

Computer architectures

Lab manual

Coordinator: prof dr ir L. Geurts

Labteachers: L. Bruynseels, G. Vranken

Table of Contents

1	About this manual.....	3
1.1	Learning goals	3
1.2	The hardware	3
1.3	The software.....	4
2	First steps	5
2.1	First working program.....	5
2.2	Memory map and the anatomy of an assembler program	7
2.2.1	Create a new project in your workspace.....	7
2.2.2	Now you can explore the files that STVD had generated for us.	8
2.3	Setup part and the infinite loop JRA	9
2.4	Exercise instructions and digital IO.....	10
2.4.1	Naming of the ports.....	10
2.4.2	Example:.....	11
2.4.3	How to find documentation?	11
2.4.4	Exercise 1.1	12
2.5	CC register	13
2.5.1	Introduction	13
2.5.2	Example 1: INC	13
2.5.3	Conditional jump: JRxx	13
2.5.4	BTJT and BTJF instructions	15
2.5.5	Exercise 1.2: if-then-else in assembler.....	15
2.6	Subroutines and loops	16
2.6.1	Exercise 1.3: a delay loop	16
2.7	Final exercise: the Knight Rider.....	17
2.8	Documentation.....	18
2.8.1	Reference manual:.....	18
2.8.2	Datasheet manual.....	18
2.8.3	Programmers manual.....	18
2.8.4	Assembler linker manual	18
2.8.5	STVD manual	18

1 About this manual

1.1 Learning goals

We will teach you the basics of assembly programming.

By programming in assembly language, you will (hopefully) better understand how concepts from higher level programming languages like C, C# and Java work. The difference between pass by value or pass by reference will be clarified. You will see how pointers and arrays work down to the level of bare metal.

You will also learn how a CPU works: program counter, accumulator and stack pointer will not be mere abstract notions after this course.

In the beginning, programming in assembly language is a bit tedious but once you find your way in all the documentation and the tools, you will discover that there is fun in it.

1.2 The hardware

This year we use a CPU board manufactured by STM ¹. The board features an STM8S105C6T6 microcontroller which has 32 kB flash memory, 2kB RAM memory and 1 kB of EEPROM.

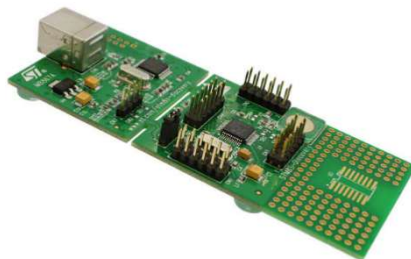


Figure 1-1: CPU board.

The left part of the board has a USB port to connect to a pc. The board is powered over the USB interface and a ST-Link interface is embedded. Hence, no external programmer is needed.

¹ STMicroelectronics is the merger of Thomson Semiconducteurs (F) and SGS Microelettronica (It).

The 4 connectors in the middle are directly connected to the 48 pins of the microcontroller.

On these 4 connectors, we attach a daughter board that is developed in Group T. It features a variety of switches, LED's, a speaker, a Hall sensor, 7 segment display, ... to enable us to design a wealth of interesting programming problems.

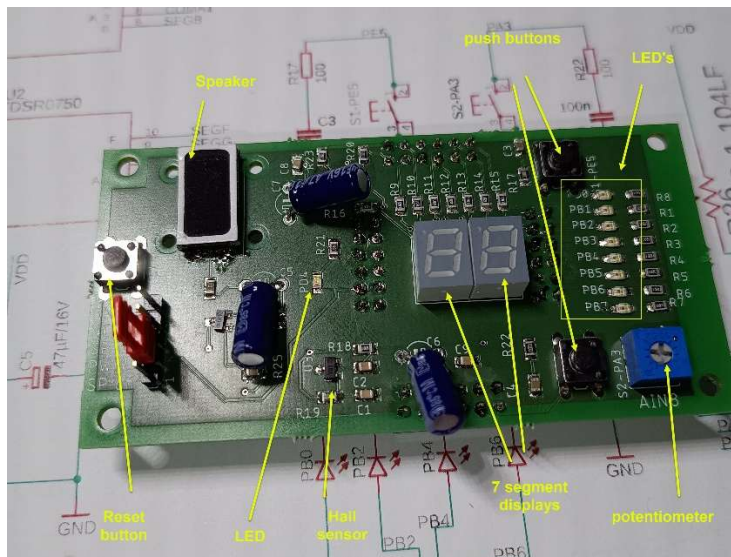


Figure 1-2: the daughter board.

