Cours de MICROINFORMATIQUE
Section de Microtechnique
Printemps 2020

Practical Exercises

Practical Exercise 2: Compilation process and motor control library	
Title:	Compilation process and motor control library for the e-puck2 miniature mobile robot.
Goal:	Getting in touch with the gcc compiler and programming a motor control library for the e-puck2 miniature mobile robot.
Duration:	4 hours
Support:	Files available to download listed and explained in the appendix.
Equipment:	e-puck2 robot, gcc, stepper motor, H-bridges

1 Introduction

1.1 Main Goal

In this practical, you will see the compiling process in detail, as you have learned during the lecture. The rest of this exercise shows all necessary steps to generate a low-level library for the e-puck2 miniature mobile robot written in C. This represents a classical task for an engineer and for an e-puck developer.

The library targeted in this exercise is a motor control library allowing to set speed and position targets.

For this work we provide you some files to download from the course website. All the documentation concerning the GNU embedded toolchain for ARM processors `arm-none-eabi` and the compiler GCC can be found in the folder where the `Eclipse` executable is installed, in the subfolder `Tools\gcc-arm-none-eabi-7-2017-q4-major-win32\share\doc\gcc-arm-none-eabi\pdf`.

1.2 Methodology

To achieve the main goal, we will go through the following steps:

- Understanding some basic features of the compiler.
- Understanding how a PWM works.
- Understanding how a stepper motor works and programming a stepper motor controller in C.
- Making a library out of it.

1.3 C compiler

To understand how the code is generated, it is important to understand how the compiler works. The main compilation steps are summarized in figure 1. In this first exercise we will explore this architecture experimentally.

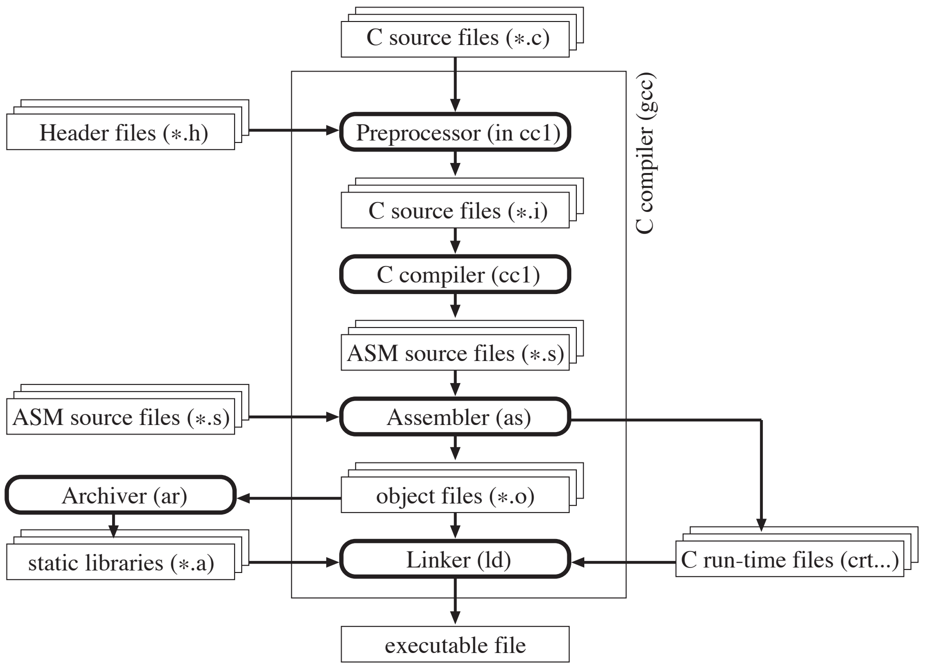

Figure 1: Compilation steps in GCC.

1.3.1 Generated code

Create a new folder in your working directory. With an editor (eg. `Notepad`) please create a source code file called `test.c` and containing the following C code:

```
int main()
{
    int i,j,out=0;
    for(i=0;i<10;i++)
        for(j=0;j<10;j++)
            out+=i+j;
    return out;
}
```

Now open a shell window (or terminal) to access to a command line. Go in the folder you created by using the commands `cd` (change directory), and `dir` (list the current directory) to verify if `test.c` is correctly created there.

