Microcontrollers Hands-on Lab
STM32F401-PMDB16

Dino Ghilardi

February 12, 2018

Contents

1 Introduction

Here we introduce the scope of the book and the knowledge base required.

1.1 General informations

The book describes the hands-on labs on the ARM-Based Polimi32 (PMDB16 board), used in the "Microcontrollers" course at Politecnico di Milano.

The processor used is a STM32F401RE, mounted on a STNucleo board that is connected to a PMDB16 board. Hardware and software of this system is described through this book, where the step-by step setup and development with this system is described. The minimal setup to do the experiments consists of

- a PC or Mac (using Linux, Windows or macOSX as operating system), with enough disk space and memory to run java and the eclipse ide.
- PMDB16 board
- ST-Nucleo FE401RE board plugged in (Other ST-Nucleo boards can be used, but the peripheral initialization may differ).

In this book we will not focus on the provided libraries for the peripheral handling, but we will go into-deep at peripheral register level, in order to be able to write the required libraries starting from the documentation provided by the semiconductor manufacturer, as the device data-sheet, the user manual and the programming manual.

1.2 Knowledge base.

Suggested book: Microcontrollers hardware and firmware for 8-bit and 32-bit devices, Progetto Leonardo, 2017, chapters 1,2,5,6,7 (The official textbook for the "microcontrollers" course)

- Basic circuit theory
- Fundamental electronics course
- Basic Binary logic and arithmetic
- Microcontrollers basic knowledge
- Basic knowledge of the ARM Cortex M4 architecture

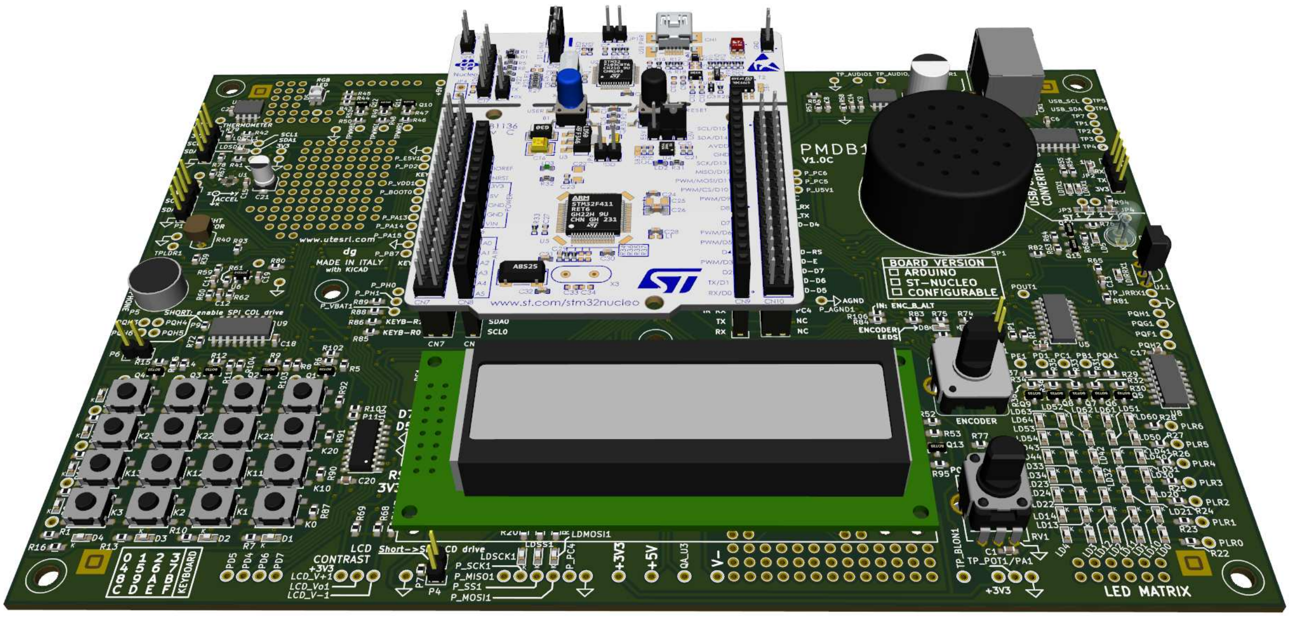


Figure 1.1: CAD Rendering of the PMDB16 board, with STNucleo board plugged in

- C programming language
- ARM assembly (not required but can be useful anyway).

1.3 Used Hardware

1.3.1 Your PC

It is... your PC, on wich the development system will be installed (see chapter 2).

1.3.2 STNucleo board

It is a development/demo board by STMicroelectronics that integrates the STM401RE microcontroller, a STLink compatible programmer/debugger, power supply circuits to get the board powered from the PC-USB connector. It connects most of the I/O of the microcontroller to some connector headers that are used to connect to the PMDB16 board. Some of the pins of the board are also connected to a connector compatible with the arduino uno shields.

See chapter 7 of the suggested textbook for a more detailed description of this board.

1.3.3 PMDB16 expansion board

The PMDB16 is a compact-size board that integrates the hardware to do various experiments in order to learn how to develop microcontroller based boards without the need to solder components or wire manually external connections. The arguments that can be covered with the board include initialization and usage of GPIOs, Analog inputs, PWM outputs, I2C, SPI, USART, timers, Interrupts etc...

Most of the signals of interest are routed to testpoints on the edge of the board, to ease measures during the experiments.

The board mechanical dimensions are a little less than A5 paper size, in order to lay down two boards on an A4 standard size) and ease transport of the board between classrooms.

The board schematics is described step by step in this book as long as it is needed for the experiments, and a full schematic is added in appendix 14.

The PMDB16 board has been developed using kiCAD, an open-source and free electronic CAD package available on www.kicad-pcb.org.

See chapter 7 of the suggested textbook for a more detailed description of this board.

2

Development tools setup

In this chapter the development tools used (eclipse IDE and plugins) will be set up..

2.1 Minimum system requirements

Operating system: Linux, Windows or macOS X

PC Hardware: Any modern PC able to run java (on which the Eclipse software is based) should do the trick, see the Eclipse website for details[2].

USB ports (1 free usb ports to connect the board. Better having two to be able to make experiments with the microcontroller USART and the on-board usb-to-usart converter)

JRE (Java runtime environment) installed on the used computer.

2.2 Software setup

2.2.1 Check Java installation and install JRE:

The development system requires a Java runtime environment (JRE) in order to run. Usually this environment is installed by default with most operating systems, but if it is not yet present it must be installed.

There are several sources for a JRE/JDK. Here are some of the more common/popular ones (listed alphabetically):

- IBM JDK [4]
- OpenJDK [5]
- Oracle JDK[6]

2.2.2 Method 1: download and install the package from openstm32

Hopefully this is the easiest method: The setup packages are available at the URL (a registration may be required):

`http://www.openstm32.org/Downloading%2Bthe%2BSystem%2BWorkbench%2Bfor%2BSTM32%2Binstalle` or visit the www.openstm32.org website and follow the links to the "System Workbench for STM32 - Bare Metal Edition", login (registration required) and download the required installer.

Note that for windows versions a driver for the ST-LINK debugger (integrated on the STNucleo board) may be needed, to be downloaded from the ST Microelectronics site www.st.com (registration required)

> https://my.st.com/content/my_st_com/en/products/embedded-software/development-tool-software/stsw-link009.license%3d1496761471507.html

or go to the www.st.com and search "stlink drivers" in the search field.

2.2.3 Method 2: Install eclipse and the extensions.

First of all it is required to download the IDE that we will use to develop anything: Eclipse, from the Eclipse site (http://www.eclipse.org/downloads/eclipse-packages/, choose Eclipse IDE for C/C++ Developers package). Eclipse is open source and free software, so it is legal to download and use it.

The eclipse project home page states:

Eclipse provides IDEs and platforms for nearly every language and architecture. We are famous for our Java IDE, C/C++, JavaScript and PHP IDEs built on extensible platforms for creating desktop, Web and cloud IDEs. These platforms deliver the most extensive collection of add-on tools available for software developers.

The code on this book have been done using Version: Neon.3 Release (4.6.3). The message from the Help->About box states:

Version: Neon.3 Release (4.6.3) Build id: 20170314-1500

```
Version: Neon.3 Release (4.6.3)
Build id: 20170314-1500
```

but any newer version of the IDE should work fine.

Note that this version requires java 8 (check that your JDK is up to that version)

The download package is a compressed file:

- .tar.gz for the Linux version
- .zip for the Windows version
- .tar.gz for the Mac osX (cocoa) version

2.2.3.1 Installing Eclipse on Linux

- Create a directory where you want to install it (it can be even a temporary directory created without root privileges). Personally I suggest to create a "microcontroller" directory wherever you want inside your home directory
- Move the downloaded file eclipse-cpp.... into that directory
- Now either
  - Open a terminal emulator
  - Enter the directory you just created with the command
    - ▷ *cd directoryPath*
  - de-compress the file with the command
    - ▷ *tar -vxf eclipse-cpp......tar.gz*
- or

    – de-compress the downloaded file into your destination directory using the graphical tool you prefer.

- Once the file has been decompressed all is done, no need for further operations.
- To execute eclipse, enter the eclipse subfolder and type ./eclipse
- To dis-install eclipse: just delete the eclipse directory you created.

If an udev entry is not automatically created, it will be needed to create it manually,

2.2.3.2 Installing Eclipse on Windows

An installer executable is available for download on the eclipse site..

2.2.3.3 Installing Eclipse on Mac OSx (cocoa)

Install the eclipse ide following the instructions from the eclipse web site.

2.2.3.4 Installing STM32 dev system inside eclipse

Launch eclipse (if it is the first run a destination directory location for the workspaces will be asked.

Select

```
Help→Install New software
```

In the "work with" field, add the OpenSTM32 repository selecting

```
ADD
```

In the new window, as "name" eter a descriptive name, like:
"System Workbench for STM32 - Bare Machine edition"
in the "location" field enter:
http://www.openstm32.org/Eclipse-updates/org.openstm32.system-workbench.site
UPDATE: Newer version for this tools (as 20170530) require the field to be set up as:
http://www.ac6-tools.com/Eclipse-updates/org.openstm32.system-workbench.update-site-v2/
Once added (and selected) this location, the **OpenSTM32** Tools item appears in the installable list of eclipse extensions.

Select

- Arm compiler for MCU
- External Tools
- OpenOCD
- OpenSTM32 Help Documentation
- OpenSTM32 IDE
- OpenSTM32 Remote Debugging Support

Then click "next" and follow instructions.

2.2.3.5 Errors after installing eclipse on Linux

If the correct java version is not installed, a message like the following will pop-up in a window while trying to start-up.

```
    JVM terminated. Exit code=1
    /usr/bin/java
    -Dosgi.requiredJavaVersion=1.8
    -XX:+UseG1GC
    -XX:+UseStringDeduplication
    -Dosgi.requiredJavaVersion=1.8
    ....
```

To solve the problem install java version 1.8 or later (jdk or jre).

In order to install oracle java 1.8 in debian 8, the following repositories should be added:

```
deb http://ppa.launchpad.net/webupd8team/java/ubuntu trusty main
deb-src http://ppa.launchpad.net/webupd8team/java/ubuntu trusty mainq
```

And use your favorite tool to install oracle-java8-installer from oracle.
Then install also oracle-java8-set-default to use it as default java version.
Note that newer versions should work also.
Refer to the documentation of the java version you are using if different from this.

If the udev rules to handle the usb debugger stlink are not installed on the system, they must be installed, downoading them from the site http://www.openstm32.org (search for "udev rules" in the site).

Probably it will be on:
http://www.openstm32.org/tiki-download_file.php?fileId=160&highlight=udev

Once downloaded, decompress and copy the file into the /etc/udev directory (or /etc/udev/rules.d, if that directory exists in your Linux distribution).

2.3 Checking the develpment tools

First we have to create a new project from the new tools installed. The procedure we will follow is:

- Open eclipse and select
  File->New->C Project
- Then, in the dialog select
  Executable->Ac6 STM32 MCU Project
- Select Ac6 STM32 MCU GCC as the toolchain (may be it is the only available)
- Select in "Project name" the name for this example (suggestion: "blink01", without quotes).
- If you do not want to use the default eclipse location for the project, un-check the "Use default location" box and browse to the directory you want to use.
- Click Next>
- Select both the available options, "Debug" and "Release" (Should be selected yet as it is the default).
- Click Next>
- Select the used CPU and Board:
- Series: STM32F4
- Board: NUCLEO-F401RE

- Click $\boxed{\text{Next>}}$
- As Project Firmware configuration select “Standard Peripheral Library”[1]
- A warning message “Target firmware has not been found locally, please install it”, if it is so, click on “Download target firmware”, accept the license and wait for the firmware to be downloaded.
  - If you downloaded the firmware, Click on “firmware installation” to get the current instructions on how to instal the firmware if the following do not work correctly. Warning: The firmware to be loaded is about 75Mb, so you should wait some time.
  - Once the firmware has ben downloaded there will be no warning anymore and an info message will appear instead as “Firmware ’STM32F4xx_DSP_StdPeriph_Lib_V1.8.0’ has been found”

Once the project has been created we will add our next code. Note that here we want to know how to use the controller at a very “low” level, so we will not use library functions, but just the defines in order to access the registers, because we want to go in-deep and understand how peripherals work and be able to write our own library, if we are in need for that (as, for instance, when an externally provided library does not fit the work we have to do).

To have a deep look into the library structure, there is a .chm help file in the directory where the file has been installed (in eclipse see Window->Preferences->System Workbench->Firmware installation to see where those files are).

Now compile the project with Project->build all (or the hammer icon).

In the console tab at the end of the messages there should be something like the following messages.

```
Generating binary and Printing size information:  arm-none-eabi-objcopy -O binary
"testNewProject.elf" "testNewProject.bin" arm-none-eabi-size "testNewProject.elf"
text data bss dec hex filename
1276 1076 1568 3920 f50 testNewProject.elf
14:28:33 Build Finished (took 211ms)
```

If the build succeedes as in the example the development system is working.

[1]The “Standard Peripheral library” is a collection of functions used to initialize peripherals at low level, without advanced functions as USB or TCP-IP.

The “Cube HAL” is a software library for HAL (Hardware Abstraction Layer) with low level drivers for USB, TCP/IP, File system, RTOS and Graphic. Those functions are not used on the PMDB16 board, so we will not need those libraries. See reference [1]

Bibliography

[1]

[2] www.eclipse.org

[3] deb http://ppa.launchpad.net/webupd8team/java/ubuntu trusty main deb-src http://ppa.launchpad.net/webupd8team/java/ubuntu trusty mainwww.eclipse.oOnce downloaded the .zip file, de-compress it in a directory and launch the eclipse executablerg, main website of thOnce downloaded the .zip file, de-compress it in a directory and launch the eclipse executablee eclipse IDE.

[4] http://www.ibm.com/developerworks/java/jdk/index.html

[5] http://openjdk.java.net/

[6] http://www.oracle.com/technetwork/java/javase/downloads/index.html

3

Hardware setup and first tests

In this chapter we will start the development environment and test the compiler an the debugger on a simple project (led blinking). We will also look at how to initialize and use GPIOs as outputs.

3.1 Hands-on Experiment 1: Creating a new project

The first step in every hardware setup is to use the less-complicated approach, with minimal hardware. In this case we will start without the PMI16 Demo board. In this section we want to check out that the nucleo board is working correctly and that the on-board programmer-debugger is detected and is working.

To achieve this, we will create a new project from scratch, describe some of the automatically generated files, add very few lines of code in order to program the microcontroller with the embedded programmer/debugger, set a breakpoint, look at the watch windows to read local and global variables, CPU Registers and I/O registers at the time of the breakpoint.

.

Create a new project (as shown in section 2.3 on page 14 ”Checking the develpment tools”)

Structure of the auto-generated code.

startup->startup_stm32f401xx.s. In this file there is the main initialization code, in the routine called Reset_Handler.

This routines prepares the c program to be executed: it copies from the flash memory to the RAM the initialization values of the static and global variables, fills with zeros the rest of the data space (it is not part of the “standard” for C compilers or operating systems but gives a better initialization, so that a C global variable that has not been initialized defaults to zero instead of being set at a random value).

After that it calls the SystemInit routine.

The SystemInit() routine (in file CMSIS->device->system_stm32f4xx.c) does the setup for the microcontroller system, initializing the embedded flash interface, the PLL and update the SystemFrequency variable.

then calls the library initialization __libc_init_array

then calls the main program, calling the high-level-language main() routine.

Note that in a typical micrococontroller program the main should *NEVER* return.

CMSIS->core->device->stm32f4xx.h This file contains the definition of various registers of the peripherals and a lot of #defines of constants that we will use to initialize peripherals.

StdPeriph_Driver->inc and StdPeriph_Driver->src Here there are the peripheral drivers routines that ease a lot the usage of the controller peripherals. Mostly we will not use those routines since our work is focused at a lower level, in order to learn to set the registers of the controller starting from the data-sheet. The description of the standard peripheral library functions is described in the document [6].

src->main.c The main(void) routine is in the src->main.c file, where it has been created with an infinite loop doing nothing.

Now Compile the project: the compilation should end with no errors and the "Console" tab should show a message ending with something like

```
text data bss  dec   hex filename
1276 1076 1568 3920  f50blink01.elf
00:33:31 Build Finished (took 4s.172ms)
```

The last lines of the message gives some statistics for the compilation and the memory usage for the different sections used by the linker.

Now, just for testing, add inside the main loop (the for(;;)) an "i++;" and declare i as int. The new main should now be:

CODE
```c
void main(void)
{
        int i=0;
        for(;;) //Just like a while (1)
        {
                i++;
        }
}
```

Compile again: there should be no errors too and the messages should be something like:.

```
Generating binary and Printing size information:...
arm-none-eabi-objcopy -O binary "blink01.elf" "blink01.bin"...
arm-none-eabi-size "blink01.elf"
text  data  bss   dec  hex filename
1288  1076  1568  3932 f5c blink01.elf
00:45:32 Build Finished (took 265ms)
```

Now put a breakpoint on the "i++;" line, so that every time that line is executed, the program is stopped letting us to see what's happening in the CPU.

- Connect the STNucleo board to the PC using the USB-A to MiniUSB-B cable.
- Click the debug button (or choose the Run->Debug menu).

- Chose Debug as...-> Ac6 STM32 C/C++ Application

Now the program is sent to the board using the gdb [5] debugger. If an error is issued, may be the correct driver for the st-link debugger is not installed. First try to close and re-open eclipse.

Once the debug is launched, the program is re-compiled, sent to the microcontroller memory and executed until it reaches the breakpoint. At the breakpoint the program is suspended and we can look at the variables. If the tab “Variables” is not present on the main screen, enable it with

- Windows->Show View->Variables.

In this case we just have “i” as variable and it is shown in the “Variables” tab. We can also see the registers content at every instruction, the I/O registers (the registers of the peripherals) selecting other available tabs[1].

It is also possible see, in the “disassembly” tab, the instructions executed in assembler, so it is visible how the C has been translated in assembly language. In the disassembly the c instruction precedes the corresponding assembly code.

Now restart the program using the Run->Resume menu (or the F8 shortcut or the relative button in the tool-bar)

At every break we will see the current value of the variable “i”, that increments at every cycle.

Adding our own code to the project skeleton The simplest way is to add lines to the main.c file (as we did before), starting from the main program, but it becomes less useful as long as the number of lines grows. After a thousand lines the program becomes less and less readable every line is added, so it is strongly suggested to group functions that have something in common in files different from “main”.

To add a file to the project, right-click on the project name, select “New-> Source File” or “New->Header file” to add to the project a .c or a .h file. Adding files to the project in this way adds them automatically to the list of files to be compiled and linked.

If we have a file that we want to re-use, it can be copied in the src directory and selecting “refresh” (or pressing the F5 key) enables the compilation of that file also.

3.1.1 WARNING on auto-generated files and PLL setup.

The files generated do not configure correctly the clock on the Nucleo board, so the timings won’t be as expected, since it does not use the internal oscillator as PLL input. To configure correctly, edit the generated file

    System_stm32F4xx.c

Around line 709 [2]

CODE
```
/* Configure the main PLL */
RCC->PLLCFGR = PLL_M | (PLL_N << 6) | (((PLL_P >> 1) -1) << 16) |
                            (RCC_PLLCFGR_PLLSRC_HSE) | (PLL_Q << 24);
```

This line sets as PLL input the HSE (external oscillator), but on the Nucleo board the external crystal is not connected, so the bit RCC_PLLCFGR_PLLSRC_HSE should not be set.

To have a correctly working system we need to change those lines to:

[1]Use the menu Windows->Show View to see which available watches are present

[2]Tip: search RCC->PLLCFGR in the code

CODE
```
/* Configure the main PLL */
RCC->PLLCFGR = PLL_M | (PLL_N << 6) | (((PLL_P >> 1) -1) << 16) |
                                (PLL_Q << 24);
```

Moreover, another register needs to be set correctly. In file System_stm32F4xx.c the PLL multiplier has to be set to the PLL input frequency, that is 16MHz, so, around line 372, we need to change the #define PLL_M 25 into

CODE
```
#define PLL_M 16
```

Once this things are changed, we will have a system core clock running at 84MHz, generated by the internal PLL fed by the internal RC oscillator.

Note that the oscillator is an RC oscillator and do not have the precision of an external crystal oscillator. The Nucleo boards have the space to mount an external crystal and its capacitors. See STNucleo board schematics.

3.2 Hands-on Experiment 2: STNucleo onboard LED blinking

In this experiment we will start using the main characteristic of a microcontroller: the integrated peripherals, starting from the GPIO: General Purpose Input/Output.

The GPIO used here are connected also to the PMDB16 board and there will be no harm in doing the experiments with the STNucleo plugged in. It also will work correctly using only the STNucleo board without the PMDB16.

The STM32F401RE (the microcontroller we have on the ST-Nucleo401RE board that we are using) has 5 GPIO ports: GPIOA, GPIOB, GPIOC, GPIOD, GPIOH (see chapter 8 of the refence manual [3] and table 8 of the data-sheet [4]).

Turning the LED ON: GPIO Initialization First of all we need to know which GPIO line il connected to the LED on the Nucleo board: we look at the STNucleo board schematic and figure out which pin of the microcontroller is connected to the led LD2 of the Nucleo Board, so we go at document [2] (or figure 3.4) , look at the schematics and see that LD2 is connected to R31, SB21, PA5 or PB13, depending on the presence of the jumpers SB42 (that is ON for the Nucleo F401RE) and SB29 (OFF on the Nucleo F401RE), so wi find that

- PA5 is the GPIO that we will use.

Peripheral initialization: We will need to set port A, pin 5 (PA5) as a push-pull output and set it to high state to turn the led ON. To be able to do this we need to look at the documentation of the controller, in particular we will look at:

- **STM32F401xD STM32F401xE Datasheet-product data**: 135 pages with electrical specifications, general information, device pin-out, internal structure and features/peripheral description, electrical connections and pin functions (including the I/O alternate function mapping), memory mapping. Most of what is needed for the hardware design of a board that uses this microcontroller is in the document.

- **PM0214 Programming manual (STM32F3, STM32F4 and STM32L4 Series Cortex ® -M4 programming manual)**: 260 pages describing the programming model, the processor, the memory model, exceptions , fault handling, Assembler instruction set and core peripherals.. The informations needed to program the core using Assembler language. In this document there is nothing about the "external" peripherals, but the internal peripherals as the SysTick timer and the NVIC (the interrupt controller) are described here.
- **RM0368 Reference manual (STM32F401xB/C and STM32F401xD/E advanced ARM ® - based 32-bit MCUs)[2]**: 841 pages describing the system architecture, internal bus, memory organization and memory map, embedded flash, power controller, clock configuration, DMA, Interrupts and all the peripherals including their structure and registers. Here we find most of the informations we need to initialize and use the peripherals.

To see which register we have to configure we need to check the Reference manual [2]. To initialize IO we need to set correctly:

- Peripheral clock
- GPIO Registers

Register definitions All the definitions are in the stm32F4xx.h file, that we need to include at the beginning of our main.c file. Using the definitions of this file we can set a register just like as it is a variable. The variable names are grouped in structures. i.e. the register RCC_AHB1ENR, that is part of the RCC subsystem is named

RCC->AHB1RSTR. Please take a moment to look at that file to become familiar to the naming convention.

Also the bits of the registers have their name defined, so that it is possible to set or clear a bit using bitwise AND and OR functions using the correct mask. About coding conventions in this file, you can see also paragraph 1.1 of [6], "1.1 Coding rules and conventions".

Peripheral Reset In chapter 6 of the reference manual [2] there is the description of the RCC (Reset and Clock Control) for the controller we are using.

RCC AHB1 peripheral reset register (RCC_AHB1RSTR). This register can reset the peripherals on the AHB1 bus. bit 0 is the GPIOARST bit, used to reset the port A. Usually there is no need to reset a GPIO port since it should work anyway, but for other peripherals the story changes, so it is much better to resed by default every peripheral we use

To reset (and restart) the GPIOA peripheral it we should do:

CODE
```
RCC->AHB1RSTR|=RCC_AHB1RSTR_GPIOARST;  //Reset the peripheral:
                                       //(set the bit 0 of AHB1RSTR)

RCC->AHB1RSTR&=~RCC_AHB1RSTR_GPIOARST; //Un-reset the peripheral:
                                       //(clear the bit 0 of AHB1RSTR)
```

The definition for the structure RCC and the defines of the constants RCC_AHB1RSTR_GPIOARST are in the auto-generated file CMSIS->core->device->stm32f4xx.h that has been added automatically to our project

Peripheral clock init/enable To understand how to enable the peripheral clock we should look at the “6.3.9 RCC AHB1 peripheral clock enable register (RCC_AHB1ENR)” section of the reference manual [2]

```
RCC->AHB1ENR |= RCC_AHB1ENR_GPIOAEN; // Set bit 0 of AHB1ENR register
                                     //(IO port A clock enabled)
```

Now that the clock is correctly set, we can configure the port A registers.

GPIO Registers initialization In section 8.4 of the reference manual there is the description of the GPIO registers, that whe have to set/check. A summary is reported in section 8.4, Table 24. (Port bit configuration table).

GPIO port mode register (GPIOx_MODER) (x = A..E and H) -section 8.4.1-. Bit 0 and 1 of the register GPIOA->MODER configure the direction of the GPIO pin A0, in particular:

bit 1-0	Port mode
00	Input (reset state)
01	General purpose output mode
10	Alternate function mode
11	Analog mode

For pin A1 we must set the bits 2-3, for pin A2 the bits 4-5 and so on.
To set the pin A5 as OUTPUT as we need we will set bits 10-11.

```
GPIOA->MODER=(GPIOA->MODER & (~GPIO_MODER_MODER5))|GPIO_MODER_MODER5_0;
                                     //Mode 01: general purpose output.
```

GPIO port output type register (GPIOx_OTYPER). -section8.4.2- The bits of the GPIOA->OTYPER sets the type of the corresponding output, having
0->Push pull output
1->Open drain output
so we should clear bit 5 of GPIOA->OTYPER. This is also the reset state for the register, so we do not need to do a

```
GPIOA->OTYPER & (~GPIO_OTYPER_OT_5); //clear bit 5 of GPIOA->OTYPER.
```

GPIO port output speed register (GPIOx_OSPEEDR). -section 8.4.3- We should look at the data-sheet to see the meaning of the various speeds available. (in particular at *Table 56. I/O AC characteristics* of the data-sheet [4] that reports the maximum frequency available at various supply voltages.)

```
GPIOA->OSPEEDR|=GPIO_OSPEEDER_OSPEEDR5; // 11=HIGH SPEED.
```

GPIO alternate function low register (GPIOx_AFRL) (x = A..E and H) and GPIO alternate function high register (GPIOx_AFRH) (x = A..E and H). Those two registers set the “alternate function” for the pin. We just need a GPIO for pin A5, so we can leave it at its reset value (0000). Anyway the register to set for an alternate function for this pin AFR[0], bits 20-23.

GPIO port pull-up/pull-down register (GPIOx_PUPDR) (8.4.4) Since we are using the pin as a push-pull output, we do not need pull-ups enabled and leave it at its default value (0 for A5).

```
GPIOA->PUPDR&=(~GPIO_PUPDR_PUPDR5); //Not needed, zero by default.
```

Setting the pin status To set the pin status on an ARM GPIO peripheral there are two ways:

- Set the corresponding bit in the GPIO port output data register (GPIOx_ODR)
- Set the corresponding "Set" bit in the GPIO port bit set/reset register (GPIOx_BSRR).

When writing in the GPIOA_ODR in bit 5, the output pin is set accordingly (to 0 if zero has been writte, or to 1 if an 1 has been written.

When writing **1** in the bit BS5 of the GPIOA_BSRR register, the output pin RA5 is set to 1.

When writing **1** in the bit BR5 of the GPIOA_BSRR register, the output pin RA5 is reset to 0.

Writing **0** in any bit of the GPIOA_BSRR does not take any effect.

Using the BSRR register it is possible to set a group of pins or reset a group of pins in a single write operation to the register and avoids possible problem with the "read-modify-write" approach.

Example: the read-modify-write problem

Suppose that two routines handle two bits of the ODR register of the same port.
One is an interrupt routine (routine B) and handles bit 1, the other is a "normal" routine (routine A) and handles bit 0 of the port. Consider this sequence of events:

1. Start with ODR=x00;
2. Routine A wants to set bit 0 and reads ODR in a variable
3. Routine A sets the bit 0 in the variable
4. *The interrupt event happens and routine B sets bit 2 in the ODR register and exits*
5. Routine A resumes its work, writing its variable (0x01) in ODR.

At the end the ODR content will be 0x01, NOT 0x03 as we need. In this case we have a bug that will show itself only when the interrupt (that is usually an asynchronous event) happens at a precise point of the code execution, making it a kind of "ghost bug" that appears randomly in our program.
Using the BSRR avoids this problem since it is an atomic operation that sets/resets bits directly without affecting other bits of the same register.

To set the pin status we can do either

```
GPIOA->ODR|=(1<<GPIO_ODR_ODR_5);
```

or

```
GPIOA->BSRRL=GPIO_BSRR_BS_5;
```

To reset the pin we can do either

```
GPIOA->ODR&=(~(1<<GPIO_ODR_ODR_5));
```

or

```
GPIOA->BSRRH=(GPIO_BSRR_BR_5>>16); //equivalent to "=(1<<5)"
```

So, inside the main loop, we add the two instructions

```
GPIOA->BSRRL=GPIO_BSRR_BS_5;
GPIOA->BSRRH=(GPIO_BSRR_BR_5>>16); //See definition of
                                   //GPIO_BSRR_BR_5 for the shift.
```

But having those two instructions leads us to a switching frequency that is far too high to be seen by the human eye, so we introduce a small delay function.

```
void smallAndUglyDelay ()
{   int i;
        for (i=0; i<100000; i++)
        {
                asm("nop"); // Assembler instruction "NOP".
        }
}
```

Then we change the main loop to:

```
for(;;)
{
        GPIOA->BSRRL=GPIO_BSRR_BS_5; //Turn LED ON
        smallAndUglyDelay();
        GPIOA->BSRRH=(GPIO_BSRR_BR_5>>16); //shift it right. (see .h file)
        smallAndUglyDelay();
}
```

Now we are going to compile and check everyghing is fine.

Create an enable a debug configuration (if it has not been done automatically)

Run->Debug configurations

Select AC6 STM Debugging

Click the “New launch configuration” icon (top left of the dialog)

Select the project to debug

Use the “debug” button to launch the new debug session, or, with a right-click on the project name in the project explorer, select “debug As->AC6 STM32 C/C++ Application”.

If there are no errors in code, it will be compiled and lauched until it reaches the beginning of the “main()” routine (a breakpoint is put automatically there)

To place or remove a breakpoint, double click on the line number on the code frame.

Try to put a couple of breakpoint at the pin set and reset instructions:

```
Breakpoint here-> GPIOA->BSRRL=GPIO_BSRR_BS_5;
Breakpoint here-> GPIOA->BSRRH=(GPIO_BSRR_BR_5>>16);
```

and execute using the run->Resume command (or the shortcut F8 or the corresponding icon) . At every run the led on the ST-Nucleo board should change its state (also the led P_SCK1 on the PMDB board, if the Nucleo board is inserted).

See experiment on page on page 71 to see an example for led flashing using interrupts.

3.3 Hands-on Experiment 3: –TODO–STNucleo on board key and led.

- *Look for the button connection on schematics*
- *GPIO configuration (input),*
- *turn led ON on keypress*
- turn led OFF on key relase.

