functionalities by acting directly on the internal ATmega's registers.

The **Scoreboard** experiment did not involve an actual scoreboard, but allowed us to get familiar with the SPI data communication method, to be used towards an hardware controller. A more complex approach than just turning on and off some leds, because of the necessity to respect a proper data transmission protocol.

The **Sonar** experiment, possibly the "coolest" of all three, put together the theorical and hardware notions of the first two projects and, before allowing us to accomplish the main objective of distance measuring, it forced us from the start to build a dedicated communication protocol for both the LCD and the ultrasonic sensor.

7.4 An open-sourced thesis

Everything that has been used or employed inside this thesis had an open source nature. Arduino is an open hardware platform and all of the other hardware components employed are freely purchasable on the global market for a fair price.

All of the software tools used to build up this thesis are open source too: the *Ubuntu* operating system, the *Linux* kernel, the `avr-libc` software distribution, the LaTeXenvironment (`tex-live`) and document editing software (`gummi`), plus the firmware program examples that have been consulted for inspiration and technical adjustments.

The thesis itself is free for publication, redistribution and modification, while citing the original author and the relevant sources is always much appreciated.

Bibliography

[1] ATmega328P microprocessor, *Device Overview*, Microchip, https://www.microchip.com/wwwproducts/en/ATmega328p

[2] Atmel Corporation, *Atmel*, Wikipedia, https://en.wikipedia.org/wiki/Atmel

[3] RISC architecture, *Reduced Instruction Set Computer*, Wikipedia, https://en.wikipedia.org/wiki/Reduced_instruction_set_computer

[4] Arduino Official Guide, *Getting Started with Arduino and Genuino UNO*, last revision 2017/01/12 by SM, arduino.cc, https://www.arduino.cc/en/Guide/ArduinoUno

[5] Assembly language definitions, *Assembly language*, Wikipedia, https://en.wikipedia.org/wiki/Assembly_language

[6] LED, *Light-emitting diode*, Wikipedia, https://en.wikipedia.org/wiki/Light-emitting_diode

[7] Interrupt signal, *Interrupt definition*, TechTerms, https://techterms.com/definition/interrupt

[8] C programming language, *C Programming/Why learn C?*, C programming Wikibook, https://en.wikibooks.org/wiki/C_Programming/Why_learn_C%3F

[9] Arduino Official Guide, *Installing Additional Arduino Libraries*, last revision 2017/02/07 by SM, arduino.cc, https://www.arduino.cc/en/Guide/Libraries

[10] AVR development environment, *AVR Libc Home Page*, Savannah software forge, https://www.nongnu.org/avr-libc/

[11] C standard library, *C Programming/Standard libraries*, C programming Wikibook, https://en.wikibooks.org/wiki/C_Programming/Standard_libraries

[12] AVR Libc Modules, AVR Libc User Manual, https://www.nongnu.org/avr-libc/user-manual/modules.html

[13] Ubuntu, *The leading operating system for PCs, IoT devices, servers and the cloud*, https://www.ubuntu.com/

[14] Debian, *The universal operating system*, https://www.debian.org/

[15] Terminal, *Terminal definition*, TechTerms, https://techterms.com/definition/terminal

[16] AVR toolchain installation, *Setting up AVR-GCC Toolchain on Linux and Mac OS X*, MaxEmbedded, http://maxembedded.com/2015/06/setting-up-avr-gcc-toolchain-on-linux-and-mac-os-x/

[17] AVR Toolchain Overview, AVR Libc User Manual, https://www.nongnu.org/avr-libc/user-manual/overview.html

[18] chmod User Manual, *chmod - change file mode bits*, man page, http://man7.org/linux/man-pages/man1/chmod.1.html

[19] avr-gcc User Manual (conforming to the original gcc manual), *gcc - GNU project C and C++ compiler*, man page, http://man7.org/linux/man-pages/man1/gcc.1.html

[20] ARDUINO UNO REV3, *Technical Specifications*, store.arduino.cc, https://store.arduino.cc/arduino-uno-rev3

[21] avrdude User Manual, *2. Command Line Options*, Savannah software forge, https://www.nongnu.org/avrdude/user-manual/avrdude_3.html#Command-Line-Options

[22] Arduino Official Library Reference, *Serial.begin() function*, arduino.cc, https://www.arduino.cc/en/serial/begin

[23] ELF binary, *Executable and Linkable Format*, Wikipedia, https://en.wikipedia.org/wiki/Executable_and_Linkable_Format

[24] C preprocessor, *C Programming/Preprocessor directives and macros*, C programming Wikibook, https://en.wikibooks.org/wiki/C_Programming/Preprocessor_directives_and_macros

[25] Multiple inclusion of C header files, *Include guard*, Wikipedia, https://en.wikipedia.org/wiki/Include_guard