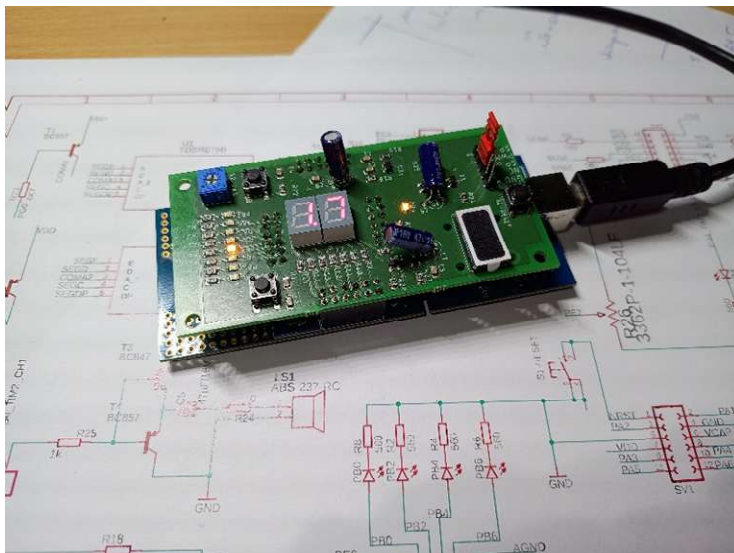


Figure 1-3: 2 boards connected.

1.3 The software

We will develop software with an IDE supplied by STM: ST Visual develop. It can be downloaded

from <https://www.st.com/en/development-tools/stvd-stm8.html>. The installation files are also on Toledo. (STM IDE for STM8)

This program is only available for Windows (10 and 11). We can lend you a windows laptop if you have none available.

The program might work on a Mac computer that runs Windows in a virtual machine, provided that the computer has Intel inside but we can not support you. If the computer runs on Apple silicon, STVD will not work. Don't waste your time if this is your configuration. The problem is the USB driver.

This program should be run as administrator. The easiest way to do this is make a shortcut on your desktop and configure it to always run the program as administrator.

Many features will be explained during the exercise, so don't worry.

2 First steps

2.1 First working program

- Download a workspace from GIT [git@gitlab.kuleuven.be:u0149338/ca-labcourse.git](https://gitlab.kuleuven.be/u0149338/ca-labcourse.git) or the zip file from Toledo.
- Assemble & link: press F7 (Build -> Build)
- Flash the board and start debugging:

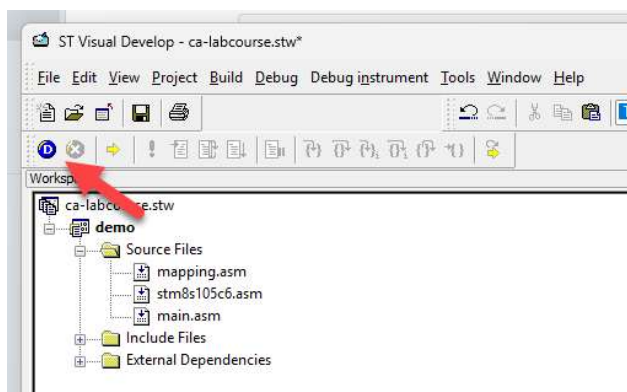


Figure 2-1: Start debugging.