During the first practical, you compiled your code using the GNU toolchain for ARM processor, which was called during the build process. We will now compile the code using only the command line, in order to see the compilation process in detail. This is the standard and basic way to compile a program. The Eclipse environment just has access to these commands. The compilation process is specified in the Makefile of the project. The command to compile the C code and generate an object files and all the intermediate steps is :

```
arm-none-eabi-gcc -save-temps=obj -mcpu=cortex-m4 -c test.c -o test.o
```

`arm-none-eabi-gcc` is the command to launch the compiler, `-save-temps=obj` is used to save the intermediate files of the compilation process, `-mcpu=cortex-m4` is called to specify the processor that will execute the code and `-c` specifies to not do the linking step. Please compile this code by typing the explained command in the shell window.

If you type this command right now in the terminal, you will get the following error :

Listing 1: compilation error

```
>arm-none-eabi-gcc -save-temps=obj -mcpu=cortex-m4 -c test.c -o test.o

'arm-none-eabi-gcc' is not recognized as an internal or external command,
an executable program or a batch file.
```

This error occurs because the command line instance doesn't know yet what is this command. We have to add the path to the folder containing the executable files in the `PATH` environment variable. The `PATH` variable is used to store the location of all the known executables the command line can call. Type the following command (a bit different depending on the OS) to add the path of the ARM toolchain packed with Eclipse_e-puck2.

Listing 2: set PATH for Windows

```
>set PATH=C:\Program Files\Eclipse_e-puck2\Tools\gcc-arm-none-eabi-7-2017-q4-major-win32\bin;%PATH%
```

Listing 3: set PATH for Linux

```
>export PATH=your_installation_path/Eclipse_e-puck2/Tools/gcc-arm-none-eabi-7-2017-q4-major/bin:$PATH
```

Listing 4: set PATH for Mac

```
>export PATH=your_installation_path/Eclipse_e-puck2.app/Contents/Eclipse_e-puck2/Tools/gcc-arm-none-
    eabi-7-2017-q4-major/bin:$PATH
```

What is important to know is that this procedure is temporary. It applies only to this command line window. If you open a new command line window or close this one, you will have to set again the `PATH` variable. Of course, the path given in the example for Windows (Listing 2) is correct for the configuration in the DLL rooms. If you installed Eclipse_e-puck2 on your own computer, you need to adapt the path.

Now if you type again the compilation command, it should work.

Task 1:

Please look at the files generated by this compilation. Which files have been generated? Which file is generated from which step of the compiler?

Task 2:

Please look in detail at the assembler code generated by the compiler. Understand the assembly language instructions by using the Cortex-M4 Generic User Guide.

Hints:
This document is available on the moodle page of the course. In particular, look at the functionality of the instructions **push**, **pop**, **mov**, **add**, **str**, **ldr**, **cmp** and **ble,...**. What is the purpose of **sub sp, sp, #20** instruction at the beginning of the main function? Try to initialize more variables in the main function and see how this instruction will change.

Task 3:

Remember the role of the instruction **#define** in C. Which step of compilation takes in charge this instruction? To look at this aspect in a real example, please create a source C file (.c) with the following source code in it. Then look at the **.i** file generated by the compilation.

```
#define  PI      3.14
#define  CIRC(R) (2 * PI * R)

int main()
{
    int circonference , rayon = 2;
    circonference  = CIRC(rayon);
    return circonference ;
}
```

1.3.2 Compilation process

Task 4:
Please use the option **-v** to observe the detailed compilation process. Check the several steps and understand the main mechanisms.
Verify that you can directly assemble assembler code with the following command:

arm-none-eabi-as test.s -o test.o

1.3.3 Compilation options

Please change the compiling command (cursor up to repeat the last commands) and the content of the program to check the influence of some compilation options.

Task 5:
Check the warning option described in table 1 on the following code:

Describe the impact of these options and their use. Do these options have an impact on the generated code?

```
int main()
{
    int i,j,out=0,k;
    for(i=0;i<10;i++)
        for(j=0;j<10;j++)
            out+=i+j;
}
```

Table 1: Compilation options and influence on compilation process

Option	Influence on compilation process	When is this option useful?
-Wreturn-type	Check si le type de variable retournée est cohérent	Une fonction void qui retourne quand même qqch. (sinon il y a conversion implicite et pas de warning)
-Wunused-variable	Check si certaines varibales sont inutilisées	Variable inutilisée
-Wall	Affiche tout le code et les adresses utilisées	Tous les Warnings classiques possibles

Task 6:
Please understand the compiling options described in table 2 on the following code:

Please describe the influence of these options and when they are useful. What is the impact on code size, use of memory, and execution speed? Please compare the number of instructions necessary to execute a code without optimization, a code with optimization with **-O2** and a code optimized at level **-O3**.