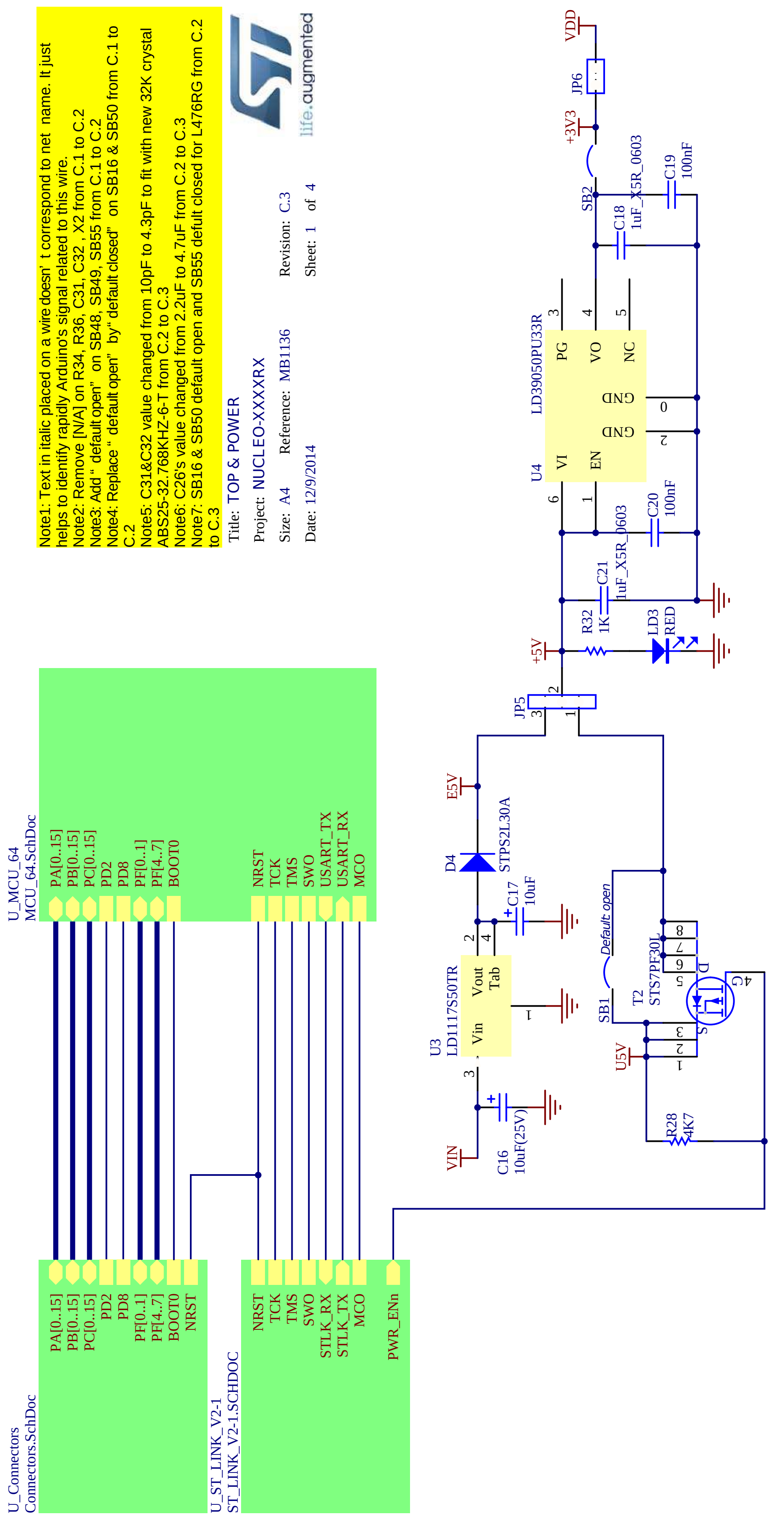
3.4 Connecting STNucleo and PMDB16

Once the STNucleo board is put on the PMDB16 main board, some tests can be done immediatly. First of all, check that turning the encoder (ENC1) turns on/off the encoder leds.

3.4.1 Hands-on Experiment 4: First test on PMDB16 board

Use hands-on experiment 2 and see if led LDSCK1 turns on while the on-board one turns off. (The line used for LD2 on the Nucleo Board is also the SCK line of the SPI)

If this does not happen, check electrical connections between the PMDB and Nucleo boards (verify that the connector has not been shifted.



Note1: Text in italic placed on a wire doesn' t correspond to net name. It just helps to identify rapidly Arduino's signal related to this wire.
Note2: Remove [N/A] on R34, R36, C31, C32 , X2 from C.1 to C.2
Note3: Add " default open" on SB48, SB49, SB55 from C.1 to C.2
Note4: Replace " default open" by" default closed" on SB16 & SB50 from C.1 to C.2
Note5: C31&C32 value changed from 10pF to 4.3pF to fit with new 32K crystal ABS25-32.768KHZ-6-T from C.2 to C.3
Note6: C26's value changed from 2.2uF to 4.7uF from C.2 to C.3
Note7: SB16 & SB50 default open and SB55 defult closed for L476RG from C.2 to C.3

Title: TOP & POWER
Project: NUCLEO-XXXXRX
Size: A4 Reference: MB1136 Revision: C.3
Date: 12/9/2014 Sheet: 1 of 4

Figure 3.1: Nucleo Board schematic, page 1

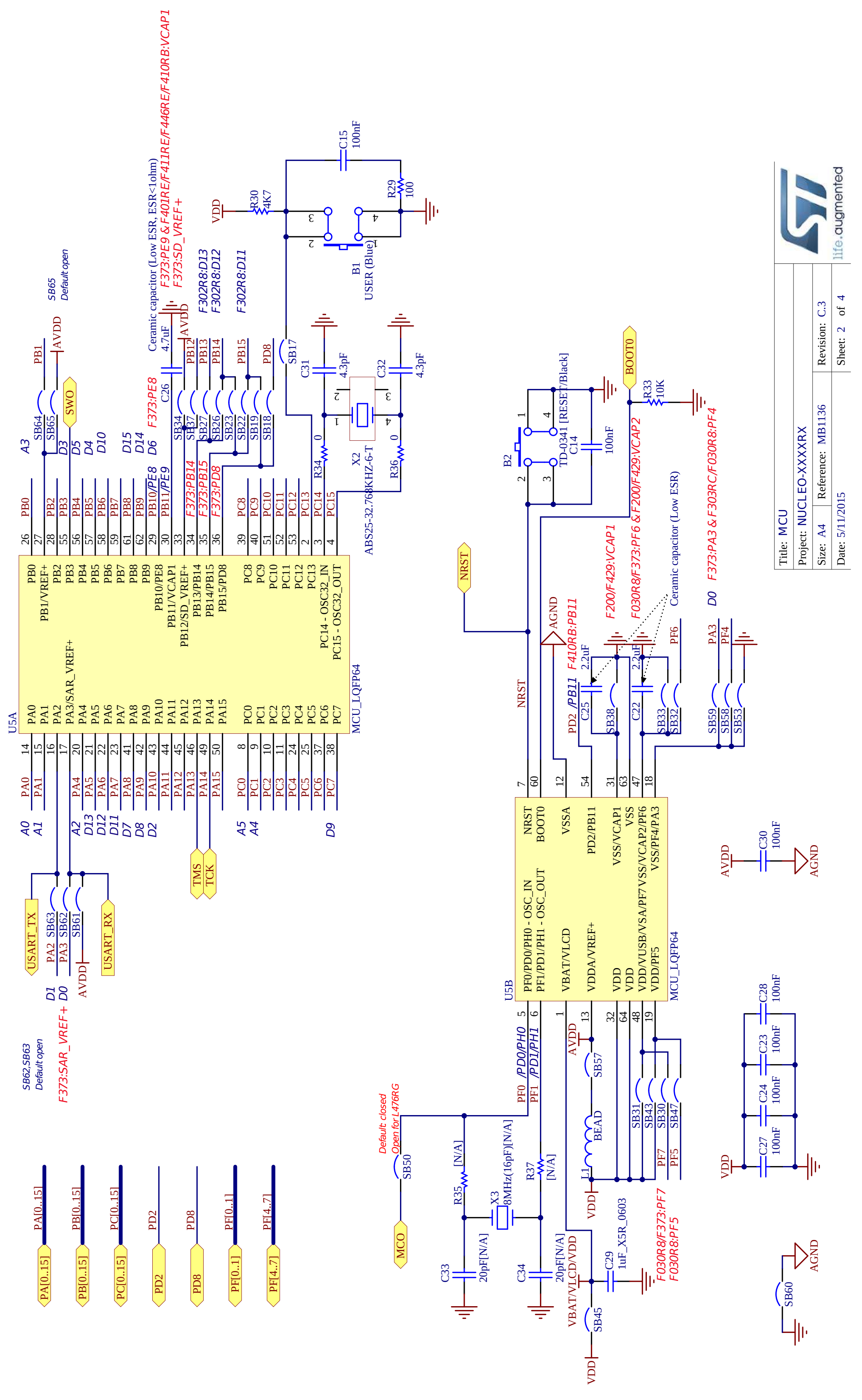


Figure 3.2: Nucleo Board schematic, page 2

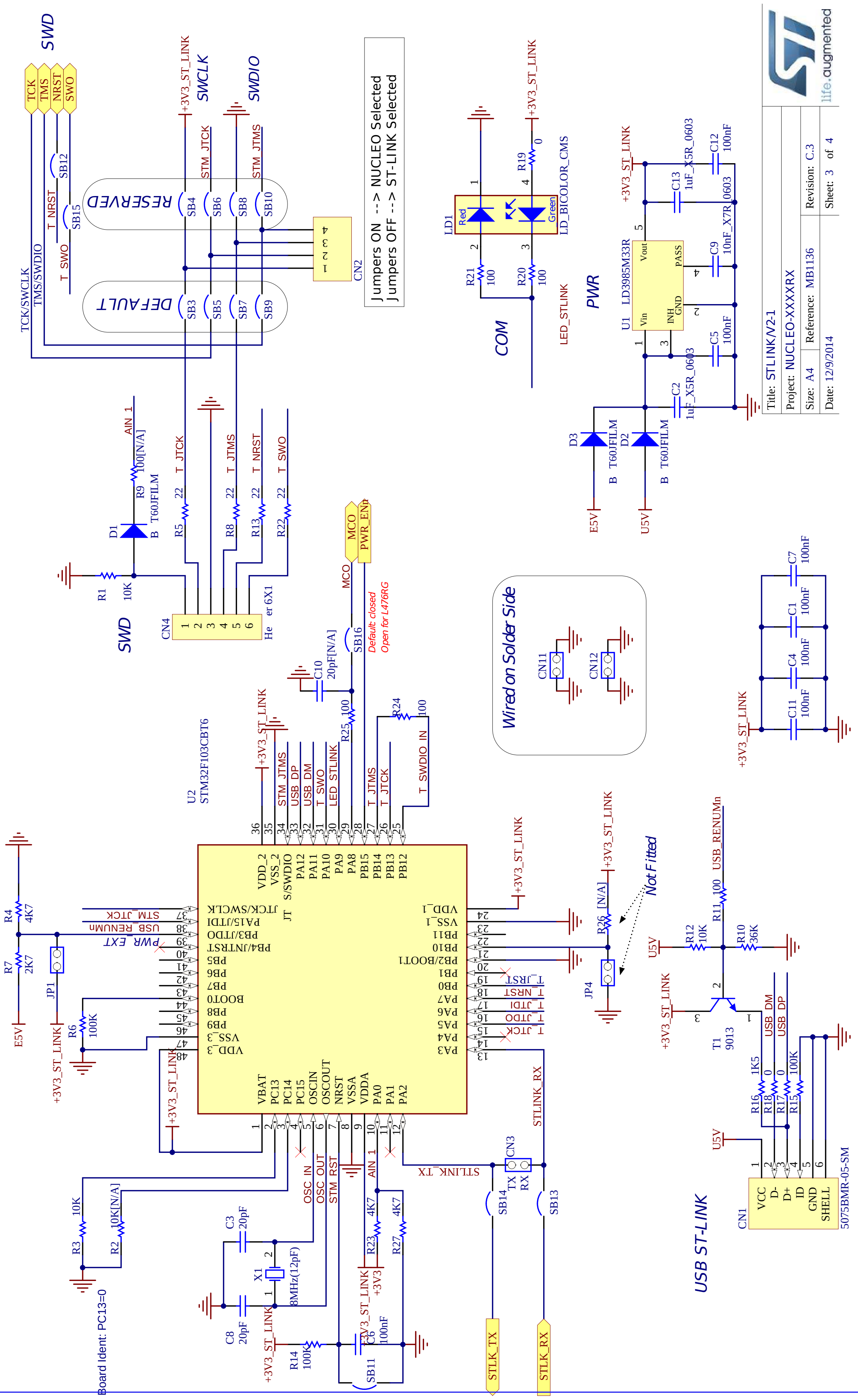


Figure 3.3: Nucleo Board schematic, page 3

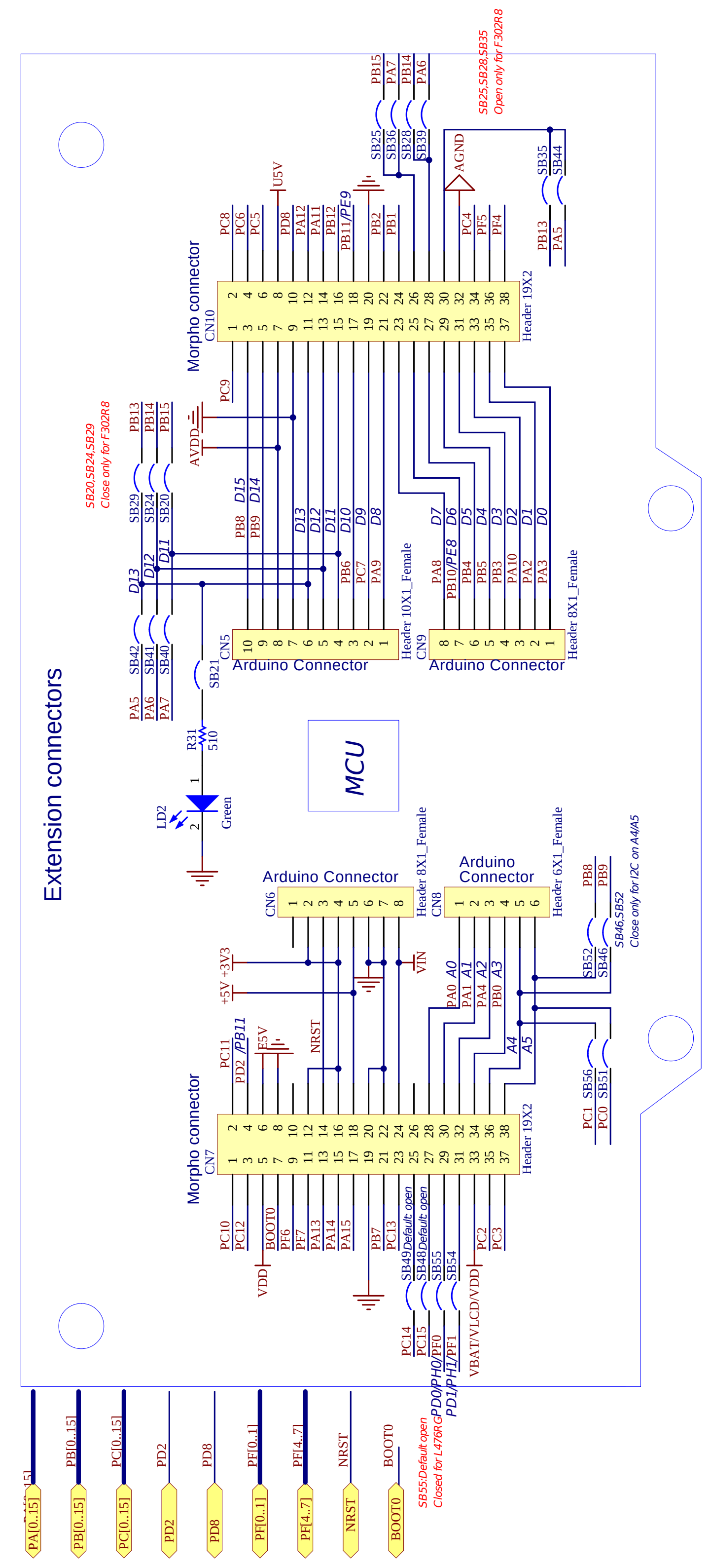


Figure 3.4: STNucleo schematic for led LD2 connections

Bibliography

[1] http://www.st.com/en/embedded-software/stm32cubef4.html

[2] UM1724 User manual STM32 Nucleo-64 boards

[3] ”RM0368 Reference manual STM32F401xB/C and STM32F401xD/E advanced ARM ® -based 32-bit MCUs”, available on www.st.com

[4] “STM32F401xD STM32F401xE ARM ® Cortex ® -M4 32b MCU+FPU, 105 DMIPS, 512KB Flash/96KB RAM, 11 TIMs, 1 ADC, 11 comm. interfaces Datasheet - production data”, available on www.st.com

[5] GDB: The GNU Project Debugger, link: https://www.gnu.org/software/gdb/

[6] ”UM1061, Description of STM32F2xx Standard Peripheral Library”, 634 pages, available on www.st.com

4

Knowledge base

In this chapter we will discuss the standard libraries provided for ARM Cortex M4 controllers. Throughout this book we rarely use them, since we want to focus on the register-level hardware handling, but those routines can be very useful in “normal” usage.

4.1 *TODO* CMSIS

From the ARM website:

“The ARM® Cortex® Microcontroller Software Interface Standard (CMSIS) is a range of projects to help accelerate software development on ARM Cortex processors, and enable better collaboration of effort in the industry to accelerate new technologies. Starting from CMSIS-CORE, a vendor-independent hardware abstraction layer for the Cortex-M processor series, the CMSIS project expanded into other areas such as reference design software component management and reference debugger interfaces. Creation of software is a major cost factor in the embedded industry. Standardizing the software interfaces across all Cortex-M silicon vendor products, especially when creating new projects or migrating existing software to a new device, means significant cost reductions.

The CMSIS enables consistent and simple software interfaces to the processor for interface peripherals, real-time operating systems, and middleware. It simplifies software re-use, reducing the learning curve for new microcontroller developers and cutting the time-to-market for devices”.

4.1.1 *TODO* CPAL Functions

Arm microcontrollers have two types of peripherals: The first type, the “core” peripherals are defined by ARM Ltd, while the “normal” peripherals are defined and implemented by the vendor that implements the microcontroller. The Core peripherals are standardized and the CMSIS framework provides some functions to handle them. This part of the framework is called CPAL (Core Peripheral Access Layer).

The core peripherals handled by this layer are (See the PM214, STM32programming manual, section 1.3.4, page 15 and section 4.1 (from page 192...) for further description [3]):

- Nested Vectored Interrupt Controller The NVIC is an embedded interrupt controller that supports low latency interrupt processing.

- System Control Block The System Control Block (SCB) is the programmers model interface to the processor. It provides system implementation information and system control, including configuration, control, and reporting of system exceptions.
- System timer The system timer, SysTick, is a 24-bit count-down timer. Use this as a Real Time Operating System (RTOS) tick timer or as a simple counter.
- Memory Protection Unit The Memory Protection Unit (MPU) improves system reliability by defining the memory attributes for different memory regions. It provides up to eight different regions, and an optional predefined background region.
- Floating-point Unit The Floating-Point Unit (FPU) provides IEEE754-compliant operations on single-precision, 32-bit, floating-point values.

Interrupt handler functions

Interrupt numbers: see typedef IRQn in stm32f4xx.h file

Interrupt CMSIS functions: see file CMSYS/core/core_cm4.h

CMSYS functions for NVIC: ("Table 45. CMSIS access NVIC functions", programming manual [3])

void NVIC_EnableIRQ(IRQn_Type IRQn) Enables the interrupt exception IRQn.

void NVIC_DisableIRQ(IRQn_Type IRQn) Disables an interrupt or exception.

void NVIC_SetPendingIRQ(IRQn_Type IRQn) Sets the pending status of interrupt or exception to 1.

void NVIC_ClearPendingIRQ(IRQn_Type IRQn) Clears the pending status of interrupt or exception to 0.

uint32_t NVIC_GetPendingIRQ(IRQn_Type IRQn) Reads the pending status of interrupt or exception. This function returns non- zero value if the pending status is set to 1.

void NVIC_SetPriority(IRQn_Type IRQn, uint32_t priority) Sets the priority of an interrupt or exception with configurable priority level to 1.

uint32_t NVIC_GetPriority(IRQn_Type IRQn) Reads the priority of an interrupt or exception with configurable priority

4.2 *TODO* St standard peripheral library

The STM32F2xx Standard Peripheral Library covers 3 abstraction levels, and includes:

- A complete register address mapping with all bits, bitfields and registers declared in C. This avoids a cumbersome task and more important, it brings the benefits of a bug free reference mapping file, speeding up the early project phase.
- A collection of routines and data structures covering all peripheral functions (drivers with common API). It can directly be used as a reference framework, since it also includes macros for supporting core-related intrinsic features, common constants, and definition of data types.
- A set of examples covering all available peripherals with template projects for the most common development tools. With the appropriate hardware evaluation board, this allows to get started with a brand-new micro within few hours.

x

Bibliography

[1] Cortex™ -M4 Devices Generic User Guide, http://infocenter.arm.com/help/topic/com.arm.doc.dui0553a/DU

5

Hands-on experiments: GPIO

5.1 Hands-on Experiment 5: Quadrature encoder input (polling)

In this experiment we will decode the quadrature encoder present on the PMDB16 board and set a variable that will be decremented/incremented depending on the rotation of the encoder axis.

A quadrature encoder (or incremental encoder) is a device that, depending on its position sets two digital lines. In figure 5.1there is the setting sequence while rotating the encoder handle.

The encoder used is an ALPS EC12D PCB encoder, used mainly for user-interfaces. There are also incremental encoders used in motion control, coupled with an electrical motor axis, to give informations about the position of the axis to the motor driver/controller. Those controllers are based exactly on the same principle.

in Figure 5.2 there is the schematic of the quadrature encoder section. Note the two leds that show the encoder lines status to ease debugging. Note also R84 that is not mounted in the STNucleo version of the board.

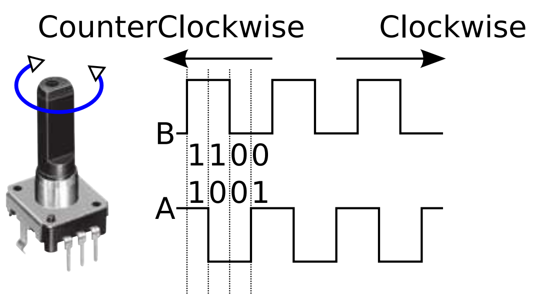


Figure 5.1: Quadrature encoder principle

Looking at the complete schematic, at page 14, 14 and 14 we see that the ENCODER_A signal is connected to PC7 and ENCODER_B_ALT to PC6 lines on the STNucleo boards, so we have to initialize those two GPIO pins.

In this experimet we will decode the incremental encoder using a polling routine (note that PC6 and PC7 have as alternate functions TIM3_CH1 and TIM3_CH2: Timers TIM2 to TIM5 can support incremental encoders directly, without the need of decoding software).

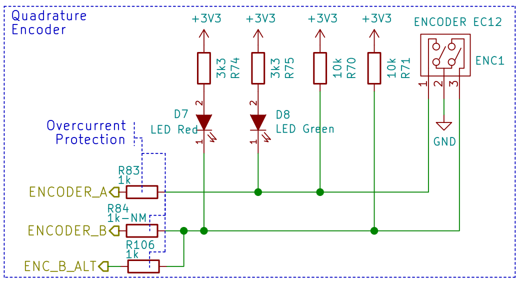


Figure 5.2: Encoder section schematic on PMDB16

5.1.1 Configuring the GPIOs

As seen in chapter 3, to configure a GPIO whe must

- Reset the peripheral
- Set the peripheral clock for the port
- Set the GPIO Registers
  - Configure P6 and P7 as Inputs
  - No need to set output type or output speed (since they are input)
  - Set Pull-up /pull down to none (we have external pull-ups -see schematics-)

5.1.1.1 Peripheral reset

Looking at the Reference manual[1], we see that the reset register to use is RCC_AHB1RSTR.

In the file stm32F4xx.h we see the definition for that register and the #define for the constant we can use[2].

So, to reset the peripheral we would do:

```
RCC->AHB1RSTR|=RCC_AHB1RSTR_GPIOCRST; //Reset peripheral
RCC->AHB1RSTR&=~RCC_AHB1RSTR_GPIOCRST;//Run the peripheral
```

This approach can be error prone, since the constant used are defined as #defines in the .h code. A much better way to proceed should re-define the RCC type using unions and bit fields, in order to be able to reset the peripheral with instructions like

```
RCCb->AHB1RSTR.GPIOCRST=1; //Reset the peripheral
RCCb->AHB1RSTR.GPIOCRST=0; //Run the peripheral
```

that is much more human-readable, using also the same names for the bits used in the data-sheet. To be able to do so we can copy the definition of RCC_TypeDef from the .h file and create a union between the 32 bit value of the register and a bitfield structure, changing the line

[1]Search for "peripheral reset" on the pdf file, or go to section 2.3.4-5-6 of the reference manual
[2]Search for "RCC" in the file stm32F4xx.h, located under CMSIS->device in our project

CODE
```c
__IO uint32_t AHB1RSTR;
```

with the code

CODE
```c
union {__IO uint32_t v;
       struct {
                int GPIOARST :1;
                int GPIOBRST :1;
                int GPIOCRST :1;
                int GPIODRST :1;
                int GPIOERST :1;
                int RESE1 :2;
                int GPIOHRST :1;
                int RESED2 :4;
                int CRCRST :1;
                int RESE3 :8;
                int DMA1RST :1;
                int DMA2RST :1;
                int RESE4 :9;
               } ;
}AHB1RSTR; /*!< RCC AHB1 peripheral reset register, Address offset: 0x10 */
```

change name of the structure from RCC_TypeDef to RCC_Typedef_UC
 and define “our” RCC as

CODE
```c
#define RCCb RCC ((RCC_TypeDef_UC *) RCC_BASE)
```

just copying the RCC define from the .h file and changing it. This part may seem a waste of time, since also the previous approach works, but it is a good programming practice to hide complexity in data instead of code, that is much more difficult to debug than a data structure. Investing time in having the source code as simple and readable as possible pays back a lot when, even after years, we will have to modify the code. Moreover, once defined this register, the definition can be re-used in other code and the time spent is a “one-time” investment.

5.1.2 Set the peripheral clock

The peripheral clock is set using the AHBxENR and APBxENR registers (see user manual, searching for “peripheral clock”, or section 6.2 and 6.3 for the clock system description)

The enable register we are interested in is AHB1ENR. See section 6.3.13 for AHB1ENR register description.

CODE
```c
RCC->AHB1ENR |= RCC_AHB1ENR_GPIOCEN; //Enable clock for GPIOC
```

Also here a better way to proceed should have a line as

CODE
```c
RCC->AHB1ENR->GPIOCEN=1; //enable clock for GPIOC
```

To have this, in RCC_typedef_UC we created before change the definition of AHB1ENR to:

CODE
```
union { __IO uint32_t value;
        struct {
                 int GPIOAEN :1;
                 int GPIOBEN :1;
                 int GPIOCEN :1;
                 int GPIODEN :1;
                 int GPIOEEN :1;
                 int RES1 :2;
                 int GPIOHEN :1;
                 int RES2 :4;
                 int CRCEN :1;
                 int RES3 :8;
                 int DMA1EN :1;
                 int DMA2EN :1;
                 int RES4 :9;
        };
} AHB1ENR; /*!< RCC AHB1 peripheral clock register, Address offset: 0x30 */
```

5.1.3 Set the GPIO Registers

Also for the GPIO registers we can change the provided definition. The complete structure for GPIO_typedef_b that we used is at the end of this chapter.

Set the pin as input setting MODER

CODE
```
GPIOCb->MODER.MODER6 = MODER_GPINPUT; //Set GP6 as input.
GPIOCb->MODER.MODER7 = MODER_GPINPUT; //Set GP7 as input.
```

Disable pull-ups setting GPUPR (no need: at reset no pull-up/down are enabled on port c, see section 8.4.4 of the reference manual).

CODE
```
GPIOCb->PUPDR.PUPDR6=PUPDR_NONE;
GPIOCb->PUPDR.PUPDR6=PUPDR_NONE;
```

—fIXME: ADD ALSO THE WAY WITH DEFINES—-

5.1.4 Reading the GPIOs

To read the GPIOs we read the "port input data register" IDR (Section 8.4.5 of the reference manual).

CODE
```
int enc_A;
int enc_B;
// *******Main loop********
for(;;)
{        enc_A=GPIOCb->IDR.IDR7;
     enc_B=GPIOCb->IDR.IDR6;
}
```

Now we can compile, connect the PMDB16 board to the PC and launch a debug session. Placing a breakpoint over the final bracket of the for cycle we will read the status of both the inputs of the encoder line.

Note that when the line is at the “1” status the led will be off, since it is connected between the positive supply and the line.

Also note that in the debug window, being declared as int, the two variables will be -1 when the line is at 3.3V (led off).

5.1.5 Decoding the signals

Looking at the sequence we see that:

- if B changes to A state: clockwise,
- if B changes to notA state: counterclockwise.
- if A chenages to B state: counterclockwise.
- if A changes to notB state: clockwise.

So to decode the movements of the encoder we will need to store the previous status of the two lines encoder_A and encoder_B, to check if one of them changed. If both of them changed status we are not able to say in which direction we moved (we ’lost’ a step).

Moreover, since the int variables are 32 bit wide, the previous variables used to read the inputs “waste” most of the memory reserved. We have quite a lot of RAM available, but here we also focus on techniques to optimise RAM usage.

As said before is a good programming practice to hide complesity in data and not in code, so we will define some structures/union to have the work done.

The encoder data structure holds space for the actual and the “old” status of the A and B lines an an integer with the calculated position.

Note that a union has been used to have both the bits for A and B lines in a single variable (for the comparation we need to do). This is declared as a global variable, outside the main.

CODE
```
struct{
  union{
    struct{ int A:1;
                  int B:1;
          };
    int status:2;
  }actual;
  union{
    struct{ int A:1;
              int B:1;
          };
        int status:2;
  }old;
  int position; //Actual position of the encoder
} encoder;
```

The decoding routine becomes:

CODE
```
/// Quadrature encoder decoding routine.
void  decodeQuadratureEncoder(void)
{         // Update the status of pins in RAM.
        encoder.actual.A=GPIOCb->IDR.IDR7; // encoder_A signal
        encoder.actual.B=GPIOCb->IDR.IDR6; // encoder_B signal
        if (encoder.old.status!=encoder.actual.status) //something changed
        {
                if (encoder.actual.B!=encoder.old.B) //B changed?
                { //B changed
                        if (encoder.actual.B==encoder.actual.A) //B following A
                        {
                                encoder.position++;
                        }
                        else encoder.position--;
                }
                else
                { //A changed.
                        if (encoder.actual.A==encoder.actual.B) //A following B
                                encoder.position--;
                        else
                                encoder.position++;
                }
                // Update the "old" value for the next comparison.
                encoder.old.status=encoder.actual.status;
        }
}
```

And the new main loop become

CODE

```
        encoder.position=0; //Initialize encoder position.
        //**********MAIN LOOP*****************************
        for(;;)
        {       decodeQuadratureEncoder();
        } //for .
```

Now we can compile and debug. Note that the data structure “encoder” does not appear in the “variable” tab. We should add it in the “expressions” tab clicking on “Add new expression” inside the tab and write (by hand) “encoder”.

5.2 Hands-on Experiment 6: Keyboard scan (polling)

In this experiment we will read the keyboard matrix at the bottom-left of the PMDB16 board ending with a routine that puts the status of the 16 keys into a global variable that can be read from all the other routines in the program.

To be able to digitally read more keys than the GPIO used to read them, a matrix-type connection of the keys is used. In this way it is possible to detect n^2 keys using $2n$ GPIOs or $m * n$ keys using $m + n$ GPIOs, with n and m integer numbers.

The typical connection, for a 4x4 matrix is shown in figure.

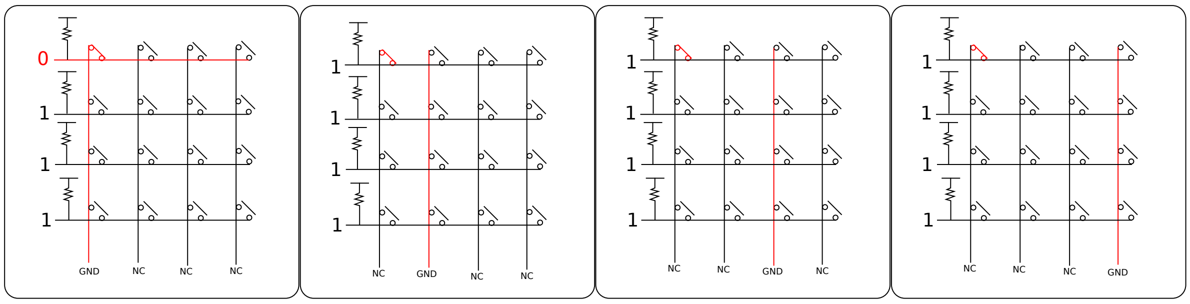


Figure 5.4: Keyboard scan (NC=Not Connected).

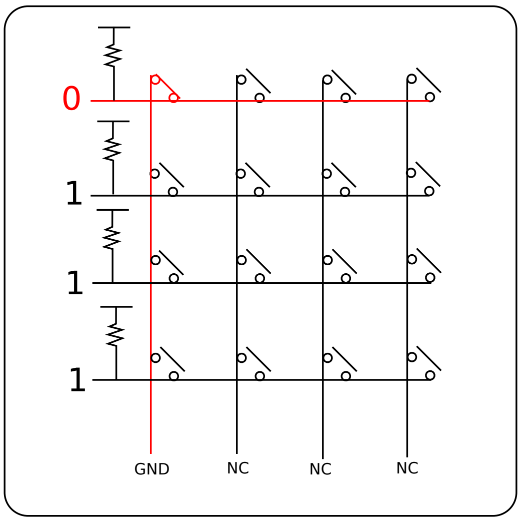


Figure 5.3: 16 key matrix arranged in a 4x4 array.

To decode such an array the following steps are done:

- activate column zero, driving low the corresponding line and leaving floating the other column lines
- read the four "row" input, to detect keypress on column zero.
- Repeat the previous step Activating column one, two, three one at-a-time.

Using this method the keypress of a key is detected.

To be able to scan the keyboard on the PMDB16 we have to look at the schematics to see which lines have to be used. Looking at pages 14 and 14 and 14 we see that the connections for the columns are implemented using some NPN transistors, driven by the lines PC8, PC9, PC10, PC11. Notice that usually the columns are connected directly to the GPIOs of the microcontroller and driven open-collector. Since the PMDB16 has been designed for didactical purposes, the use of the transistors avoids possible misconfiguration of the outputs that can damage the microcontroller (i.e. configuring the outputs as push-pull instead of open drain and pressing simultaneously two keys, shorting them).

From the same schematics we see also that the row lines are PC2, PC3, PC12 and PC13. Also here we notice on the schematic that there are some 1k resistors connected in series to the row lines that usually should be directly connected to the microcontroller. Also those resistors are put there to avoid possible damage from misconfiguration of the lines.