- Meet the debugger

- Start/pause the program

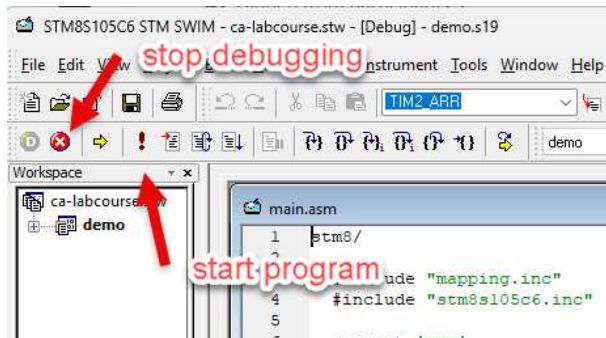


Figure 2-2: start and stop the program.

- Set a breakpoint and step through the program.

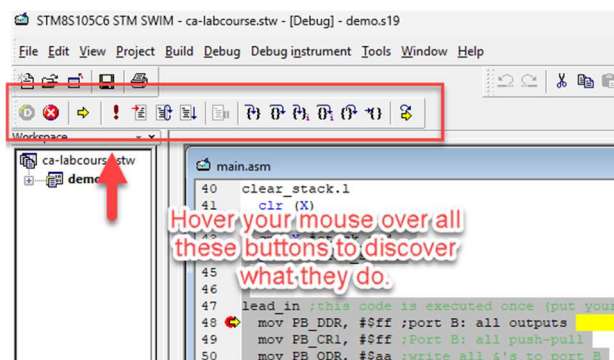


Figure 2-3: debugging commands

- Inspect the registers:

On the menu: View -> Core registers

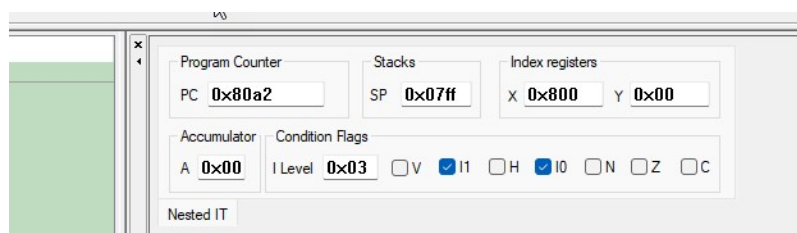


Figure 2 4: View core registers.

- Inspect memory:

On the menubar: View -> Memory

Take home:

Number notation. Decimal, binary, hexadecimal

- Decimal: `mov PB_DDR, #255`
- Hexadecimal: `mov PB_DDR, #$FF` (\$ = Motorola notation, used in this program)
- Binary: `move PB_DDR, #%11110000`

Choose the format that best suits your need.

Take home:

Immediate and direct addressing

- `ld a, #5` => load the number 5 into the accumulator.
- `ld a, 5` => load the contents of memory address 5 into the accumulator.

2.2 Memory map and the anatomy of an assembler program

2.2.1 Create a new project in your workspace

- Right click on 'ca-labcourse.stw', -> add new project to workspace.
- Choose a project name that is not too long.
- Make sure to select a project location in the same folder where you downloaded the workspace!
- Click OK and let STVD create the new folder.

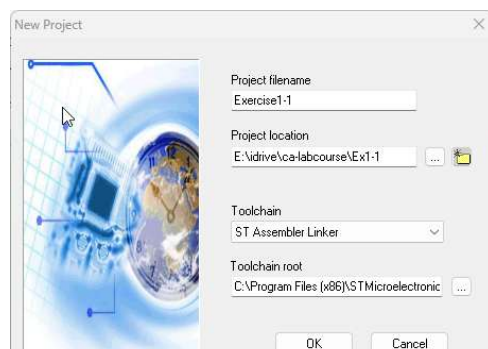
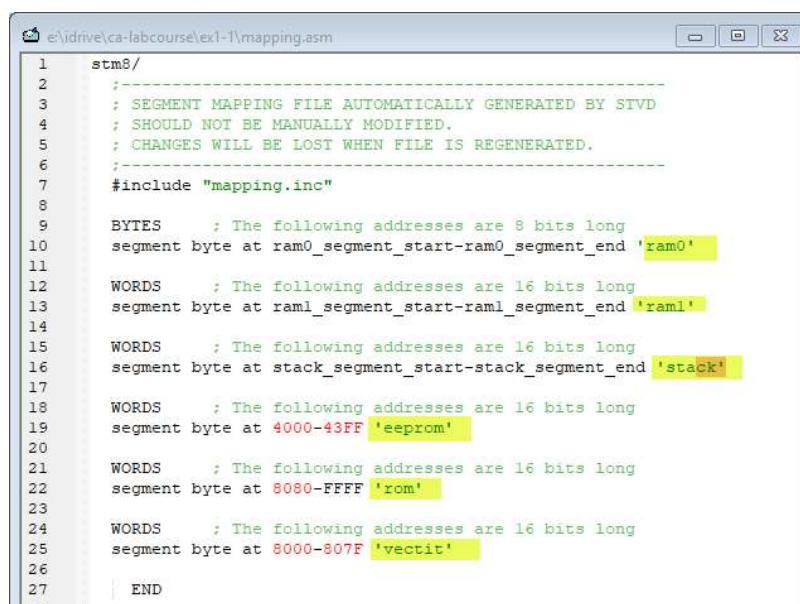


