Manual ARM Microcontroller Course 2017 E.T.S.V. Scintilla

Cursuscommissie

March 22, 2017

Contents

Chapter 1

Introduction

Intro

Welcome to the ARM microcontroller course 2017 of ETSV Scintilla. In this course we hope to teach you some of the basics of programming a microcontroller, the common caveats and the fun of getting a mixed hardware and software project to work. We've chosen for an ARM microcontroller as these are more and more common, with a lot of chip manufacturers having their own implementations. The goal is to build an audio recorder and player using a Nucleo-F446RE board, made by ST, and a shield with auxiliary hardware. In order to build this product, we'll talk about using digital input and outputs, analog peripherals, SPI, I2S and SDIO. Using this, you should be able to design your own audio recorder by the last evening.

This course has been made possible at low cost with aid from our sponsors. We would like to thank STMicroelectronics for supplying the Nucleo F446RE development boards, Eurocircuits for the PCBs, Cirrus Logic for their audio codecs and microphones and Molex for the SD connector.

1.1 What is a microcontroller

A microcontroller is a programmable chip, based around a processor. This chip can be programmed with your own code. You are thus able to control the behavior of the chip by writing software. This makes microcontrollers incredibly versatile, reconfigurable and highly suitable to implement complex digital behaviors. Microcontrollers can be found in almost all electronics around us: from your alarm clock to your cellphone and your car.

Apart from this processor, microcontrollers can have many other peripherals. These peripherals are built-in hardware blocks that can be read by and controlled from the processor. Some example peripherals are general purpose input and output pins (GPIO), analog-to-digital converters, digital communication busses and many more. Microcontrollers come in many different types with different clock speeds, memory sizes, types and amounts of peripherals and processor architectures.

1.2 Introduction to the project

In this course, we will use the STM32F446RE microcontroller. This microcontroller is part of the ARM M4 family of microcontrollers, designed around an ARM core processor. For a microcontroller, this is a relatively powerful device. In this course, we will use this microcontroller to build an audio player and recorder.

in the final project, he microcontroller will communicate with a WM8731 audio codec to read out microphone data and store it internally or on an SD card. The audio can then be played back over your headphones using the codec. Additional buttons and a potentiometer allow you to control the final device.

A coarse planning[1] is as follows

1. Setting and LED and reading button input using GPIO
2. Reading analog values using the ADC
3. Controlling the codec over SPI
4. Sending audio between the codec and the microcontroller over I2S
5. Storing audio on the SD card and completing the audio recorder

[1]Changes may occur

1.3 Memory mapped I/O

The peripherals in the microcontrollers are based on the concept of memory mapped IO. This means that there are no separate instructions to control the peripherals, but rather they have setting registers, part of the microcontroller memory map. A register is nothing more than a location at a certain address in the device memory. For example, you can configure the microcontroller to output the contents of the GPIOA_ODR register (general purpose input/output port A - output data register) to pins PA0-PA15. When you write data to this position in the memory, the logic values in these memory cells connect to output pins and allow for digital control of the output pins.

The other way around, the GPIOA_IDR register (general purpose input/output port A - input data register) can be read as if it were a normal memory cell in the device memory, but instead of connecting to a memory cell, it connects to a microcontroller pin. This allows to read input pins as if they were variables in the microcontroller memory.

From here it is a small step to extend the concept of memory mapped IO to further use. It can be used to control peripherals by writing digital control settings to specified registers and reading ADC values, incoming data, etc. from the peripherals. All of this by setting and reading registers as if they were variables in the device memory.

Chapter 2

C Projects

A C project usually consists of several source files (.c files) and headers (.h files), to improve the structure of the project and make code reusable. Usually, each source file has a corresponding header file, while header files also often exist separately.

2.1 Headers (.h files)

Header files declare various data types (for example typedefs and public function prototypes) and macros (for example register addresses). This also means that header files have no function at all without being included in a source file, since these declarations describe no memory content. Header files are included in another file using an include statement, and can be included in a path relative to the source file the include statement is in, or from the system standard includes.

Snippet 2.1 Example of an include statement

```c
// Include from a path relative to the source file
#include "userheader.h"
// Include from the system header path
#include <systemheader.h>
```

2.1.1 Public function prototypes

A function prototype is the definition of a function, but without its actual content. Thus, it declares the existance of the function, but not its behavior. This enables a program to use functions that are not defined yet, for example when some function calls another function that is only declared later in the source file, or even in another file, which can be very practical to limit file and improve project structure. Header files that contain public function prototypes need corresponding source files (usually with the same filename, except for the extension) that actually define the functions. Function prototypes in headers are called public function prototypes since the corresponding functions can be used in any source file that includes the corresponding headers.

Snippet 2.2 Example of a function prototype

```c
// Function prototype, note the ;
uint16_t round_sixteen_bit(uint32_t input);
```

2.1.2 Defines

When using header files, one might want to include one header file from another header file (usually because the source file corresponding with one header file needs functions from a source file corresponding with another header file). This way, it is possible that a header is included multiple times, which would make the code impossible to compile, since this means the content from a header file is included multiple times, which would redefine everything in this header file. To work around this, a define statement can be used at the start of a header file. Then, an ifdef statement is used to test if this header file was already included (and an ifndef statement for not included). In more complicated software, the functionality of some part of code can also be changed depending on some definition, in a comparable way. A define statement can also define some value (instead of just its existence), like in a macro.

Snippet 2.3 Example of a define statement

```c
// Test if already defined
#ifndef _HEADER_NAME_H
#define _HEADER_NAME_H
    // Header file contents
// Final header_name_h for clarity in long ifdef statements
#endif /* _HEADER_NAME_H */
```

2.1.3 Macros

A macro is a define that defines some part of code, which is usually easier to remember than the actual code. This also adds some flexibility to the code functionality, since they can be edited before compiling (for example, to change the pin an LED is connected to). Another use in this course is the the definition the microcontorller register map, so one does not need to remember the 32 bit (for the microcontroller used in this course) hexadecimal addresses they correspond to (like 0xDEADBEEF, but usually an even more difficult number). The header files that descibe register maps are often provided by the manufacturer of the microcontroller, while the header files that describe what the pins are connected to, like a red LED, might be provided by the hardware designer (however we did not, for teaching purposes).

Snippet 2.4 Example of a macro

```c
// Define, now with value (a led on pin 5)
#define LED_PIN 5
#define PORT_REGISTER 0xDEADBEEF

// Use in some code, not related to this course
void led_blink() {
    gpio_blink(PORT_REGISTER, LED_PIN);
}
```

2.2 Source files (.c files)

The source files contain the actual code for a project. A single source file usually includes multiple functions that are related to eachother, and has a corresponding header file. For example, the gpio.c and gpio.h files might contain both the functions to set an IO pin direction and to read or write it.

2.2.1 Functions

Source file contain functions. Most often, these functions are already defined in a function prototype in a header. These functions are public functions. Functions may also be defined without a header, in which case thay are not usable in another source file. These functions are private functions.

2.2.2 Main source file

The main source file is simply the file that contains the main() function. This is the function that is called when the program starts. The return type of this function is int by convention, and a return value different from 0 indicates an error (in microcontroller code, a program should usually never exit, since there is only one program to run). Usually, the main source file has the same name as the project.

Snippet 2.5 Example of a main function

```c
// Standard main function
int main(void){
    // Microcontroller code usually loops
    for(;;) {
        // Some code
    }
}

// Return types, used in actual programs
int main(void){
    // Some code

    // Return something else than 0 on error
    if(test_error()) return 1;
    // Return 0 on success
    return 0;
}
```

2.3 CMSIS

CMSIS-CORE (ARM Cortex Microcontroller Software Interface Standard) provides an abstraction layer for the Cortex-M processors.

- Standardized definitions for SysTick, NVIC[1], System Control Block registers, MPU and FPU registers, and core access functions.
- System exception names (vendor-independent)
- Naming conventions for device-specific interrupts.
- System initialization functions like SystemInit() for coniguring the clock.
- Intrinsic functions used to generate CPU instructions that are not supported by standard C.
- Variables to determine the system frequency (making it easy to setup SysTick)

Files

- stm32f446re.h
- system_stm32f446re.h
- core_cm4.h
- startup_stm32f446re.h

[1]Nested Vector Interrupt Controller

Chapter 3

Programming C

Due to its low level description, C is a relatively simple language to get started with: it features only a minimum of possible operations, so there is not that much to learn.

3.1 Data Types

Every operation needs at least one variable. There are two types of variables: integer type variables (for integer numbers) and floating point variables (for non-integer numbers). In ARM microcontrollers it is strongly advised to use unsigned integer type variable with a size of 32 bits or less standard. Longer data types or floating point variables take more time to perform operations on and thus decrease performance and increase power consumption. Keep in mind that an ARM has 32 bit registers: using data types smaller than this will not save memory, but will just leave the rest of the bits in the register unused.

3.1.1 Integer Data types

The integer type variables can be found in table 3.1. This table shows that there is no Boolean type variable. To be able to process operations which require a true/false input, any other integer data type can be used. Giving the value "0" as the argument of a function will execute the same as a logic "false" and any other value will be processed as a logic "true". The code snippet in Snippet 3.1 will thus switch on an LED. The LED will switch on for every value of x, as long as it not 0.