At this time we have to configure:

- PC8, PC9, PC10, PC11 as OUTPUTS, push-pull (they have to drive the transistors)
- PC2, PC3, PC12, PC13 as INPUTS (they read the row status)

In order to have a better organization in code, we create a different routine to initialize I/O pins and ports. In our code we will use the GPIO redefine that is at the end of this chapter, to have a code more readable.

Note that before calling this routine the GPIOC peripheral should be reset and with the clock enabled using

CODE
```
RCC_b->AHB1RSTR.GPIOCRST = 1; // Set reset bit on port C
RCC_b->AHB1RSTR.GPIOCRST = 0; // Clear reset bit on port C
RCC_b->AHB1ENR.GPIOCEN = 1; // Set the clock enable bit for GPIOC
//or, using the standard method: RCC->AHB1ENR|=RCC_AHB1ENR_GPIOCEN;
```

And the init routine will be:

CODE
```
void keyborard_io_init(void)
{     // Row input configuration, PC2, PC3, PC12, PC13.
      GPIOC_b->MODER.MODER2 = MODER_GPINPUT;
      GPIOC_b->MODER.MODER3 = MODER_GPINPUT;
      GPIOC_b->MODER.MODER12 = MODER_GPINPUT;
      GPIOC_b->MODER.MODER13 = MODER_GPINPUT;

      //Columns: PC8, PC9, PC10, PC11
      // Column output configuration: PC8
      GPIOC_b->MODER.MODER8 = MODER_GPOUTPUT;
         // or, using the standard method: GPIOC->MODER=(GPIOC->MODER &
          ↪  (~GPIO_MODER_MODER8))|GPIO_MODER_MODER8_0;
      GPIOC_b->OTYPER.OT8 = OTYPE_PUSH_PULL;
         // or, using the standard method: GPIOC->OTYPER &= (~GPIO_OTYPER_OT_8);
      GPIOC_b->OSPEEDR.OSPEEDR8 = OSPEED_HIGH;
         // or, using the standard method: //GPIOC->OSPEEDR |=
          ↪  GPIO_OSPEEDER_OSPEEDR8;

          // Column output configuration: PC9
      GPIOC_b->MODER.MODER9 = MODER_GPOUTPUT;
      GPIOC_b->OTYPER.OT9 = OTYPE_PUSH_PULL;
      GPIOC_b->OSPEEDR.OSPEEDR9 = OSPEED_HIGH;

      // Column output configuration: PC10
      GPIOC_b->MODER.MODER10 = MODER_GPOUTPUT;
      GPIOC_b->OTYPER.OT10 = OTYPE_PUSH_PULL;
      GPIOC_b->OSPEEDR.OSPEEDR10 = OSPEED_HIGH;

      // Column output configuration: PC11
      GPIOC_b->MODER.MODER11 = MODER_GPOUTPUT;
      GPIOC_b->OTYPER.OT11 = OTYPE_PUSH_PULL;
      GPIOC_b->OSPEEDR.OSPEEDR11 = OSPEED_HIGH;
}
```

Having initialized the ports we will write the keyboard scan routine, starting from the column enable code.

CODE
```
/// Keyboard scan routine
void keyboardScan(void)
{    static unsigned char currentColumn=0;
     //readRows(currentColumn); ---todo---
     if (currentColumn<3)
         currentColumn++;
     else
         currentColumn=0;
     enableColumn(currentColumn);
}
```

Notice that the row read code have to be before the current column update, since the controller is so fast that a delay is needed between the column set and the input read. Organizing the routine in this way the delay can be in the main loop (or in the interrupt routine) that will drive this read and we will not hang the processor wasting time in a wait cycle here. It is much better having routines that do part of the work at every call, but without wait cycles inside of them than have a routine tat wastes time. Organizing software in this way allows to move the keyboard scan routine in an interrupt routine that will give the timings without using too much processing power.

Now we need to write the routine that enables the correct column of the keyboard. Since we need to enable one of the lines PC8,9,10,11, we can use the BSRR register. Using our redefined structure for GPIOs (see section 14 in this chapter) we can write:

CODE
```
void enableColumn(unsigned char currentColumn)
{   //resets PC8, PC9, PC10, PC11
    GPIOC_b->BSRR.value= (0b1111<<(8+16));
    //sets the current column output.
    GPIOC_b->BSRR.value= 1<<(8+currentColumn);
}
```

While, using only the Standard peripheral library defines it should be:

CODE
```
void enableColumn(unsigned char currentColumn)
{   //resets PC8, PC9, PC10, PC11 turning OFF the column
    GPIOC->BSRRH= (0b1111<<8);
    //sets the current column output.
    GPIOC->BSRRL= 1<<(8+currentColumn);
}
```

We will also add in the main loop the call to keyboardScan

CODE
```
for (;;)
{ keyboardScan();
}
```

Now we will test the routine putting a breakpoint at the first line of the routine “enableColumn”.

At every break we should see the leds at the bottom of the keyboard turning on, one-at-a-time. (D1, D2, D3 and D4, see schematic at page 14).

After the column scan routine, it is time to red the column data. We want to read the keyboard status and be able to recognize easily in the remaining program which key has been pressed, we define a data structure where we will store the keyboard data. We will use a union between a structure with the single keys (as one-bit integers) and an integer that holds them all.

CODE
```
// Keyboard data structure
union{  struct{
                int k0:1; //one bit, status of key 0.
                int k1:1; //one bit, status of key 1.
                int k2:1; //one bit, status of key 2.
                int k3:1; //one bit, status of key 3.
                int k4:1; //one bit, status of key 4.
                int k5:1; //one bit, status of key 5.
                int k6:1; //one bit, status of key 6.
                int k7:1; //one bit, status of key 7.
                int k8:1; //one bit, status of key 8.
                int k9:1; //one bit, status of key 9.
                int kA:1; //one bit, status of key 10.
                int kB:1; //one bit, status of key 11.
                int kC:1; //one bit, status of key 12.
                int kD:1; //one bit, status of key 13.
                int kE:1; //one bit, status of key 14.
                int kF:1; //one bit, status of key 15.
              };
         unsigned int value;
     } keyboard;
```

We will declare this structure as global, so that the routines outside the keyboard-scan will be able to read the key status.

Looking at the schematic and the board layout we see that the button name we use in the software (we choose to folollow the "map" on the silkscreen in the rectangle named "keyboard") is different from the name of the components "buttons" on the board (that are K0, K1, K2, K3, K10, K11, K12, K13, K20, K21, K22, K23, K30, K31, K32, K33). The different naming is common in electronics, so we will should get used to it.

Following the schematic we have the following routing between signals.

Keys	Electrical signal (net)	Software row name	GPIO
C,D,E,F	KEYB-R0	ROW_3	PC3
8,9,A,B	KEYB-R1	ROW_2	PC2
4,5,6,7	KEYB-R2	ROW_1	PC13
0,1,2,3	KEYB-R3	ROW_0	PC12

Looking at the schematic we also have this mapping for the column drive signals

Keys	column drive signal	GPIO	Sw currentColumn
3,7,B,F	KEYB-C0	PC8	0
2,6,A,E	KEYB-C1	PC9	1
1,5,9,D	KEYB-C2	PC10	2
0,4,8,C	KEYB-C3	PC11	3

Now we need to be able to read the row status, so we define some macros to make the code more readable.

CODE
```
#define READ_ROW_0() (GPIOC_b->IDR.IDR12==0)
#define READ_ROW_1() (GPIOC_b->IDR.IDR13==0)
#define READ_ROW_2() (GPIOC_b->IDR.IDR2==0)
#define READ_ROW_3() (GPIOC_b->IDR.IDR3==0)
```

Now we write the function readRows, that read the row data storing them in the right position in the data structure “keyboard”.

CODE
```
void readRows(unsigned char currentColumn)
{ switch (currentColumn)
           { case 0: keyboard.k0=READ_ROW_0();
                     keyboard.k4=READ_ROW_1();
                     keyboard.k8=READ_ROW_2();
                     keyboard.k12=READ_ROW_3();
                     break;
             case 1: keyboard.k1=READ_ROW_0();
                     keyboard.k5=READ_ROW_1();
                     keyboard.k9=READ_ROW_2();
                     keyboard.k13=READ_ROW_3();
                     break;
             case 2: keyboard.k2=READ_ROW_0();
                     keyboard.k6=READ_ROW_1();
                     keyboard.k10=READ_ROW_2();
                     keyboard.k14=READ_ROW_3();
                     break;
             case 3: keyboard.k3=READ_ROW_0();
                     keyboard.k7=READ_ROW_1();
                     keyboard.k11=READ_ROW_2();
                     keyboard.k15=READ_ROW_3();
                     break;
            } //switch
}//readRows()
```

Now we can de-comment the readRows function in the keyboardScan routine

CODE
```
/// Keyboard scan routine
void keyboardScan(void)
{    static unsigned char currentColumn=0;
     readRows(currentColumn);
     if (currentColumn<3)
         currentColumn++;
     else
         currentColumn=0;
     enableColumn(currentColumn);
}
```

Now we can debug the keyboard scan routine, putting a breakpoint at the beginning of the readRows function, in order to follow it step-by-step.

We can now fund an interesting bug in our code: Since the controller we are using is quite fast, the time that elapses between the set of the next column and the reading of the row data is so small that the electrical signals do not have the time to settle, so our program will work perfectly if we have a breakpoint at the readRows routine but does not work well if it is running free. To avoid this problem we need to introduce a delay between two subsequent calls of the keyboardScan routine. This can be acheaved calling the scan routine from a time-driven interrupt routine, or with an “ugly” delay in the main loop

CODE
```
for (;;)
{ keyboardScan();
```

```
        //ugly delay
        for (int i=0; i<1000;i++)
        {       asm("nop"); //just an instruction to avoid the compiler to
        ↪   optimize-out an empty loop.
        }
}
```

Now we can debug our program letting it running free for a while, pressing a key and stop it using the 'pause' button. The variable keyboard will have the field keyboard.kx corresponding to the key "x" set to "-1" (remember it is an integer and a signed integer of one bit can have only the values 0 and -1) and the variable keyboard.value will be 1 presisng k0, 2 pressing k1, 4 pressing k2 ...32768 pressing kF.

To activate an event on keypress we can now just compare the keyboard.value with the key combination we want to detect, as 1 for k0, 3 for k0 and k1 pressed simultaneously and so on, like:

CODE
```
#define KEY_0_ONLY_PRESSED (keyboard.value==0x01)
if (KEY_0_ONLY_PRESSED)
{       do_something();
}
```

Note that this is very different from

CODE
```
#define KEY_0_PRESSED (keyboard.k0==1)
if (KEY_0_PRESSED)
{       do_something();
}
```

since in the first case ONLY the key 0 must be pressed to make the expression true, while in the second example the condition will be true also if any other key is pressed simultaneously with k0.

5.3 Chapter appendix: the re-definition of GPIO type (complete)

Here it is our modified version for the GPIO_typedef, so that the single bit of the registers are accessible using C structures (and unions). To access the register as a whole the union with "value" is provided, i.e. an instruction as

```
GPIOA->MODER.value
```

will access the 32 bit value as it did with the original structure using

```
GPIOA->MODER.
```

CODE
```
typedef struct {
  union   {
    __IO uint32_t value;
    struct
    {
      volatile int MODER0 :2;
      volatile int MODER1 :2;
      volatile int MODER2 :2;
      volatile int MODER3 :2;
      volatile int MODER4 :2;
      volatile int MODER5 :2;
      volatile int MODER6 :2;
      volatile int MODER7 :2;
      volatile int MODER8 :2;
      volatile int MODER9 :2;
      volatile int MODER10 :2;
      volatile int MODER11 :2;
      volatile int MODER12 :2;
      volatile int MODER13 :2;
      volatile int MODER14 :2;
      volatile int MODER15 :2;
    };   /*!< GPIO port mode register, Address offset: 0x00 */
  } MODER; //union
  union
  {
    __IO uint32_t value;
    struct
    {
      int OT0 :1;
      int OT1 :1;
      int OT2 :1;
      int OT3 :1;
      int OT4 :1;
      int OT5 :1;
      int OT6 :1;
      int OT7 :1;
      int OT8 :1;
      int OT9 :1;
      int OT10 :1;
      int OT11 :1;
      int OT12 :1;
      int OT13 :1;
      int OT14 :1;
      int OT15 :1;
    };
  } OTYPER; /*!< GPIO port output type register, Address offset: 0x04 */
  union
  {
    __IO uint32_t value; // To access the register at a whole (all 32 bits).
    struct
    {
      int OSPEEDR0 :2;
      int OSPEEDR1 :2;
      int OSPEEDR2 :2;
      int OSPEEDR3 :2;
      int OSPEEDR4 :2;
      int OSPEEDR5 :2;
      int OSPEEDR6 :2;
      int OSPEEDR7 :2;
      int OSPEEDR8 :2;
      int OSPEEDR9 :2;
      int OSPEEDR10 :2;
      int OSPEEDR11 :2;
      int OSPEEDR12 :2;
      int OSPEEDR13 :2;
      int OSPEEDR14 :2;
      int OSPEEDR15 :2;
    };
  } OSPEEDR; /*!< GPIO port output speed register, Address offset: 0x08 */
  union
  {
    __IO uint32_t value;
    struct
    {
      int PUPDR0 :2;
      int PUPDR1 :2;
      int PUPDR2 :2;
      int PUPDR3 :2;
      int PUPDR4 :2;
      int PUPDR5 :2;
      int PUPDR6 :2;
      int PUPDR7 :2;
      int PUPDR8 :2;
      int PUPDR9 :2;
      int PUPDR10 :2;
      int PUPDR11 :2;
      int PUPDR12 :2;
      int PUPDR13 :2;
      int PUPDR14 :2;
      int PUPDR15 :2;
    };
  } PUPDR; /*!< GPIO port pull-up/pull-down register, Address offset: 0x0C */
  union
  {
    __IO uint32_t value;
    struct
    {
      int IDR0 :1;
      int IDR1 :1;
      int IDR2 :1;
      int IDR3 :1;
      int IDR4 :1;
      int IDR5 :1;
      int IDR6 :1;
      int IDR7 :1;
      int IDR8 :1;
      int IDR9 :1;
      int IDR10 :1;
      int IDR11 :1;
      int IDR12 :1;
      int IDR13 :1;
      int IDR14 :1;
      int IDR15 :1;
    };
  } IDR; /*!< GPIO port input data register, Address offset: 0x10 */
  union
  {
    __IO uint32_t value;
    struct
    {
      int ODR0 :1;
      int ODR1 :1;
      int ODR2 :1;
      int ODR3 :1;
      int ODR4 :1;
      int ODR5 :1;
      int ODR6 :1;
      int ODR7 :1;
      int ODR8 :1;
      int ODR9 :1;
      int ODR10 :1;
      int ODR11 :1;
      int ODR12 :1;
      int ODR13 :1;
      int ODR14 :1;
      int ODR15 :1;
    };
  } ODR; /*!< GPIO port output data register, Address offset: 0x14 */
  __IO uint16_t BSRRL; /*!< GPIO port bit set/reset low register,  Address offset: 0x18 */
  __IO uint16_t BSRRH; /*!< GPIO port bit set/reset high register, Address offset: 0x1A */
  union
  {
    __IO uint32_t value;
    struct
    {
      int LCK0 :1;
      int LCK1 :1;
      int LCK2 :1;
      int LCK3 :1;
      int LCK4 :1;
      int LCK5 :1;
      int LCK6 :1;
      int LCK7 :1;
      int LCK8 :1;
      int LCK9 :1;
      int LCK10 :1;
      int LCK11 :1;
      int LCK12 :1;
      int LCK13 :1;
      int LCK14 :1;
      int LCK15 :1;
    };
  } LCKR; /*!< GPIO port configuration lock register, Address offset: 0x1C */
  union
  {
    __IO uint32_t value;
    struct
    {
      int AFRL0 :4;
      int AFRL1 :4;
      int AFRL2 :4;
      int AFRL3 :4;
      int AFRL4 :4;
      int AFRL5 :4;
      int AFRL6 :4;
      int AFRL7 :4;
    };
  } AFRL; /*!< GPIO alternate function registers, Address offset: 0x20-0x21 */
  union
  {
    __IO uint32_t value;
    struct
    {
      int AFRL8 :4;
      int AFRL9 :4;
      int AFRL10 :4;
      int AFRL11 :4;
      int AFRL12 :4;
      int AFRL13 :4;
      int AFRL14 :4;
      int AFRL15 :4;
    };
  } AFRH; /*!< GPIO alternate function registers, Address offset: 0x22-0x23 */
} GPIO_TypeDef_b;

#define GPIOAb ((GPIO_TypeDef_b *) GPIOA_BASE)
#define GPIOBb ((GPIO_TypeDef_b *) GPIOB_BASE)
#define GPIOCb ((GPIO_TypeDef_b *) GPIOC_BASE)

#define MODER_GPINPUT         0x00
#define MODER_GPOUTPUT  0x01
#define MODER_ALTERNATE_FUNCTION 0x02
#define MODER_ANALOG 0x03
//defines used for gpio OTYPE.
#define OTYPE_OPEN_DRAIN 1
#define OTYPE_PUSH_PULL 0
//Defines used for OSPEED:
#define OSPEED_LOW    0b00
#define OSPEED_MEDIUM 0b01
#define OSPEED_FAST   0b10
#define OSPEED_HIGH   0b11
//Defines for pull-up-pulldown in PUPDR
#define PUPDR_NONE 0
#define PUPDR_PULLUP 1
#define PUPDR_PULLDOWN 0x02
// 0x03 is reserved, so no define for that value.
```

6

Hands-on experiments: SPI

In this chapter we will setup and use the SPI of the microcontroller, driving the led matrix on the PMDB16 board.

6.1 PMDB16 SPI chain hardware description: The led matrix, the SPI expansion connector.

One of the SPIs of the controller (SPI1) is connected to a chain of shift registers, that are used to expand I/O capabilities of the controller. The chain is made of:

- U5: a 74HC595 SIPO that drives the five column drive transistors of the led matrix
- U8: a 74HC595 SIPO that drives the seven rows of the led matrix
- U3 (not mounted in the ST-Nucleo version): A 74HC595 SIPO that drives the LCD module lines. This component is optional and not needed to use the LCD screen using GPIOs from the controller. It can be used to drive the LCD using the SPI and develop that function. If U3 is enabled (With P4 or P7 shorted), the GPIOs used for parallel drive of the LCD MUST be configured as inputs. U3 is mounted only in "Arduino" and "Configurable" versions. If the parallel drive pins are needed for other functions, solderable jumpers JPRS1, JPE1, JPD4, JPD5, JPD6, JPD7 can be cut and disconnected from the LCD To disconnect the Backlight line remove R52 (See board schematic).
- U9 (Not mounted in the ST-Nucleo version): A 74HC595 SIPO that drives the keyboard lines.
- U10 (Not mounted in the ST-Nucleo version): a 74HC... PISO that reads the keyboard rows via the SPI)
- When U3, U9 and U10 are not mounted, they are bypassed by jumpers, in order to have anyway a complete chain. P11 should be removed if an external SPI peripheral is connected to the board (so that the controller MISO line is correctly connected).

In this chain, the last two bytes sent before the strobe (SS) signal are the ones used to drive the led array.

The led matrix will be driven one-column at a time, turning on the relative leds in the column and U5 will take care of driving the transistors that connect the column common cathode to the 0V line..

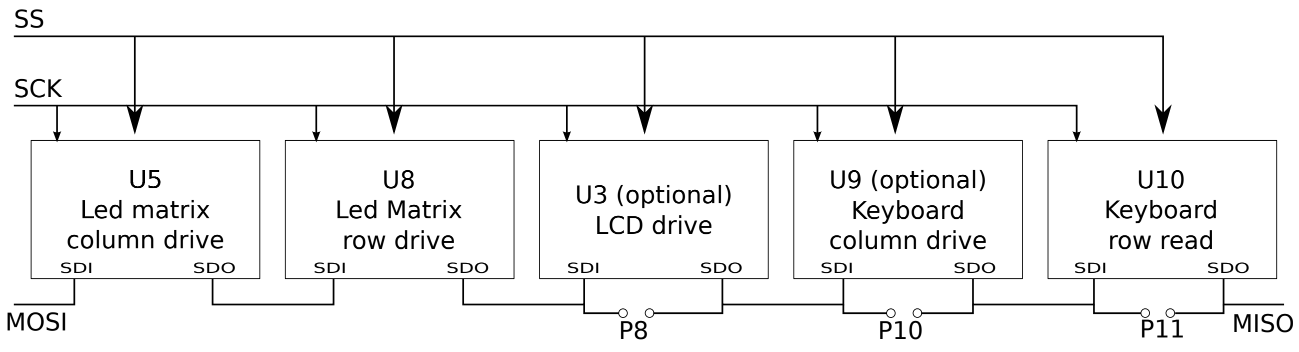


Figure 6.1: PMDB16 SPI Chain block diagram

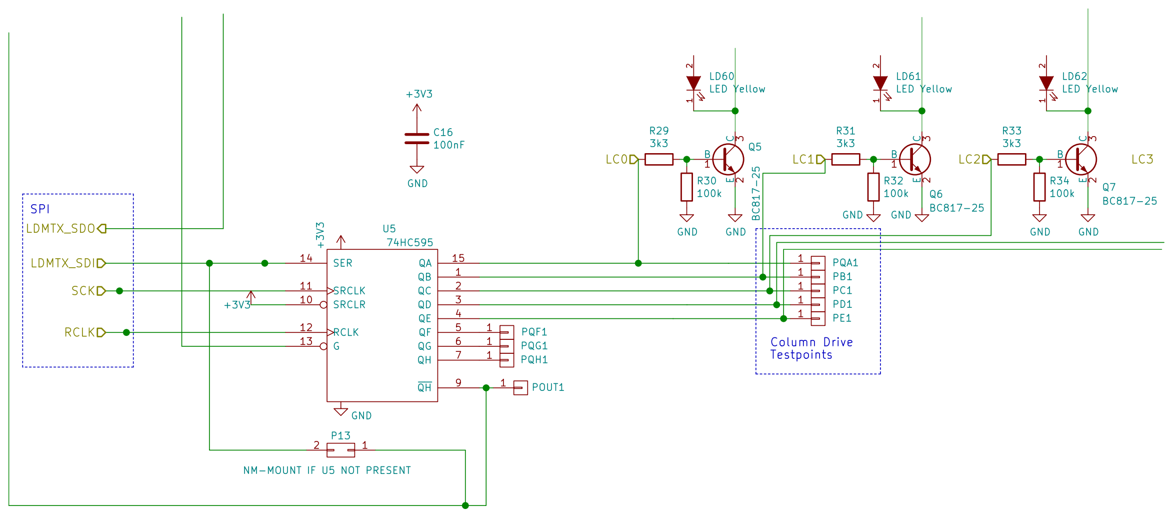


Figure 6.2: Detail of the led matrix column drive circuit (columns 3 and 4 removed)

The row drive is implemented with U6, that drives directly the leds anode.

Since the chain has as first shift register U5 and second U8, we will have that the last byte before the strobe signal will set U5 (the column drive) and the previous one the row data.

Looking at figure 6.5 (taken from the sn74HC595 data-sheet) we see that in the rising edge of the SRCLK the input signal is sampled, and at the falling edge the input signal can change state, so we will need to configure the SPI accordingly.

6.2 Hands-on Experiment 7: *WORK-IN-PROGRESS* Scanning the led matrix via SPI

In this experiment we will drive the led matrix, showing a small image on it.

6.2.1 Configuring the SPI

To configure the SPI1 we first take a look at the user manual [3], chapter 20 (Serial Peripheral Interface). From the user manual we see figure 6.6 where we note that the timings we need will be with the bit CPOL=1 (to have the clock at the high level when idle, so that the led LDSCK1 that shows our

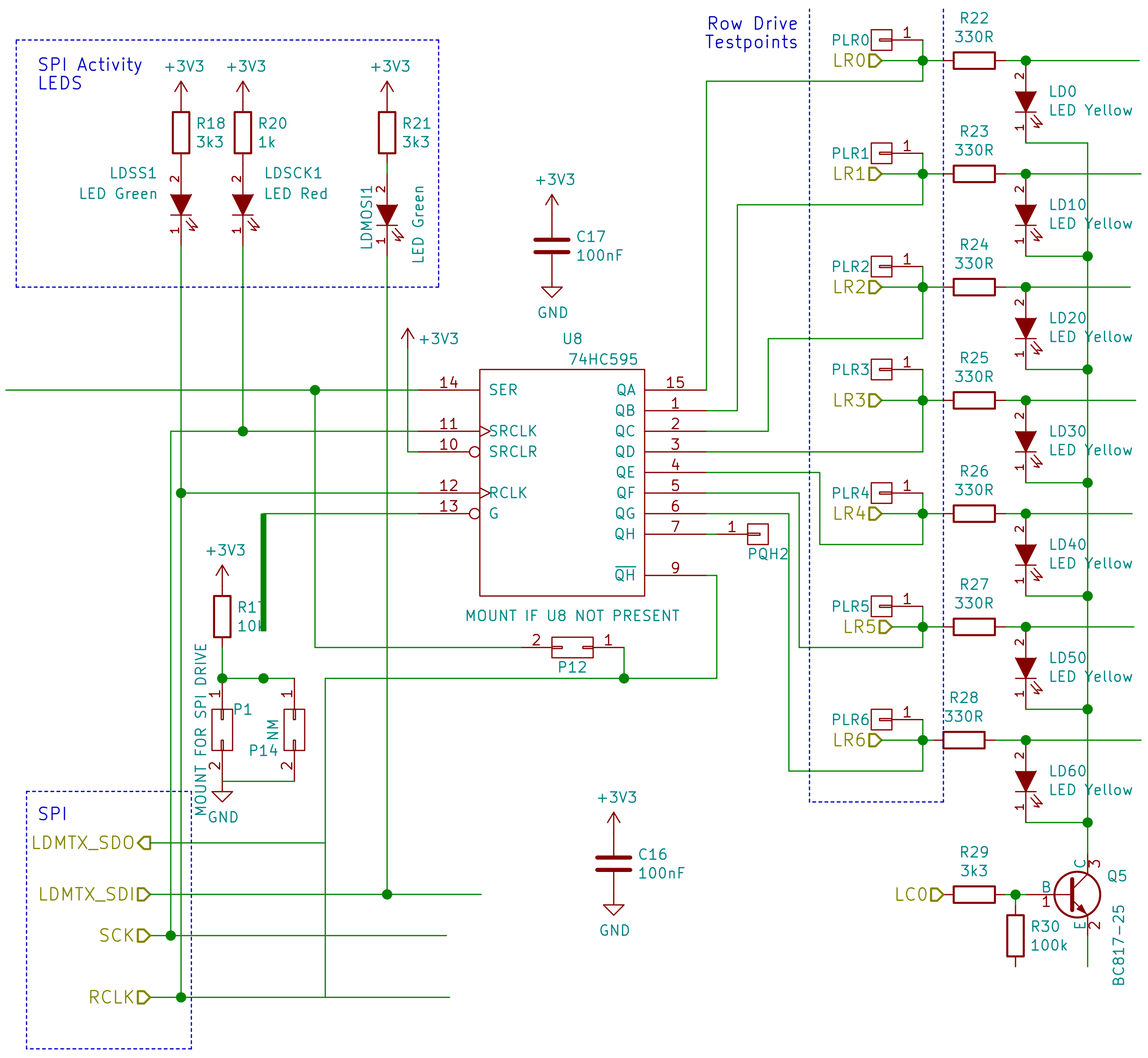


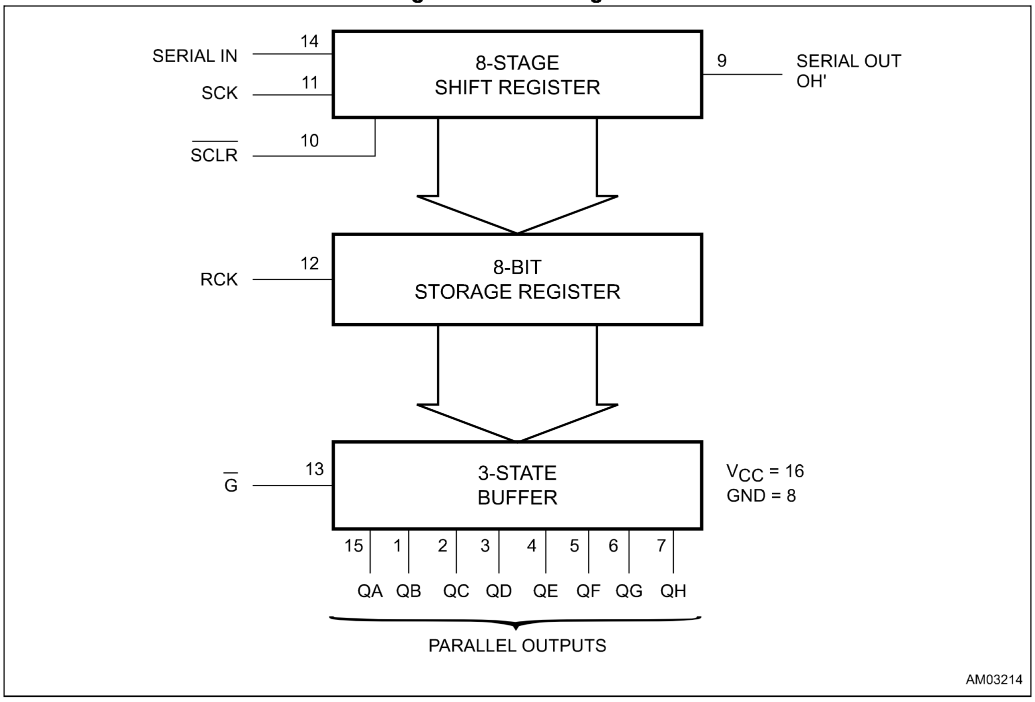
Figure 6.3: Detail of the LED matrix row drive circuit

spi activity will be OFF when there is no SPI activity) and CPHA=1 (sampling on the rising edge, changing on the falling edge).

Also in paragraph 20.3.3 (Configuring SPI In Master Mode) we find the correct procedure to initialize the SPI, that we will follow, but first (not reported in the procedure), we will create the SPI_init() routine to hold all the initialization and we must reset the peripheral and enable its clock.

CODE
```
/// SPI Initialization routine
/// Note: Port A and B must be enabled outside of this routine
/// (resetting and enabilng clock).
void SPI_init(void) {
        // Reset SPI Peripheral
        RCC->APB2RSTR |= RCC_APB2RSTR_SPI1RST; // Reset bit to 1
        RCC->APB2RSTR &= ~((uint32_t) RCC_APB2RSTR_SPI1RST ); //Reset bit to
         ↪  zero.
        // Enable the SPI Peripheral clock
        RCC->APB2ENR |= RCC_APB2ENR_SPI1EN;
        // Reset CR1 and CR2 to zero (to be sure, but they are the defaults at
         ↪  reset)
```

Figure 2. Block diagram



1. This block diagram has not be used to estimate propagation delays.

Figure 6.4: 74HC595 Block diagram

```
        SPI1->CR1=0;
        SPI1->CR2=0;
```

Then we will follow the procedure step-by-step.

- Select the BR[2:0] bits to define the serial clock baud rate (see SPI_CR1 register). For our application we do not need a particular baud rate, so we select 111 (the slowest) for our tests.

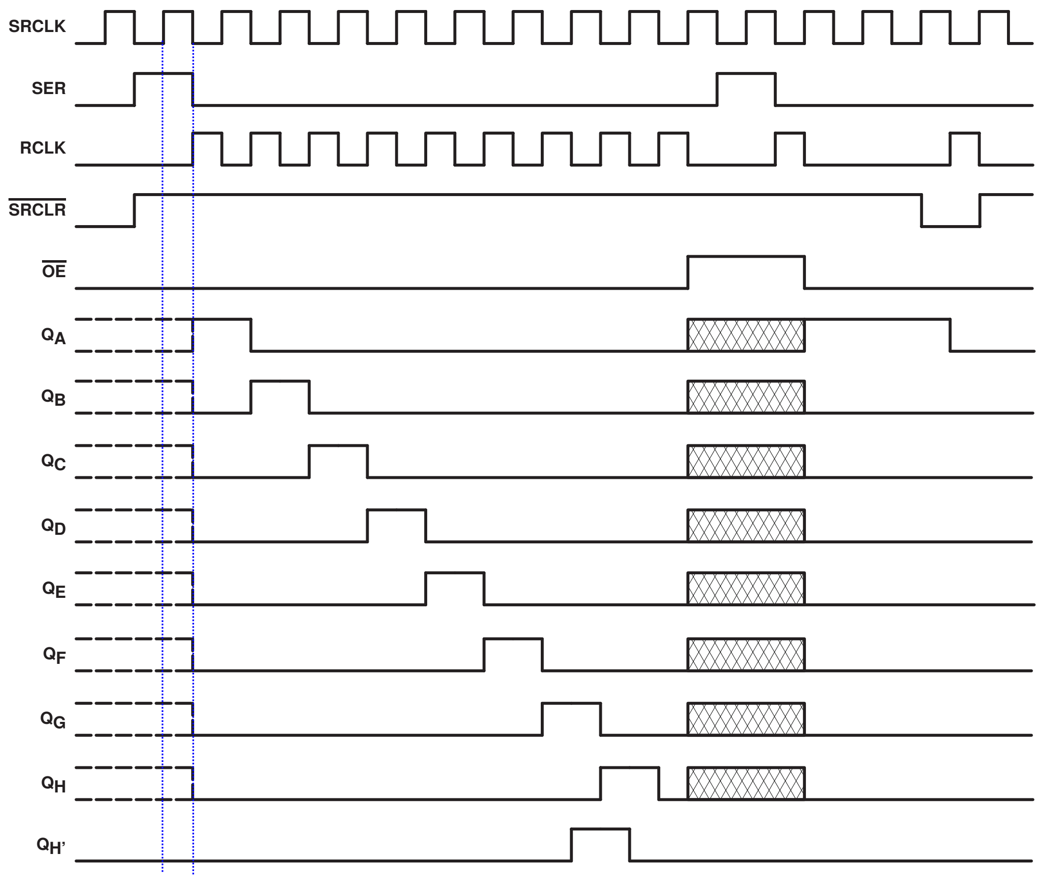
CODE
```
        // Peripheral initialization, as described in the
        // User manual, par. 20.3.3; Configuring the SPI in master mode
        // 1) Baud rate selection: 111=fpp/256
        SPI1->CR1|=SPI_CR1_BR; // 111 for baud rate selection.
```

- Select the CPOL and CPHA bits to define one of the four relationships between the data transfer and the serial clock (see Figure 6.6). This step is not required when the TI mode is selected.