Figure 2-5: Create a new project.

- Select the MCU: stm8s105c6

2.2.2 Now you can explore the files that STVD had generated for us.

Under “Source files” there is main.asm, the starting point for own work. Mapping.asm is automatically generated and should not be modified by programmers. We will briefly examine its contents:



```

1  stm8/
2  ;-----
3  ; SEGMENT MAPPING FILE AUTOMATICALLY GENERATED BY STVD
4  ; SHOULD NOT BE MANUALLY MODIFIED.
5  ; CHANGES WILL BE LOST WHEN FILE IS REGENERATED.
6  ;-----
7  #include "mapping.inc"
8
9  BYTES    ; The following addresses are 8 bits long
10 segment byte at ram0_segment_start-ram0_segment_end 'ram0'
11
12 WORDS    ; The following addresses are 16 bits long
13 segment byte at ram1_segment_start-ram1_segment_end 'ram1'
14
15 WORDS    ; The following addresses are 16 bits long
16 segment byte at stack_segment_start-stack_segment_end 'stack'
17
18 WORDS    ; The following addresses are 16 bits long
19 segment byte at 4000-43FF 'eeprom'
20
21 WORDS    ; The following addresses are 16 bits long
22 segment byte at 8080-FFFF 'rom'
23
24 WORDS    ; The following addresses are 16 bits long
25 segment byte at 8000-807F 'vectit'
26
27 END
28
  
```

Figure 2 6: memory map in code

This file is a description of the memory layout of the MCU: there are 6 blocks:

- ram0 from ram0_segment_start to ram0_segment_end
- ram1 from ram1_segment_start to ram1_segment_end
- stack
- eeprom
- rom
- vectit

Ram = random access memory (R/W), rom = read only memory, eeprom = electrically erasable memory. Stack is in ram with a special function, vectit (interrupt vector table) is in rom with a special function.

The values for xxx_segment_start and xxx_segment_end can be found in mappings.asm.

These values can be different for each member of the big STM8s family of microcontrollers. This

is the reason we had to configure the cpu when we set up the project.

Take home: the total memory space of the MCU is divided in 6 blocks.

2.3 Setup part and the infinite loop JRA

Now we will have a look at main.asm



Figure 2-7: anatomy of the program

- Beginning and end of file. Autogenerated. Beware: there must a CR/LF after the end statement.
- Header files: you must include the `stm105c6.inc` line yourself in every new source file. This file contains symbolic names for every port and register in the MCU. By doing so, we don't have to care about absolute addresses and that is a good thing.

- Line 6: all that follows this instruction goes into rom!
- `main`: this is the entry point of the program. The MCU will always begin execution at this point.
- Line 47: `jra infinite_loop`: jump always to the label `infinite_loop`. Consequences:
 - Code from `main` -> `infinite_loop` is executed only once. Typically used for initialization.
 - Code between `infinite_loop` and `jra` is executed repeatedly. Be aware: the mcu never stops. The clock is ticking for ever.
- Lines 49-51: interrupts are covered later
- Line 53: all that follows this instruction goes into the interrupt vector table. Not in the same area of the rom where the program goes. When the mcu does a power on reset, the program counter is loaded with the instruction found at the first entry of this table. Line 54 of this program fills the table with the correct value.