In microcontrollers it is very common to use an integer as an array of bits. For example a 16 bits unsigned integer (uint16_t) can not only be used to store a 16 bit number, but could also be used to store 16 bits. For this we have to know how unsigned integers are stored. This type of integers is stored as a binary number. If we consider the decimal number "1234", this can be stored in binary as "0b0000010011010010". The "0b" is a marking which determines that a number is given in binary and the zeros at the beginning are added to show this is a 16 bit number. Microcontrollers often have setting registers where one bit e.g. determines if a hardware peripheral is switched on. To alter this specific bit one can use bitwise operators, which will be discussed later in this manual.

Apart from the binary "0b" notation, hexadecimal notation is very common as well. Instead of base 10 (decimal) or base 2 (binary) we now use base 16. This notation can be considered to be a short form of binary notation which groups four bits in one digit. The star of a hexadecimal number is indicated with "0x". In hexadecimal notation one can now count 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F. In this base 16, 0x0 is the same as 0b0000, 0x5 is the same as 0b0101 and 0xE is the same as 0b1110. The longer decimal number 1234 can be converted to hexadecimal by separating the binary number in groups of four digits and rewriting these groups to hexadecimal. "0b0000 0100 1101 0010" in hexadecimal notation will become "0x04D2". Hexadecimal numbers have the advantages of both binary and decimal notation: it is grouped per bit, but the notation is still short.

Snippet 3.1 Example of an integer as a Boolean

```c
uint16_t var = 1615;
if (var) {
    led_set();
} else {
    led_reset();
}
```

Table 3.1 Integer data types in C

Name	Syntax	Range	Size (bits)
Boolean	Non-existent	There are no Booleans in standard C	1
Signed character	int8_t	-128 to 127	8
Unsigned character	uint8_t	0 to 255	8
Signed integer	int16_t	$-32,768$ to $32,767$	16
Unsigned integer	uint16_t	0 to $65,535$	16
Signed long integer	int32_t	-2^{31} to $2^{31}-1$	32
Unsigned long integer	uint32_t	0 to $2^{32}-1$	32
Signed long long integer	int64_t	-2^{63} to $2^{63}-1$	64
Unsigned long long integer	uint64_t	0 to $2^{64}-1$	64

3.1.2 Floating point data types

Apart from the integer data types, non-integer or "floating point" data types can be used as well. Floating point operations are generally harder for a microcontroller to process. For example, it is harder to perform $1.23456 \cdot 10^1 + 9.87654 \cdot 10^{-2}$ than to perform $12345678 + 98765432$. This implies that floating point operations take longer to calculate and thus decrease performance. It can be concluded that it is unwise to use floating point numbers when this is not necessary. The possible floating point types can be found in Table 3.2. The long double indicated here is specified according to the "IEEE 754 quadruple-precision binary floating-point format", but implementations of the long double may differ per system.

Table 3.2 Floating point data types in C

Name	Syntax	Sign bit	Exponent bits	Fraction bits	Size (bits)
Floating point	float	1	8	23	32
Double floating point	double	1	11	52	64
Long double floating point	long double	1	15	112	128

3.1.3 Arrays

An array is an indexed lists of a certain data type. These arrays can e.g. be used to store lists of variables, as we will do later to map an output sample number to an output value. As in every properly thought out programming language, the first entry in the array is numbered "0". As an example, the example code in code snippet 3.2 generates a list of squares of the first 8 integer numbers and returns the value of the fifth square. This should be 25.

Snippet 3.2 Example code for reading and writing of an array

```c
// generate a list of 8 signed numbers of 16 bits named "y"
uint16_t list[8];

// fill the list with the squares of the index numbers
for (uint16_t index = 0; index < 8; index++) {
    list[index] = index * index;
}

// return the value of the fifth square
return list[5];
```

3.1.4 Structs

C is no object oriented language, but has a feature which comes close to object oriented storing of variables. This can be done by defining a so called "struct" type variable. A struct can be considered to be an object with several variables in it. Structs are used often where there is a clear repetition in data sets. As an example, a microcontroller has several sets of input and output pins (or "general purpose input and output (GPIO) ports", more about this later). Each GPIO port has (amongst others) a setting for pin modes. Data could be organized in a very convenient way if we could make an object "GPIO port" with as one of its internal variables a value for the pin mode for that port. In code snippet 3.3 the type definition (similar to a class) for the GPIO ports is given, a struct (similar to an object) *"GPIOA"* is created, and one of its variables is changed and returned.

Snippet 3.3 Example of defining and working with a struct

```
// give a type definition for the GPIO structs
typedef struct
{
  uint32_t MODER;    // GPIO port mode register
  uint32_t OTYPER;   // GPIO port output type register
  uint32_t OSPEEDR;  // GPIO port output speed register
  uint32_t PUPDR;    // GPIO port pull-up/pull-down register
  uint32_t IDR;      // GPIO port input data register
  uint32_t ODR;      // GPIO port output data register
  uint32_t BSRR;     // GPIO port bit set/reset register
  uint32_t LCKR;     // GPIO port configuration lock register
  uint32_t AFR[2];   // GPIO alternate function registers
} GPIO_TypeDef;

// initialize GPIO port ”GPIOA”
GPIO_TypeDef GPIOA;

//Set the variable ”MODER” in the struct ”GPIOA” to ”0x0001”
GPIOA->MODER = 0x0001;

// return the variable ”MODER” of struct ”GPIOA”
return GPIOA->MODER;
```

3.1.5 Enumerated type

An enumerated type is a limited list of keywords, using symbolic names to make a program clearer to the programmer. This data type will be useful when you want to implement a state machine, as in code snippet 3.4. The defined keywords can be used directly in code.

Snippet 3.4 Example of a state machine using enumerated type

```
//define the enumerated type States with three possible values
typedef enum
{
    start_state,
    wait_state,
    process_state
} states_t;

//declare and initialize mystate to startState.
states_t mystate = start_state;
```

3.1.6 Unions

A Union is a bit similar to the struct, but all of its elements are stored at the same memory address. As such only one element can be accessed at the same time. For microcontrollers this is useful to convert one datatype into another. An example can be found in snippet **??**.

Snippet 3.5 Example of union type

```
// the union consists of its command and its individual bytes
typedef union spicommand_t
{
    uint32_t spicommand
    uint8_t commandbytes[4];
};

// set the bytes of the spi command
spicommand_t mycommand;
mycommand.commandbytes[0] = 0xAA;
mycommand.commandbytes[1] = 0xBB;
mycommand.commandbytes[2] = 0xCC;
mycommand.commandbytes[3] = 0xDD;
// now mycommand.spicommand contains 0xDDCCBBAA
```

3.2 Operators

To perform operations on variables, operators can be used. These operators can be categorized into 4 main categories:

1. Mathematical (arithmetic) operators
2. Comparison operators
3. Logical operators
4. Bitwise operators

This section gives a brief overview of these operators.

3.2.1 Standard Operators

A list of standard mathematical operators in C can be found at: `http://en.wikipedia.org/wiki/Operators_in_C_and_C%2B%2B#Arithmetic_operators`
A list of standard comparison operators can be found at: `http://en.wikipedia.org/wiki/Operators_in_C_and_C%2B%2B#Comparison_operators.2Frelational_operators`

3.2.2 Logical Operators

Logical operator do operations on word. This means That a variable is processed as a logical "false" if its value is 0 and is processed as a logical "true" if it has any other value. A list of the possible logical operators is given in Table 3.3.

Table 3.3 List of logical operators on words

Name	Syntax	Application
Logical NOT	`!a`	Returns the logical inverse of a
Logical OR	`a || b`	Returns "true" if a, b or both are true
Logical AND	`a && b`	Returns "true" if a and b are both true

Table 3.4 shows how the operation `0b1100 && 0b0110` is performed.

Table 3.4 Processing of a logical operation on a word

a	1	1	0	0	$\rightarrow$	TRUE	
b	0	1	1	0	$\rightarrow$	TRUE	&&
					return value:	TRUE	

These operators base their input on a whole word. There are also operations which perform logical operations based in each individual bit in a word. These are called bitwise operators

3.2.3 Bitwise operators

A bitwise operator performs the logical operation not per word, but per bit. A list of possible bitwise operators is given in table 3.5

Table 3.5 List of bitwise operators

Name	Syntax	Application
Bitwise NOT	`~a`	Flips all bits in a
Bitwise OR	`a	b`	ORs the first bit of a with the first bit of b, etc.
Bitwise AND	`a & b`	ANDs the first bit of a with the first bit of b, etc.
Bitwise exclusive OR	`a ^ b`	XORs the first bit of a with the first bit of b, etc.
Bitwise left shift	`a << b`	Shifts the bits in a b places to the left
Bitwise right shift	`a >> b`	Shifts the bits in a b places to the right

Table 3.6 shows how the operation `0b1100 & 0b0110` is performed. This shows that the logical and operation is performed for every column and not for the whole word. This will later prove very useful for reading, setting and clearing specific bits.