CODE
```
        // 2) CPOL and CPHA
        SPI1->CR1|=SPI_CR1_CPOL;
        SPI1->CR1|=SPI_CR1_CPHA;
```

- Set the DFF bit to define 8- or 16-bit data frame format (We do not need it since it is yet zero since we cleared the whole control registers CR1 and CR2 before.

CODE
```
        // 3) DFF yet to zero
```



NOTE: ▨ implies that the output is in 3-State mode.

Figure 1. Timing Diagram

Figure 6.5: 74HC595 Timing diagram

- Configure the LSBFIRST bit in the SPI_CR1 register to define the frame format. This step is not required when the TI mode is selected. We will not use TI mode and the bit is yet zero since we cleared CR1 and CR2.

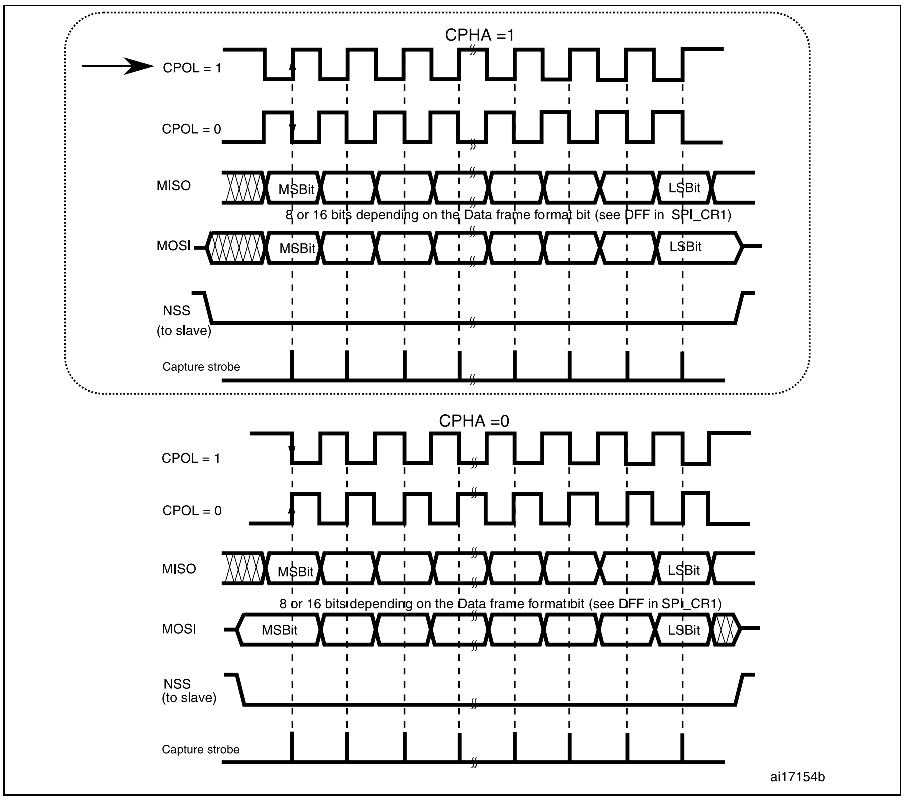
CODE
```
        // 4) LSBFIRST bit yet to zero.
```

- If the NSS pin is required in input mode, in hardware mode, connect the NSS pin to a high-level signal during the complete byte transmit sequence. In NSS software mode, set the SSM and SSI bits in the SPI_CR1 register. If the NSS pin is required in output mode, the SSOE bit only should be set. This step is not required when the TI mode is selected.

CODE
```
        // 5) set SSM and SSI.
        SPI1->CR1|=SPI_CR1_SSM;
        SPI1->CR1|=SPI_CR1_SSI;
```

- Set the FRF bit in SPI_CR2 to select the TI protocol for serial communications.

CODE
```
        // 6) FRF yet zero.
```



1. These timings are shown with the LSBFIRST bit reset in the SPI_CR1 register.

Figure 6.6: SPI timings of the STM32F401RE SPI

- The MSTR and SPE bits must be set (they remain set only if the NSS pin is connected to a high-level signal).

```
        // 7) Set MSTR bit and SPE bit
        SPI1->CR1|=SPI_CR1_MSTR;
        SPI1->CR1|=SPI_CR1_SPE;
```

This procedure is also not enough to configure the I/O pins to work as SPI, so we will need to set the MODER register to the alternate function and select the correct alternate function. From the schematics of the PMDB16 board (141414) and the alternate function table of the datasheet we have the following table:

Function	Pin
MISO	PA6
MOSI	PA7
SCK	PA5
SS/STROBE	PB6

So we will initialize the GPIOs accordingly.

```
        // Initialize GPIOs for SPI
        GPIOA_b->MODER.MODER5 = MODER_ALTERNATE_FUNCTION;
        GPIOA_b->MODER.MODER6 = MODER_ALTERNATE_FUNCTION;
        GPIOA_b->MODER.MODER7 = MODER_ALTERNATE_FUNCTION;
        GPIOA_b->OTYPER.OT5 = OTYPE_PUSH_PULL;
    // OT6 is the MISO input, no need to set OTYPER.
    GPIOA_b->OTYPER.OT7 = OTYPE_PUSH_PULL;
        GPIOA_b->OSPEEDR.OSPEEDR5 = OSPEED_HIGH;
    GPIOA_b->OSPEEDR.OSPEEDR6 = OSPEED_HIGH;
        GPIOA_b->OSPEEDR.OSPEEDR7 = OSPEED_HIGH;
```

To set correctly the AFR (Alternate Function Register), we should look at the reference manual (for the meaning of the bits of the register AFRL) and at the data-sheet [4] to see the correct value for the SPI Alternate function (Table 9: Alternate Function Mapping).

CODE
```
        GPIOA_b->AFRL.AFRL5=0x05; //5=SPI1 alternate function
        GPIOA_b->AFRL.AFRL6=0x05; //5=SPI1 alternate function
        GPIOA_b->AFRL.AFRL7=0x05; //5=SPI1 alternate function
```

Then we configure GPIOB as output, since we use it as a strobe signal for the shift registers.

CODE
```
        // Configure GPIOB pin 6 as output, used as strobe signal
        // for the shift registers.
        GPIOB_b->MODER.MODER6= MODER_GPOUTPUT; // Configure GPIOB pin 6 in max
         ↪   speed
        GPIOB_b->OSPEEDR.OSPEEDR6=OSPEED_HIGH;
        GPIOB_b->OTYPER.OT6=OTYPE_PUSH_PULL;
}
```

6.2.1.1 Debugging the configuration and first transmission test

To transmit and receive the data from the spi chain, we just need to write a byte in the transmission register and read the received data from the same register once the communication ended. In our case we need only to transmit, but we are forced anyway to read the received data to avoid a peripheral read overrun error, so we first read the byte then we transmit the new one.

To be sure not to overwrite a running transmission we will check that the previous one ended before starting a new one.

....

debug... breakpoint at every data write.

...See the clock led blink (few light, since it is a fast blink at the clock frequency) once for every transmission.

6.2.2 Sending two bytes and the strobe signal

6.2.3 Column scanning

Routine

Debug

6.2.4 Row data and image on the matrix

Routine

Debug

6.2.5 Moving to a separate routine (that can be called from a timer interrupt routine)

Reorganizing code, no delays into the routine.

6.3 Hands-on Experiment 8: *TODO* Char display on encoder move

Use the incremental encoder to select different chars shown on the led array.

7

Hands-on: ADC

7.1 Used hardware

On the PMDB16 board there are two analog sources: one is a potentiometer, connected as a voltage divider between the 3V3 power and ground and the second is a LDR (photoresistor), also connected to a resistor to form another voltage divider (see figure 7.1 on the following page). Two resistors (R93 and R77) are used to limit the GPIO pin current in the case the GPIOpins are misconfigured.

Before being able to use those input we must take a look at the schematics to know where those analog signals are connected and which GPIO pins we will need to configure.

7.1.1 Reading the schematics to find GPIO lines

Starting from the "Analog In" schematic sheet we see the source for the analog signals (see figure 7.1 on the next page). They are connected to two "off-page" connectors that show that those signals are connected also on some other page of the schematics.

In the master sheet of the design schematic the Analog In sheet is connected to the "Development board Connectors" sheet (see figure 7.2 on page 65), where the two analog lines change their reference from ADC_LDR and ADC_POT to LDR and POT.

In the "Development Board Connections" page of the schematic (fig. 7.3 on page 66) those signals are connected to *PA0 and PA1* lines of the Nucleo board.

At the end of the schematic analysis we have that

Potentiometer handle rotation affects the voltage on **PA1** pin

Light on the photoresistor R40 affects the voltage of the **PA0** pin.

7.2 Hands-on Experiment 9: ADC READ: Potentiometer input

7.2.1 Short ADC peripheral description

The section 11 of the Reference manual [3] describes the ADC peripheral, with its main features:

- 12-bit successive approximation analog-to-digital converter configurable as 12-bit, 10-bit, 8-bit or 6-bit resolution ADC

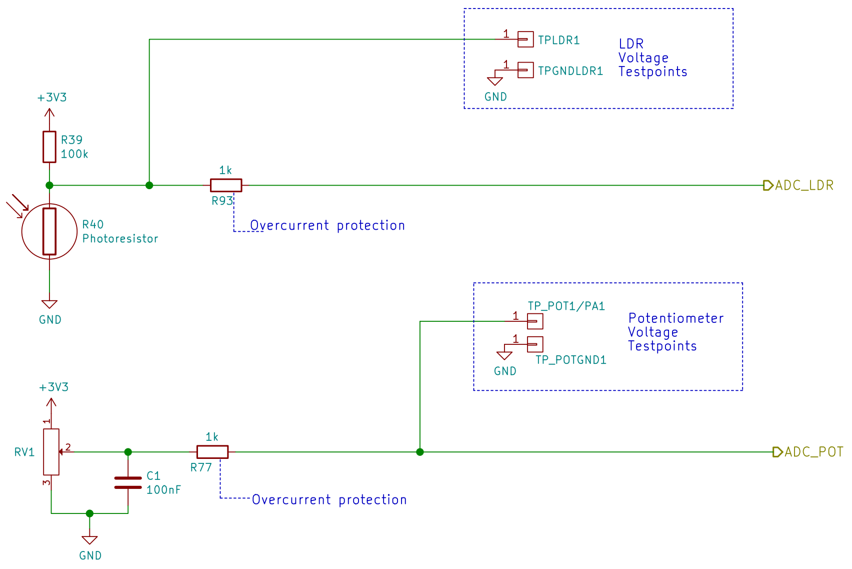


Figure 7.1: Analog sources on PMDB16 board (Analog In sheet).

- 16 external multiplexed analog inputs
- 3 internal multiplexed analog channels (temperature sensor, Vbat, Temperature)
- Interrupt generation at the end of conversion, end of injected conversion, and in case of analog watchdog or overrun events
- Single and continuous conversion modes
- Scan mode for automatic conversion of channel 0 to channel ‘n’
- Data alignment with in-built data coherency
- Channel-wise programmable sampling time
- External trigger option with configurable polarity for both regular and injected conversions
- Discontinuous mode
- DMA request generation during regular channel conversion

As first experiment with ADC we will configure it in the simplest way: we will do a single conversion on a single channel.

Looking at the device data-sheet [4], at “Table 8. STM32F401xD/xE pin definitions” we find that the pins PA0 and PA1 have as “Additional Functions” :

PA0: ADC1_IN0

PA1: ADC1_IN1

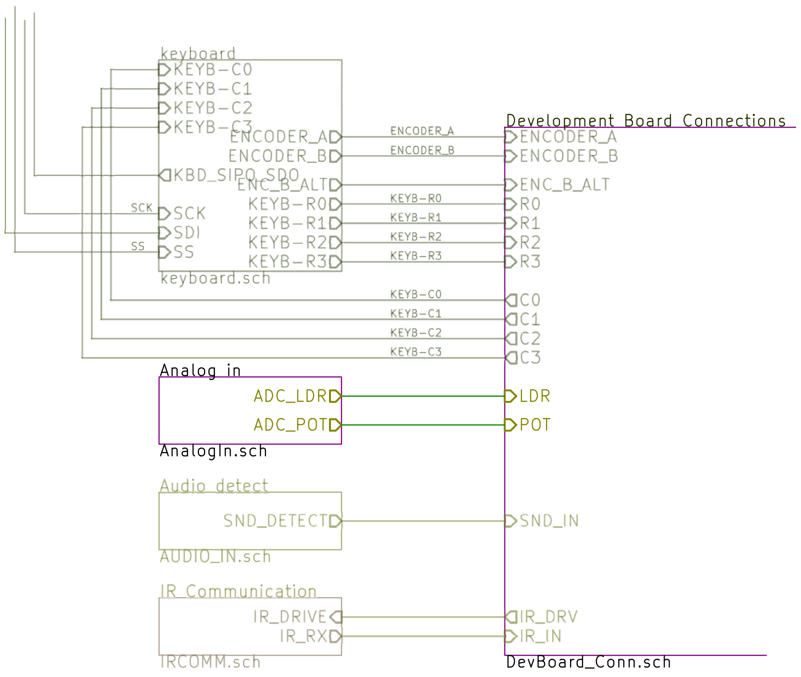


Figure 7.2: Analog connections between different sheets.

7.2.2 GPIO configuration and ADC setup

So we need to configure PA1 pin as ADC_IN1, setting the MODER register accordingly. Also we will set up PA0 in the same way (we will use it later)

CODE
```
// Reset and enable clock to GPIOA
RCC_b->AHB1RSTR.GPIOARST=1; //Reset peripheral
RCC_b->AHB1RSTR.GPIOARST=0; //Leave reset
RCC_b->AHB1ENR.GPIOAEN=1; //Enable the GPIO A peripheral clock.
// Configure analog input pins.
GPIOAb->MODER.MODER0=MODER_ANALOG; //PA0 pin as analog        (LDR)
GPIOAb->MODER.MODER1=MODER_ANALOG; //PA1 pin as analog.     (Potentiometer)
```

or, if you have not re-defined RCC_b and GPIO:b as the bitfield-oriented structures

CODE
```
RCC->AHB1RSTR|=RCC_AHB1RSTR_GPIOARST; //Reset peripheral
RCC->AHB1RSTR&=~RCC_AHB1RSTR_GPIOARST;//Leave reset
RCC->AHB1ENR|=RCC_AHB1ENR_GPIOAEN; //Enable the GPIO A peripheral clock.
GPIOA->MODER |= ((GPIO_MODER_MODER0) | (GPIO_MODER_MODER1)); //PA0 and PA1
 ↪  analog inputs
```

Since the pins are in analog mode, we do not need to set output type and pin speed and other output features.

As usual, to have more readable code we put those instructions in a routine.

CODE
```
/// GPIOA 0 and 1 setup as analog input.
void GPIOs_ADC_Setup(void)
{// Initialize port A pins A0 and A1 as ANALOG INPUTS.
        RCC_b->AHB1RSTR.GPIOARST=1; //Reset GPIOA peripheral
        RCC_b->AHB1RSTR.GPIOARST=0; //Leave Reset.
```

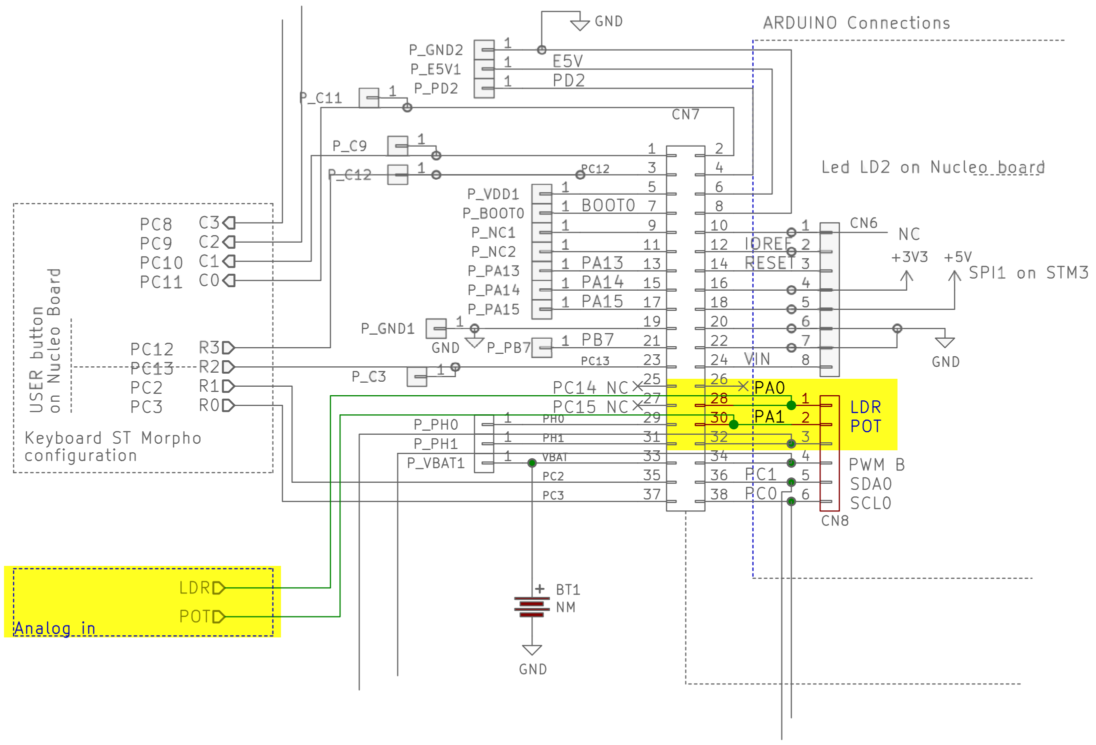


Figure 7.3: Analog connections to the Nucleo board.

```
        RCC_b->AHB1ENR.GPIOAEN=1; //Enable the GPIO A peripheral clock.
        GPIOAb->MODER.MODER0=MODER_ANALOG; //GPIOA PA0 analog (LDR)
        GPIOAb->MODER.MODER1=MODER_ANALOG; //GPIOA PA1 analog (potentiometer)
        GPIOA->MODER |= ((GPIO_MODER_MODER0) | (GPIO_MODER_MODER1));
}
```

As with any other peripheral we need to reset it, connect the clock to the peripheral and set it up for the single ADC conversion. Note that here we do not redefine our structures since ADC1 registers do not have byte-size write access and we would not be able to write them correctly.

CODE
```
// RESET AND CLOCK for ADC1
RCC_b->APB2RSTR.ADC1RST=1; //reset
RCC_b->APB2RSTR.ADC1RST=0; //exit reset
RCC_b->APB2ENR.ADC1EN=1;   //enable ADC clock
```

In the Reference manual [3], in the 11.3.4 section there is the description of the single conversion mode, while in the 11.12 section there is the description of the ADC registers.

First we need to turn the ADC peripheral ON.

CODE
```
ADC1->CR2 = ADC_CR2_ADON;//=1; //ADC_CR2 ADON=1 to turn ADC on.. anything else
↪  to zero (single conversion)
```

The other bits of the ADC Control Register 2 (ADC_CR2) [sec. 11.12.3, page 231 in the reference manual) are all set to zero by this instruction, so this instruction configures the ADC_CR2 register in the following way:

SWSTART=0: Don't start any conversion now

EXTEN=0: External trigger disabled

EXTSEL[3:0]=000 Select channel 0 for external trigger (don’t care since it is disabled)

JSWSTART=0 Don’t start injected channel conversion (we don’t use them now)

JEXTEN=0 Don’t use external trigger for injected channel (we don’t use them now)

JEXTSEL[3:0]=0000 External event for injected group. (Don’t care since it is disabled)

ALIGN=0 RIGHT aligned adc conversion result (we want a 12 bit result from 0 to 4095, so right-align is our choice)

EOCS=0 Overrun detection disabled (since we do not use DMAs here)

DDS=0 No DMA

CONT=0 No continuous conversion (we want a single conversion now)

ADON=1 ADC peripheral turned ON.

Then we configure the other ADC control register (sec. 11.12.2, page 231 in the reference manual)

CODE
```
ADC1->CR1 = 0; //channel 0 for analog watchdog., all others to zero (default).
```

This configuration for the bits in the register mean:

OVRIE=0 No overrun interrupt enable (we don’t use IRQs in this experiment)

RES[1:0]=00 12 bit resolution for the conversion

AWDEN=0 No Analog watchdog function used

JAWDEN=0 No Analog watchdog function on injected channels.

DISCNUM[2:0]=000 Discontinuous mode channels count=0 (don’t care since we are not using discontinuous conversion (see JDISCEN bit)

JDISCEN=0 Discontinuous mode on injected channels disabled

DISCEN=0 Discontinuous mode disabled for regualr channels

JAUTO=0 Automatic injected group conversion disabled

AWDSGL=0 No analog watchdog on single channel

SCAN=0 No scan mode

JEOCIE=0 No interrupt for injected channels end of conversion (we don’t use IRQs in this experiment)

AWDIE=0 No Analog Watchdog interrupt enable (we don’t use IRQs in this experiment)

EOCIE=0 End of conversion interrupt disabled (we don’t use IRQs in this experiment)

AWDCH[4:0]=0000 Analog channel watchdog select bits. Don't care since we do not use this feature now.

Now we set up the sampling time for every channel. See Reference manual sections 11.12.4 and 11.12.4 ADC for sample time register 1 and 2

To set channels 0 and 1 sample time we need to set ADC1_SMPR2 bits 0-2 for channel 0 (LDR) and bits 3-5 for analog input 1 (potentiometer)

```c
ADC1->SMPR1 = 0; //Sampling time: 3 cycles for all (not needed since we will
 ↪  use only channels 0 and 1.
ADC1->SMPR2 = 0b111111; //Sampling time, 3 bits for every channel.
```

At this point we have the following code for ADC setup .

```c
/// ADC peripheral setup.
void ADC_Setup(void)
{  // Reset and set clock to ADC1 Peripheral.
   //RCC->APB2RSTR |= RCC_APB2RSTR_ADCRST; //Reset the ADC peripheral,
    ↪  standard way
   RCC_b->APB2RSTR.ADC1RST = 1;
   //RCCAPB2RSTR &= ~(RCC_APB2RSTR_ADCRST); //Leave reset for ADC peripheral,
    ↪  standard way.
   RCC_b->APB2RSTR.ADC1RST = 0;
   //RCC->APB2ENR |= RCC_APB2ENR_ADC1EN; //Enable ADC1, standard way.
   RCC_b->APB2ENR.ADC1EN = 1; //Initialize ADC1
   ADC1->CR2 = ADC_CR2_ADON;// ADON=1 to turn ADC on.. anything else to zero
    ↪  (single conversion)
   ADC1->CR1 = 0; //channel 0 for analog watchdog., all others to zero
    ↪  (default).
   ADC1->SMPR1 = 0; //Sampling time: 3 cycles for all (not needed since we
    ↪  will use only channels 0 and 1.
   ADC1->SMPR2 = 0b111111; //Sampling time, 3 bits for every channel. longest
    ↪  sample time for ch0 and 1
}
```

7.2.3 ADC single acquisition channel select

To select which channel to convert we need to set the ADC_SQR3 register, just like if a single conversion was a "regular sequence of one channel".

```c
ADC1->SQR3 = 1; //1 for Analog input 1 (PA1, potentiometer input)
```

Defining a more handful routine:

```c
/// Analog channel selection. channel must be between 0 and 18
void set_ADC_Channel(unsigned char channel)
{  ADC1->SQR3 = channel;
}
```

7.2.4 Starting a conversion and reading it

To start a conversion we need to set the bit SWSTART in the ADC control register 2 ADC1_CR2, so we create a function (or a macro) to ease the readability of the code

CODE
```
#define ConversionStart() ADC1->CR2|=ADC_CR2_SWSTART
```

To detect the end of conversion we must look at the ADC Status register ADC1_SR and the EOC bit, so we define a macro to check if the conversion ended.

CODE
```
#define CONVERSION_NOT_ENDED ((ADC1->SR & ADC_SR_EOC) == 0)
```

Now we can write a main reading the ADC.

CODE
```
int main(void)
{   unsigned int ADCReadValue1=0; //ADC conversion result.
        GPIOs_ADC_Setup();
        ADC_Setup();
        set_ADC_Channel(1);//Set channel to convert.
        for (;;) //main loop
        {   ConversionStart();
                while(CONVERSION_NOT_ENDED); //wait conversion end
                ADCReadValue1 = ADC1->DR; // Read analog input value..
        } //main loop
} //main
```

Now we can compile and debug. Put a breakpoint at the Conversion start and look at the "ADCRead-Value1" variable (if the "variables" view is not present in your debug prespective, enable it once the debug perspective is enable using Window->Show view->variables)

7.3 Hands-on Experiment 10: ADC READ LDR AND Potentiometer input

Here we will modify the code of the previous example in order to acquire both the channels. Since we yet initialized also the LDR input with those routines, we jus tneed to change the code in the main function to acquire both the channels. First we will create a couple of variables to hold the new data and to hold the used channel.

To write a routine that scans both channels:

- Add a new variable ADCReadValue0 to store the conversion for channel 0
- Add a new variable (unsigned char) to hold the current channel selected
- In the main loop add a switch that, when the conversion ends, reads the conversion results, sets the new channel to be converted and starts the new conversion.

To check for conversion end we add a define to check the end of conversion condition

CODE
```
#define CONVERSION_ENDED ((ADC1->SR & ADC_SR_EOC) != 0)
```

The new main function reading both channels will be:

CODE
```
int main(void)
{   unsigned int ADCReadValue0=0; //ADC conversion result 0.
        unsigned int ADCReadValue1=0; //ADC conversion result 1.
    unsigned char channel=0; // used channel.
        GPIOs_Setup();
    ADC_Setup();
        set_ADC_Channel(channel); //Set channel to convert.
    ConversionStart(); //start the first conversion.
    for (;;) //main loop
    {  if (CONVERSION_ENDED)
       {  switch (channel)
          { case 0: ADCReadValue0= ADC1->DR; // Read analog input value (LDR).
                    channel=1; //set the next channel number.
                    break;
            case 1: ADCReadValue1= ADC1->DR; // Read analog input value for
             ↪  (Potentiometer).
                    channel=0; //Set channel for the next read.
                    break;
            default: //should never arrive here, but we code "defensively"
                    channel=0;
                    break;
          } //switch.
          set_ADC_Channel(channel); //Set the new channel
          ConversionStart();
       }
     } //main loop
} //main
```

Debug the program and watch the content of ADCReadValue0 and ADCReadValue1 changing when LDR light changes and potentiometer handle is rotated.

8

Hands-on: Interrupts

8.1 Hands-on Experiment 11: Timer interrupt on fixed time (led flashing on sysTick timer)

In this experiment we will start from the "led blink" that we used in chapter 3 and we will change it to blink the led in an interrupt routine driven by the sysTick timer.

8.1.1 The SysTick timer

The sysTick timer is a "core peripheral", present on every arm cortex M processor, so it is described in the "Programming manual"[3] and not in the "Reference manual" where "external" peripherals are described.

The SysTick timer is a 24 bit down-counter timer with auto-reload function. The auto-reload function loads automatically into the counter register (STK_VAL) the value stored in the STK_LOAD register any time the timer underflows and the reload feature can be used to set the time between the underflows (timer period). This timer can be used to trigger an interrupt every time it underflows. Also a flag named COUNT_FLAG can be used to detect if the timer did count to zero since last time that flat was read, in order to use the timer in polling mode, without using the interrupt functions.

The clock signal for this timer can be chosen between AHB clock (processor clock) or AHB clock /8. In figure 8.1 there is a schematic representation of the clock subsystem for ARM cortex M4 cores.

Note that when the processor is halted for debugging the counter does not decrement.

8.1.2 The interrupt vector table and the weak aliases.

In section 3.1 we have seen that between the generated files there was an assembler file named

```
/startup/startup_stm32F401xx.s
```

In this file we can find the real startup code for the controller (the one that initializes some hardware as the memory controller, initializes the libraries and calls the "main()" function).

Among all the other things, in this file it is defined the interrupt vector table, as

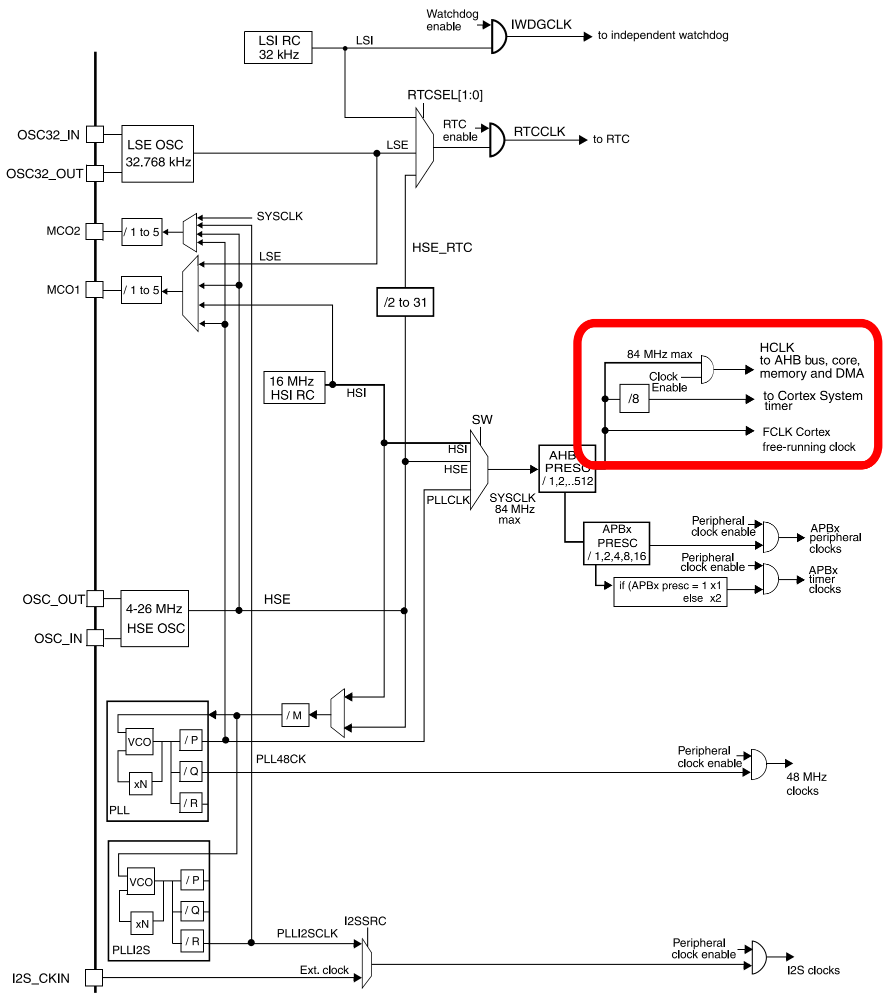


Figure 8.1: ARM Cortex-M4 clock tree

```
.section .isr_vector,"a",%progbits
.type g_pfnVectors, %object
.size g_pfnVectors, .-g_pfnVectors g_pfnVectors:
.word _estack
.word Reset_Handler
.word NMI_Handler
.word HardFault_Handler
.word MemManage_Handler
.word BusFault_Handler
.word UsageFault_Handler
.word  0
.word  0
.word  0
.word  0
.word  SVC_Handler
.word  DebugMon_Handler
```

```
.word  0
.word  PendSV_Handler
.word  SysTick_Handler
......
```

This vector table is located at the very beginning of the FLASH memory, defined in the linker script LinkerScript.ld[1]. Since at the power-on the vector table is located by default at address 0x00000000, the SystemInit() function in the file system_stm32F4xx sets the register SCB->VTOR (the vector table Offset Register) to that address (FLASH_BASE | VECT_TABLE_OFFSET)[2]. The final effect is that in the vector table there will be at the reset handler position, the the address of the routine named "Reset_Handler", at the position of System Tick interrupt handler the address of the routine named SysTick_Handler, etc.

In the programming manual, section 2.3.4, [Figure 11. Vector table] there is a description of the vector table and the vector addresses, where the sequence of vectors is reported (bottom-up). The same table is also available in the reference manual, table 38, section 10.2: External interrupt/event controller (EXTI).

Moreover, in the same assembler file "startup_stm32F401xx.s" we find also a lot of "weak defines" for those functions, so that if a function with the name specified in the vector table is not defined, it defaults to a known function. For the system tick handler (and other IRQ handling routines) we have:

```
.weak SysTick_Handler
.thumb_set SysTick_Handler,Default_Handler
```

That means that if we do not define a SysTick interrupt handler routine with name "SysTick_Handler", the "Default_handler" routine will be called instead.

The "Default_Handler" routine is defined in the same file as

```
.section .text.Default_Handler,"ax",%progbits
Default_Handler:
Infinite_Loop:
     b Infinite_Loop
     .size Default_Handler, .-Default_Handler
```

and it is just a never-ending loop.

This means that *if we enable an interrupt without writing the corresponding interrupt handling routine*, when this irq is triggered our program *will jump to this infinite loop and stay there*, since only a higher level interrupt can stop (temporarily) this loop. A common mistake is to mis-type the name of the handler and while debugging see the program looping forever in the "Default_Handler" loop.