When could this type of optimization (**-O3** or **-funroll-loops**) generate big problems (imagine an embedded system with sensors and actuators)? How can we force a correct optimization in this case?

Hint : **-funroll-loops** only works when an optimization level is set (**-O1**, **-O2**, etc.)

```
int main()
{
    int i,j,out=0,k;
    for(i=0;i<10;i++)
        for(j=0;j<10;j++)
            out+=i+j;
    return out;
}
```

Table 2: Compilation options and influence on generated code

Option	Influence on compilation process	When is this option useful?
-O0		
-O1		
-O2		
-O3		
-funroll-loops		
-funroll-loops -O3		

2 Setup of the project

From this practical exercise and for the next ones, you will not have to configure the project anymore. The projects are now already created and you just need to open them.

To open the project for this practical session, in Eclipse_e-puck2, go to **File->Open Projects from File System**, then click on **Directory...** and choose the folder containing the code. Here it is the folder **move** contained into **TP2_Move**. Click finish and you should have the project open.

Another new thing is that the project contains a debug configuration too. If you look on the **debug configurations** menu of Eclipse, you will see a new debug configuration called **TP2_Move**. If you select it and go on the tab **Startup**, you will see a variable **$COM_PORT** instead of a real com port. This is because now we will use an internal variable created by us into Eclipse to tell which com port to use. This variable will be used by all the debug configurations of the next practical sessions. Like this, only this variable need to be changed when the com port changes, and not every debug configuration.

To change the **COM_PORT** variable, simply return on the **main** tab of the debug configuration, then click on the **Variables...** button and on **Edit variables...**. Here you will be able to change the value of the variable to put the correct com port for your case.

3 Programming a PWM signal to drive a LED

3.1 Introduction

During Practical 1, you used a timer and an interrupt routine to toggle a LED with a given frequency. Here, we are going to drive a LED using a PWM (Pulse Width Modulation) signal, which will allow us to change the intensity of the LED. This will require to configure a Timer with an Output Compare channel (Fig. 2), and to connect a GPIO on which the LED is connected to this channel in order to drive the LED with the PWM signal.

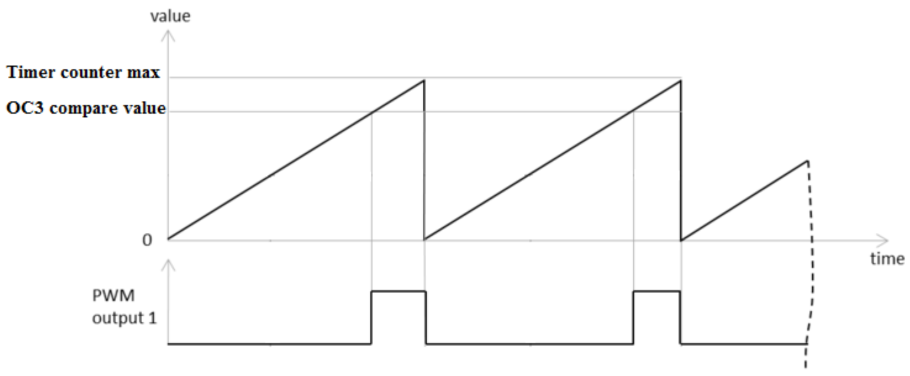

Figure 2: PWM signal generated using a Timer and Output Compare in PWM mode 2. (https://visualgdb.com/tutorials/arm/stm32/pwm/)

Note : The figure 2 shows a PWM mode 2. PWM mode 1 does the same, except the output signal is inverted.

3.2 Configuration of the GPIO

Only specific GPIO can be driven in output using a PWM signal. If you open the datasheet of the STM32F40xxx and go to table 7 `STM32F40xxx pin and ball definitions`, you will see all the possible alternate functions for each pin of the microcontroller (4). In this example, we will select the Pin 14 of the Port D `PD14` that is connected to the channel 3 of the timer 4 `TIM4_CH3`. This GPIO is connected to the Front Led of the e-puck2, thus it is perfect to test the effect of a PWM signal on the intensity of a LED!