Table 3.6 Processing of a bitwise operator

a	1	1	0	0	
b	0	1	1	0	&
return value:	0	1	0	0	

3.2.4 Compound assignment operators

It is very common to perform bitwise operation where a certain variable is both one of the arguments, as well as the location to store the result of the operation. For this a shortened form called a "compound assignment operator" can be used. This allows to for shorter code over which the programmer has a better overview. For example the logical operations in code snippet 3.6 perform exactly the same operation. A full list of these operators can be found at http://en.wikipedia.org/wiki/Operators_in_C_and_C%2B%2B#Compound_assignment_operators

Snippet 3.6 Example of compound statements

```
// initializing the variables
uint8_t var_x = 0b01010101;
uint8_t var_y = 0b00001111;

// performing a bitwise operation on variable x
var_x = var_x & var_y;

// performing the same operation again using a compound assignment operator
var_x &= var_y;
```

An example which uses a lot of these bitwise operators is the resetting of a specific bit. It might sound simple to set a single "1" to a "0", but takes quite some steps to clear bit 5 as is Table 3.7. To perform this operation, the code in code snippet 3.7 is used. This code is explained step by step in table 3.8.

Table 3.7 Example of clearing of a single bit

Bit number	7	6	5	4	3	2	1	0
Current value of x	1	0	1	0	1	0	1	0
Desired value of x	1	0	0	0	1	0	1	0

Snippet 3.7 Example of the clearing of a single bit in a register

```
// initializing the variables
uint8_t var_x = 0b10101010;
uint8_t bit_to_clear = 5;

// clearing the bit to clear
var_x &= ~(1 << bit_to_clear);
```

Table 3.8 Step by step explanation of the code in snippet 3.7

#	Description	Substituted code
1	Original statement	`x &= ~(1 << bit_to_clear);`
2	Rewriting &= to the full form	`x = x & ~(1 << bit_to_clear);`
3	Substituting x and bitToClear	`x = ~(1 << 5);`
4	Rewriting 1 to binary	`x = 0b10101010 & ~(0b00000001 << 5);`
5	Performing the bitshift between brackets	`x = 0b10101010 & ~(0b00100000);`
6	Performing the bitwise NOT operation	`x = 0b10101010 & 0b11011111;`
7	Performing the AND operation	`x = 0b10001010;`

3.3 Statements in C

To build logical blocks with these operators, statements are added to determine when and how to perform the logical operations. This can be done using control statements. This section will cover some basic control statements.

3.3.1 Conditional Statements

There are two types of conditional statements: if statements and switch cases. In this manual we assume you are familiar with if statements. The syntax for a C if statement is as given in code snippet 3.8, both with or without the "else" statement.

Snippet 3.8 Example of an if statement with and without an else clause

```c
int16_t var_x = -3;
int16_t var_y;

// set y as the absolute value of x
if (var_x > 0){
    var_y = var_x;
} else {
    var_y = - var_x;
}

// set x to its absolute value
if (var_x < 0) {
    var_x = - var_x;
}
```

Alternative to the if statement, a switch statement can be used. The given argument determines to which line in the statement the program will jump. In code snippet 3.9 an example of a morning routine is given. If you wake up on time, there is time to take a shower and have breakfast, if there is little time left, you will skip some steps and if your wake up very early or too late you will go back to sleep.
The same routine could be realized using if statements, but in many cases the switch statement is more insightful and convenient. Switch statements are extremely useful to implement state machines and execute some code depending on a state variable.

Snippet 3.9 Example of a switch statement

```c
// variable for time until your lecture starts in quarters of an hour
int16_t time_to_lecture = 7;
int16_t next_time = 0;

// choose what to skip depending on how much time you have
switch(time_to_lecture){
    case 4:
        next_time = 3;
    break;
    case 3:
        // if you have 3 quarters of an hour start by taking a shower
        take_a_shower();
        next_time = 2;
    break;
    case 2:
        // if you have 2 quarters of an hour have some breakfast
        have_breakfast();
        next_time = 1;
    break;
    case 1:
        // leave directly if there is only on 1 quarter of an hour left
        leave_for_lecture();
        next_time = 0;
    break;
    default:
        // in the case that the time until the lecture is more than 4 quarters of an hour or less
            than 1 (0 or negative), go back to sleep.
        go_back_to_sleep();
        next_time = time_to_lecture -1;
        // end of the default routine
    break;
}

// update time
time_to_lecture =  next_time;
```

3.3.2 Iteration Statements

C knows two types of iterations: for loops and while loops. It is assumed that the reader knows how to work with these loops. The syntax for these loops is given in code snippet 3.10 and code snippet 3.11.

Snippet 3.10 Example of a for loop

```c
// perform a piece of code 10 times
for (uint16_t index=0; index < 10; index++) {
    //write the code to loop here
}
```

Snippet 3.11 Example of a while loop

```c
// blink an LED as long as a button is pushed
while ( button_read() ) {
    led_blink();
}
```

3.4 Functions

If code is to be used multiple places, it is advised to make functions of these blocks of code. This manual assumes you know what functions are. The C syntax for functions is as given in code snippet 3.12. Use a reference to the function inside the `main()` scope and the code in the function block will be executed. The declaration of a function needs to be before the first call of the function. To do that you can use function prototypes, where you specify the name, return type and number and types of the arguments.

Snippet 3.12 Example of a function

```c
// function prototype of the function multiply
int16_t multiply(int16_t, int16_t);

int main() {
    // define some variables
    int16_t var_a = 4;
    int16_t var_b = 5;

    // calculate the value for c using the multiply function
    int16_t c = multiply(var_a, var_b);
}

// definition of the function multiply
int16_t multiply (int16_t var_x, int16_t var_y) {
    return var_x * var_y;
}
```

Chapter 4

Starting a Project

To accompany this guide there is a video walk-through available on https://www.youtube.com/watch?v=HxGEBEWRyy8
In Eclipse go to **File** → **New C Project**. Enter a name for the project, and select *STM32F4xx C/C++ Project* from the project type dropdown. From Toolchains choose the *Cross ARM GCC*.
In the next menu, for the Nucleo-F446 in the Target processor needs to be changed to *STM32F446xx*, the flash size to 512KB, and Content: Empty.
Keep the suggested settings in the Folders menu and Select Configurations menu.
In the Cross GNU ARM Toolchain make sure to select the Toolchain **GNU Tools for ARM Embedded Processors**, and if necessary locate the *bin* folder of the toolchain. Click Finish. Now a simple project is set up with an empty `main()` function.
To make optimal use of our Eclipse installation, right-click on the project, and select Properties. Browse to **C/C++ Build** → **Settings** → **Devices**. Locate the STM32F446RE, select it and click Okay. You can now test the installation by pressing the **Build** icon.

4.0.1 Flashing the Program Memory

To load the binary onto the microcontroller, the STLink Utility can be used[1], as well as a debugger.

STLink Flash under Linux

1. In Eclipse, go to Project Properties → C/C++ Build → Settings → Build Steps
2. Add the command to Post-build steps, and give it a description *Create Binary*:
   `arm-none-eabi-objcopy -S -O binary "${ProjName}.elf""${ProjName}.bin"`
3. Now you can run st-flash by clicking Run → External Tools → stlinkv2

4.0.2 Debugging

This part assumes you installed OpenOCD, as in section A.1.4 for Windows or A.2.2 for GNU/Linux.

1. Build the project and make sure the executable file exist.
2. Go to Run → Debug Configurations...
3. Select the **GDB OpenOCD Debugging** group and click New.
4. In the new configuration, the Main tab should already be filled in. Click on the Debugger tab.
5. Add the following line to the Config Options
   -f board/st_nucleo_f4.cfg
6. Click on the Common tab, and select *Shared file*
7. Click Apply and Close.

[1]Fundamentally this could also be used on windows, but we did not include this in this course, if you want to find out how this works, look at the manual of the 2015 course

Chapter 5

GPIO

5.1 IO settings

The primary interface of the microcontroller with the outside world are the general purpose input/output pins (GPIO). The hardware which describes the implementation of this functionality for the STM32 microcontrollers is shown in figure 5.1. In this image you can see that a GPIO pin is both rather reconfigurable and multifunctional. Some GPIO pins can have alternate functionality, such as for instance an ADC, a communication peripheral, or a timer in-/output. This functionality can be selected through setting the following registers.[1] The GPIO pins are distributed over GPIO ports. Each port contains 16 pins, but not each pin is connected to the outside world, as that depends on the chip package.

- MODER - This is the mode register, in which the functionality of the IO pin can be selected.[2]
- OSPEEDR - This is the output speed regiter, this register controlls the speed of the IO pin.
- OTYPER - This allows to select the IO drive mode, either PP (push-pull) or OD (open-drain)
- PUPDR - This register can enable a pull-up or pull-down resistor, to tie a certain pin high or low.
- IDR - Input data register, this register allows one to read the state of the GPIO input.
- ODR - Output data register, this can be used to set the output state of the IO pin.
- BSRR - The IO set or reset register, this could be used to change the output state of selected IO pins within a port in one clock cycle.
- AFRH & AFRL - The alternate function select registers, these can be used to select the alternative functionality of the IO pin. The AFRH register does this for the upper 8 pins of the port and the AFRL for the lower 8 pins.

[1]See STM32F446_Complete.pdf on page 174.
[2]See STM32F446_Complete.pdf on page 184.