Conclusion: Looking at all those files we found out that to write the System tick timer interrupt handling routine we just need to write a routine with the correct name:

```
SysTick_Handler().
```

[1]around line 62 for memory areas definition and about at line 72 for the .isr_vector section

[2]lines around 517-520, or search for SCB->VTOR in that file

8.1.3 SysTick timer initialization.

The CMSIS-CORE framework gives us a routine to initialize the system tick timer and enable the relative interrupt.

To see the definitions of the core functions provided, see (in your project folder) CMSIS->core->core_cm4.h

Inside that file we find a comment with:

CODE
```
/*************************************************************************
↪  **
 *                Hardware Abstraction Layer
  Core Function Interface contains:
  - Core NVIC Functions
  - Core SysTick Functions
  - Core Debug Functions
  - Core Register Access Functions
*************************************************************************/
```

Then we see the function definition and description:

CODE
```
/** \brief System Tick Configuration
The function initializes the System Timer and its interrupt,
and starts the System Tick Timer.
Counter is in free running mode to generate periodic interrupts.
\param [in] ticks Number of ticks between two interrupts.
\return 0 Function succeeded. \return 1 Function failed.
\note When the variable <b>__Vendor_SysTickConfig</b> is set to 1,
then the function <b>SysTick_Config</b> is not included. In this case,
the file <b><i>device</i>.h</b> must contain a vendor-specific
implementation of this function.
*/ __STATIC_INLINE uint32_t SysTick_Config(uint32_t ticks) {
...
```

So, to initialize the timer to generate an interrupt periodically (namelly 1000 IRQs per second) we should use[3]:

CODE
```
uint32_t returnCode;
returnCode = SysTick_Config(SystemCoreClock/1000);
```

Enabling an interrupt without defining an interrupt routine enables the default interrupt handler routine, which is an infinite loop where the program will hang, so now we need to create such routine (SysTick_Handler).

To have a led blink every second we will count interrupt events and every 1000 counts we will toggle the led status.

CODE
```
/// Systick interrupt handler.
void SysTick_Handler(void)
{   static int count=COUNT_RELOAD;
        if (count>0)
```

[3]NOTE: remember to set correctly the PLL configuration register to use internal oscillator. See section 3.1.1 for correct initialization of the clock subsystem.
Hopefully this will be fixed in future version of the development environment.

```
                count--;
        else
        {   count=COUNT_RELOAD;
                if (Led_is_OFF)
                        Led_ON();
                else
                        Led_OFF();
        }
}
```

Before this code (yet "beautified" and made very human readable) we should define some macros and symbols.

CODE
```
/// Turns led ON
#define Led_ON() GPIOA->BSRRL = GPIO_BSRR_BS_5
/// Turns led OFF
#define Led_OFF() GPIOA->BSRRH = (GPIO_BSRR_BR_5 >> 16)
/// TRUE if led is OFF
#define Led_is_OFF (GPIOA_b->IDR.IDR5==0)
/// Number of sysTicks before led toggling.
#define COUNT_RELOAD 1000;
```

Remember to remove from the main(), in the main loop, the code that toggles the led if you're starting from the "led blink" project we did on section 3.2.

Compile and test. To change the blinking period try changing COUNT_RELOAD (example: decreasing it (i.e. to 200)).

8.1.4 Alternate Systick initalization procedure

The same initialization can be done without using the CMSIS routines, writing directly the registers. This way is more complex, but gives more control. As an example this way you will set directly the interrupt priority level and choose if the interrupt has to be enabled or not. See the programming manual about the meaning of the registers involved.

CODE
```
        SysTick->LOAD=(uint32_t)(SystemCoreClock/1000-1UL);
        // Set irq priority
        SCB->SHP[11]=0xF0;
        //Load the timer with 1 value so that very soon it will underflow and
         ↪ reload the value we've chosen.
        SysTick->VAL=1;
        // Enable SysTick interrupt
        SysTick->CTRL = SysTick_CTRL_CLKSOURCE_Msk | //CLKSOURCE to 1:
         ↪ Processor clock (AHB). (bit 2)
                        SysTick_CTRL_TICKINT_Msk   | // TICKINT to 1: enable
                         ↪ clock interrupt (systick irq) (bit 1)
                        SysTick_CTRL_ENABLE_Msk;     // Enable the systick
                         ↪ timer (bit 0).
```

8.2 Hands-on Experiment 12: Systick interrupt driven keyboard scan

In this experiment we will make the keyboard scan routine so that no “ugly” delays will be used, starting from experiment 5.2.

- Initialize the SysTick timer as in the previous experiment
- Use the same routine described in the experiment 5.2 for the keyboard scan
- Remove from the main loop the keyboardScan call
- Put the keyboardScan call inside the systick timer interrupt routine in order to call it every 10ms or 5ms or 1ms. Choose the best solution.

8.3 Hands-on Experiment 13: **work-in-progress** Interrupt driven quadrature encoder decoding

In this experiment we will start from the quadrature encoder decoding software we developed in section 5.1, enabling the interrupt on change for the two lines encoder_A and encoder_B, thus calling automatically the decoding routine.

8.3.1 External interrupts

Interrupt generation by GPIOs is handled by the External interrupt/event controller (EXTI) (Section 10.2 of the reference manual [3]), that states:

> “The external interrupt/event controller consists of up to 23 edge detectors for generating event/interrupt requests. Each input line can be independently configured to select the type (interrupt or event) and the corresponding trigger event (rising or falling or both). Each line can also masked independently. A pending register maintains the status line of the interrupt requests.”

In the same section we have also, in table 38, the vector table, where we can find the different external interrupt sources. Between them are the following EXTI interrupts.

Interrupt	IRQ number	Description
EXTI0	6	EXTI Line0 interrupt
EXTI1	7	EXTI Line1 interrupt
EXTI2	8	EXTI Line2 interrupt
EXTI3	9	EXTI Line3 interrupt
EXTI4	10	EXTI Line4 interrupt
EXTI9_5	23	EXTI Line[9:5] interrupts
EXTI16 / PVD	1	EXTI Line 16 interrupt / PVD through EXTI line detection interrupt
EXTI17 / RTC_Alarm	41	EXTI Line 17 interrupt / RTC Alarms (A and B) through EXTI line interrupt
EXTI18 / OTG_FS WKUP	42	EXTI Line 18 interrupt / USB On-The-Go FS Wakeup through EXTI line interrupt
EXTI21 / TAMP_STAMP	3	EXTI Line 21 interrupt / Tamper and TimeStamp interrupts through the EXTI line

To enable a generic external interrupt, we need to program the trigger registers with the desired edge detection and enable the relative interrupt writing the corresponding interrupt enable register.

8.3.2 PC6 and PC6 external interrupt setup

To initialize and enable the interrupts on lines PC6 and PC7 we will have to:

- Enable the clock to the SYSCFG peripheral
- Set PC6 and PC7 as external interrupt sources
- Enable interrupt lines EXTI 6 and 7 (set the interrupt mask bit)
- Enable both rising and falling edge on lines EXTI 6 and 7
- Enable the interrupt for lines 9:5 in the NVIC

First we have to enable the clock to the SYSCFG peripheral, in order to set up the external interrupts. This can be done using the functions provided in stm32F4xx_rcc.h (See manual for APB2Periph register)

CODE
```
RCC_APB2PeriphClockCmd(RCC_APB2Periph_SYSCFG, ENABLE);
```

Or directly writing the register APB2ENR[4].

CODE
```
RCC->APB2ENR |= RCC_APB2Periph_SYSCFG;
```

Setting PC6 and PC7 as external interrupt lines

CODE
```
//EXTICR: set PC6 and PC6 as external interrupt lines.
↪  SYSCFG->EXTICR[1]|=(SYSCFG_EXTICR2_EXTI6_PC|SYSCFG_EXTICR2_EXTI7_PC);
```

[4]As usual, see the reference manual on the APB2ENR register definition

Or, Using library functions defined in stm32f4xx_syscfg.c

```
//SYSCFG_EXTILineConfig(EXTI_PortSourceGPIOC, EXTI_PinSource6);
    //SYSCFG_EXTILineConfig(EXTI_PortSourceGPIOC, EXTI_PinSource7);
```

Setting to 1 the interrupt mask bits for EXTI lines 6 and 7.
   Using library function:

```
        EXTI_IMR (EXTI_IMR_MR6|EXTI_IMR_MR7);
```

or directly writing the register EXTI_IMR

```
        EXTI->IMR|=EXTI_IMR_MR6|EXTI_IMR_MR7; //unmask irqs from lines 6 and 7.
```

Enable rising AND falling edge

```
        // Enable edge: rising on lines 6 and 7
        EXTI->RTSR|=(EXTI_RTSR_TR6|EXTI_RTSR_TR7);
        // Enablelle also Falling edge on the same lines.
        EXTI->FTSR|=(EXTI_FTSR_TR6|EXTI_FTSR_TR7);
```

Enable the EXTI_9_5 interrupt

```
// Enable interrupt (see programming manual sec. 4.3 for provided functions).
NVIC_EnableIRQ(EXTI9_5_IRQn);
```

or, setting the register directly

```
// EXTI9_5_IRQn is defined as 23, (table 38 in the reference manual or see .h
    file) so
// we have to set bit 23 register ISER 0
// See structure NVC in stm32F4xx.h.
        NVIC->ISER[0]|=(1<<(EXTI9_5_IRQn)); // Set bit 23 of register
            NVIC_ISER0
```

Now the interrupts on those two lines are enabled. we just have to remove the decodeQuadratureEncoder(); from the main loop and write a new interrupt routine. The name of the routine should be EXT9_5_IRQHandler (see startup_stm32f401xx.s file for the interrupt vector definition and the names assigned to the interrupt routines.

```
void EXTI9_5_IRQHandler(void)
{
        decodeQuadratureEncoder();
}
```

Now compile and debug. Watch the variable encoder.position change while the axis is rotated.

8.4 Hands-on Experiment 14: *TODO* Interrupt driven ledMatrix scan (timer)

Use a timer-driven routine to scan at a fixed rate the led matrix. Tip: use the routine defined in chapter 6

8.5 Hands-on Experiment 15: *TODO* Keyboard+LedMatrix: Ghost keypress problem.

Use a submatrix 4x4 in the led array to show wich keys are pressed and to see “ghost key” problem.

8.6 Hands-on Experiment 16: *TODO* Interrupt on GPIO: Sound peak input

Hardware used, Schematic ecc...

Code.

8.7 Hands-on Experiment 17: *WORK-IN-PROGRESS*: ADC with EOC IRQ

In this section we want to read an analog signal, convert it and put the converted value in a global variable as soon the conversion ends, without having to poll continuously the end-of conversion bit in the status register. Then we want to switch to the second analog source of the board and do the same for that one.

To obtain this behaviour we will use the “End Of Conversion” ADC interrupt and doing nothing in the main loop of the program (leaving room for other functions there). We will need to write an interrupt handling routine that

- reads the current ADC conversion value
- switches to the next channel to acquire
- starts the next conversion.

We will save the conversion values in the global variables ADCReadValue0 and ADCReadValue1.

The ADC IRQ handler routine needs to be called “void ADC_IRQHandler (void)” (see the vector table defined in the file “startup_stm32F401.s“, in the startup folder of the project, around lines 170 and 326).

The interrupt routine will implement a state machine that, depending on the current channel sets one or another variable, switches to the next channel and starts the new conversion. As status of our state machine we will use the “channel” variable, declared static inside the interrupt routine.

CODE
```
unsigned int ADCReadValue0;
unsigned int ADCReadValue1;
void ADC_IRQHandler (void)
{  static unsigned char channel=0;
   switch (channel)
   {
    case 0: //conversion of channel 0 ended.
      ADCReadValue0 = ADC1->DR;
      channel = 1;
      set_ADC_Channel(channel);
      break;
    case 1: //conversion of channel 1 ended.
      ADCReadValue1 = ADC1->DR;
      channel = 0;
      set_ADC_Channel(channel);
      break;
    default: //should never arrive here.
      channel = 0;
    } //switch
       ConversionStart(); //Start next conversion
}//ADC_IRQ_Handler
```

Then before entering the main loop we need to:

- Configure the GPIOs (just the same as the function we used for ADC setup “GPIOs_ADC_Setup()”, in section 7.2.2 on page 65)
- Configure the ADC, (just the same as the function we used for ADC setup “ADC_Setup()”, in section 7.2.2 on page 68)
- Enable ADC interrupt
- Start the first conversion on channel 0 (just the same as the function we used for ADC conversion, in section 7.2.4 on page 69).

To enable the EOC interrupt we must find which is the interrupt number to be activated. In stm32f4xx.h there is a #define with ADC_IRQn that sets it to 18. Looking also at the data-sheet we find that 18 is the correct number for ADC interrupt. The priority for the ADC interrupt is 25, hard-wired in the controller.

CODE
```
NVIC->ICPR[0]  = (1<<ADC_IRQn); //Clear irq 18 pending flag, to avoid spurious
 ↪  interrupts.
NVIC->ISER[0] |= (1<<ADC_IRQn); //enable interrupt number 18.
//NVIC_EnableIRQ(ADC_IRQn); //CMSIS function for IRQs enable, just the same as
 ↪  the line before.
ADC1->SR &= ~ADC_SR_EOC; //clear EOC flag.
ADC1->CR1 |=ADC_CR1_EOCIE ;// Enable end of conversion interrupt.
```

Then we use the same routines as in the ADC chapter. Our new main will be:

CODE
```
int main(void)
{
  GPIOs_ADC_Setup();
  ADC_Setup();
  set_ADC_Channel(0); // select channel 0 as first conversion channel.
  // Priority:  25 (fixed)
  NVIC->ICPR[0] = (1 << ADC_IRQn); //Clear irq 18 pending flag.
  NVIC->ISER[0] |= (1 << ADC_IRQn); //enable interrupt number 18.
  //NVIC_EnableIRQ(ADC_IRQn); //CMSIS function to enable IRQ 18, same as above.
  ADC1->SR &= ~ADC_SR_EOC;//clear EOC irq flag.
  ADC1->CR1 |= ADC_CR1_EOCIE; // Enable end of conversion interrupt.
  ConversionStart(); //Start first conversion and wait for interrupt.
  for (;;) //main loop
  {  //do something else here...
         asm ("nop");//just a placeholder...
  } //main loop
}// main
```

Debug using “Add watch expression” to enable the display of the global varialbes ADCReadValue0 and ADCReadValue1 in the “expressions” tab of the debugger[5].

8.8 Hands-on Experiment 18: *TODO* ADC single channel continuous conversion

??In this experiment we will modify the previous code so that a single channel will be converted without the need to restart the conversion every time.

8.9 Hands-on Experiment 19: *WORK-IN-PROGRESS* ADC continuous conversion scan

In this experiment we want to read and convert both the analog inputs of the PMDB16 board, as we did in experiment 17 on page 79, but using the continuous conversion and scan of regular channels, so that the operations to be done during the interrupt routine will be reduced: There will be no need to restart a new conversion and the switch from one channel to the other since it will be automatically handled by the ADC peripheral.

The registers to set to have a conversion sequence are:

- ADC_CR1: Setting the SCAN bit enables the scan mode for the ADC.
- ADC_SQR1, ADC_SQR2, ADC_SQR3: Setting those registers (ADC Regular Sequence Register 1,2,3) set the scan sequence and the number of channels to be scanned. Bits 20-23 of the SQR1 register set the number of channels to scan. Other nybbles of registers SQR 1,2,3 hold the channel to be converted at every step. See the reference manual, section 11.12 for an accurate description of those registers.

Then we will set CONT bit in ADC_CR2 register to have continuous conversion, (Just like in the experiment 18) so that at the end of the scan a new scan will start.

Note that if the register ADC_DR is not read before the ADC ends the next conversion, the ADC will go in OVERRUN, the OVR bit in the Status register (ADC_SR) will be set and the scan sequence will be stopped. To avoid this condition we need to enable the ADC End-Of-Conversion interrupt in order to read the data from the ADC_SR fast enough.

Note that, in debug mode, a conversion scan is not stopped, so the ADC will go in overrun every time we stop our program execution. To exit the overrun condition we need to clear manually the OVR bit and re-start the sequence.

The new ADC setup routine will be as the ones before at the beginning (Peripheral reset and clock setup for ADC, ADC peripheral setup)

CODE
```
void ADC_Setup (void)
{ // Reset and set clock to ADC1 Peripheral.
```

[5]If there is no “Expressions” tab on your debug perspective, use Window->Show View->Expressions

```
  //RCC->APB2RSTR |= RCC_APB2RSTR_ADCRST;
//Reset the ADC peripheral, standard way
  RCC_b->APB2RSTR.ADC1RST = 1;
  //RCCAPB2RSTR &= ~(RCC_APB2RSTR_ADCRST);
//Leave reset for ADC peripheral, standard way.
  RCC_b->APB2RSTR.ADC1RST = 0;
  //RCC->APB2ENR |= RCC_APB2ENR_ADC1EN; //Enable ADC1, standard way.
  RCC_b->APB2ENR.ADC1EN = 1; //Initialize ADC1
  ADC->CCR|=0b11<<16; //ADC prescaler, divide by 8 to have more time to
   ↪ respond to irq.
  ADC1->CR2 = ADC_CR2_ADON; // ADON=1 to turn ADC on.. anything else to zero
   ↪ (single conversion)
  ADC1->CR1 = 0; //channel 0 for analog watchdog., all others to zero
   ↪ (default).
  ADC1->SMPR1 = 0; //Sampling time: 3 cycles for all (not needed since we will
   ↪ use only channels 0 and 1.
  ADC1->SMPR2 = 0b111111; //Sampling time, 3 bits for every channel. longest
   ↪ sample time for ch0 and 1 (480 clk cycles)
  ADC1->CR1|=ADC_CR1_SCAN; //set the SCAN bit for ADC1.
  ADC1->CR2|=ADC_CR2_CONT; //Set continuous conversion for ADC1
  ADC1->CR2|=ADC_CR2_EOCS; // set EOC bit at the end of every conversion.
  ADC1->SQR1&=~(0b1111<<20); //clear number of converted channels.
  ADC1->SQR1|=1<<20; //1 means 2 channels to convert in the sequence
  ADC1->SQR3=(0<<5)+1; //Convert channels 1 and 0 as regular sequence channels.
}
```

//ADC1->LTR=1024

8.10 Hands-on Experiment 20: **TODO**ADC Analog watchdog function with IRQ

In this experiment we want the ADC to check if the conversion result value is inside an interval.

The ADC converter in the STM32F401RE controller can check automatically if the conversion result is between tho thresholds. Any time the converted value is outside the two threshold set, the bit "AWD" in the ADC status register (ADC_SR) is set (Note: must be cleared by software).

To set the high and low value of the interval there are two registers: the HTR (High Thrishold Register) and LTR (Low Threshold register).

Irq used: ...

xxx..

Bibliography

[1] http://infocenter.arm.com/help/index.jsp?topic=/com

[2] .arm.doc.dui0553a/Babieigh.html, in the “Chapter 4. Cortex-M4 Peripherals”

[3] ”PM0214 Programming manual (STM32F3, STM32F4 and STM32L4 Series Cortex®-M4 programming manual)”, from www.st.com website.

9

Hands-on: USART

9.1 PMDB16 Hardware used

9.2 Hands-on Experiment 21: *TODO* Send a single char

9.3 Hands-on Experiment 22: *TODO* Receive a single char

9.4 *TODO* Ringbuffer implementation

9.5 Hands-on Experiment 23: *TODO* Send a string

9.6 Hands-on Experiment 24: *TODO* Receive a string

9.7 Hands-on Experiment 25: *TODO* Simple command interpreter

10

Hands-on: I^2C

10.1 Hardware used

In the PMDB16 board two external peripherals are connected to the I^{2}C bus: a temperature sensor LM75 and an accelerometer LIS2DE.

10.1.1 *I^{2}C bus description*

10.1.1.1 Hardware interface

The I^{2}C bus uses two electrical connections (lines) between the units connected to the bus. The output stages of every peripheral is connected using open-drain (or open-collector) outputs, in a wired-and configuration. That means that two pull-up resistors (one for the SCL and one for the SDA line) are mandatory for the i2c bus.

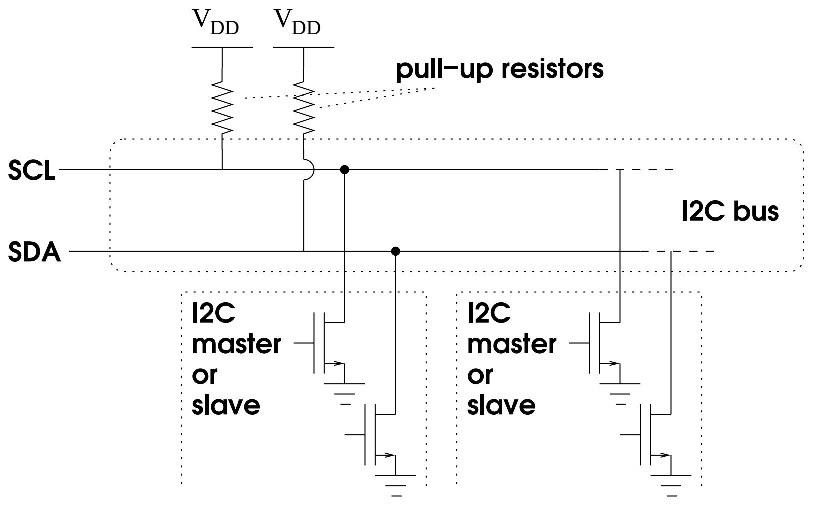


Figure 10.1: I^{2}C wired-and configuration of the bus lines

This connection implies that if two units on the bus are trying to set the line at a different status (i.e. one unit is trying to set it to zero and another is trying to set it to 1), the zero “wins”. In other words, *the dominant bit on the bus is zero*.

10.1.1.2 Protocol definition

The I^2C bus is a multi-master synchronous bus that allows more peripherals to communicate using only two signals: clock (SCL) and data (SDA), originally defined by Philips Semiconductors. The main features of the bus are:

- Both the signals (SDA and SCL) are bi-directional and can be set by any of the units that are communicating on the bus.

- Every communication is started by a master that generates the clock signal.

- If a communication is running no-one talks until it has finished

- If two masters start communicating, an arbitration procedure is used to decide which one can continue the communication and which one must stop.

- If a slave needs more time to elaborate a request can "slow down" the bus "stretching" the clock (forcing SCL to 0 making the bit time longer: the master will wait the clock to return to 1 before continuing the transmission of the next bit). The mechanism is known as "clock stretching". This allows "fast" and "slow" peripherals to communicate without pre-defining a bit rate.

The arbitration is also non-destructive[1] and is based on a "recessive/dominant" bit encoding. In fact the SDA and SCL signals are connected in a wired-and configuration between units, so that if two units transmit two different bits simultaneously, the 0 (that is the *DOMINANT* level) "wins". The unit that is transmitting the "1" (*RECESSIVE*) sees that the line status is different from the one it is transmitting and stops transmission. The unit that wins the arbitration does not even notice that someone else was trying to use the bus.

The protocol specifies that during a communication SDA can change only when the clock line is low, so changing SDA when the clock is high is used to issue the "start" and "stop" conditions.

To address a peripheral, the first data sent on the bus after a START condition must be the peripheral address. The peripheral address can be a 7 bit address or an 10 bit address, dependingo n the peripheral. If the first byte after START condition is 11110xx0 it means that the xx bits are the MSB of the address and the remaining part of the address will be sent as the next byte on the bus.

On the PMDB16 board we have only 7 bit address peripherals.

For a more detailed description of the I^2Cbus, refer to [1] and [2]. For a description of the peripheral implemented in the used controller, refer to the Reference manual [3].

10.1.2 The I2c bus on PMDB16 board, Connectors, I^2C 0 and I^2C 1.

Two of the I^2C interfaces available on the STM32F401 controller are connected to P2 and P15 connectors and are labeled SDA1/SCL1 and SDA0/SCL0.

[1] The transmitted bits are still valid when one of the two masters detects it lost the arbitration so there is no need to re-transmit the data packet on arbitration

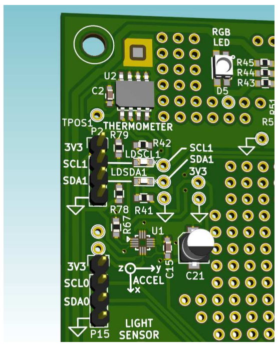


Figure 10.2: I2C connectors on the top-left board side

10.1.3 I^{2}C bus initialization (I2C bus 1)

To be able to send and receive data on the I^2C bus we need to initialize the ports and the I^{2}C peripheral.

As a first step we look at the PMDB16 schematics, looking for the pins and the peripheral used to communicate with the on-board I^{2}C sensors.

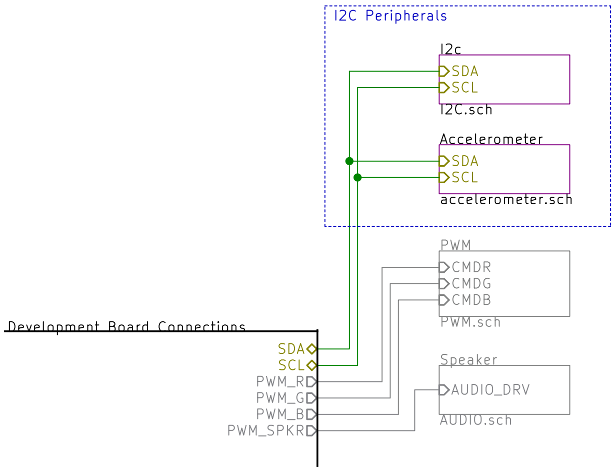


Figure 10.3: Main schematic page: i2c connections

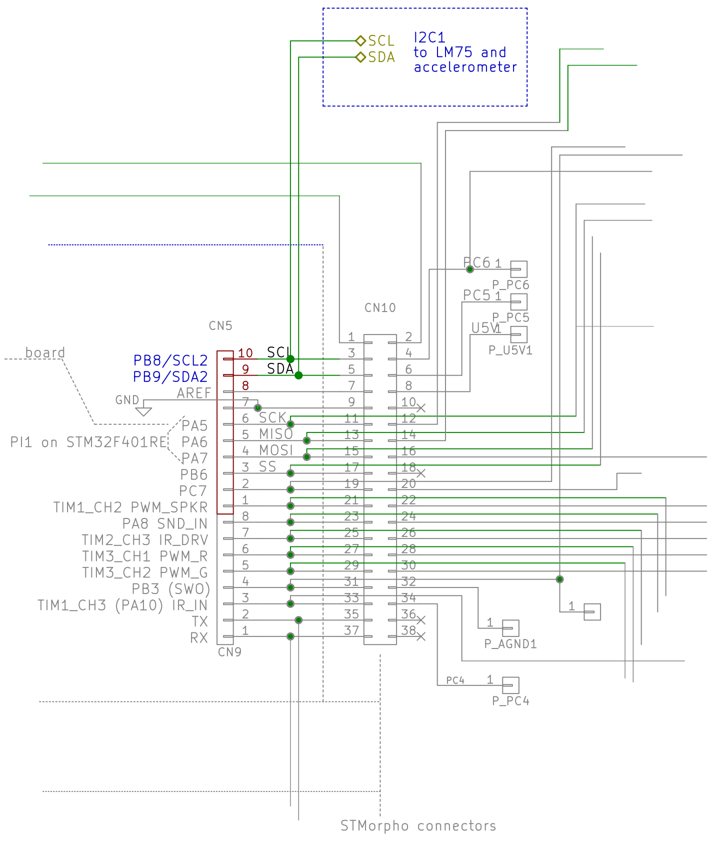


Figure 10.4: I^2C bus connections to the Nucleo board

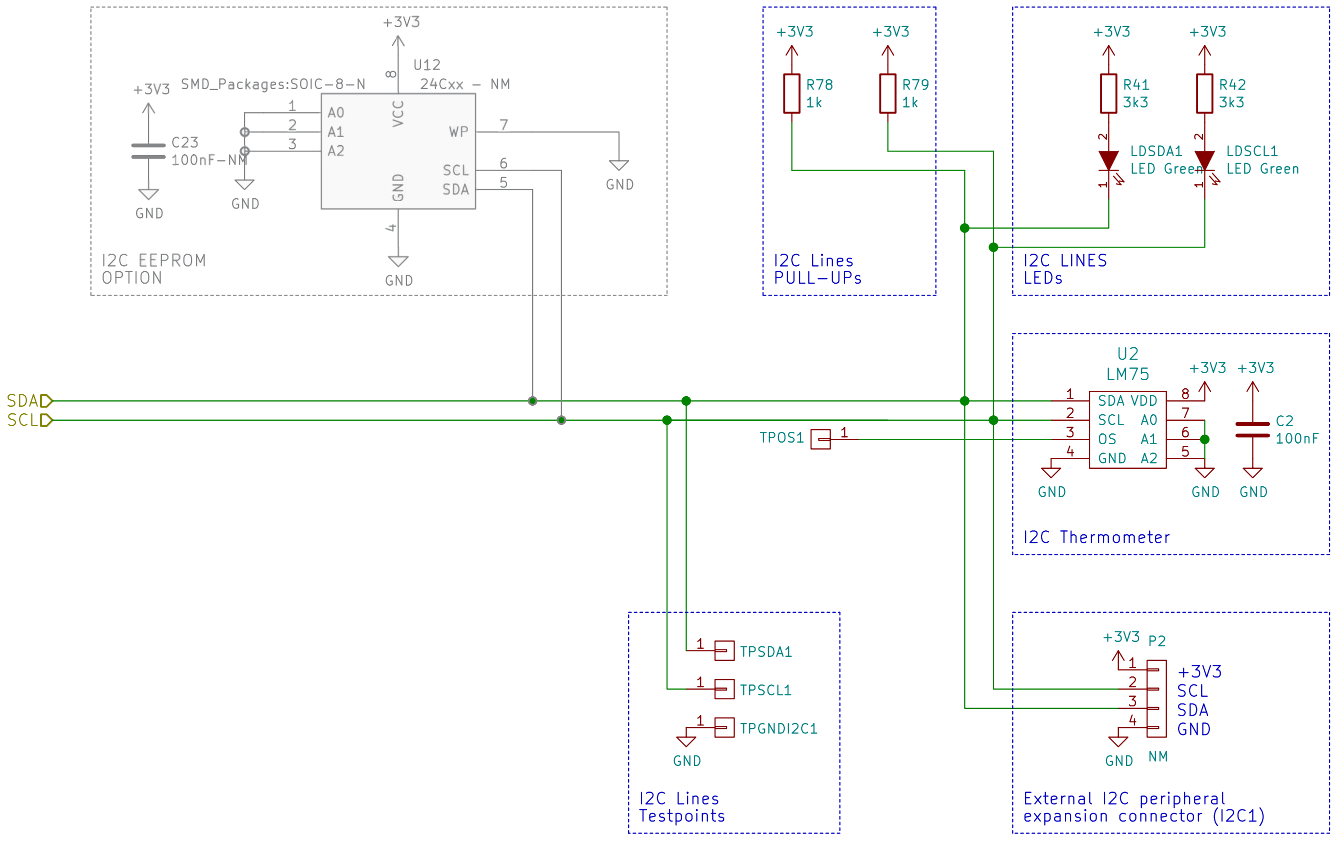


Figure 10.5: I^2C connection of the LM75

The ports connected to the on board sensors are PB8 and PB9, using their alternate function SCL2 and SDA2, so we will need to initalize and configure GPIOs and i2c peripheral.

- GPIOs PB8 and PB9

1. Mode: Alternate function I^2C ,
2. output type: open-drain.
3. speed: ...any of the allowed ones can work correctly, choose one.
4. Alternate function select: ...we need to search for the correct value in documentation.

- The I^2C peripheral 2 to act as a master for the bus.

Starting with the GPIO initialization we need to search in the data-sheet [4] which is the correct value for the alternate function register and we find in in table 9, Alternate function mapping, at page 46, where we find that for I2C1_SCL and I2C1SDA functions for GPIOs PB8 and PB9 we must use alternate function number 4.

So we define a define a set-up function:

CODE
```c
void initialize_GPIO_as_I2C(void)
{//Set clock to GPIOB peripheral
 RCC_b->AHB1ENR.GPIOBEN=1;         // Reset the GPIOB peripheral, with our
  ↪ better defines.
 RCC_b->AHB1RSTR.GPIOBRST=1;
 RCC_b->AHB1RSTR.GPIOBRST=0;  //Exit GPIO from reset state.
  //Configure PB8 and PB9 as alternate function I2C.
 GPIOBb->AFRH.AFRL8=0x04; // in datasheet: alternate 04 for GPIOB means I2C
  ↪ peripheral.
 GPIOBb->AFRH.AFRL9=0x04; // in datasheet: 04 for GPIOB means I2C peripheral.
 GPIOBb->MODER.MODER8=MODER_ALTERNATE_FUNCTION; //Alternate function.
 GPIOBb->MODER.MODER9=MODER_ALTERNATE_FUNCTION; //Alternate function.
 GPIOBb->OTYPER.OT8=OTYPE_OPEN_DRAIN;
 GPIOB_b->OTYPER.OT9=OTYPE_OPEN_DRAIN;
 GPIOBb->OSPEEDR.OSPEEDR8=OSPEED_MEDIUM;
 GPIOBb->OSPEEDR.OSPEEDR9=OSPEED_MEDIUM;//|=GPIO_OSPEEDER_OSPEEDR9;
}
```

Note that in this routine we used our own way to set bit, based on the data-structure we created in chapter 5. Also for RCC we have used a re-definition where the AHB1ENR was re-defined in the usual way.