Take home

- `main`, `infinite_loop` are LABELS. They MUST be far left in the file. Labels must be unique. Case insensitive
- `jra`, `clr`, `incw`, are INSTRUCTIONS and MUST be indented. Case insensitive
- symbolic names like `stack_end` are CASE SENSITIVE.

2.4 Exercise instructions and digital IO

- In this first exercise, we will toggle one or more of the LED's and read from the inputs.

2.4.1 Naming of the ports.

- Each port has a name: PA, PB, PC, ...
- Each port can have up to 8 pins. PA0..PA7, PB0..PB7
- The names of the pins are printed on the PCB: eg the LED's are PB0..PB7

Now, we need to **configure** them.

- Input or output?
- For inputs: activate internal pull up? Yes/no
- Outputs: push pull or open drain.

Assignment: as a budding electronics engineer you must know what pull up/open drain/push-pull means! If you don't know it yet, try to find out in your courses or the internet.

- Each port has 5 registers that control its operation:
 - DDR: datadirection 1 = output, 0 = input
 - ODR: Output data register. The latch that set the state of the output pin when configured as output.
 - IDR: Input register
 - CR1 Control register 1
 - CR2 control register 2.

We will learn the function of these registers very soon.

2.4.2 Example:

```
mov PB_DDR, #$ff ; write ff (hex) to register PB_DDR (output)
bres PA_DDR, #5; Reset port A, pin 5 = input.
mov PB_ODR, #10 ; write 0000 1010 (10 in binary) tot output B
bset PB_ODR, #6 ; set the 6th bit in PB_ODR.
```

Can you tell from the schematic if the outputs are push-pull or open drain?

Do the inputs need the internal pull up resistor?

2.4.3 How to find documentation?

All this is documented in the stm8 reference manual (RM0016, Reference manual.) in chapter 11, General purpose I/O ports. This book is common to all parts in the STM8 family of mcu's.

To make your life easier, here is the part you need today:

11.9.4 Port x control register 1 (Px_CR1)

Address offset: 0x03

Reset value: 0x00 except for PD_CR1 which reset value is 0x02.

7	6	5	4	3	2	1	0
C17	C16	C15	C14	C13	C12	C11	C10
rw	rw	rw	rw	rw	rw	rw	rw

Bits 7:0 C1[7:0]: Control bits

These bits are set and cleared by software. They select different functions in input mode and output mode (see Table 21).

– In input mode (DDR = 0):

0: Floating input

1: Input with pull-up

– In output mode (DDR = 1):

0: Pseudo open drain

1: Push-pull, slope control for the output depends on the corresponding CR2 bit

Note: This bit has no effect on true open drain ports (refer to pin marked "T" in datasheet pin description table).

Figure 2 8: configuration register 1 (from the reference manual)

2.4.4 Exercise 1.1

Continue with the project you created in 2.2.1

To be executed once:

- Configure all pins in port B as output.
- What is the default configuration state? Push-pull or OD? What do we need here?
- Configure pin PA3 as input.
- Pull up activated or not? What is the default value after reset?

To be executed continuously (endless loop)

- Copy the contents of PA_IDR to PB_ODR

Demonstrate to your lab teacher when done.

Instructions used:

```
mov PB_DDR, #$ff
```

```
bset (bit set)
```

```
bres (bit reset)
```

ld a, PA_IDR

2.5 CC register

2.5.1 Introduction

Every instruction might have and probably has an effect on the contents of the CC register. The programmers manual (PM0044 Programming manual) will tell you all the details. This book is by far the most important reference book from STM that you will use in this course. Download the book from Toledo. You will need it in every lesson.

2.5.2 Example 1: INC

STM8 instruction set

PM0044

INC	Increment	INC
Syntax	INC dst	e.g. INC counter
Operation	dst <= dst + 1	
Description	The destination byte is read, then incremented by one, and the result is written to the destination byte. The destination is either a memory byte or a register. This instruction is compact, and does not affect any registers when used with RAM variables.	