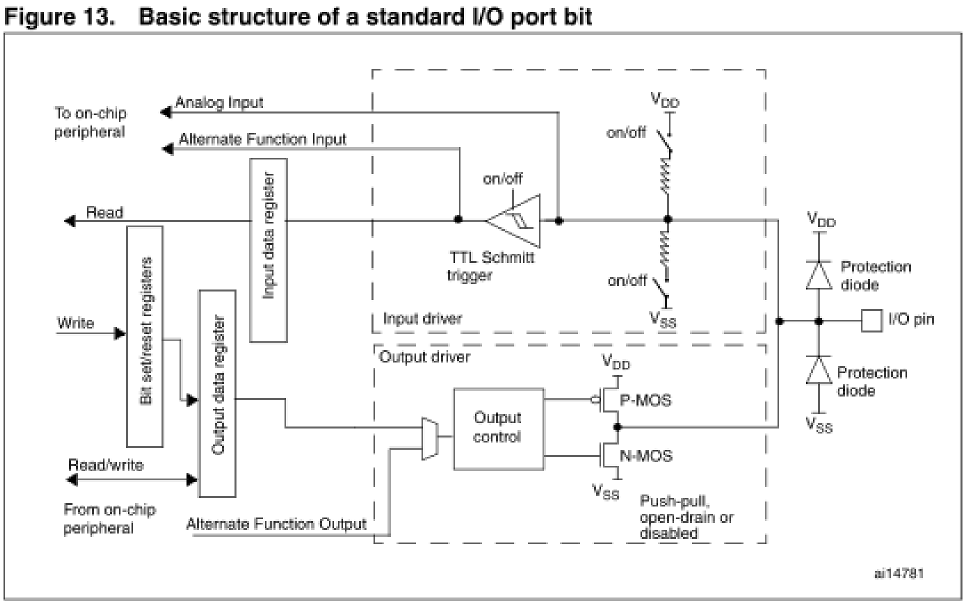

Figure 5.1: The GPIO as used in STM microcontrollers.

5.2 Exercises

5.2.1 Setting up the clock

In order to actually be able to write to the GPIO registers, the GPIO clock domain **needs** to be enabled. This should be done by enabling the clock for required GPIO port, by setting bits in the AHB1ENR register.[3]

Exercise 1: turning on an LED

The first exercise is the getting a so called ”blinky” to work. For this excercise, we are going to write a c library for the header file in 5.1. This header specifies the names for the functions which will be implemented. Step by step, we will write code in the corresponding file gpio.c, implementing these functions. The header file can also be found at: `https://www.scintilla.utwente.nl/docs/cursus/MicrocontrollerCourse2017/ProjectHeaders`

The steps to do this are as follows:

1. Start a new C project and add the gpio.h header file
2. Create a library file for the GPIO: gpio.c
3. In the gpio library, include the corresponding header (”gpio.h”) and the library containing the microcontroller’s register definitions (<stm32f4xx.h>)
4. In the main file, include ”gpio.h”
5. In the init function, enable the clock domain for the GPIO port of interest and setup the mode of the GPIO pins of interest.
6. In the on function, write the code to turn on the red LED.
7. In the off function, write the code to turn off the red LED.
8. In the main function, call the init function
9. In the while(1) loop in the main function, call the on function, wait and call the off funtion

To find what connections on the shield are connected to which pins, the schematic of the shield is included in appendix B.

Important: use open drain mode (red LED stays on dim in push-pull mode).
Please try to write the functions for setting up an LED and switching it on/off, in such a way that they implement the function protypes as given in **gpio.h**. This header file can be extended to support other colors.

[3]See STM32F446_Complete.pdf page 141

Exercise 2: reading a button input

To make the functionality of our program somewhat more interesting, we will now add color changing functionality to the code.[4] The colors that can be produced by the LED are:

- RED
- YELLOW
- GREEN
- WHITE
- CYAN
- BLUE
- MAGENTA

We build on top of the previous code:

1. Read the state of the knobs.
2. If a knob was pressed, change the color.

Note: Configure the IO pin connected to the knob in such a way that is actually able to detect if it is being pressed. (Hint: examine the schematic and figure 5.1, to find how to do this.)
Write code to read out a button. Pull-up resistors need to be configured to do this. Make functions for initializing the button pins and reading a button, anticipating for libraries.

5.2.2 BONUS: Pin change interrupts

An issue which you will encounter is that the colour of the blinking LED will not immediately change if you change the knob. This is due to the fact the implementation of reading a knob as suggested in the previous excercise polls the state of the knob instead of begin able to react immediately. A solution to this problem would be to use interrupts, this halts the linear processing of code, and by means of an external stimulus (a pin change), lets the processor jump to a specific piece of code.

Implement the color change using pin change interrupts.

You could use `http://www.scintilla.utwente.nl/docs/cursus/MicrocontrollerCourse2015/examples/Function_generator/user_IO.c` *as inspiration.*

[4]The led on the shield is an RGB LED

Snippet 5.1 Contents of the gpio.h header file

```c
/** @file gpio.h
*
* @brief Provides an abstraction layer to the LED(s) and Buttons.
*
* @par
*
*/

#ifndef _GPIO_H
#define _GPIO_H

#ifdef __cplusplus
extern "C" {
#endif

#include <stdint.h>

void gpio_led_red_init();
void gpio_led_red_on();
void gpio_led_red_off();
void gpio_led_red_toggle();

void gpio_button_init();
uint8_t gpio_button_0_read();
uint8_t gpio_button_1_read();


#ifdef __cplusplus
}
#endif

#endif /* _GPIO_H */
```

Chapter 6

ADC

In the previous chapter we focused on the most basic and essential peripheral of the microcontroller: the GPIO. In this chapter we will focus on the analog to digital converters that are part of the STM342F446 microcontroller. There are 3 ADCs in the STM342F446, which have a 12-bit resolution. However, the microcontroller has much more pins that can be used to sense analog voltages. This is accomplished using analog multiplexers, which can route the voltages from different pins to an ADC. These multiplexers can also be used to route a few internal voltages to the ADC such as an internal temperature sensor, or the $V_{battery}$ supply voltage.

To function properly, the ADC requires a proper ADCCLK clock, from the ADC prescaler. An overview of this can be seen in figure 6.1, from the STM32F411 reference manual.

The ADC can be started from various triggers, both internal and external, and can generate interrupts when done. It also supports several modes: single conversion mode, free running mode (convert a single channel continuously), and group conversion (convert a group of channels in sequence, in arbitrary order). When reading the reference manual, the difference between injected and regular channels can be a bit unclear, so for clarity: the injected channels can have a higher priority than the regular ones, making it possible to interrupt a regular conversion group for an injected conversion.

6.1 Relevant registers

1. RCC_APB2ENR - In this register the clock for the relevant ADC should be enabled.
2. ADCx_SR - In this register the status of the ADC can be read (for example to see whether a conversion has finished).
3. ADCx_CR1 - ADC Control register 1, In this register parts of the ADC operation mode can be set (for example the resolution).
4. ADCx_CR2 - ADC Control register 2, In this register parts of the ADC operation mode can be set (for example the data-alignment).
5. ADCx_SMPR1/ADC_SMPR2 - These registers can be used to set the ADC sampling time.
6. ADCx_SQR1/ADCx_SQR2/ADCx_SQR3 - regular sequence register (This register can be used to set the different ADC channels when using the ADC in continouos or scan mode).
7. GPIOn_MODER - This sets the alternate function of the GPIO pin.
8. ADCx_DR - This is the register in which the conversion result is stored.

A more in depth description of the registers can be found in the STM32F446_Complete.pdf manual, chapter 13.

6.2 ADC Exercise

Setup the ADC in such a way that the that depending on the angle of the potentiometer, the color of the LED is changed. Use the header as provided as a guideline for writing the code.

- Set the potentiometer pin to analog mode. [1]