In the practical 1, the `PD14` GPIO used to drive the LED was set to output, as we just wanted to drive the LED with two states (high and low). Here, we will have to use the pin in `Alternate Function (AF)` mode.

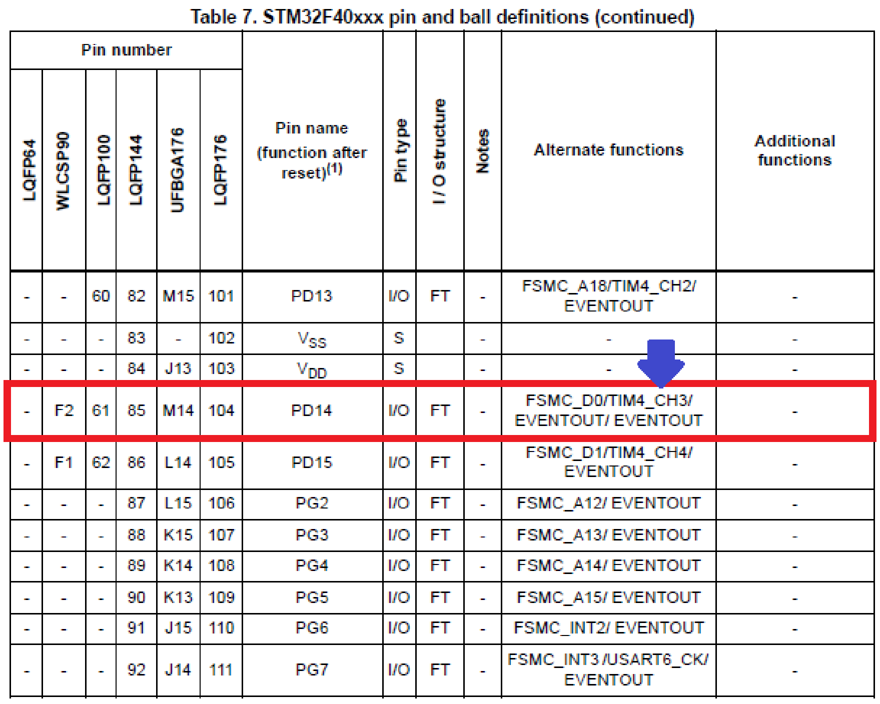

Table 7. STM32F40xxx pin and ball definitions (continued)

Pin number						Pin name (function after reset)(1)	Pin type	I / O structure	Notes	Alternate functions	Additional functions
LQFP64	WLCSP90	LQFP100	LQFP144	UFBGA176	LQFP176						
-	-	60	82	M15	101	PD13	I/O	FT	-	FSMC_A18/TIM4_CH2/ EVENTOUT	-
-	-	-	83	-	102	V_{SS}	S		-	-	-
-	-	-	84	J13	103	V_{DD}	S		-	-	-
-	F2	61	85	M14	104	PD14	I/O	FT	-	FSMC_D0/TIM4_CH3/ EVENTOUT/ EVENTOUT	-
-	F1	62	86	L14	105	PD15	I/O	FT	-	FSMC_D1/TIM4_CH4/ EVENTOUT	-
-	-	-	87	L15	106	PG2	I/O	FT	-	FSMC_A12/ EVENTOUT	-
-	-	-	88	K15	107	PG3	I/O	FT	-	FSMC_A13/ EVENTOUT	-
-	-	-	89	K14	108	PG4	I/O	FT	-	FSMC_A14/ EVENTOUT	-
-	-	-	90	K13	109	PG5	I/O	FT	-	FSMC_A15/ EVENTOUT	-
-	-	-	91	J15	110	PG6	I/O	FT	-	FSMC_INT2/ EVENTOUT	-
-	-	-	92	J14	111	PG7	I/O	FT	-	FSMC_INT3 /USART6_CK/ EVENTOUT	-

Figure 3: STM32F40xxx pin and ball definitions, from the datasheet

Task 7:
Create a function in **gpio.c** and **gpio.h** called **gpio_config_output_af_pushpull** for example. Then, use the Reference Manual to configure the pin in **Alternate Function (AF)** mode by changing the configuration of the register **MODER**.

Then, you will have to specify that the pin is driven by the Channel 3 of Timer 4, by modifying the corresponding bits in the `AFR` register of the **GPIOD**. This is done by using the datasheet, table 9, you should obtain an `AF` number corresponding to the function and pin. Then, use this number in the reference manual (Sec 8.4.10) to identify how to set `AFRH14`.