If you did not re-define it, the “official” defines and data structure can be used, with the following code

CODE
```c
void initialize_GPIO_as_I2C(void) {
        RCC->AHB1ENR |= RCC_AHB1ENR_GPIOBEN; // Bit GPIOBEN of AHB1ENR
         ↪ register set to 1.
        // Reset the GPIOB peripheral
        RCC->AHB1RSTR|=RCC_AHB1RSTR_GPIOBRST;  //Reset the GPIO B peripheral.
        RCC->AHB1RSTR&=~RCC_AHB1RSTR_GPIOBRST; //Exit GPIO from reset state.
        RCC->AHB1ENR |= RCC_AHB1ENR_GPIOBEN;   //Enable peripheral clock to
         ↪ GPIO B
        //Conigure PB8 and PB9 as alternate function I2C.
        GPIOB->MODER &= (~GPIO_MODER_MODER8); // Clear moder8 bits
        GPIOB->MODER &= (~GPIO_MODER_MODER9); // Clear moder9 bits
        GPIOB->MODER |= (GPIO_MODER_MODER8_1 | GPIO_MODER_MODER9_1);
         ↪  //Alternate function for PB8 and PB9
        GPIOB->OTYPER |= (GPIO_OTYPER_OT_8 | GPIO_OTYPER_OT_9); // OPEN DRAIN.
         ↪          // Set speed for GPIOB8 and 9.
         ↪          GPIOB->OSPEEDR|=GPIO_OSPEEDER_OSPEEDR8;
        GPIOB->OSPEEDR|=GPIO_OSPEEDER_OSPEEDR9;
        // Select alternate function number 4 for GPIOB 8 and 9.
        GPIOB->AFR[1]=(GPIOB->AFR[1]&((0x0F<<0)|(0x0F<<4)))|(0x04<<0)|(0x04<<4
         ↪  );
}
```

10.1.4 I2C Peripheral initalization (after GPIO pins have been configured)

After the gpio configuration now we need to setup the I^{2}C peripheral in order to act as a master for the bus. In the reference manual [3] we find section "18.3.3 I^{2}C master mode" where we see that we need to:

- Reset the i2c1 peripheral (not reported in that section but has to be done for every peripheral we use)
- Enable the clock signal to the peripheral (not reported in that section but has to be done for every peripheral we use)
- Configure its registers for master mode, following the correct sequence
- Set the peripheral speed.

For reset and clock we proceed as usual: look into the reference manual for the RCC peripheral to find reset and clock enable bits for the I2C1 peripheral, so we find the I2C1RST bit in one of the "xxxRST" registers and the I2C1EN bit in one of the "xxxENR" registers, so:

CODE
```
RCC_b->APB1RSTR.I2C1RST=1; // Put I2C1 peripheral into RESET
RCC_b->APB1RSTR.I2C1RST=0; // Leave RESET state for I2C1 periph.
RCC_b->APB1ENR.I2C1EN=1;   // Clock enable for I2C1 peripheral
```

or, using the original defines from the stm32f4xx.h file, if RCC has not been re-defined as RCC_b with more human-readable fields.

CODE
```
RCC->APB1RSTR |= RCC_APB1RSTR_I2C1RST;   //Reset peripheral I2C1
RCC->APB1RSTR &= ~(RCC_APB1RSTR_I2C1RST);//Leave reset status for I2C1
RCC->APB1ENR  |= RCC_APB1ENR_I2C1EN;    //Enable clock for I2C1 peripheral
```

After the peripheral has been reset, we must follow a particular sequence in setting the next registers to put the peripheral in master mode. In section 18.3.3, "I^{2}C master mode" paragraph of the reference manual [3]

As previously, we could think about re-defining the I2C_Typedef structure from the file stm32f4xx.h in order to have a less error-prone code and easier programming of the registers, but since the registers of the i2c peripheral can be accessed only on word and half-word and the compiler can generate byte access instructions for the bit-fields, it is better not to do that and keep working on the original data structure.

CODE
```
// REQUIRED SEQUENCE in MASTER mode:
//                  Program I2C_CR2 Register in order to generate correct timings
I2C1->CR2 |= 8; // 2-> 2 MHz , 4=4MHz. 8=8MHz... peripheral clock frequency.
//                   ... Configure clock control registers...
I2C1->CCR = 0x50; //0x28; //Timings. FIXME: VERIFICARE VALORE.
//                  ... Configure the rise time register
I2C1->TRISE = 0x09; // See datasheet for 100kb/s
//                  Program the I2C_CR1 register to enable the peripheral
I2C1->CR1 |= I2C_CR1_PE; //Set the bit PE (Peripheral Enable) of the I2C1_CR1
 ↪  register.
```

After this init we are ready to start communication as masters on i2c bus. For the initialization procedure as stand-alone functions see the appendix at the end of this chapter.

10.1.5 I2C: Reading a register from an external peripheral

The sequence to use to read a register from an I^{2}C external peripheral when the pointer register is not yet set to the address we want to read is the following (see also figure 10.6 on the next page):

1. Set the external peripheral "pointer register" to the address we want to read, so that the peripheral knows that it has to send the content of that location during the next read operation. In "figure 164 Transfer sequence diagram for master transmitter" (the reference manual, section "18.3.3 I^{2}C Master mode", in the "Master transmitter" subsection) there is the sequence for the 7-bit address master transmitter, that we will follow.

2. Read data. At every byte the external peripheral "pointer register" will be incremented and we will continue reading as we need. In figure 165, in the reference manual there is the 7-bit master receiver sequence that we will follow.

So the complete sequence of operations that we will do is:

- Generate a START condition on the bus
- Send the slave device address, left shifted by 1 and with the LSB set to 0, meaning "WRITE OPERATION" for the next byte[2].
- Check if the device did send the ACK (Acknowledge) bit. If ACK is not present abort the communication.
- Send the register address we want to read (temperature register=0x00). This will be written in the "pointer register" of the LM75.
- Check if the device did send the ACK (Acknowledge) it. If ACK is not present abort the communication.
- Generate a new START condition on the bus (RE-START), (Note: no stop condition has been generated at the previous step).
- Send the slave device address, left shifted by 1 and with the LSB set to 1, meaning "READ OPERATION", so the next bytes read will be the one pointed by the "pointer register".

[2]On the PMDB16 board we have only I2C peripherals with 7 bit addresses.

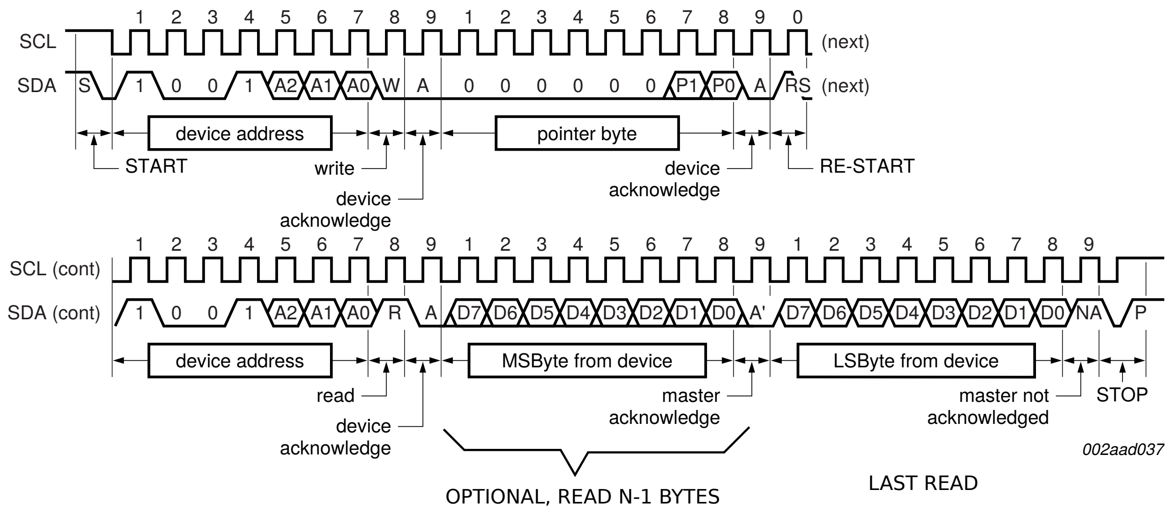


Figure 10.6: I²C register read operation (NXP LM75B datasheet)

- Check if the device did send the ACK (Acknowledge) bit. If ACK is not present abort the communication.
- Read a byte from the slave (i.e. generate the clock signal and receive data), sending an ACK if we want to continue reading the next byte or sending a NACK (Not acknowledge) to tell the slave that the communication ends.
- If needed Repeat the previous step (execute it only one time to receive 8 bits, two for 16 bits and so on).
- Generate a STOP condition on the bus.

To do such an operation we will write some support routines, starting from one reading a single 8 bit register from the peripheral.

10.1.6 8 bit (1 byte) register read

First we need to check if the I²C bus is busy and wait until it is not. In our board there is only one master (the controller we are programming), but remember that the I²C bus is multi-master, so in other configurations may be some other masters are talking on the bus.

To detect a bus busy condition, the bit "BUSY" in the I²C status register 2 (I2C_SR2) register should be used. It is set to 1 by hardware when SCL or SDA lines go low and cleared by hardware when a stop condition is detected. Note that the stop condition is generated when a master ends its operations on the bus.

CODE
```
        while (I2C1->SR2 & I2C_SR2_BUSY); //wait the bus to be free
```

Now we can start our communication, creating the start condition

CODE
```
        I2C1->CR1 |= I2C_CR1_START; //Start condition.
```

Since the start condition needs some time to be sent on the bus, we need to wait until it has been generated with the correct timings. In the status register 1 (I2c_SR) the bit SB is set to 1 when the start condition is generated. See reference manual section “18.6 I^{2}C registers” for full register description.

CODE
```c
        while (!((I2C1->SR1 & I2C_SR1_SB))); //Wait until start condition bit
        ↪ has been generated.
```

Reading the SR1 register and writing the data register clears the SB bit and starts the transmission now we send the 7 bit peripheral address on the bus. Since we want to write the pointer address of our external i2c peripheral we need to set the “write” bit, wo we send the peripheral address left shifted by one bit and clear the LSB. To make it more readable we also define a couple of constants

CODE
```c
#define OP_WRITE 0
#define OP_READ 1
I2C1->DR = (address<<1) | OP_WRITE; //Transmit address, write bit set to 0.
```

Since the hardware automatically detected that we are sending the slave address, it will automatically set the ADDR bit of the Status register 1 (SR1)

CODE
```c
while ((I2C1->SR1 & I2C_SR1_ADDR)==0); //wait until ADDR has been sent.
```

Now we check if the peripheral did answer with the acknowledge bit. If not we need to abort subsequent operations, issue a stop condition and exit the routine.

We also setting an error variable we will use at the end of the routine before sending the STOP condition. We can not exit immediatly on failure because at the end of our operations we need to generate a stop condition to free the bus.

CODE
```c
        //*** IF NO ACK has been received, ABORT.
        if ((I2C1->SR1 & I2C_SR1_AF) == 0) //Acknowledge has been received on
        ↪ address, go on.
        { // ...all the subsequent operations to read-write etc...
                //....
}
        else
                err = 1; //Acknowledge failure on first address send.
```

Inside the if we will continue our communications. To continue sending data we must read both the status registers in sequence and then write the data register. Immediatly after writing the data register the byte will be sent.

CODE
```c
      //***SECOND BYTE: THE REGISTER ADDRESS***
                sr1 = I2C1->SR1; // Consecutive read of SR1 and SR2 needed
                ↪ (can omit this since the previous "if" did read SR1)
                sr2 = I2C1->SR2; // Read SR2 mandatory to continue
                ↪ transmission.
                I2C1->DR = registerAddress; //Send the pointer register content
```

Now we need to wait until the transmission ends.

CODE
```c
                while ( ((I2C1->SR1 & I2C_SR1_BTF) == 0) && ((I2C1->SR1 &
                ↪ I2C_SR1_AF)==0) ); //wait until the byte transfer finished
                ↪ or NACK received
```

Now we check again for acknowledge.

CODE
```
if ((I2C1->SR1 & I2C_SR1_AF) == 0) //Acknowledge has been received, go on.
                {
                  // continue with communication, restart etc...
                }
                        else
                err = 2; //No acknowledge after register address sent
```

Inside the if (acknowledge received) we send another start condition

CODE
```
                  //*** RE-send START condition (RESTART)***
                                I2C1->CR1 |= I2C_CR1_START; //Repeated start
                                 ↪  condition.
                                while (!((I2C1->SR1 & I2C_SR1_SB))); //Wait
                                 ↪  until the START condition has been
                                 ↪  generated.
```

Re-send the device address.

CODE
```
                      //*** RE-SEND the device address, but this time with the
                       ↪  READ operation.
                            I2C1->DR = (address<<1) | OP_READ;
                while ((I2C1->SR1 & I2C_SR1_ADDR) == 0); //wait until the ADDR
                 ↪  has been sent. (no comm running).
```

Check for acknowledge once again

CODE
```
                          //*** If no ACK received abort ***
                if ((I2C1->SR1 & I2C_SR1_AF) == 0) //Acknowledge has been
                 ↪  received (Acknowledge failure is zero, go on.
                                {
                         //...continue with operations here...
                        }
                        else
                                err = 3; //No acknowledge after second
                                 ↪  peripheral address.
```

Now we are ready to read the data, inserting our code in the "true" branch of the previous if. Since in 8 bit register read this will be the last byte we will read, we should send a NACK at the end of the operation, so we need to clear the ACK bit in control register CR1 (see reference manual).

CODE
```
//*** READ With NACK (it is the las tybe we will read).
I2C1->CR1 &= (~I2C_CR1_ACK); //Send a NACK after the next read. TODO before
 ↪  clearing the ADDR flag.
sr1 = I2C1->SR1; //read SR1 READ SR1 AND SR2 in sequence TO CLEAR ADDR FLAG.
sr2 = I2C1->SR2; //read SR2 READ SR1 AND SR2 in sequence TO CLEAR ADDR FLAG.
// and wait for the end..
while ((I2C1->SR1 & I2C_SR1_RXNE) == 0);// Wait the read operation to finish.
res = I2C1->DR; // get the read byte and store it in the res variable.
```

At the end (outside all the ifs) we send the stop condition to free the bus.

CODE
```
        I2C1->CR1 |= I2C_CR1_STOP; //Generate the stop condition.
```

And finally exit the routine returning the byte value read or the error left-shifted 16 times, so that if the returned value is <256 means no error.

CODE
```
        if (err == 0)
                return res; //return the 0-255 value read.
        else
                return err<<16; //error, number outside the 8 bit limit..
        }
```

See the appendix of this chapter for the whole code for the routine.

Note that in this procedure we did not check for an eventual arbitration lost: Since on our board we are the only master present we will never have two masters talking at the same time, so using only our PMDB16 on-board peripherals there will never be an arbitration procedure to loose.

10.1.7 1-2-3 bytes bit register read

To read n bytes we can modify the 8 bit routine that we just wrote, creating a new function

CODE
```
I2C_ReadNBytesRegister(unsigned char address, unsigned char registerAddress,
 ↪  unsigned char bytesToRead)
```

that has the number of bytes to be read as an argument.

The only part that will change in communication is that the NACK will be sent only for the last byte to be read, so we will change just the last acknowledge-check if (the one that issues err=3 if evaluates to false) and we can change it from

CODE
```
//*** If no ACK received abort ***
if ((I2C1->SR1 & I2C_SR1_AF) == 0) //Acknowledge has been received
 ↪  (Acknowledge failure is zero, go on.
{
         //...continue with operations here...
}
else
        err = 3; //No acknowledge after second peripheral address.
```

to:

CODE
```
for (i=0; i<bytesToRead; i++)
{                             //*** If no ACK received abort ***
        if ((I2C1->SR1 & I2C_SR1_AF) != 0)
        {       err=3+i;                                  break; //abort.
         ↪                             }
        else//Acknowledge has been received (Acknowledge failure is zero, go
         ↪  on).
        {
                //  ...Read operations for the bytes to be read, setting ACK
                 ↪  accordingly.
        }
}
```

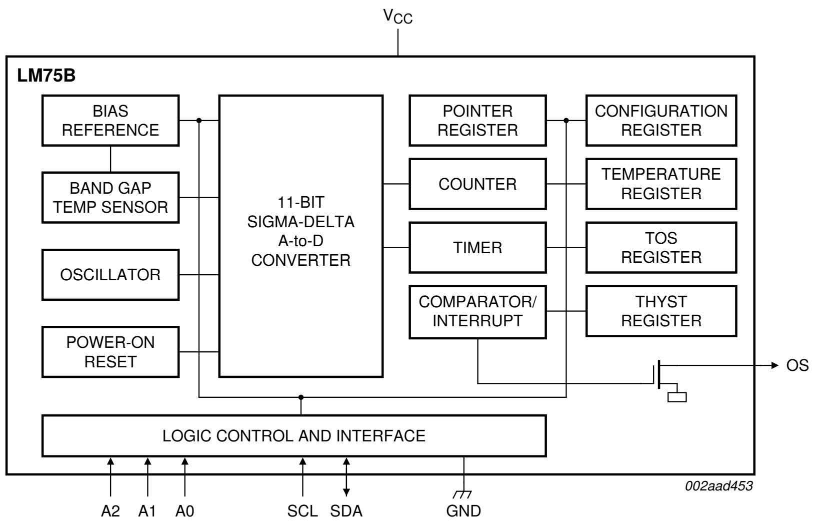


Figure 10.7: LM75B block diagram

Of course remember to declare int i at the beginning of the routine.

Inside the "else" branch of the previous for we add the read operations:

CODE
```
//...
else//Acknowledge has been received (Acknowledge failure is zero, go on).
{        if (i==bytesToRead-1) //Set the NACK for the last byte only.
                        I2C1->CR1 &= (~I2C_CR1_ACK); //Send a NACK after the
                         ↪  next read.
     else
                        I2C1->CR1 |= I2C_CR1_ACK; //Send an ACK to say it is
                         ↪  NOT the last byte we read.
        // Start read operation
        sr1 = I2C1->SR1; //read SR1 READ SR1 AND SR2 TO CLEAR ADDR or RXNE
         ↪  FLAG.
        sr2 = I2C1->SR2; //read SR2 READ SR1 AND SR2 TO CLEAR ADDR or RXNE
         ↪  FLAG.
        // and wait for the end..
        while ((I2C1->SR1 & I2C_SR1_RXNE) == 0);// Wait the read operation to
         ↪  finish.
    res = (res<<8)|I2C1->DR; // get the read byte and store it in the res
     ↪  variable left-shifting the old ones.
} //else (ack received.
```

10.2 Hands-on Experiment 26: HANDS-ON: LM75 temperature sensor

10.2.1 Sensor description

The LM75, LM75A, LM75B integrated circuits are temperature sensors that integrate also an ADC converter and an I^{2}C interface. On the PMDB16 board we have a LM75B temperature sensor connected to the i2C bus 1. For a detailed LM75B description please read this device's data-sheet [3].

10.2.1.1 LM75 Register table and functions

The LM75 has some internal registers that can be used to read the temperature and set some additional features of the peripheral. In the data-sheet there is a detailed description of those registers and, in table 5 we find:

Address	Register	R/W	length	Function
0x00	Temperature	r	2	Temperature register (Measured Temp data)
0x01	Configuration	r/w	1	Configuration register
0x02	Thyst	r/w	2	Hysteresis register
0x03	Tos	r/w	2	Overtemp shutdown threshold

The temperature register stores the current temperature as measured by the sensor.

The configuration register is used to set the OS pin behaviour and to put the device in shutdown mode. The only bit we care of in our experiments is the SHUTDOWN bit, but it is set to 0 (normal mode, device operating) by default.

The last two registers are used to control the OS output, that can be used to operate as a thermostat or interrupt signal when a certain temperature is reached. This output is connected to the testpoint TPOS1 on the PMDB16 board.

See section “7.4.2 Configuration register” in the LM75 data-sheet for a detailed register description.

10.2.2 Reading temperature

To read the temperature from the LM75, we need to read the temperature register. All the needed operations (sample, ADC conversion and all the timings) are automatically handled by the LM75 once it is not in shutdown mode. (At power-up it is not in shutdown mode, so we do not need any initialization for this peripheral).

To be able to start all this, we need to know the 7-bit address on the I^{2}C of the LM75B we are communicating with. In the data-sheet (Section “7.3 Slave address”) we find that the address is $0b1001A_2A_1A_0$, where the three LSb’s are defined by the voltage level at the pins A0, A1, A2 of the integrated circuit.

From circuit schematics (figure 10.5 on page 90) we see that all three pins are connected to ground, so the I^{2}C address of our temperature sensor will be $0b1001000$.

To get the first byte of the temperature register only (i.e. the temperature in degrees) we can use now

CODE
```
int TempHi;
TempHi = I2C_ReadNBytesRegister(LM75_I2C_ADDRESS, LM75_TEMP_REG_ADDRESS,1);
```

Now try to execute it, putting the TempHi assignment inside the main loop, put a breakpoint and watch the TempHi variable set to current temperature. Also look at the leds LDSCL1 and LDSDA1 than should blink at any i2c communication, since they turn on every time the I2C lines go low.

Try putting a finger on the sensor to see if the temperature changes.

To have the complete value of themperature we should read all the 16 bits, throw away the 5 LSBs and divide the number by 8.

CODE
```c
int tempRegister;
float temperature;
tempRegister=I2C_ReadNBytesRegister(LM75_I2C_ADDRESS, LM75_TEMP_REG_ADDRESS,2);
temperature=(float)(tempRegister>>5)/8;
```

As before, put the two last code lines inside the main loop and look at their value using breakpoints and watches [3]

TODO If you have yet done the experiments using the LCD module (chapter 13), you can display those temperatures on the LCD.

10.3 Hands-on Experiment 27: *TODO* Accelerometer

Setup

Read acceleration

10.4 Chapter Appendix: Code.

10.4.1 Code: Initialization of PB8 and PB9 pins as I2C peripheral SDA and SCK lines.

CODE
```c
void initialize_GPIO_as_I2C(void) {
    // Set clock to GPIOB peripheral
    RCC_b->AHB1ENR.GPIOBEN=1;
    RCC_b->AHB1RSTR.GPIOBRST=1; // Reset the GPIOB peripheral WARNING: do it
     ↪  only if you
    RCC_b->AHB1RSTR.GPIOBRST=0; // did not reset the same peripheral before.
    //Configure PB8 and PB9 as alternate function I2C.
    GPIOBb->AFRH.AFRL8=0x04; // in user manual: 04 for GPIOB means I2C
     ↪  peripheral.
    GPIOBb->AFRH.AFRL9=0x04; // in user manual: 04 for GPIOB means I2C
     ↪  peripheral.
    GPIOBb->MODER.MODER8=MODER_ALTERNATE_FUNCTION; //Alternate function.
    GPIOBb->MODER.MODER9=MODER_ALTERNATE_FUNCTION; //Alternate function.
    GPIOBb->OTYPER.OT8=OTYPE_OPEN_DRAIN;
    GPIOBb->OTYPER.OT9=OTYPE_OPEN_DRAIN;
    GPIOBb->OSPEEDR.OSPEEDR8=OSPEED_MEDIUM;
    GPIOBb->OSPEEDR.OSPEEDR9=OSPEED_MEDIUM;
}
```

10.4.2 Code: I2C1 Peripheral init

CODE
```c
void initialize_I2C1_Peripheral(void)
{   RCC_b->APB1RSTR.I2C1RST=1; // Put I2C1 peripheral into RESET
    RCC_b->APB1RSTR.I2C1RST=0; // Leave RESET state for I2C1 periph.
    RCC_b->APB1ENR.I2C1EN=1;   // Clock enable for I2C1 peripheral
    // REQUIRED SEQUENCE in MASTER mode:
    //                 Program I2C_CR2 Register in order to generate correct
     ↪  timings
```

[3] If the the "variables" tab is missing in the eclipse window you can enable it using Window->Show view->Variables

```
    I2C1->CR2 |= 42; // 2-> 2 MHz , 4=4MHz. 8=8MHz... peripheral clock
    ↪ frequency.
    //                 ... Configure clock control registers...
    I2C1->CCR = 0x00D5;// value for 100kb/s, 42MHz APB1 clock.
    //                ... Configure the rise time register
    I2C1->TRISE = 0x09; // See datasheet for 100kb/s
    //                Program the I2C_CR1 register to enable the peripheral
    I2C1->CR1 |= I2C_CR1_PE; //Set the bit PE (Peripheral Enable) of the
    ↪ I2C1_CR1 register.
}
```

10.4.3 8 bit register read routine

Here we have the 8 bit register read routine, a little bit "beautified" in order to be better readable (with some defines and macros added before the routine.)

CODE

```
//defines and macros used to make more readable the code
#define OP_READ 1
#define OP_WRITE 0
#define ACKNOWLEDGE_RECEIVED ((I2C1->SR1 & I2C_SR1_AF) == 0)
#define I2C1_BUS_IS_BUSY (I2C1->SR2 & I2C_SR2_BUSY)
#define I2C1_ADDR_NOT_SENT ((I2C1->SR1 & I2C_SR1_ADDR)==0)
#define I2C1_START_CONDITION_PENDING (!((I2C1->SR1 & I2C_SR1_SB)))
#define I2C1_READ_RUNNING ((I2C1->SR1 & I2C_SR1_RXNE) == 0)
#define I2C1_beginStartCondition() I2C1->CR1 |= I2C_CR1_START
#define I2C1_setNACKforTheNextRead() I2C1->CR1 &= (~I2C_CR1_ACK)
#define I2C1_beginStopCondition() I2C1->CR1 |= I2C_CR1_STOP;

/// Reads an 8 bit register from the peripheral via the i2c bus.
/* TIP: Delay that can be used just to see different steps on oscilloscope..
        for (i=0; i<100; i++) {  asm("nop");  }
*/
// 8 bit i2c register read routine. Returns err<<24 in case of error, the read
 ↪ byte on success.
int I2C_Read8bitRegister(unsigned char address, unsigned char registerAddress)
{  unsigned int sr1, sr2; //used for dummy reads of the registers.
   unsigned char err = 0;
   unsigned int res=0; // return value if no error..
   while (I2C1_BUS_IS_BUSY); // Wait for bus to be free. do not start
    ↪ transmission if busy.
// *** Start condition
   I2C1_beginStartCondition(); //Start condition.
   while (I2C1_START_CONDITION_PENDING);  //Wait until start condition bit has
    ↪ been generated.
   // ***FIRST BYTE: THE SLAVE ADDRESS***
   I2C1->DR = (address<<1) | OP_WRITE; //Transmit address, write bit set to 1.
   // We should also check for an arbitraton lost, but on our board we are the
    ↪ only master, so
   // we will never loose an arbitration.
   while (I2C1_ADDR_NOT_SENT); //wait until ADDR has been sent.
   //TIP: If needed Insert a small delay here to better see the different
    ↪ bytes on oscilloscope
   // *** IF NO ACK has been received, ABORT.
   if (ACKNOWLEDGE_RECEIVED)
   {   //Acknowledge has been received on address, go on.
```

```c
       //***SECOND BYTE: THE REGISTER ADDRESS***
       // We should also check for an arbitraton lost, but on our board we are the only master, so
       // we will never loose an arbitration on i2c bus.
       //sr1 = I2C1->SR1; // Consecutive read of SR1 and SR2 needed, but the IF before did read SR1 yet.
       sr2 = I2C1->SR2; // Read SR2 mandatory to stay in 7-bit address mode.
       I2C1->DR = registerAddress; //Send the pointer register content
       while ( ((I2C1->SR1 & I2C_SR1_BTF)==0) && ((I2C1->SR1 & I2C_SR1_AF)==0) ); //wait until the byte transfer finished or NACK received
       //TIP: If needed insert a small delay here to better see the different bytes with an oscilloscope
       if (ACKNOWLEDGE_RECEIVED)
       { //Acknowledge has been received on register address, go on.
         //*** RE-send START condition (RESTART)***
         I2C1_beginStartCondition(); //Repeated start condition.
         while (I2C1_START_CONDITION_PENDING); //Wait until the START condition has been generated.
         //*** RE-SEND the device address, but this time with the READ/WRITE bit set to "READ".
         I2C1->DR = (address<<1) | OP_READ;
         while (I2C1_ADDR_NOT_SENT); //wait until the ADDR has been sent.
         //*** If no ACK received abort ***
         if (ACKNOWLEDGE_RECEIVED) //Acknowledge has been received?
         { //*** READ With NACK (it is the las tybe we will read).
            I2C1_setNACKforTheNextRead(); //Send a NACK after the next read.
            // Start read operation
            sr1 = I2C1->SR1; //read SR1 READ SR1 AND SR2 TO CLEAR ADDR FLAG.
            sr2 = I2C1->SR2; //read SR2 READ SR1 AND SR2 TO CLEAR ADDR FLAG.
            // after clearing the ADDR flag the reception starts.
                while (I2C1_READ_RUNNING);// Wait the read operation to finish.
            res = I2C1->DR; // get the read byte and store it in the res variable.
          } else
                           err = 3; //No acknowledge after second peripheral address sent.
        } else //Acknowledge not received, return with error.
                        err = 2;
   } else
        err = 1; //Acknowledge failure on first address send.
// TIP: Insert a small delay here if you want to see better the sequence on oscilloscope.
I2C1_beginStopCondition(); //Send stop condition. (no wait here to check if the condition has been issued).
if (err == 0)
       return res; //return the 0-255 value read.
else
       return err<<24; //error, number outside the 8 bit limit..
} // I2C_Read8bitRegister()
```

10.4.4 8-16-24 bit register read routine

The routine is yet "beautified" to make it more readable and uses the same defines shown for the previous routine:

CODE

```
//defines and macros used to make more readable the code
#define OP_READ 1
#define OP_WRITE 0
#define ACKNOWLEDGE_RECEIVED ((I2C1->SR1 & I2C_SR1_AF) == 0)
#define I2C1_BUS_IS_BUSY (I2C1->SR2 & I2C_SR2_BUSY)
#define I2C1_ADDR_NOT_SENT ((I2C1->SR1 & I2C_SR1_ADDR)==0)
#define I2C1_START_CONDITION_PENDING (!((I2C1->SR1 & I2C_SR1_SB)))
#define I2C1_READ_RUNNING ((I2C1->SR1 & I2C_SR1_RXNE) == 0)
#define I2C1_beginStartCondition() I2C1->CR1 |= I2C_CR1_START
#define I2C1_setNACKforTheNextRead() I2C1->CR1 &= (~I2C_CR1_ACK)
#define I2C1_beginStopCondition() I2C1->CR1 |= I2C_CR1_STOP;
```

This routine takes three arguments: the i2c peripheral address, the register address inside the peripheral and the number of bytes to read (maximum 3 for this routine).

CODE

```
int I2C_ReadNBytesRegister(unsigned char address, unsigned char registerAddress, unsigned char bytesToRead) {
   unsigned int sr1, sr2; //used for dummy reads of the registers.
   unsigned char err = 0;
   unsigned int res=0;
   int i;
   if (bytesToRead>3) return -1; //This routine works only for 1,2,3 bytes to read (since the 8 MSbits are used to return an error.
   while (I2C1_BUS_IS_BUSY); // Wait for bus to be free. do not start transmission if busy.
   // *** Start condition
   I2C1_beginStartCondition(); //Start condition.
   while (I2C1_START_CONDITION_PENDING);  //Wait until start condition bit has been generated.
   // ***FIRST BYTE: THE SLAVE ADDRESS***
   I2C1->DR = (address<<1) | OP_WRITE; //Transmit address, R/W bit set to 0 (write).
   // We should also check for an arbitraton lost, but on our board we are the only master, so
   // we will never loose an arbitration on i2c bus.
   while (I2C1_ADDR_NOT_SENT); //wait until ADDR has been sent.
   //TIP: If needed Insert a small delay here to better see the different bytes on oscilloscope
   // *** IF NO ACK has been received, ABORT.
   if (ACKNOWLEDGE_RECEIVED)
   {  //Acknowledge has been received on address, go on.
      //***SECOND BYTE: THE REGISTER ADDRESS***
      // We should also check for an arbitraton lost, but on our board we are the only master, so
      // we will never loose an arbitration on i2c bus.
      //sr1 = I2C1->SR1; // Consecutive read of SR1 and SR2 needed, but the IF before did read SR1 yet.
      sr2 = I2C1->SR2; // Read SR2 mandatory to stay in 7-bit address mode.
      I2C1->DR = registerAddress; //Send the pointer register content
      while ( ((I2C1->SR1 & I2C_SR1_BTF)==0) && ((I2C1->SR1 & I2C_SR1_AF)==0) ); //wait until the byte transfer finished or NACK received
      //TIP: If needed insert a small delay here to better see the different bytes with an oscilloscope
      if (ACKNOWLEDGE_RECEIVED)
      {        //Acknowledge has been received on register address, go on.
         //*** RE-send START condition (RESTART)***
        I2C1_beginStartCondition(); //Repeated start condition.
        while (I2C1_START_CONDITION_PENDING); //Wait until the START condition has been generated.
        //*** RE-SEND the device address,but this time with the READ/WRITE bit set to "READ".
            I2C1->DR = (address<<1) | OP_READ;
            while (I2C1_ADDR_NOT_SENT); //wait until the ADDR has been sent.
            for (i=0; i<bytesToRead; i++) // data read loop
            {   //*** If no ACK received abort ***
                if ((I2C1->SR1 & I2C_SR1_AF) != 0)
                {    err=3+i;
                     break; //abort exiting the for loop.
                }
                else//Acknowledge has been received (Acknowledge failure is zero, go on).
                {    if (i==bytesToRead-1) //Set the NACK for the last byte to read.
                                            I2C1->CR1 &= (~I2C_CR1_ACK); //Send a NACK for the last byte only.
                     else
                        I2C1->CR1 |= I2C_CR1_ACK; //Send an ACK to say it is NOT the last byte we read.
                     // Start read operation
                     sr1 = I2C1->SR1; //read SR1 READ SR1 AND SR2 TO CLEAR ADDR or RXNE FLAG.
                     sr2 = I2C1->SR2; //read SR2 READ SR1 AND SR2 TO CLEAR ADDR or RXNE FLAG.
                     // and wait for the end..
                     while (I2C1_READ_RUNNING);// Wait the read operation to finish.
                     res = (res<<8)|I2C1->DR; // get the read byte and store it in the res variable left-shifting the old ones.
                } //else (ack received)
           }//for
        } else //Acknowledge not received, return with error.
              err = 2;
     } else
          err = 1; //Acknowledge failure on first address send.
    // TIP: Insert a small delay here if you want to see better the sequence on oscilloscope.
    I2C1_beginStopCondition(); //no wait here to check if the condition has been issued.
    if (err == 0)
        return res; //return the value read.
    else
        return err<<24; //error, number outside the 8 bit limit..
}
```

Bibliography

[1] http://www.i2c-bus.org/fileadmin/ftp/i2c_bus_specification_1995.pdf

[2] https://www.nxp.com/docs/en/user-guide/UM10204.pdf

[3] https://www.nxp.com/docs/en/data-sheet/LM75B.pdf, “LM75B Digital temperature sensor and thermal watchdog”

11

Hands-on: PWM

11.1 Hardware used

11.2 Hands-on Experiment 28: *TODO* PWM: Audio output, fixed note

11.3 Hands-on Experiment 29: *TODO* PWM: Select tone with potentiometer

11.4 Hands-on Experiment 30: *TODO* PWM: play music using the keyboard

11.5 Hands-on Experiment 31: *TODO* RGB LED

three PWM from the same peripheral

Hands-on: RGB led controlled by encoder and potentiometer

encoder: color wheel

potentiometer: fade to black-purecolor-white.