[1]See the schematic provided in appendix B

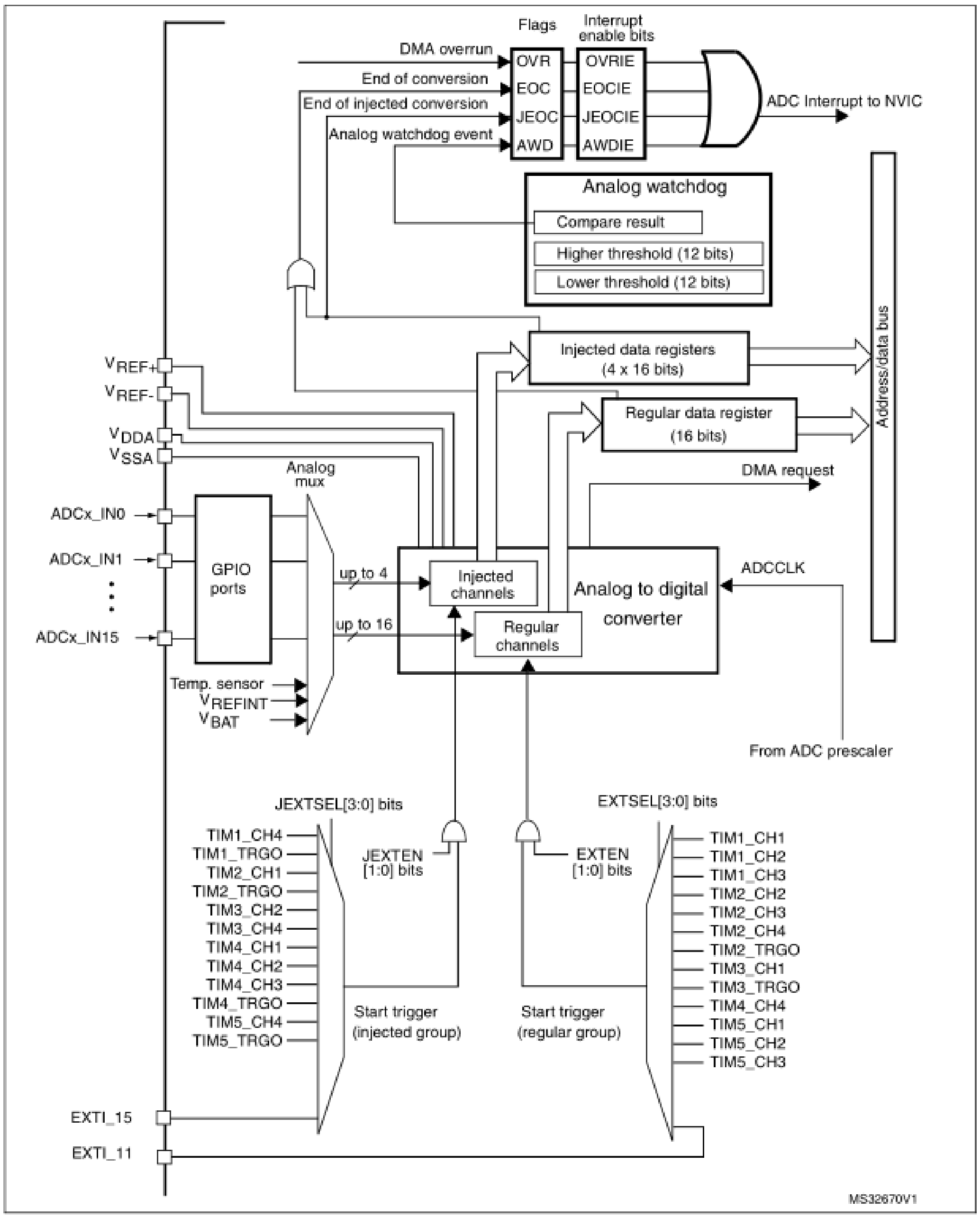

Figure 6.1: ADC block diagram

- Enable the clock for either ADC1 or ADC2.
- Turn on the ADC.
- Set the adc to continouos conversion mode.
- The continuous loops through a given sequence of input pins. Set the length of this sequence to 1[2] and select the relevant multiplexer setting.[3]
- Start conversions.
- Use the status register to wait for the conversion to finish.
- Read the ADC value.

6.2.1 Bonus:

Setup the ADC using DMA so an address in memory always represents the most recent potentiometer value.

[2] ADCx_SQR1

[3] See table 10 in the STM32F446_concise.pdf manual, and figures 69 or 70 in the STM32F446_complete.pdf manual

Chapter 7

Serial communication

7.1 SPI

One of the most basic and most common forms of communication between chips is the Serial Peripheral Interface bus (SPI). This bus is not a physical component, but rather a convention how to implement bi-directional data transfer. In our project, the microcontroller uses the SPI protocol to communicate settings and controls (but not the audio data) to the audio codec.

Standard, bi-directional SPI is a three-wire interface. There are two data lines, one for each direction, and a clock line, driven by one of the chips. The chip that drives the clock line is called **Master**, the device receiving the clock signal is called the **Slave**. The wires between the chips are:

- MOSI (Master Out - Slave In) - line transmitting data from the master to the slave
- MISO (Master In - Slave Out) - line transmitting data from the slave to the master
- SCK (Serial Clock) - clock line determining the timing of the data transfer. Sometimes referred to as SCLK.
- SS (Slave select) - An extra wire to select the active slave, in case multiple slaves share the same SPI bus. One of these lines is required per slave. Sometimes referred to as NSS (N indicating "not", as this is an active low control) or Chip Select (CS)

The general idea is that the Master first selects a Slave by driving the corresponding SS line low. Then, at every rising or falling clock edge (depending on settings), the master reads the MISO line and the slave reads the MOSI line. This transfers one bit of data in both directions, just like a shift register. On the other clock edge, the master and slave set up new data for the MOSI and MISO lines, respectively. To transmit n bits of data in a certain direction, n clock cycles are thus required.

There are two parameters for SPI. The clock polarity (CPOL) determines the clock value in the idle state. The clock phase (CPHA) selects wether data should be transmitted at the first or second clock edge after SS becomes low. The can be translated to sampling data at either the rising or falling clock edge. You will need to look into the datasheets what SPI mode is required by the slave and how to set up the master accordingly.

The microcontroller has integrated SPI hardware: it can transmit and receive SPI data in dedicated communication hardware. You do not have to use GPIO functions and transmit data bit by bit. Instead, you can write outgoing data to a SPI data register and read incoming data from another register. The generation of the clock and data signals is done automatically in dedicated hardware blocks in the microcontroller. This decreases complexity, increases communication speed and allows for parallel SPI communication and other microcontroller functions.

7.2 Exercises

For the project, we are going to implement a one-directional SPI bus between the microcontroller and the codec for the control interface. As there is only data going from the microcontroller to the codec, there will be no MISO line. This SPI interface is used to send control words (CWs) to the codec, writing the control registers in the codec, indicated in Table 30 in the WM8731 datasheet. At the course website, you can download a header file and a C file as a start.[1].

[1] https://www.scintilla.utwente.nl/docs/cursus/MicrocontrollerCourse2017/ProjectHeaders

Exercise 3: Configuring the SPI bus

The shield is designed such that the SPI3 bus can be used to communicate to the codec. A timing diagram for this interface is given in Figure 33 in the WM8731 datasheet. To tell the WM8731 we are using the 3-wire interface, the MODE line to the codec first needs to be set high.
In the `WM8731_init` function:

1. Set the MODE pin on the codec high using GPIO. (don't forget to enable the corresponding GPIO clock)
2. Configure the SPI pins:
   - Configure the MOSI pin as alternate function and set the alternate function to SPI3_MOSI. [2]
   - Configure the SCK pin as alternate function and set the alternate function to SPI3_SCK.
   - As we are going to use software slave management, configure the microcontroller pin going to the codec NSS pin as a GPIO output and set it to a logic high.
3. Enable the clock to SPI3
4. Set up SPI3 control register:
   - Set the SPI to minimal speed for most reliable communication over the relatively long wires.
   - Set the SSI bit to high. This tells the SPI hardware to start SPI transmissions as soon as new data is available: the slave is always selected.
   - Write the other required bits (correct clock phase/polarity, 16-bit data lenght, etc.) in the control register. Only enable SPI after writing all settings.
5. Reset the codec at the end of your initialization sequence. You can use the supplied function `WM8731_config_reset` . Note that this function is dependent on the function `WM8731_spi_out`, which we are going to implement in the next assignment.

Exercise 4: Writing SPI data

To transmit a CW to the codec over SPI, we will use the `WM8731_spi_out` function. This function will transmit one word of SPI data. In higher level functions, layers are added to send specific commands to the codec. In the `WM8731_spi_out` function, you will need to:

1. Start an SPI transfer of the SPI word by writing it to the output register.
2. Wait until the transmit buffer is empty (use the SPI status register).
3. Wait until SPI is not busy anymore.
4. Send a sync pulse over the NSS line, as indicated in Figure 33 in the WM8731 datasheet.

Test this function by periodically writing test data over the SPI interface and measure it using the oscilloscope. Choose smart test data to debug the functionality of the interface (e.g. a value with both zeroes and ones in an easily recognizable pattern). You might want to add a short delay between your transmission sequences. A simple but ugly way to implement this delay for test purposes is given in 7.1.

Snippet 7.1 Example code for a delay implementation

```
for(volatile uint16_t counter = 0; counter < 10000; counter++){/*do nothing in this loop*/}
```

Exercise 5: Writing control words

Next, we are going to write the actual control words required to set up the codec to bypass the audio of the onboard microphone to the headphone output. For this, we are going to call the function `WM8731_config_micbypass`. This function forms an abstraction layer to easily write the control registers. In short, this function uses calls the `WM8731_spi_out` function with pre-programmed data to set up the codec for this bypass mode. Call this function once during your initialization. You should now be able to hear microphone audio over the headphone output.

[2] Refer to "Table 11. Alternate function" in the concise manual ("STM32F446_concise.pdf") for an overview of the alternate functions.

Exercise 6: BONUS : Controlling the volume

Two WM8731_config functions have been given: a reset function and a microphone bypass function. Add a function to control the output volume of the codec. Base this function on the "WM8731_CW" calls in the `WM8731_config_micbypass` function.

Exercise 7: BONUS : A volume knob

Combine your ADC functions and the volume control function in the previous exercise to turn the potentiometer into a volume knob.