Task 8:
Use the Reference Manual to configure the register **AFR** to select the right alternate function for the pin 14 so that it is driven by the Channel 3 of the Timer 4.

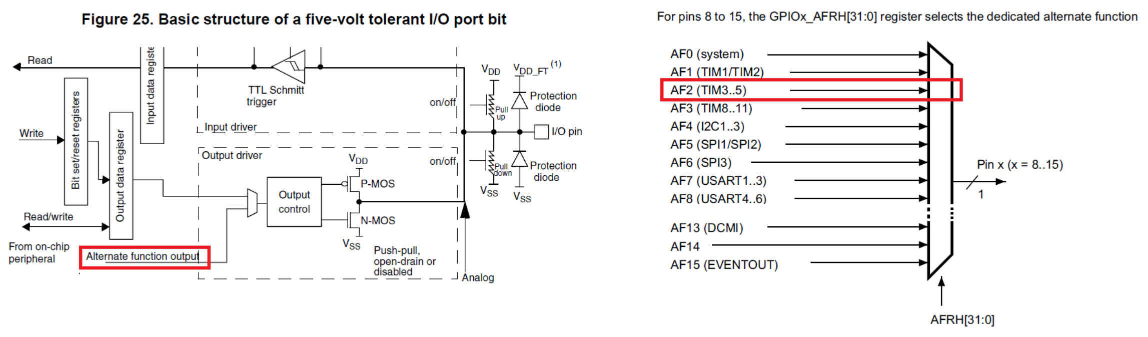

Figure 4: Configuration of the GPIO in alternate function mode, see Sec. 8 of the Reference Manual

Now, you have finished to configure the GPIO, let's configure the Timer and Output Capture to obtain the desired PWM on the pin 14. The Timer 4 should be configured similarly to what was done in Practical 1 with Timer 6, except that we do not require a timer interrupt in this case (List. 5).

Listing 5: Timer 4 configuration

```
// enable TIM4 clock
RCC->APB1ENR |= RCC_APB1ENR_TIM4EN;

// configure TIM4
TIM4->PSC = PRESCALER_TIM4 - 1;    // Note: final timer clock  = timer clock / (prescaler + 1)
TIM4->ARR = COUNTER_MAX_TIM4 - 1; // Note: timer reload takes 1 cycle, thus -1

// enable TIM4
TIM4->CR1 |= TIM_CR1_CEN;
```

Task 9:
Determine the Prescaler and Counter maximum value for Timer 4. Hint: The blinking of the LED should be fast in order to not being seen. Something around 80-100Hz.

The final step is to configure the output compare of Timer 4, Channel 3. The schema-bloc of the Output compare architecture can be seen in Fig. 5 and can be seen in the reference Manual. We already identified the registers of timer 4 that need to be configured.

Figure 161. Output stage of capture/compare channel (channel 1)