12

Hands-on: Advanced PWM: IRcomm

12.1 Hands-on Experiment 32: *TODO* Using PWM to generate the IR burst at 38 kHz

12.2 Hands-on Experiment 33: *TODO* PWM and timer to send the IR encoded data

12.3 Hands-on Experiment 34: *TODO* Receiving IR data using GPIO input AND a timer

13

Hands-on: Alphanumeric LCD Module

13.1 Hardware used

The PMDB16 board integrates a standard alphanumeric LCD display module, that can be driven directly using GPIOs. Depending on board version also the SPI can be used to drive that LCD.

The LCD is an Hitachi HD44780 LCD controller compatible LCD display. The HD44780 is a de-facto standard for industrial alphanumeric LCD modules. Newer chips require timings that can be a little bit different, so reading the data sheet of the used module helps to use it correctly.

The LCD can be driven in 8bit or 4 bit mode. On the PMDB16 board the 4 bit mode is used in order to drive it with only 7 GPIOs.

Datasheet: The driver and the protocol, 8bit and 4bit communication

13.2 Hands-on Experiment 35: *TODO* LCD Initialization

GPIO init

LCD init

13.3 Hands-on Experiment 36: *TODO* Write a char at a given location

13.4 Hands-on Experiment 37: *TODO* Write a string at a given location

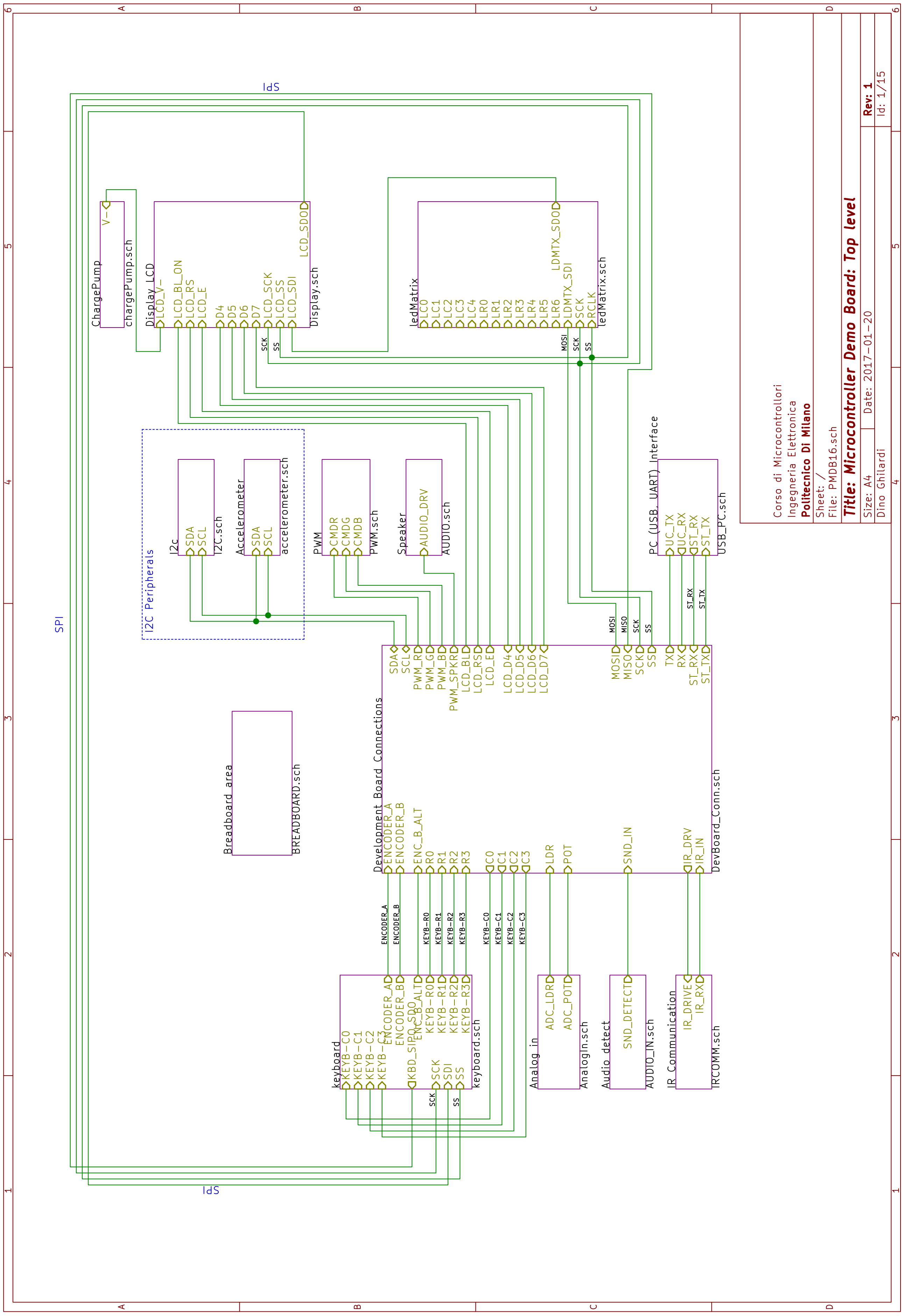
13.5 Hands-on Experiment 38: *TODO* Programming chars: drawing bars.

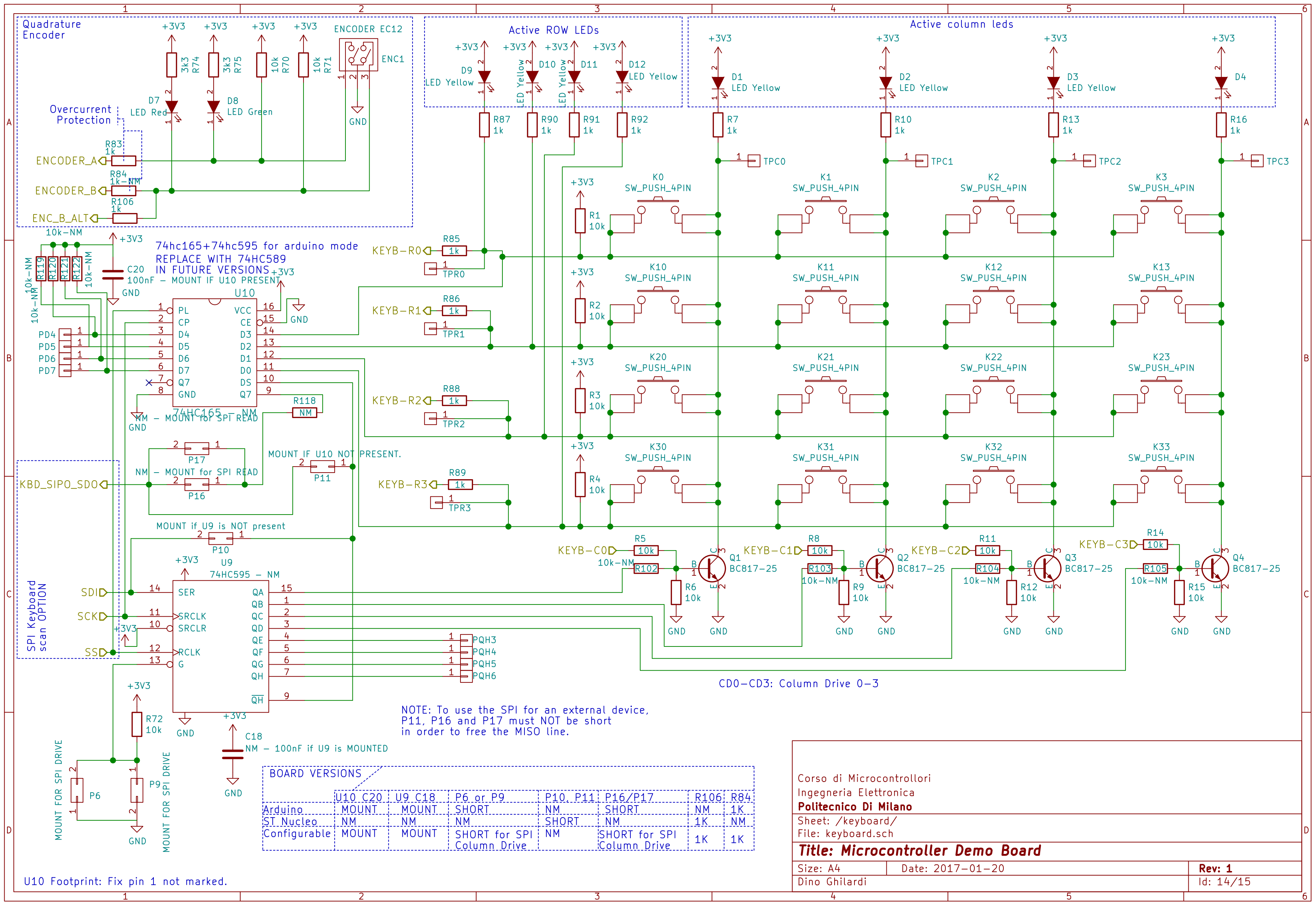
Drawing a bar on LCD driven by the encoder or ADC.

14

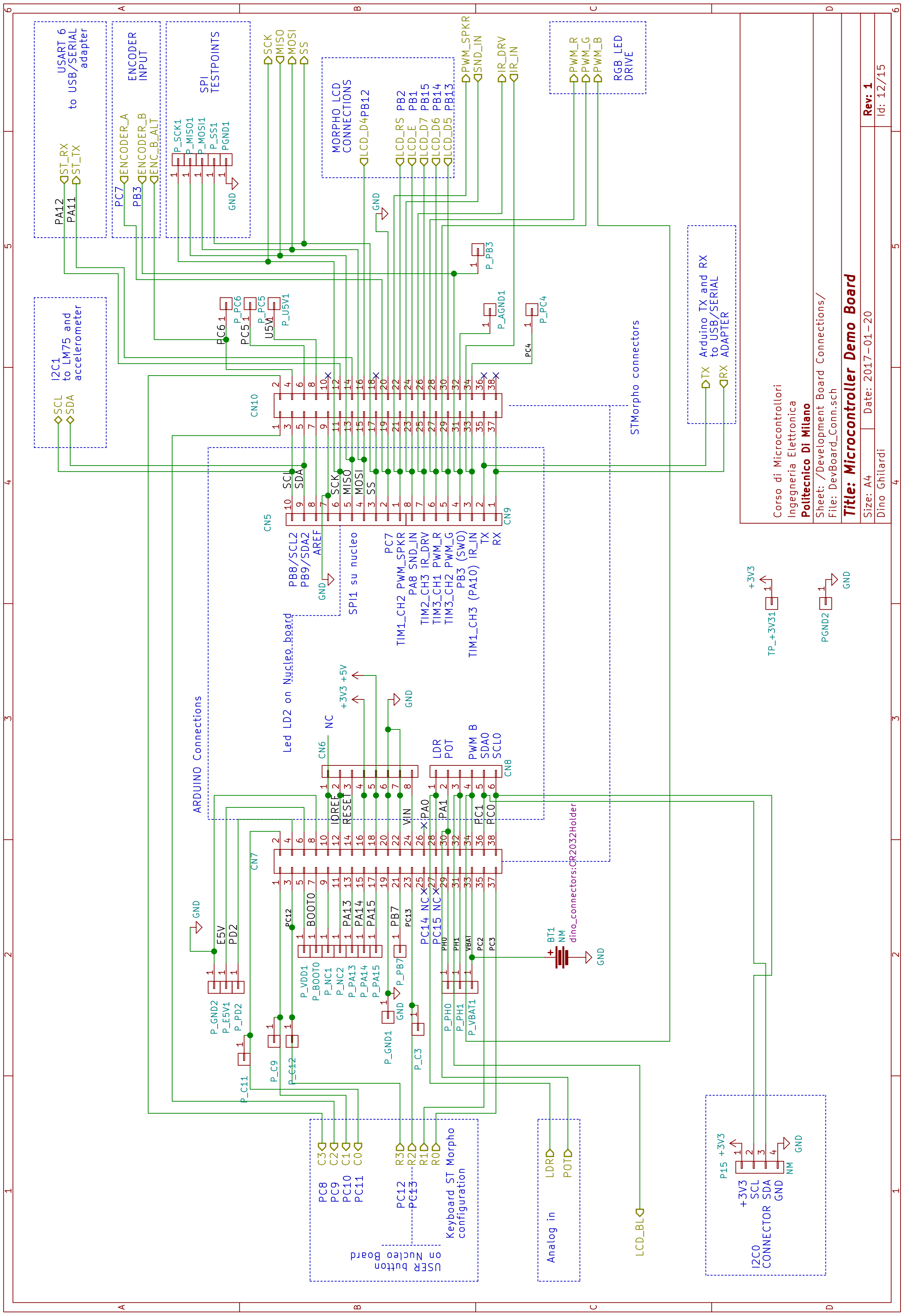
Appendix: PMDB16 Schematics

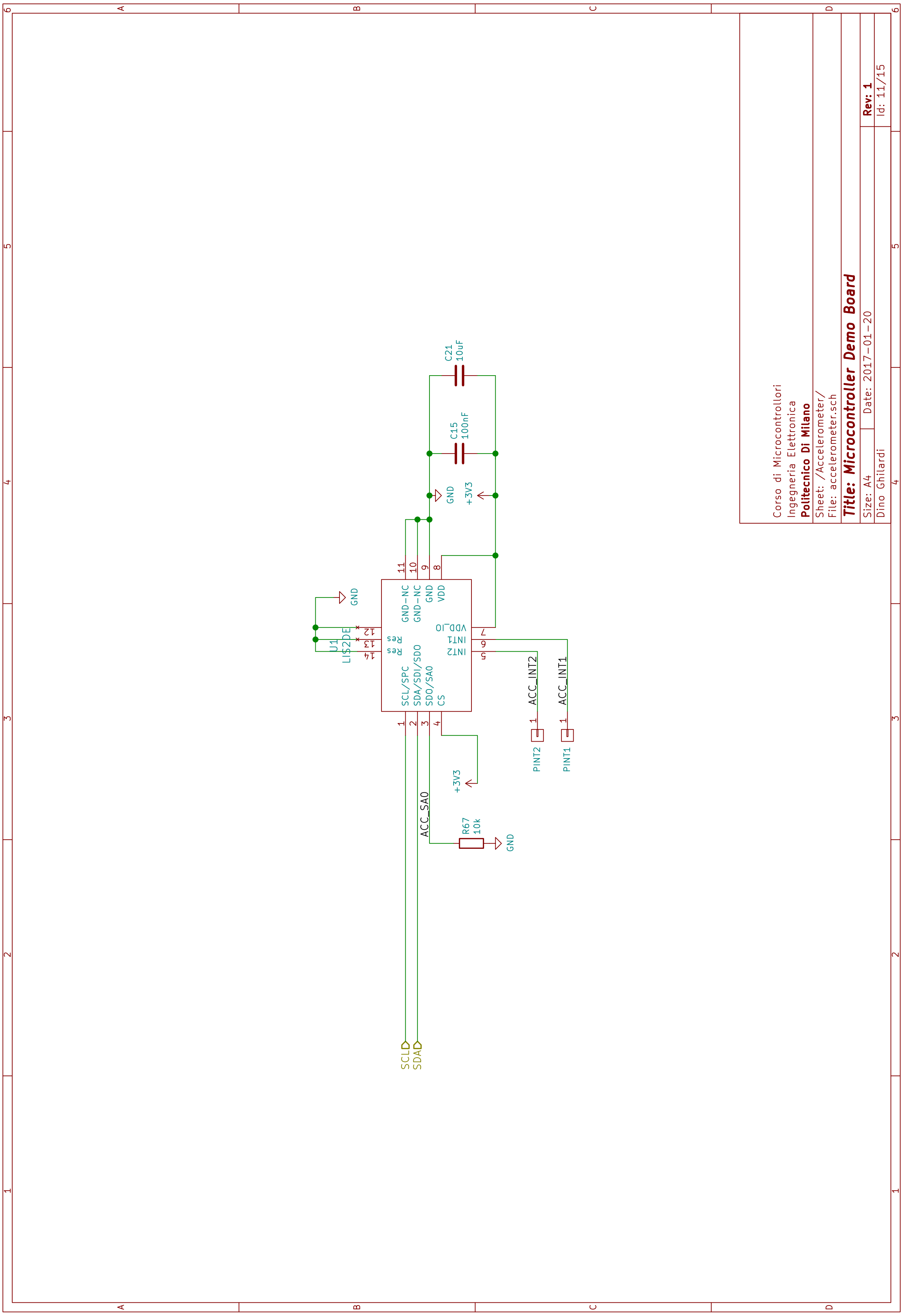
In this appendix there are the schematics for the PMDB16 main board.

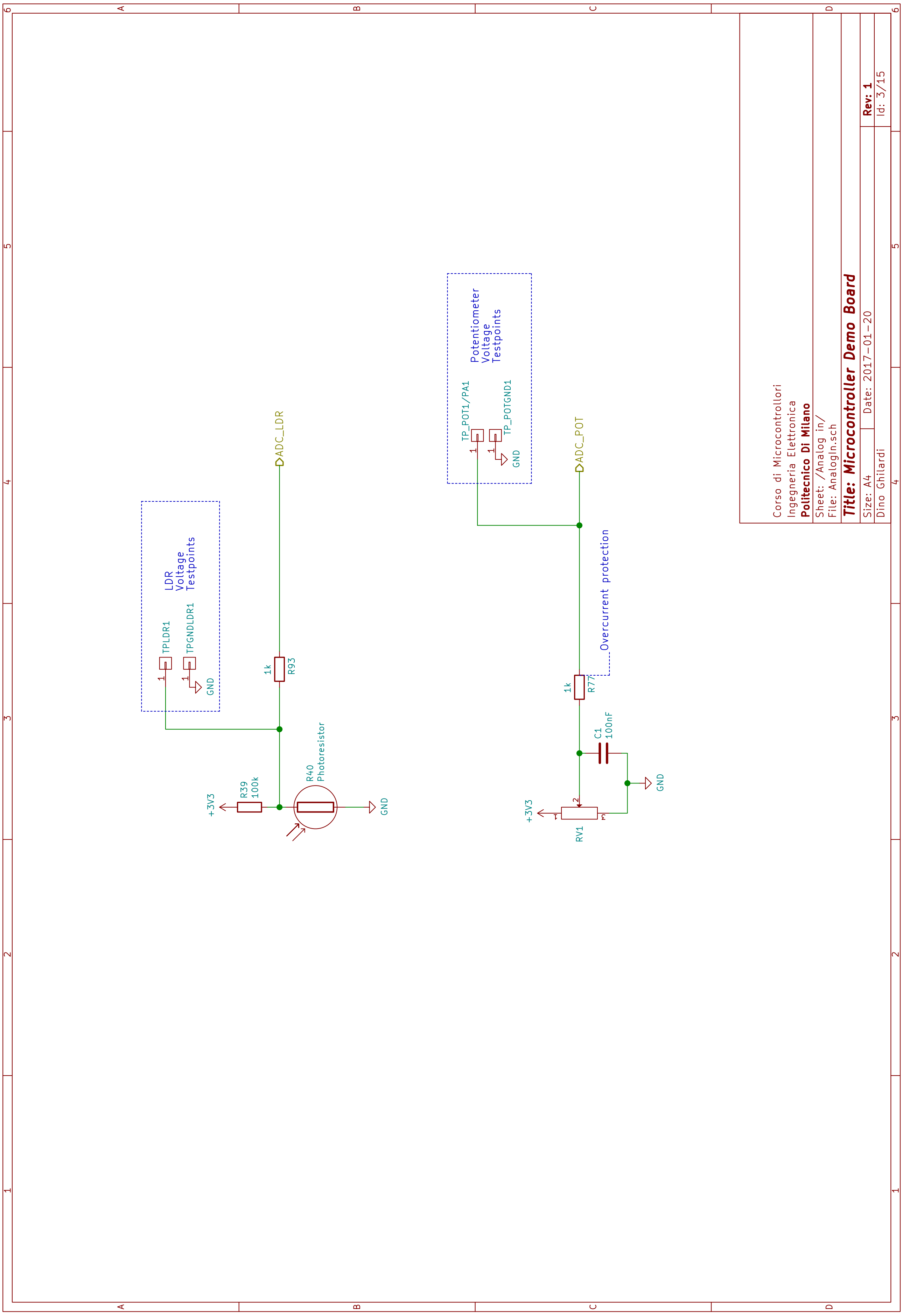

Corso di Microcontrollori
Ingegneria Elettronica
Politecnico Di Milano
Sheet: /
File: PMDB16.sch
Title: Microcontroller Demo Board: Top level
Size: A4
Date: 2017-01-20
Rev: 1
Dino Ghilardi
Id: 1/15
SPI
ChargePump
chargePump.sch
V-
Display_LCD
Display.sch
LCD_V-
LCD_BL_ON
LCD_RS
LCD_E
D4
D5
D6
D7
LCD_SCK
LCD_SS
LCD_SDI
LCD_SDO
ledMatrix
ledMatrix.sch
LC0
LC1
LC2
LC3
LC4
LR0
LR1
LR2
LR3
LR4
LR5
LR6
LDMTX_SDI
LDMTX_SDO
SCK
RCLK
MOSI
SS
I2C Peripherals
I2c
I2C.sch
SDA
SCL
Accelerometer
accelerometer.sch
PWM
PWM.sch
CMDR
CMDG
CMDB
Speaker
AUDIO.sch
AUDIO_DRV
PC (USB_UART) Interface
USB_PC.sch
UC_TX
UC_RX
ST_RX
ST_TX
Development Board Connections
DevBoard_Conn.sch
PWM_R
PWM_G
PWM_B
PWM_SPKR
LCD_BL
LCD_RS
LCD_E
LCD_D4
LCD_D5
LCD_D6
LCD_D7
MISO
TX
RX
ENCODER_A
ENCODER_B
ENC_B_ALT
R0
R1
R2
R3
C0
C1
C2
C3
LDR
POT
SND_IN
IR_DRV
IR_IN
Breadboard_area
BREADBOARD.sch
keyboard
keyboard.sch
KEYB-C0
KEYB-C1
KEYB-C2
KEYB-C3
KBD_SIPO_SDO
SDI
KEYB-R0
KEYB-R1
KEYB-R2
KEYB-R3
Analog_in
AnalogIn.sch
ADC_LDR
ADC_POT
Audio_detect
AUDIO_IN.sch
SND_DETECT
IR_Communication
IRCOMM.sch
IR_DRIVE
IR_RX

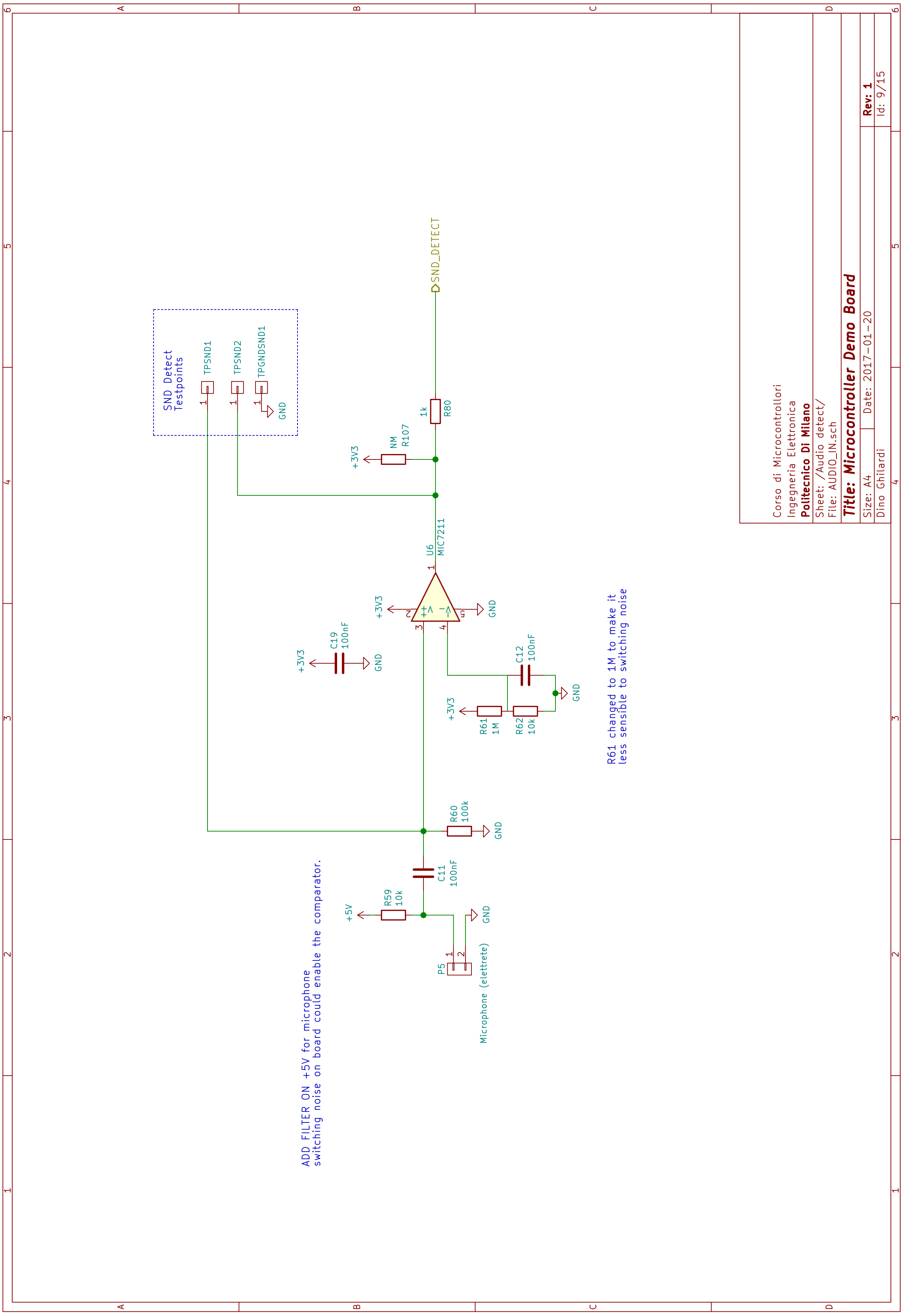


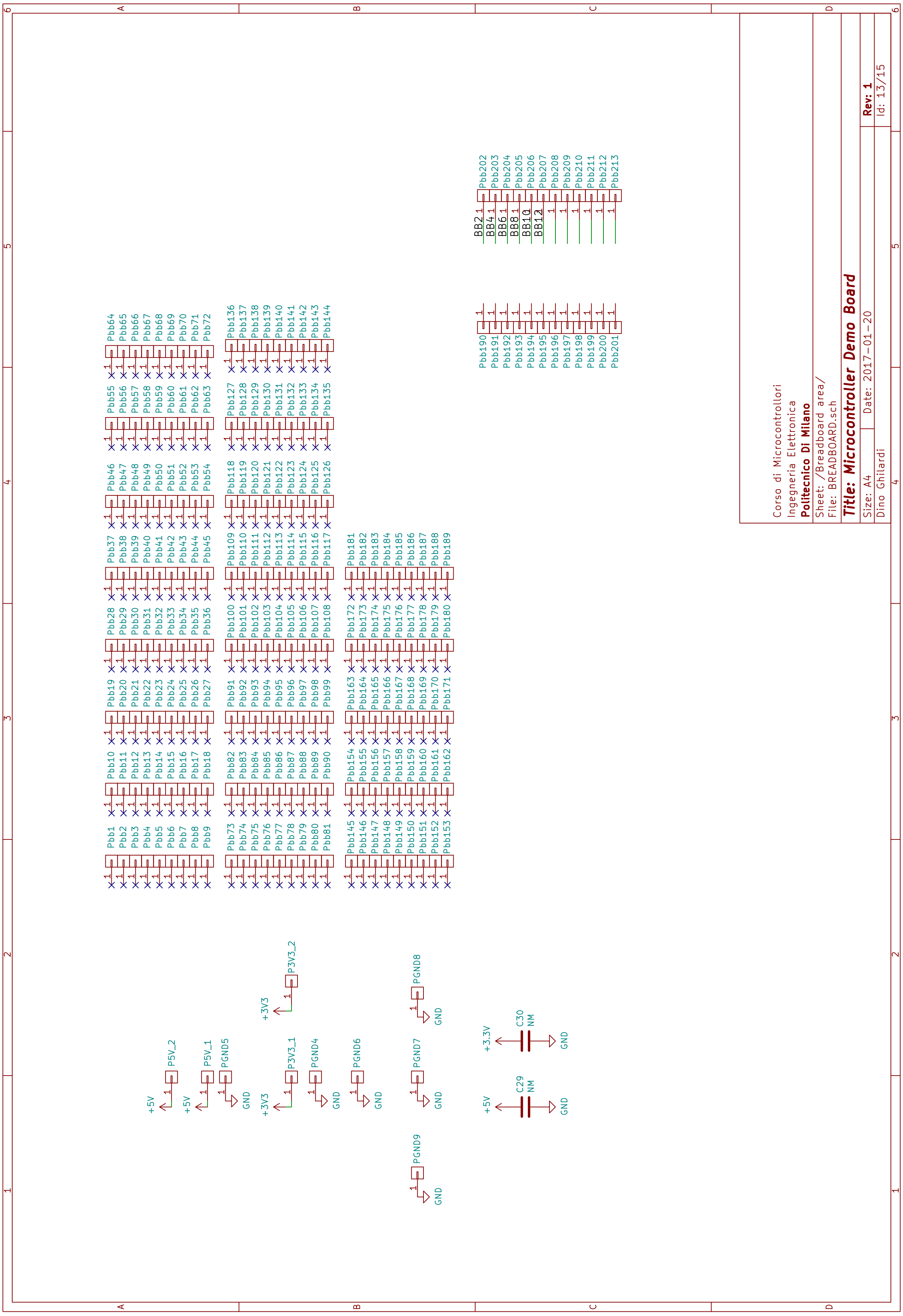
BOARD VERSIONS	U10 C20	U9 C18	P6 or P9	P10, P11	P16/P17	R106	R84
Arduino	MOUNT	MOUNT	SHORT	NM	SHORT	NM	1K
ST Nucleo	NM	NM	NM	SHORT	NM	1K	NM
Configurable	MOUNT	MOUNT	SHORT for SPI Column Drive	NM	SHORT for SPI Column Drive	1K	1K

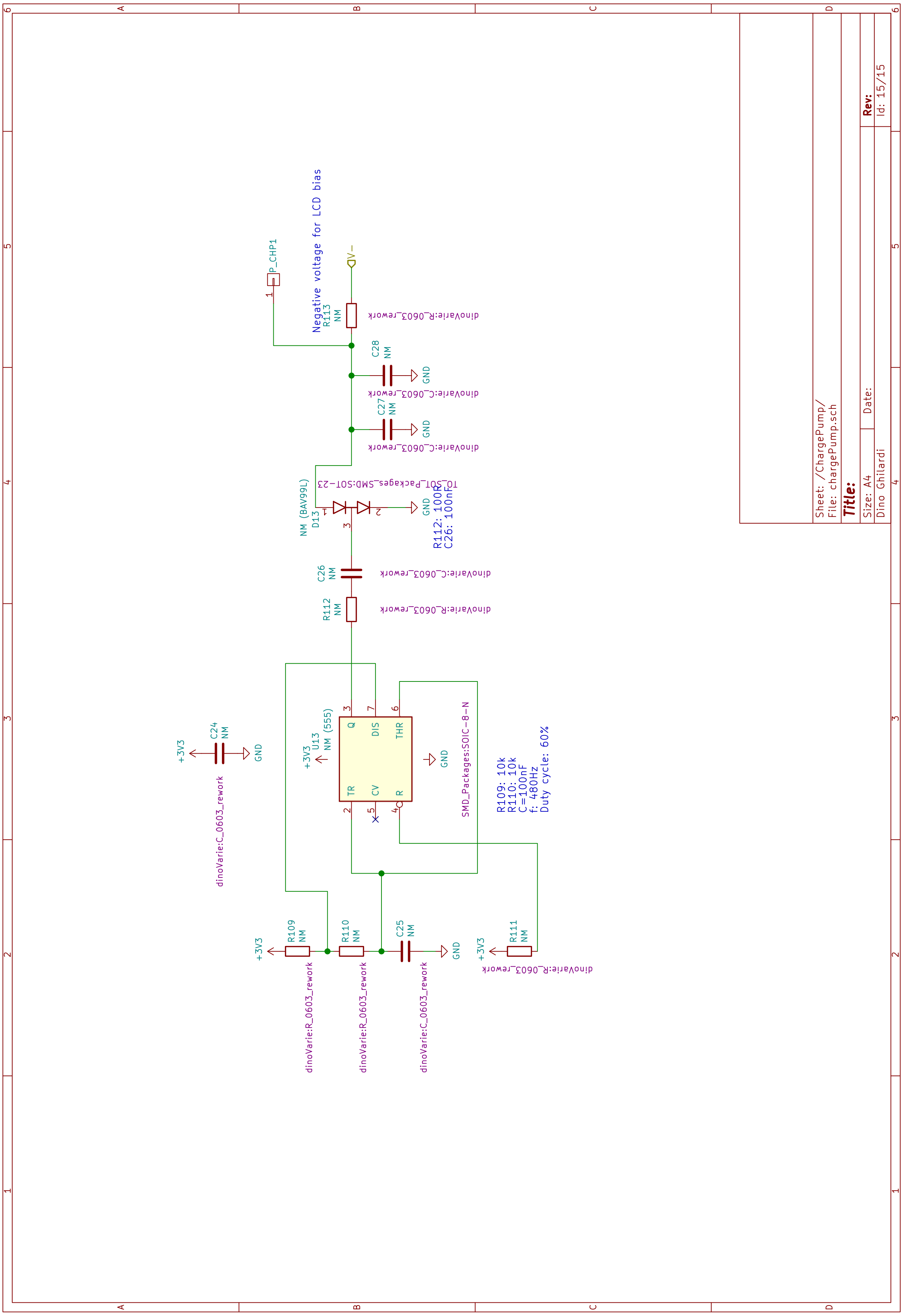

Corso di Microcontrollori
Ingegneria Elettronica
Politecnico Di Milano
Sheet: /Development Board Connections/
File: DevBoard_Conn.sch
Title: Microcontroller Demo Board
Size: A4
Date: 2017-01-20
Rev: 1
Id: 12/15
Dino Ghilardi
USART 6 to USB/SERIAL adapter
ENCODER INPUT
SPI TESTPOINTS
MORPHO LCD CONNECTIONS
RGB LED DRIVE
I2C1 to LM75 and accelerometer
Arduino TX and RX to USB/SERIAL ADAPTER
STMorpho connectors
ARDUINO Connections
Led LD2 on Nucleo board
SPI1 su nucleo
CN10
CN5
CN9
CN6
CN8
CN7
BT1
dino_connectors:CR2032Holder
Keyboard ST Morpho configuration
USER button on Nucleo Board
Analog in
I2C0 CONNECTOR
LCD_BL
GND
+3V3
+5V