7.3 I^{2}S

Now we have a working control interface to the codec, we are going to implement a data interface. The audio data from and to the codec is transmitted over the Inter-IC Sound protocol, I^{2}S. A timing diagram for this data transfer is given in Figure 7.1. I2S is a communication protocol similar to SPI: in implements data transfer using data lines and clock lines. The data is set up at the falling edge of the clock and is sampled at the rising edge of the clock. Again, there is a master generating the communication clock signal, and a slave receiving the clock signal. A slight difference with SPI is that the first bit of data is set up after the first falling edge after selecting a channel, instead of immediatelly. Furthermore, there is a word select line prescent (often abrieviated as WS or Left/Right Channel, LRC), indicating wether the transferred data belongs to the left or right side of a stereo input or output.

I^{2}S and SPI are based on the same principle of serial communication using a clock line and a data line. The two are so similar that the used microcontroller implements both forms of communication in the SPI hardware peripheral.

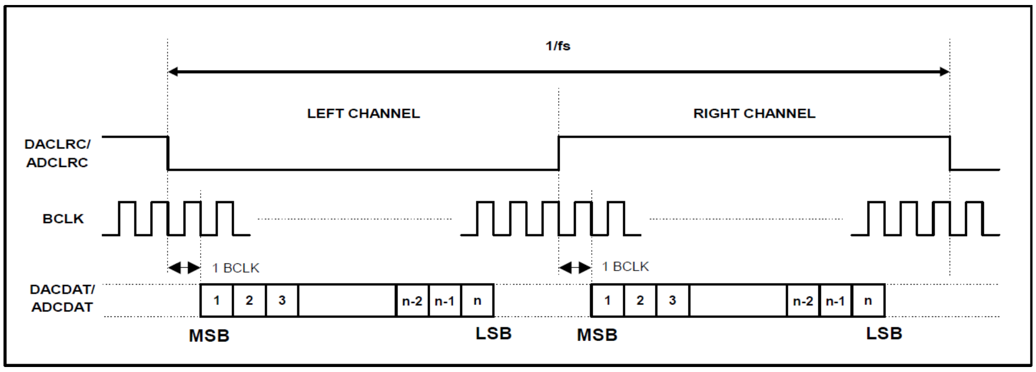

Figure 7.1: Audio data transfer to the WM8731 in I^{2}S mode

7.4 Exercises

For the project, we are going to implement two I^{2}S busses: one bus for communicating microphone samples from the codec to the microcontrolelr, and one bus for communicating playback samples from the microcontroller to the codec. In our case, the codec will be the master. This way, the codec generates a well-defined time base for the sampling of the audio and generates corresponding I^{2}S signals. At the course website, you can again download a header file and a C file as a start (`https://www.scintilla.utwente.nl/docs/cursus/MicrocontrollerCourse2017/ProjectHeaders`).

Exercise 8: Testing the WM8731 I^{2}S configurations

For this first exercise we are going to initialize the I^{2}S peripheral in the codec only. The microcontoller will ignore the I^{2}S data for now. In your system initialization, call the function `WM8731_config_init` to set up the codec for 16bit, 8kHz audio communication over I^{2}S to start the stream of I^{2}S data coming from the codec DAC. The functions `WM8731_i2s_dac_init` and `WM8731_i2s_adc_init` are called in this function but are not implemented yet; we thus only initialize the codec end of the I^{2}S bus. Connect a jumper wire between the ADC data output and the DAC data input of the codec. These pins are best accessible from the pinheaders on the backside of the Nucleo board. This wire should feed the ADC data directly to the DAC. As a result, you should be able to hear the microphone audio over your headphones, albeit with decreased audio quality, as we are only sampling at 8kHz.

Exercise 9: I^{2}S communication

To implement a full duplex I^{2}S communucation, we are first going to initialize the interface from the microcontroller to the codec DAC.

1. Remove the jumper wire used in the previous exercise.
2. In the `WM8731_i2s_dac_init` function, initialize the clock, serial data and word select lines from the codec ADC to the microcontroller as I^{2}S pins by writing the correct GPIOx_MODER and GPIOx_AF bits. To find the correct alternate functions, refer to Table 11 in the concise manual (STM32F446_concise.pdf).
3. In the same function, set up the I^{2}S peripheral:
   - Enable the clock to the SPI2/I^{2}S peripheral by setting the correct bit in the Reset and Clock Control (RCC).
   - Enable the I^{2}S interrupt using the snippet in Snippet 7.2.
   - Set the SPI/I^{2}S peripheral to I^{2}S mode in the SPIx_I2SCFGR register.
   - Set the I^{2}S peripheral to slave transmit I^{2}S mode for 16 bit audio in the same register.
   - Write the initial data 0x0000 to the data register.
4. In the `WM8731_i2s_adc_init`, initialize the clock, serial data and word select lines from the microcontroller to the codec DAC as I^{2}S pins by writing the correct GPIOx_MODER and GPIOx_AF bits. Again, refer to Table 11 in the concise manual (STM32F446_concise.pdf).
5. In the same function, set up the I^{2}S peripheral:
   - Enable the clock to the SPI2/I^{2}S peripheral by setting the correct bit in the Reset and Clock Control (RCC).
   - Set the SPI/I^{2}S peripheral to I^{2}S mode in the SPIx_I2SCFGR register.
   - Set the I^{2}S peripheral to slave receive I^{2}S mode for 16 bit audio in the same register.

   In the previous exercise you initialized the I^{2}S communication on the codec. Now, you should now have made code that initializes the I^{2}S pins and interrupt at the microcontroller. Now, a problem arises. The used microcontroller has a silicon bug (a hardware design mistake) for the I^{2}S slave mode for the settings we use: "In this case, the master and slave will be desynchronized throughout the whole communication."[3]. As a workaround, the code in 7.3 is used as an abstraction layer over the SPI interrupt, ensuring correct synchronization. Note that this function gives calls to the functions `i2s_send` and `i2s_receive`.
6. Make an implementation for the `i2s_receive` function. As the codec is configured to send microphone data (a mono signal), the incoming data is the same for the the left and right channel. If we receive data from the right channel (check the CHSIDE bit in the SPI Status Register), save the incoming data in a variable.
7. Make an implementation for the `i2s_send` function. When checking the channel side bit, keep in mind that you are setting the data for the next data transmission, which is in the other side. Furthermore, the jack plug is connected inverse: the left left headphone output of the codec is connected to the right side of the jack connector, and vice versa. In both cases, for the left and right channel, transmit the microphone data you received in the `i2s_receive` function. You should now be able to playback the microphone data over your headphone.

Snippet 7.2 Initialization of the SPI/I^{2}S interrupt

```
// Enable TXE interrupt
SPI2->CR2 |= SPI_CR2_TXEIE;
// Enable interrupt through the interrupt controller using CMSIS function
NVIC_EnableIRQ(SPI2_IRQn);
```

[3]Refer to the STM32F446 Errata Sheet

Snippet 7.3 Workaround for the I^{2}S silicon bug

```c
static int8_t i2s_count = 0;
static enum {LEFT, RIGHT} last_side = LEFT;

void SPI2_IRQHandler(void) {
    // Reset the counter at an edge of the word select
    if (last_side == LEFT) {
        if (   SPI2->SR & SPI_SR_CHSIDE  ) i2s_count = 0;
    } else {
        if ( !(SPI2->SR & SPI_SR_CHSIDE) ) i2s_count = 0;
    }

    // Prepare new output data every last sequence
    if (i2s_count == 11) {
        i2s_send();
    } else {
        SPI2->DR = 0x0000;
    }

    // Read new input data every first sequence
    if (i2s_count == 0) {
        i2s_receive();
    }

    // Update the side state variable
    if (SPI2->SR & SPI_SR_CHSIDE) {
        last_side = RIGHT;
    } else {
        last_side = LEFT;
    }
    // Increase the counter
    i2s_count++;
}
```

Exercise 10: BONUS : Audio manipulation

Use a form of processing over your audio before playing it back. Some suggestions on easy to implement processing:

- Amplify your audio by multiplying it with a constant.
- In a looping manner, fill an array with one second of audio data. The next second, play back the audio you just stored. Repeat this saving and playing sequence.
- Implement a delay function. Using an array, continiously store data and play back the data recorded a second ago.

Chapter 8

Final Project

8.1 Audio processing environment

For the final project, we are going to implement an audio recorder. For this, we give you an existing microcontroller project to start with. This project handles the complete initialization and transfer of audio samples in a streamlined pipeline. You are going to work in the audio processing library (`audio_proc.c`). This library contains one function: `audio_proc`. This function is called by the pipeline when new incoming samples are available. In this function, you will need to set the next output samples. A more detailed parameter description is given above this function in the comment. In Eclipse, you might need to expand this comment.

Currently, audio comes in as 16 bit samples at 8kHz. You can increase the sampling frequency to 48kHz in `WM8731.h`. A higher sampling rate gives better audio quality, but decreases the time you have to process a sample, and increases the required memory to store a certain period of audio.

8.2 Exercises

Exercise 11: Importing the audio processing project

- Download the `http://www.scintilla.utwente.nl/docs/cursus/MicrocontrollerCourse2017/audio_proc.zip` file and extract it.
- Create a new project according to section 4 of the manual.
- Import the header files from the "audio_proc.zip" from into the project, by copying them into the src folder in the project directory. Refresh the src directory ('Right click > Refresh' or 'F5') if the files do not show up in the Eclipse Project Explorer.
- Debug the project (section 4.0.2). When you connect a pair of headphones to the jack on the shield, you should be able to hear ambient sound from it now.