[26] C functions declaration and definition, *What is the difference between a definition and a declaration?*, user's question on StackOverflow, https://stackoverflow.com/questions/1410563/what-is-the-difference-between-a-definition-and-a-declaration

[27] ATmega 328P datasheet, Sparkfun, https://cdn.sparkfun.com/assets/c/a/8/e/4/Atmel-42735-8-bit-AVR-Microcontroller-ATmega328-328P_Datasheet.pdf

[28] Arduino Hacking, *ATmega168/328P-Arduino Pin Mapping*, arduino.cc, https://www.arduino.cc/en/Hacking/PinMapping168

[29] Maurizio Palesi, *The DLX Instruction Set Architecture Summary*, CiteSeerX scientific literature digital library, http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.710.734&rep=rep1&type=pdf

[30] Fritzing, open source and community-driven circuit board designing software, http://fritzing.org/home/

[31] Red Led, *LED - Basic Red 5mm*, item description, Sparkfun, https://www.sparkfun.com/products/9590

[32] AVR mixed programming example, *Combining C and assembly source files*, AVR Libc User Manual, https://www.nongnu.org/avr-libc/user-manual/group__asmdemo.html

[33] C language variables, *C Programming/Variables*, C programming Wikibook, https://en.wikibooks.org/wiki/C_Programming/Variables

[34] ATmega 328P assembler instruction set, Microchip, http://ww1.microchip.com/downloads/en/devicedoc/atmel-0856-avr-instruction-set-manual.pdf

[35] Assembly `.global` keyword, *"global main" in Assembly*, user's question on StackOverflow, https://stackoverflow.com/questions/17882936/global-main-in-assembly

[36] Stack program memory area, *Memory Layout of C Programs*, GeeksForGeeks, https://www.geeksforgeeks.org/memory-layout-of-c-program/

[37] Tutorial on LEDs, *Light-Emitting Diodes (LEDs)*, Sparkfun, https://learn.sparkfun.com/tutorials/light-emitting-diodes-leds/all

[38] `while` loop construct tutorial, *C Programming while and do...while Loop*, Programiz, https://www.programiz.com/c-programming/c-do-while-loops

[39] SPI full-duplex data transmission protocol, *Serial Peripheral Interface*, Wikipedia, https://en.wikipedia.org/wiki/Serial_Peripheral_Interface

[40] SPI Tutorial, Serial Peripheral Interface, Sparkfun, https://learn.sparkfun.com/tutorials/serial-peripheral-interface-spi/all

[41] MAX7219 8-LED digit display driver, Maxim Integrated, https://datasheets.maximintegrated.com/en/ds/MAX7219-MAX7221.pdf

[42] 1088AS 8x8 led matrix tutorial, *How to Use the MAX7219 to drive an 8x8 LED display Matrix on the Arduino*, Best-Microcontroller-Projects.com, https://www.best-microcontroller-projects.com/max7219.html

[43] SPI's Daisy Chain configuration, *Serial Peripheral Interface*, Wikipedia, https://en.wikipedia.org/wiki/Serial_Peripheral_Interface#Daisy_chain_configuration

[44] LCD display CFAH1602XYYHJP v2.0 datasheet, CrystalFontz, https://www.crystalfontz.com/products/document/964/CFAH1602XYYHJP_v2.0.pdf

[45] ASCII-compatible character encoding, *ASCII*, Wikipedia, https://en.wikipedia.org/wiki/ASCII

[46] HC-SR04 ultrasonic distance measurement sensor datasheet, ElecFreaks, https://www.mouser.com/ds/2/813/HCSR04-1022824.pdf

[47] LCD Programming Example using 'C', *Eight-bit interface using software time delays*, Donald Weiman, Alfred State College of Technology, State University of New York, http://web.alfredstate.edu/faculty/weimandn/programming/lcd/ATmega328/LCD_code_gcc_8d.html

Thanks

I would like to thank each and every one who supported me throughout this (very, very long) experience of studying Computer Engineering. My parents, without whom I would have had many more difficulties with my studies and with my life in general, for having been so supportive. My friends, who have always been encouraging about my accomplishment, to the point that became necessary for me to prove them they were right, that I could actually make it! To my thesis supervisor, professor Stefano Mattoccia, who allowed me to have something to work on that was a bit unusual between IT and Computer Engineers, but nonetheless stimulating. To everybody who understood the struggle I was passing through, by having to deal with all the difficulties that my university course brought to me, during these heavy years.

I would also like to dedicate my work to all those friends and classmates that did not make it through with their studies, who approached together with me the School of Engineering and Architecture, but ultimately left before having gained a title. They paradoxically inspired me to move forward, to have my chance of victory, not *on* them but for them, to make them feel that, after all, it was all definitely worth a try. It has not been a loss of time! I have finally become the professional figure I had always dreamt to be. I wanted the bigger picture. I think I have finally found it in how computers are designed, built and organised.

Riccardo Muggiasca
March 5th, 2019