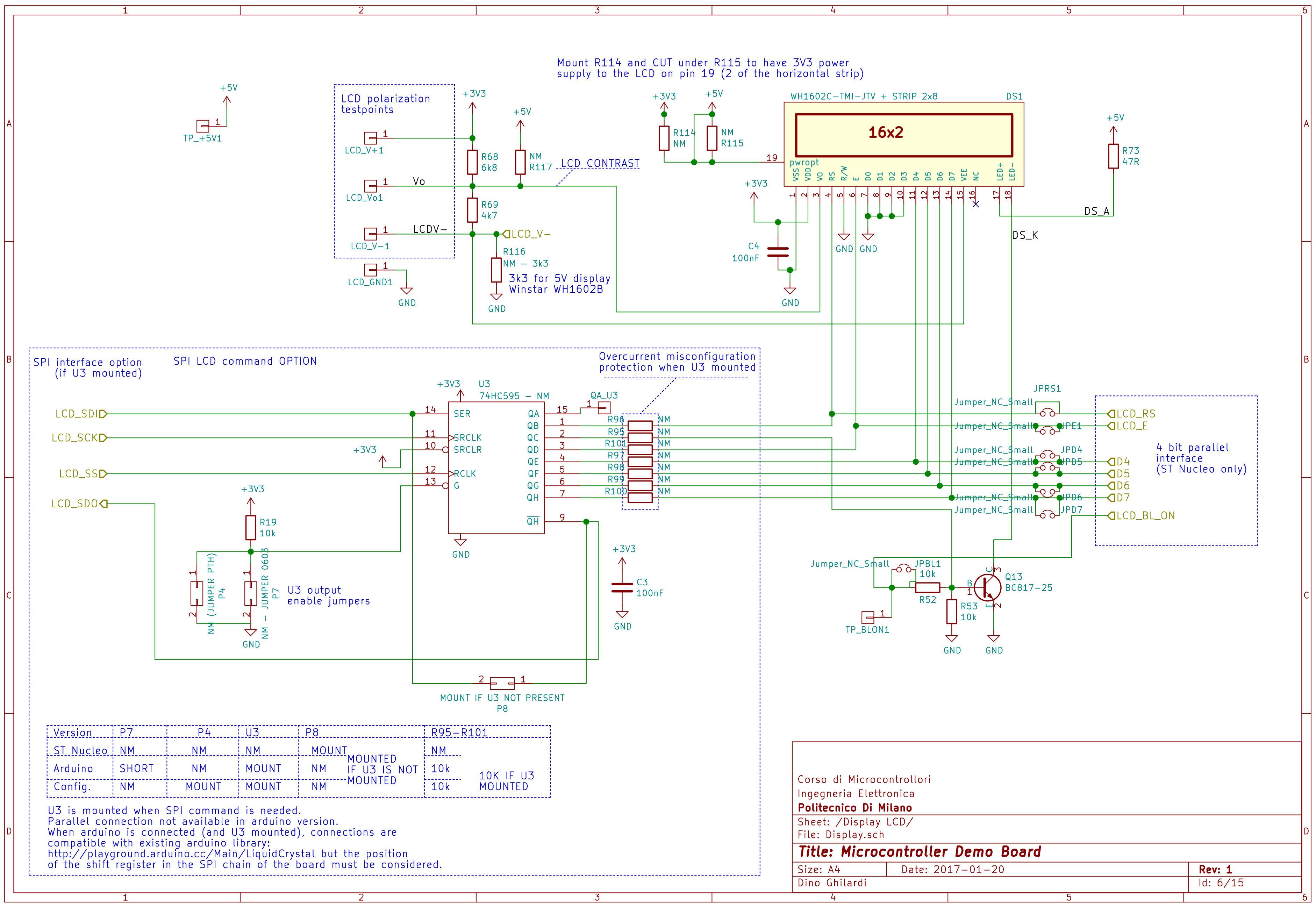

C21
10uF
C15
100nF
GND
+3V3
U1
LIS2DE
GND-NC
GND-NC
GND
VDD
VDD_IO
INT1
INT2
Res
Res
SCL/SPC
SDA/SDI/SDO
SDO/SA0
CS
ACC_INT2
ACC_INT1
PINT2
PINT1
ACC_SA0
R67
10k
SCL
SDA
Corso di Microcontrollori
Ingegneria Elettronica
Politecnico Di Milano
Sheet: /Accelerometer/
File: accelerometer.sch
Title: Microcontroller Demo Board
Size: A4
Date: 2017-01-20
Rev: 1
Dino Ghilardi
Id: 11/15


LDR Voltage Testpoints
TPLDR1
TPGNDLDR1
GND
ADC_LDR
1k
R93
+3V3
R39
100k
R40
Photoresistor
GND
Potentiometer Voltage Testpoints
TP_POT1/PA1
TP_POTGND1
GND
ADC_POT
Overcurrent protection
1k
R77
C1
100nF
GND
+3V3
RV1
Corso di Microcontrollori
Ingegneria Elettronica
Politecnico Di Milano
Sheet: /Analog in/
File: AnalogIn.sch
Title: Microcontroller Demo Board
Size: A4
Date: 2017-01-20
Rev: 1
Dino Ghilardi
Id: 3/15


SND_DETECT
R80
1k
R107
NM
+3V3
U6
MIC7211
+3V3
GND
C19
100nF
+3V3
GND
C12
100nF
GND
+3V3
R61
1M
R62
10k
R61 changed to 1M to make it
less sensible to switching noise
R60
100k
GND
C11
100nF
R59
10k
+5V
GND
P5
Microphone (elettrete)
ADD FILTER ON +5V for microphone
switching noise on board could enable the comparator.
SND Detect
Testpoints
TPSND1
TPSND2
TPGNDSND1
GND
Corso di Microcontrollori
Ingegneria Elettronica
Politecnico Di Milano
Sheet: /Audio detect/
File: AUDIO_IN.sch
Title: Microcontroller Demo Board
Size: A4
Date: 2017-01-20
Rev: 1
Dino Ghilardi
Id: 9/15


Corso di Microcontrollori
Ingegneria Elettronica
Politecnico Di Milano
Sheet: /Breadboard area/
File: BREADBOARD.sch
Title: Microcontroller Demo Board
Size: A4
Date: 2017-01-20
Rev: 1
Dino Ghilardi
Id: 13/15
P5V_2
P5V_1
PGND5
P3V3_1
P3V3_2
PGND4
PGND6
PGND7
PGND8
PGND9
+5V
+3V3
GND
C29
C30
NM
+3.3V
BB2
BB4
BB6
BB8
BB10
BB12


Negative voltage for LCD bias
V-
P_CHP1
R113 NM
C28 NM
C27 NM
GND
dinoVarie:R_0603_rework
dinoVarie:C_0603_rework
D13 NM (BAV99L)
TO_SOT_Packages_SMD:SOT-23
R112: 100R
C26: 100nF
C26 NM
R112 NM
U13 NM (555)
+3V3
Q
DIS
THR
TR
CV
R
SMD_Packages:SOIC-8-N
R109: 10k
R110: 10k
C=100nF
f: 480Hz
Duty cycle: 60%
C24 NM
R109 NM
R110 NM
C25 NM
R111 NM
Sheet: /ChargePump/
File: chargePump.sch
Title:
Size: A4
Date:
Rev:
Dino Ghilardi
Id: 15/15



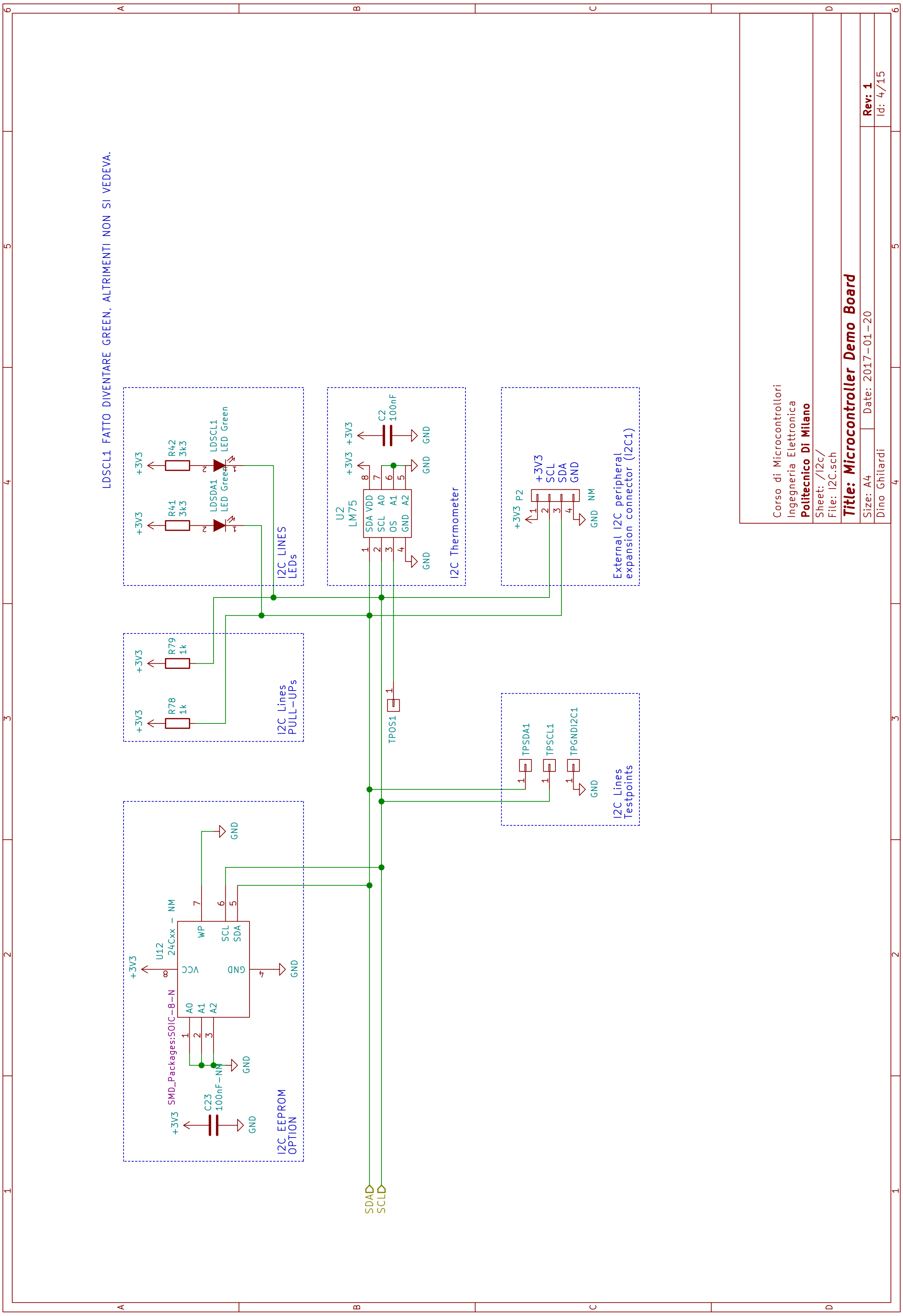
Version	P7	P4	U3	P8	R95–R101
ST Nucleo	NM	NM	NM	MOUNT	NM
Arduino	SHORT	NM	MOUNT	NM	10k
Config.	NM	MOUNT	MOUNT	NM	10k

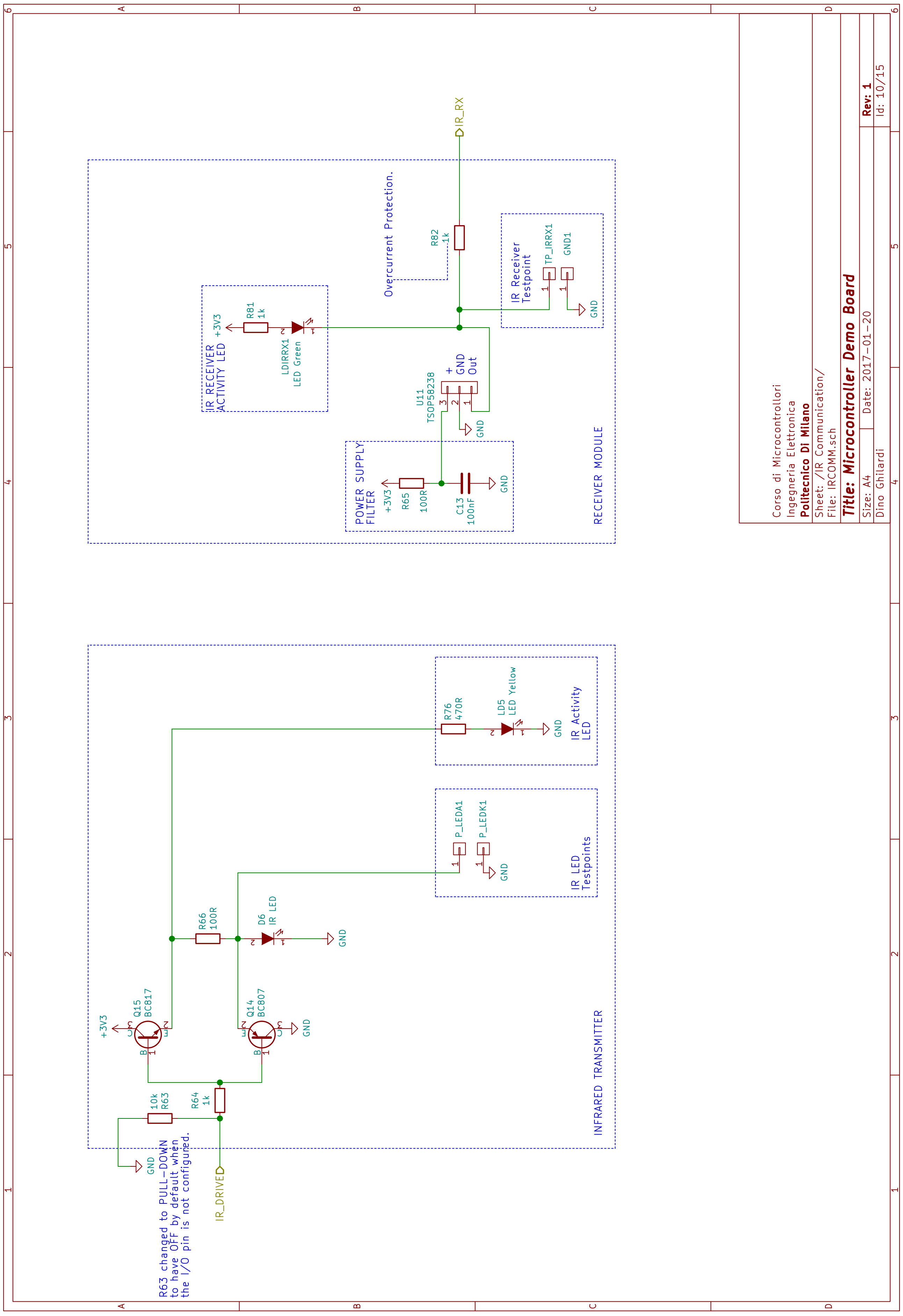
MOUNTED IF U3 IS NOT MOUNTED

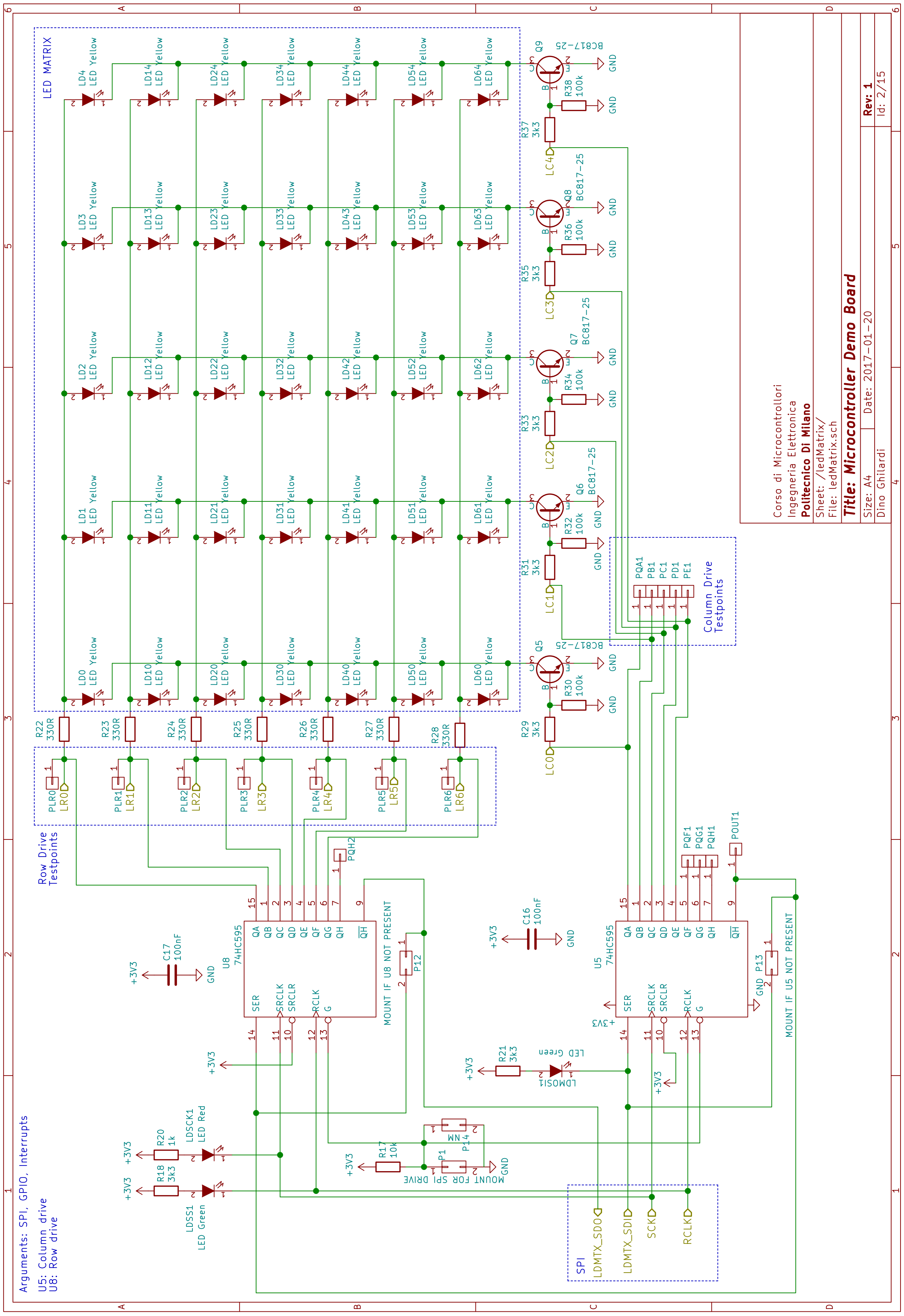
10K IF U3 MOUNTED

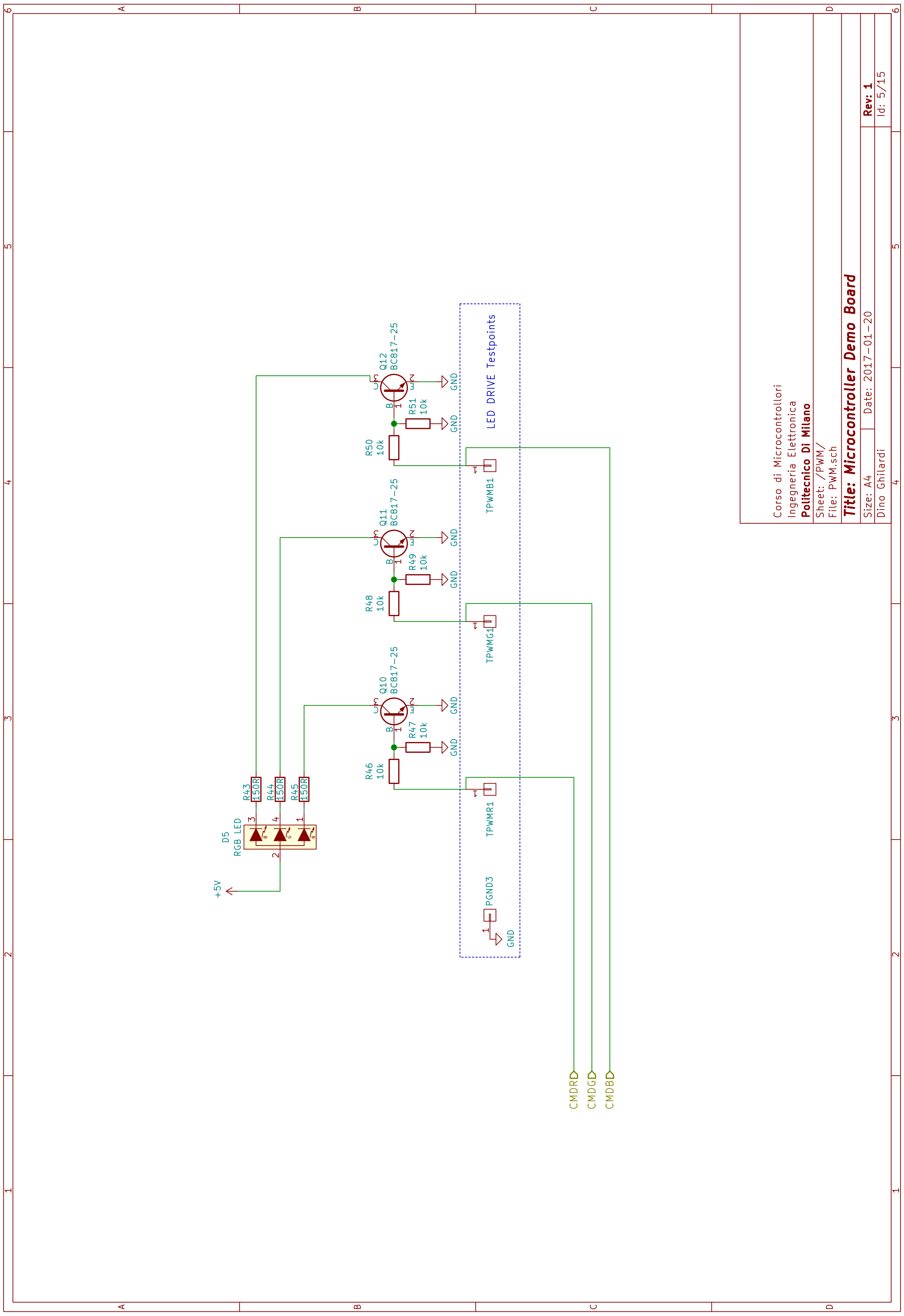
U3 is mounted when SPI command is needed.
Parallel connection not available in arduino version.
When arduino is connected (and U3 mounted), connections are compatible with existing arduino library:
http://playground.arduino.cc/Main/LiquidCrystal but the position of the shift register in the SPI chain of the board must be considered.

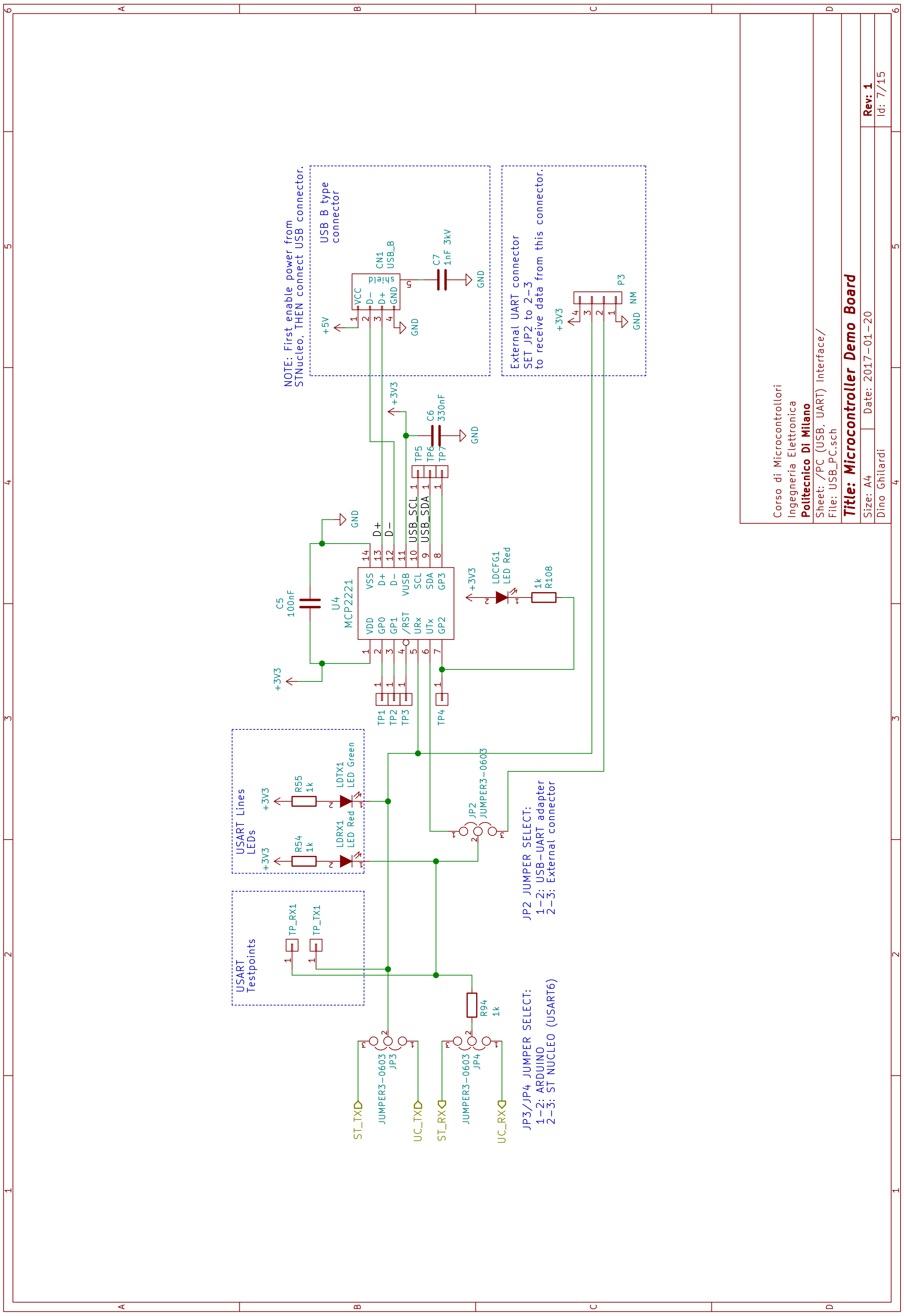
Corso di Microcontrollori
Ingegneria Elettronica
Politecnico Di Milano
Sheet: /Display LCD/
File: Display.sch
Title: Microcontroller Demo Board
Size: A4 Date: 2017-01-20 **Rev: 1**
Dino Ghilardi Id: 6/15


LDSCL1 FATTO DIVENTARE GREEN, ALTRIMENTI NON SI VEDEVA.
+3V3
R42
3k3
LDSCL1
LED Green
R41
3k3
LDSDA1
LED Green
I2C LINES
LEDs
U2
LM75
SDA VDD
SCL A0
OS A1
GND A2
C2
100nF
GND
I2C Thermometer
P2
+3V3
SCL
SDA
GND
NM
External I2C peripheral
expansion connector (I2C1)
R79
1k
R78
1k
I2C Lines
PULL-UPs
TPOS1
TPSDA1
TPSCL1
TPGNDI2C1
I2C Lines
Testpoints
U12
24Cxx - NM
SMD_Packages:SOIC-8-N
WP
SCL
SDA
VCC
GND
A0
A1
A2
C23
100nF-NM
I2C EEPROM
OPTION
SDA
SCL
Corso di Microcontrollori
Ingegneria Elettronica
Politecnico Di Milano
Sheet: /I2c/
File: I2C.sch
Title: Microcontroller Demo Board
Size: A4
Date: 2017-01-20
Rev: 1
Dino Ghilardi
Id: 4/15


IR_RX
Overcurrent Protection.
R82
.1k
IR RECEIVER
ACTIVITY LED
+3V3
R81
1k
LDIRRX1
LED Green
IR Receiver
Testpoint
TP_IRRX1
GND1
GND
U11
TSOP58238
+
GND
Out
GND
POWER SUPPLY
FILTER
+3V3
R65
100R
C13
100nF
GND
RECEIVER MODULE
R76
470R
LD5
LED Yellow
GND
IR Activity
LED
P_LEDA1
P_LEDK1
GND
IR LED
Testpoints
R66
100R
D6
IR LED
GND
+3V3
Q15
BC817
Q14
BC807
GND
10k
R63
R64
1k
GND
IR_DRIVE
INFRARED TRANSMITTER
R63 changed to PULL-DOWN
to have OFF by default when
the I/O pin is not configured.
Corso di Microcontrollori
Ingegneria Elettronica
Politecnico Di Milano
Sheet: /IR Communication/
File: IRCOMM.sch
Title: Microcontroller Demo Board
Size: A4
Date: 2017-01-20
Rev: 1
Dino Ghilardi
Id: 10/15


LED MATRIX
Row Drive Testpoints
Column Drive Testpoints
Arguments: SPI, GPIO, Interrupts
U5: Column drive
U8: Row drive
MOUNT IF U8 NOT PRESENT
MOUNT IF U5 NOT PRESENT
MOUNT FOR SPI DRIVE
SPI
LDMTX_SDO
LDMTX_SDI
SCK
RCLK
U8 74HC595
U5 74HC595
Corso di Microcontrollori
Ingegneria Elettronica
Politecnico Di Milano
Sheet: /ledMatrix/
File: ledMatrix.sch
Title: Microcontroller Demo Board
Size: A4
Date: 2017-01-20
Rev: 1
Dino Ghilardi
Id: 2/15


LED DRIVE Testpoints
Q12
BC817-25
Q11
BC817-25
Q10
BC817-25
R51
10k
R50
10k
R49
10k
R48
10k
R47
10k
R46
10k
R43
150R
R44
150R
R45
150R
D5
RGB LED
+5V
GND
TPWMB1
TPWMG1
TPWMR1
PGND3
CMDR
CMDG
CMDB
Corso di Microcontrollori
Ingegneria Elettronica
Politecnico Di Milano
Sheet: /PWM/
File: PWM.sch
Title: Microcontroller Demo Board
Size: A4
Date: 2017-01-20
Rev: 1
Dino Ghilardi
Id: 5/15


NOTE: First enable power from STNucleo, THEN connect USB connector.
USB B type connector
CN1 USB_B
C7 1nF 3kV
External UART connector
SET JP2 to 2-3 to receive data from this connector.
P3
U4 MCP2221
C5 100nF
C6 330nF
LDCFG1 LED Red
R108 1k
USB_SCL
USB_SDA
TP1 TP2 TP3 TP4 TP5 TP6 TP7
USART Lines LEDs
LDTX1 LED Green
LDRX1 LED Red
R55 1k
R54 1k
USART Testpoints
TP_RX1 TP_TX1
JP2 JUMPER3-0603
JP2 JUMPER SELECT: 1-2: USB-UART adapter 2-3: External connector
R94 1k
JP3 JUMPER3-0603
JP4 JUMPER3-0603
JP3/JP4 JUMPER SELECT: 1-2: ARDUINO 2-3: ST NUCLEO (USART6)
ST_TX UC_TX ST_RX UC_RX
+5V +3V3 GND
Corso di Microcontrollori
Ingegneria Elettronica
Politecnico Di Milano
Sheet: /PC (USB, UART) Interface/
File: USB_PC.sch
Title: Microcontroller Demo Board
Size: A4 Date: 2017-01-20 Rev: 1
Dino Ghilardi Id: 7/15

...and thousands of new exciting applications and hands-on experiments coming soon...