Exercise 12: Recording audio

We are going to record audio in the device RAM. The easiest place to store the data is in an array.

- Create a long array of 16 bits unsigned integers in the `audio_proc.c` file. Your compiler gives errors if you use too much RAM, so go ahead and try how far you can go. Keep in mind that the microphone input is mono, so we do not need to save the left and right samples. How many seconds of audio is this?
- In the `audio_proc` function, make a counter that keeps counting as long as the record button is pressed. When the button is released, save the value of the counter and reset the counter.
- While the recoding button is pressed, save the audio samples in the array.

To test this, record some audio and look in the debugger wether audio is saved in the array.

Exercise 13: Playing audio

Now we have recorded audio, we want to be able to play back audio from the same array.

- In the `audio_proc` function, when the play button is pushed, make a counter that keeps counting as long as the play button is pressed. When the button is released, reset the counter.
- While recoding, send the audio sample that corresponds to the counter index to the headphones.

Exercise 14: Human interface

To make the audio recoder function more like a commercial recorder, switch the LED red when recording and green when playing audio. You can use our GPIO functions (refer to `gpio.h`) or you can add your own gpio functions to this library and use these.

Exercise 15: BONUS : Volume control

Control the audio codec playback volume with the potmeter:

- Read the potmeter value using the `ADC_pot_read` function from `ADC.h`.
- Set the playback volume using the `WM8731_config_volume` function from `WM8731.h`.

Appendix A

Setup Eclipse and Toolchain

In this course Eclipse is used as the IDE, as it's cross-platform and highly customizable. It is recommended to use a clean install of Eclipse for C/C++. Besides Eclipse you'll need to install some Eclipse plugins and drivers for STLink, the device that connects to your Nucleo board. The guide is written with Windows and GNU/Linux in mind, but it should work on OSX as well. For 64-bit windows the packages are available on http://www.scintilla.utwente.nl/docs/cursus/MicrocontrollerCourse2017/Software

A.1 Windows

A.1.1 Eclipse

1. Go to http://www.eclipse.org/downloads/packages/eclipse-ide-cc-developers/neon1 and download **Eclipse IDE for C/C++ Developers**. Make sure you install the correct version (x64 or x86).
2. Unpack and install.
3. If the installer prompts you that you need a newer Java Runtime Environment (JRE), download the newest JRE.
4. In Eclipse, go to Help → Install New Software
5. click on Add..., fill in Name: *GNU ARM Eclipse Plug-ins*. Location: http://gnuarmeclipse.sourceforge.net/updates and press OK.
   In case you get an *Unable to get repository* error, please check http://gnuarmeclipse.github.io/blog/2017/01/29/plugins-install-issue/
6. select all but the Freescale Project Templates, click Next and install.

A.1.2 Toolchain

1. Get the toolchain from https://launchpad.net/gcc-arm-embedded. (gcc-arm-none-eabi***-win32.exe)[1]
2. Install the toolchain, but in the final window disable "Add path to the environment variable".
3. Download the latest build tools (gnuarmeclipse-build-tools-win32-2.*-*-setup.exe) from https://github.com/gnuarmeclipse/windows-build-tools/releases
4. Run the installer, remember the path of the Build Tools.
5. In Eclipse, go to Window → Preferences. C/C++ → Build → Global Tools Paths
6. Locate the installed Build tools, enter the path in Build Tools folder.
7. Select *GNU Tools for ARM Embedded Processors*, locate the toolchain and enter the path in Toolchain folder, and click Apply.

[1]On 13/2/2017: https://launchpad.net/gcc-arm-embedded/5.0/5-2016-q3-update/+download/gcc-arm-none-eabi-5_4-2016q3-20160926-win32.exe

A.1.3 STLink v2.1 driver

1. Download the STLink v2.1 driver from http://www.scintilla.utwente.nl/docs/cursus/Microcontrolle Software/stsw-link009.zip
2. Extract files and run dpinst_amd64.exe for a 64-bit system, or dpinst_x86.exe for a 32-bit system.

A.1.4 Debugger - OpenOCD

This is optional, but a debugger might help you a lot. We're using OpenOCD as it's available for all platforms, and easily integrates with the Eclipse plugins we installed.

1. Download the latest stable (0.10.*) version of OpenOCD for your architecture from https://github.com/gnuarmeclipse/openocd/releases/download/gae-0.10.0-20170124/gnuarmeclipse-openocd- 10.0-201701241841-setup.exe.
2. Follow the installation procedure.
3. In Eclipse go to Window → Preferences → Run/Debug → String Substitutions
4. Fill in the path to the bin directory of OpenOCD in the Value field of *openocd_path*, then click OK.
5. The next steps only apply after making a project, see section 4.0.2.

A.1.5 Packs - Device Support

1. In Eclipse, open the Packs perspective.
2. Click on the Refresh button. It will now load all available packs from Keil.
3. Select the device menu, locate the STM32F4 series and install the package.

A.2 Linux

This guide assumes a working Java runtime environment. It was tested using OpenJDK 1.8, and should work equally well with a recent version of Oracle JRE.

A.2.1 Toolchain and Eclipse

This guide works for Eclipse NEON 1 and Eclipse Mars. The newer NEON 2 is **not** supported.

1. Get the toolchain from https://launchpad.net/gcc-arm-embedded (gcc-arm-none-eabi***-linux.tar.bz2)
2. Extract to a directory of your liking.
3. Install Eclipse Luna for C/C++ development, if you haven't already.[2]
4. open Eclipse, set a workspace and click on Help → Install New software.
5. click on Add..., fill in Name: GNU ARM Eclipse Plug-ins. Location: http://gnuarmeclipse.sourceforge.net/updates and press OK
6. select all but the Freescale Project Templates, click Next and install.
7. In case you get an *Unable to get repository* error, please check http://gnuarmeclipse.github.io/blog/2017/01/29/plugins-install-issue/
8. In Eclipse, go to Window → Preferences. C/C++ → Build → Global Tools Paths
9. Select GNU Tools for ARM Embedded Processors, locate the toolchain and enter the path in Toolchain folder.

[2]Either via your package manager, or by downloading from http://www.eclipse.org/downloads/

STLink flasher for Linux

The STLink Utility provided by STMicroelectronics is Windows only, but an open source alternative is available on GitHub.

1. Go to `http://github.com/texane/stlink`, download the Zip file
2. Extract the zip file to a preferred location.
3. In a terminal cd to the directory the files are in, and build stlink using:
   ```
   ./autogen.sh
   ./configure
   make
   sudo make install
   ```
4. get the path of st-flash using:
   ```
   whereis st-flash
   ```
5. In Eclipse, click Run → External Tools → External Tools Configurations
6. Click on Program, and then on New. Name the new configuration *st-linkv2 flash*, and paste the path to st-flash in the Location field.
   Working directory: ${project_loc}/Release
   Arguments: write ${project_name}.bin 0x8000000
7. Click Apply and close.

A.2.2 Debugger - OpenOCD

This is optional, but a debugger might help you a lot. We're using OpenOCD as it's available for all platforms, and easily integrates with the Eclipse plugins we installed.

1. Download the latest stable (0.10.*) version of OpenOCD for your architecture from `http://sourceforge.net/projects/gnuarmeclipse/files/OpenOCD/GNULinux/`.
2. Extract the package to a directory.
3. In Eclipse go to Window → Preferences → Run/Debug → String Substitutions
4. Fill in the path to the bin directory of OpenOCD in the Value field of *openocd_path*, then click OK.
5. The next steps only apply after making a project, see section 4.0.2.

A.2.3 Packs - Device Support

1. In Eclipse, open the Packs perspective.
2. Click on the Refresh button. It will now load all available packs from Keil.
3. Get a cup of coffee.
4. Select the device menu, locate the STM32F4 series and install the package.
5. Drink the coffee.