Instruction overview

mnem	dst	Affected condition flags						
		V	It	H	lO	N	Z	C
INC	Mem	V	-	-	-	N	Z	-
INC	A	V	-	-	-	N	Z	-

V ⇒ $(A7.M7 + M7.R7 + R7.A7) \oplus (A6.M6 + M6.R6 + R6.A6)$
Set if the signed operation generates an overflow, cleared otherwise.

N ⇒ R7
Set if bit 7 of the result is set (negative value), cleared otherwise.

Z ⇒ R7.R6.R5.R4.R3.R2.R1.R0
Set if the result is zero (0x00), cleared otherwise.

Figure 2 9: The INC instruction

Examine for yourself how the CMP, CLR, BTJF, BTJT instructions work: what do they do and how is the CC affected?

2.5.3 Conditional jump: JRxx

Examples:

- JREQ: the jump (goto) is executed if Z = 1
- JRNE: the jump (goto) is executed if Z = 0

PM0044

STM8 instruction set

JRxx	Conditional Jump Relative Instruction	JRxx
Syntax	JRxx dst e.g. JRxx loop	
Operation	PC = PC+lgth PC <= PC + dst, if Condition is True	
Description	Conditional relative jump. PC is updated by the signed addition of PC and dst, if the condition is true. Control, then passes to the statement addressed by the program counter. Else, the program continues normally.	

Instruction overview

mnem	dst	Affected condition flags						
		V	I1	H	I0	N	Z	C
JRxx	Mem	-	-	-	-	-	-	-

Instruction List

mnem	meaning	sym	Condition	Op-code (OC)	
JRC	Carry		C = 1		25
JREQ	Equal	=	Z = 1		27
JRF	False		False		21
JRH	Half-Carry		H = 1	90	29
JRIH	Interrupt Line is High			90	2F
JRIL	Interrupt Line is Low			90	2E
JRM	Interrupt Mask		I = 1	90	2D
JRMI	Minus	< 0	N = 1		28
JRNC	Not Carry		C = 0		24
JRNE	Not Equal	<> 0	Z = 0		26
JRNH	Not Half-Carry		H = 0	90	28
JRNM	Not Interrupt Mask		I = 0	90	2C

Figure 2. 10: the JRxx instruction (from Programmers Manual)

Example:

```

13 ; clear RAM0
14 ram0_start.b EQU $ram0_segment_start
15 ram0_end.b EQU $ram0_segment_end
16 ldw X,#ram0_start
17 clear_ram0.1
18 clr (X)
19 incw X
20 cpw X,#ram0_end
21 jrnc clear_ram0

```

Figure 2.11: jrnc instruction

Line 21 is the test (look in the PM if you want to know what is tested). If the test passes (is true) execution is transferred to line 17. If the condition of the is not met, execution continues on line 22. Remember: `clear_ram0` is a label must be on the far left of the line. Labels are case sensitive.

2.5.4 *BTJT and BTJF instructions*

Very often, one bit needs to be tested. In such cases, BTJF and BTJT come in very handy:

- BTJT = Bit Test and Jump if True.
 - BTJT PA, #5, next_label: Test bit 5 on port A and jumps to next_label if true.
- BTJF = Bit Test and Jump if False.

2.5.5 *Exercise 1.2: if-then-else in assembler.*

In this exercise we will implement the famous if-then-else pattern in assembler. We build on the work already done in the previous exercise that was started in paragraph 2.4.4

The assignment is simple:

- After reset, all LED's are off.
- If button S2-PA3 is shortly pressed, the pattern \$AA should be sent to the LED's on port B. This means that the LED's with an even number are lit.
- If button S1-PE5 is shortly pressed, the pattern \$55 should be sent to the LED's on port B. This means that the LED's with an odd number are lit.
- If the button is released, the LED's keep their state. In fact we have implemented a set/reset flipflop!
- What to do:
 - Configure PB as output if not done. Turn all LED's off.
 - Configure PA and PE as input. Think about the pull up stuff
 - Draw a flowchart. If you don't draw a flowchart, there is very very little chance that you will finish this exercise in time.
 - Make your life ease: use BTJF or BTJT
 - Secret (I tell it only to you): if you press a button , the corresponding input goes low.
 - Show flowchart and implementation to the labteacher