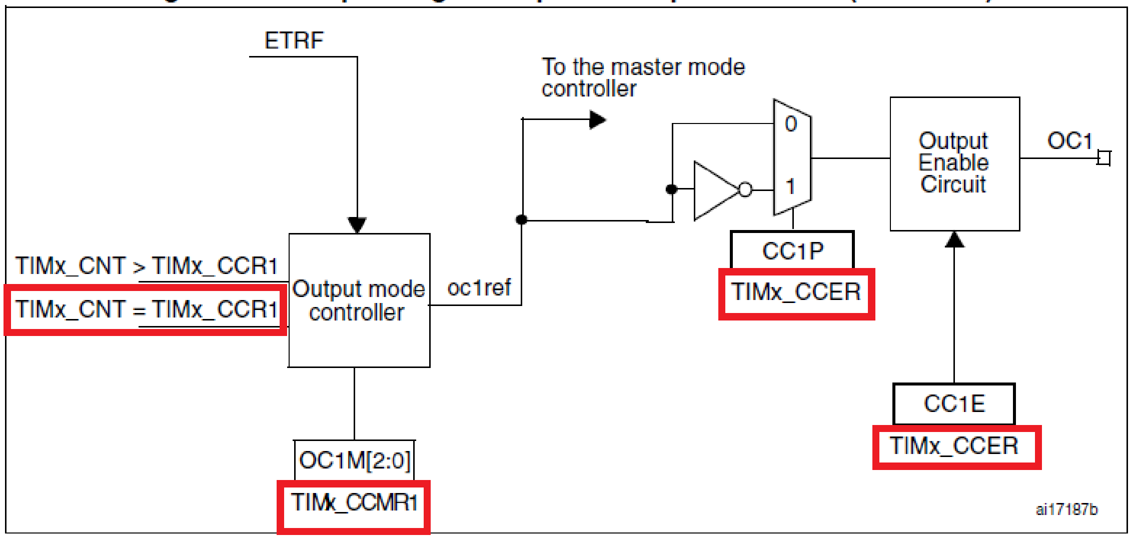

Figure 5: Output stage of capture/compare channel (channel 1), with the registers involved. The same scheme can be applied to channel 3, see Chap. 8 of the Reference Manual.

Task 10:

Configure the three registers of **TIM4: CCMR2, CCR3, CCER** in order to obtain the PWM signal on pin 14. Use **PWM mode 1**.
Build and run your code. You can use the debugger and the register visualization to verify that the registers are correctly configured.
You can use the selector for example to change the value of **CCR3** and thus, change the intensity of the Front LED.
Use the oscilloscope to visualize the PWM signal with different duty cycle.

Hint : Take a look on the point 18.3.9 of the reference manual. It explains how to configure the timer to use the PWM.

4 Programming a library to drive the stepper motor of the e-puck2

5 Motor control functions

The e-puck2 robot is equipped with two stepper motors. These motors have two windings. The power in these windings is controlled by 4 signals (a, b, c and d) for each motors. Figure 6 shows how these four signals have to be activated to make the motor turn. Every signal transition is considered as a step. A rotation of 360^o of the wheel corresponds to 1000 steps (20 steps per revolution of the motor combined with a gear reduction of 50:1). In order to calculate the distance run by the robot, note also that the robot wheel's perimeter is about 13cm.

If you want to get more information on stepper motors you can look on
`http://en.wikipedia.org/wiki/Stepper_motor` or
`http://fr.wikipedia.org/wiki/Moteur_pas_a_pas` (in french)

or in any motor design course, for example Prof. Perriard's electromechanical conversion course.

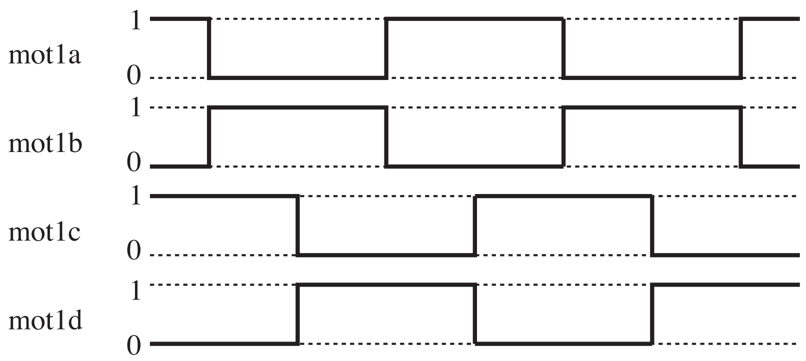

Figure 6: Activation sequence for the motor 1 control signals.

5.1 Code organization

Take a moment to think about the possible implementation of a stepper motor control allowing to change the speed and the direction of the motor rotation. This can be managed, for instance, by a state machine (figure 7) where you can control the state changing direction (go forward or backward) and the speed of state transitions (speed of the robot).

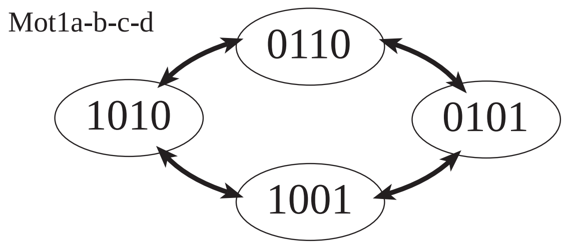

Figure 7: State machine with the states of motor control.

The 4-bit value in the figure 7 indicates the values you have to put in the 4 binary signals that control the windings of the stepper motor. To make the motors turn, you will have to make these signals change in a cyclical way according to figure 7. Two timers, one for each motor (we recommend timers 6 and 7) will be used to do so: each timer interrupt will make a motor make one step. Of course, you will have to set the interval between timer interrupts according to the desired speed of the robot.

Think also how to have a step counter that allows you to move for a given number of steps. Do not consider trapezoidal speed profiles, just rectangular (on-off) speed profiles.