Appendix B

Project shield schematic

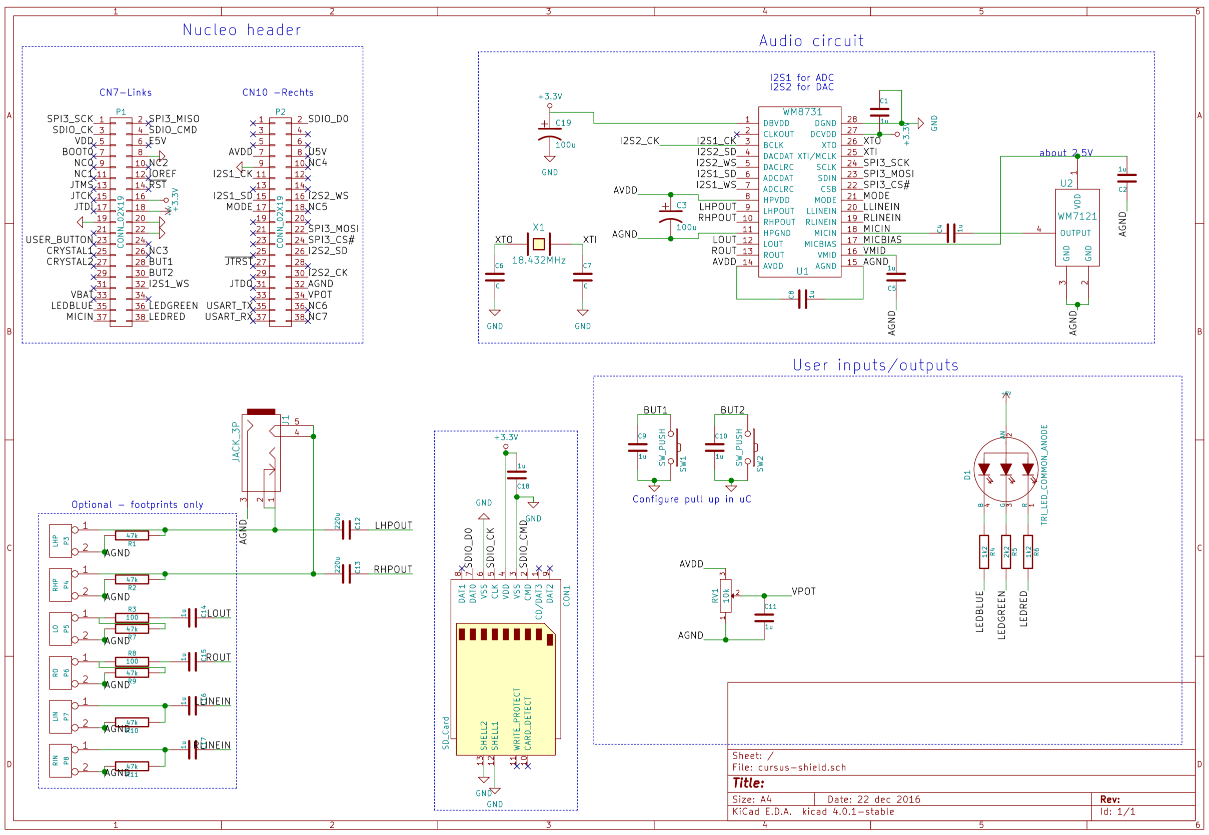

Figure B.1: The schematic of the shield.

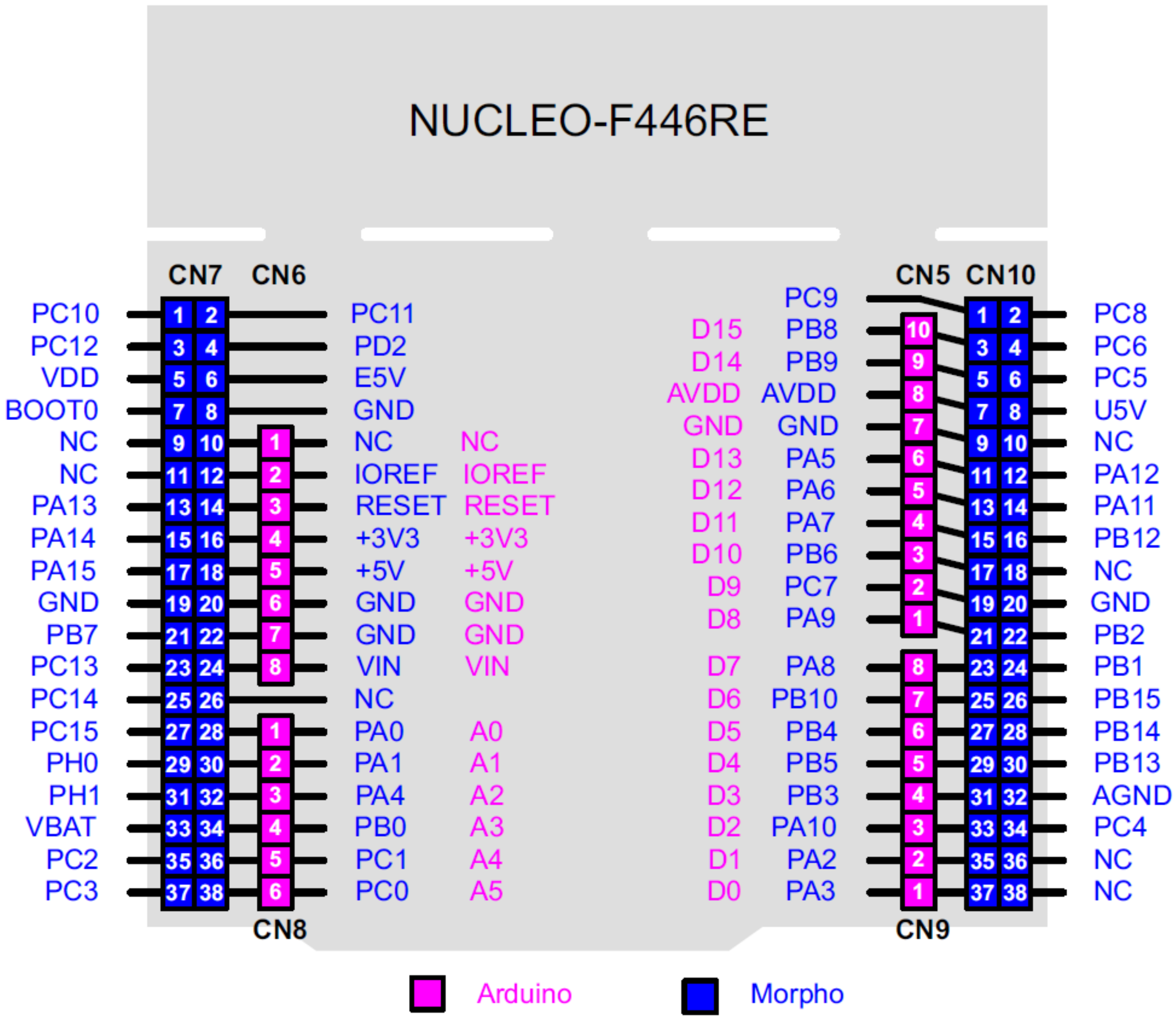

Figure B.2: The pinout of the morpho header of the nucleo board.

Appendix C

FAQ

C.0.1 Eclipse

I can't find Packs

go to the Eclipse menu → Window → Open Perspective → Other...
select the Packs perspective in the list.

Warning "Please check if the SystemClock_Config() settings match your board!"

This is not a problem, you can ignore this or remove the line starting with #warning in the file *_initialize_hardware.c*

Where do I place my .h and .c files?

In the project tree you see a few folders. The *.h* header files should go in the *include* directory. The *.c* source files should go in the *src* directory.

When I turn the pin for a LED color high, the LED goes off.

Very confusing indeed. Look in figure B.1 and you can see that the common pin for the RGB LED is the positive supply.

Appendix D

Working without Eclipse

D.1 System Clock Setup

The STM32F446 has two main high frequency oscillator modes: using the internal RC oscillator HSI[1] or an external crystal oscillator HSE[2]. On the provided Nucleo-446 the external oscillator crystal is unpopulated, so you have to use the internal RC oscillator. According to the datasheet[3] the internal RC oscillator operates at 16MHz. The maximum frequency the STM32F446 can run at is 180MHz. We can use the PLL to increase the operating frequency of the microcontroller.
The frequency of the PLL output, f_{PLL} is defined as:

$$f_{\text{PLL}} = f_{\text{in}} \frac{PLLN}{PLLM \cdot PLLP}$$

The PLLN, PLLM and PLLP are condifuration bits of the PLLCFGR register, with the following conditions:

- $50 \leq \text{PLLN} \leq 432$
- $2 \leq \text{PLLM} \leq 63$
- $f_{\text{in}} \frac{PLLN}{PLLM} \leq 432\text{MHz}$

Snippet D.1 Example code to set systemclock to 180MHz on internal oscillator

```
// Enable internal RC oscillator
RCC->CR |= RCC_CR_HSION;

// Wait for locked oscillator
while((RCC->CR & RCC_CR_HSIRDY) != RCC_CR_HSIRDY);

// Configure the PLL to increase system frequency
RCC->PLLCFGR =
    RCC_PLLCFGR_PLLSRC_HSI
    | (8 << 0)      // PLLM = 8 -> fvcoin = 2MHz
    | (180 << 6)    // PLLN = 180 -> fvco = 180*2MHz = 360MHz
    | (0 << 16)     // PLLP = 0 -> fsys = 360/2 = 180MHz
    | (0b010 << 28);    // PLLR = 010 -> Main PLL division factor for I2S

// Enable the PLL
RCC->CR |= RCC_CR_PLLON;

// Wait for locked PLL
while((RCC->CR & RCC_CR_PLLRDY) != RCC_CR_PLLRDY);

// Select system clock
RCC->CFGR &= ~RCC_CFGR_SW;
RCC->CFGR |= RCC_CFGR_SW_PLL;

// Wait for Sysclk to be set by PLL
while((RCC->CFGR & RCC_CFGR_SW_PLL) != RCC_CFGR_SW_PLL);
```

[1]High Speed Internal
[2]High Speed External
[3]STM32F446_complete.pdf 6.2.2, page 118

List of Tables

Code Snippets