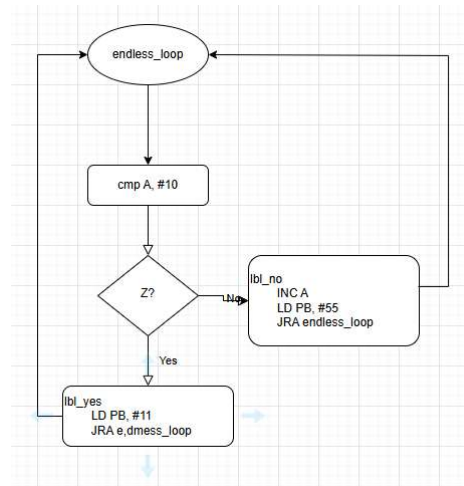


Figure2.15: example of flowchart
(created with app.diagrams.net)

2.6 Subroutines and loops

Another important construct in any programming language is the subroutine. It helps to structure code and make it maintainable. ‘Good’ code behaves correctly and is easy to maintain. This applies to any program language.

In assembler the construct is very easy: you just have to put the call statement.

Ex : call myRoutine (my routine is a label)

When the work is done, you just need to call the ret statement

```

...
    call myRoutine ; call the subroutine
...
...
myRoutine ; label to indicate the start of the routine
    do work
    do more work
    ret ; end of subroutine.
  
```

2.6.1 Exercise 1.3: a delay loop

Assignment:

- In your existing program, create a subroutine name myDelay. The body of this subroutine

has only 1 instruction: nop (apart from the ret statement) The correct location of this subroutine is just after the endless loop. By doing this, the code in the subroutine will not be executed unless called from another place.

- Call this subroutine from within your endless loop. Check with the debugger (step by step execution) that you jump into the routine.
- In the main program (endless loop) toggle one of the LED's (BCPL PB_ODR, #1)
- Within the myDelay routine:
 - Load the X register with \$ffff (X is 16 bit => LDW instead of LD)
 - Program a loop such that X is decremented by 1 in every iteration. Exit from the loop when X becomes 0
 - Exit the subroutine: RET
 - In Java-like code, this would be

```
void myDelay() {
    X = $FFFF;
    While (X > 0) {
        Dec X;
    }
    Return;
}
```

We recommend to draw a flowchart before you start coding. It can make you save time. But it is your life...

You will see that this routine takes quite some time to finish. Experiment with different start values in X.

Would there be a way to set this start at runtime?

If you master this, you are able to implement all looping patterns in assembly language:

```
For...next
Do ... while() {...}
While() {...}
```

2.7 Final exercise: the Knight Rider

The Knight Rider was a popular TV series in the previous century. The hero in this series drove a car named KITT. This car looked like a Pontiac Firebird Trans-Am 1982 but was loaded with a ton of electronic gadgets including AI. There is a bar of LED's (scanners) that can pulse in different

patterns and/or sweep rapidly or slowly.

See the effect: <https://youtu.be/WxE2xWZNfOc?si=nEfsrvOJK9wTXSYt>

We will now simulate this in the LED's of PB.

- Start with PB0 and PB1 on; other LED's off
- After a delay (you have now a delay routine) shift the pattern one position.
- Repeat till you led's fall off at the other side.
- Restart. (or shift them back to the other side).

There are instructions for rotate and shift left or right. Find them in the programmes manual.

Check what these instructions do in the CC register

Jump accordingly.

...

2.8 Documentation

2.8.1 Reference manual:

- Applies to all members of stm8 family

2.8.2 Datasheet manual

- Specific for each member of stm8 family. Each device may have different amount of memory, I/O, number of timers, ...

2.8.3 Programmers manual

Be sure to download this manual. You will it very often.

2.8.4 Assembler linker manual

If you want to know more about assembler directives (the lines that start with a '#')

2.8.5 STVD manual