5.2 Functions

All the functions are already declared in **motor.c** and **motor.h**. What is asked is to complete them.

Task 11:

Open the project **TP2_move** (available on the moodle page) and complete the necessary functions to control the two stepper motors of the e-puck2 robot with a certain speed and/or a given target position. The next points will help you through the different steps:

- **identify the pins of the microcontroller that are linked with the H-bridges that are controlling the left and right motors**
- **look at the motor.c file in order to read the definitions of the functions to implement**
- **write a stepper motor driver for constant speed using timers TIM6 and TIM7 for the right and left motor in motor.c, using the timer interrupt to make one motor step. TIM6 is similar to TIM7, you can reuse the code already written for TIM7 in the previous TP. Use a timer frequency of about 100kHz (already defined in the code), then the speed will be given by the counter value you will assign to the ARR register. The more the counter value will be, the slower the steps will be. Don't use a speed above 13cm/s**
- **add motor.c in the makefile on the line listing the .c files to compile**
- **run your code**

Test these functions with a simple demo that moves forward for 10 cm, turns 180 degrees, and comes back to the starting point. In your main you should have only calls to the functions given above, no other motor manipulation.

WARNING:
It is important to set each phase of the motor to 0 when the robot does not have to move, or if the program is stopped, thus ensure that no phase is active continuously, otherwise you risk damaging the motors!

5.3 Library

Compile your project again to be sure to have the `.o` files. The idea here is in the file explorer window (not Eclipse) to move out of the project's folder `motor.c` and to only let `motor.o` and `motor.h`. Then in the Makefile, delete the mention to `motor.c` and add `motor.o` to the variable `LIB_OBJS`. Make sure the Makefile has been saved and you can now clean the project and recompile it.

Task 12:
The effect on the results at the execution is the same, but what is the difference from a developer point of view? What are the advantages of this configuration? What are the disadvantages?

To understand how a library is generated, look at Figure 8.

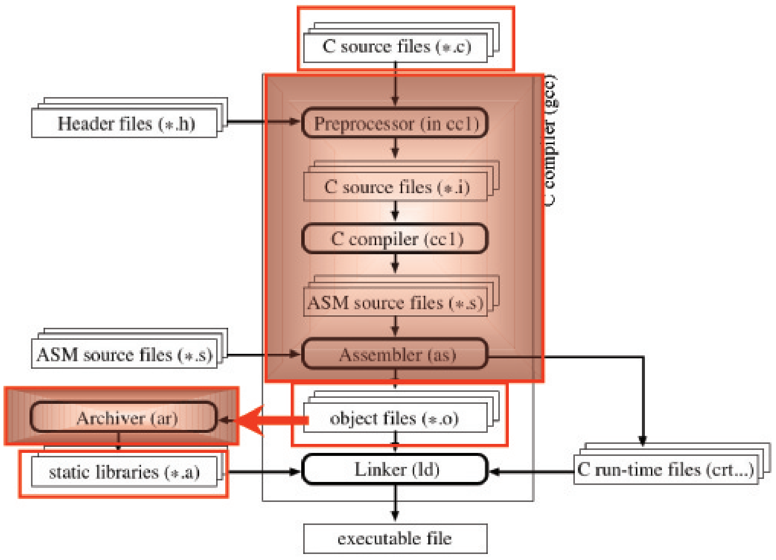

Figure 8: Path for the generation of a library.

Now let's create a library archive which will contain `motor.o`, `gpio.o`, `timer.o` and `selector.o`. To create the library, return to the command line window used previously (or open a new one and reconfigure the PATH), go to the directory containing the files of the projects (command cd) and type the following command:

```
arm-none-eabi-ar -q libtp2.a gpio.o motor.o timer.o selector.o
```

Now in the file explorer window (not Eclipse), a `libtp2.a` file should have appeared. Move out of the project's folder `motor.o`, `gpio.o`, `timer.o` and `selector.o`, then edit the Makefile to delete all the mention to these files (.c and .o) and add `libtp2.a` to the variable `LIBS`. You should now be able to compile the project.

Task 13:

Do you see a difference? What is the difference? When do we use a library instead of simply object files? Understand what **ar** is really doing by looking to its definition under **http://en.wikipedia.org**.

If your program (that includes the library) doesn't work, it might be that you did a mistake in your library. Always check that your library is correct before trying to use it somewhere else!