

Computer architectures

Lab manual

Coordinator: prof dr ir L. Geurts

Labteachers: L. Bruynseels, G. Vranken

Table of Contents

1	About this manual.....	1
1.1	Learning goals	1
1.2	The hardware.....	1
1.3	The software.....	2
2	First steps	1
2.1	First working program.....	1
2.2	Memory map and the anatomy of an assembler program	3
2.2.1	Create a new project in your workspace.....	3
2.2.2	Now you can explore the files that STVD had generated for us.	3
2.3	Setup part and the infinite loop JRA	4
2.4	Exercise instructions and digital IO.....	6
2.4.1	Naming of the ports.....	6
2.4.2	Example:.....	7
2.4.3	How to find documentation?	7
2.4.4	Exercise 1.1	8
2.5	CC register	9
2.5.1	Introduction	9
2.5.2	Example 1: INC	9
2.5.3	Conditional jump: JRxx	9
2.5.4	BTJT and BTJF instructions	11
2.5.5	Exercise 1.2: if-then-else in assembler.....	11
2.6	Subroutines and loops	12
2.6.1	Exercise 1.3: a delay loop	12
2.7	Final exercise: the Knight Rider.....	13
2.8	Documentation.....	14
2.8.1	Reference manual:.....	14

2.8.2	Datasheet manual.....	14
2.8.3	Programmers manual.....	14
2.8.4	Assembler linker manual.....	14
2.8.5	STVD manual	14
3	Variables, interrupts and timers	1
3.1	Variables	1
3.2	Example	1
3.3	Run the example	3
3.4	Interrupts	6
3.4.1	Introduction: polling vs interrupts	6
3.4.2	Interrupts on digital inputs.....	7
3.5	Exercise 1	8
3.6	Timers	9
3.6.1	Timer 2 as PWM generator	10
3.7	Exercise 2	11
3.8	TIM3 as timer	11
3.9	Exercise 3	11
3.10	Challenge	11
4	Addressing modes	1
4.1	What have we learned so far	1
4.1.1	Error with dll registration.....	1
4.1.2	If then else and loops	1
4.1.3	Digital I/O.....	2
4.1.4	Set up a workspace	2
4.1.5	TIM2	3
4.1.6	TIM3	3
4.1	Addressing modes	4
4.1.1	Immediate addressing.....	4
4.1.2	Direct addressing	4
4.1.3	Indexed addressing	4
4.1.4	Exercises	5
5	Appendices.....	1

6	Documentation	4
6.1	The instruction set	4
6.2	The reference manual	4
6.3	The datasheet.....	4
6.4	The IDE.....	4

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 STM¹. The board features an STM8S105C6T6 microcontroller which has 32 kB flash memory, 2kB RAM memory and 1 kB of EEPROM.

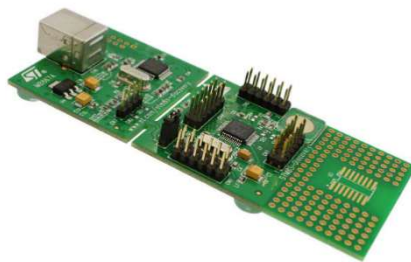


Figure 1-1: CPU board.

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.

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

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

variety of switches, LED's, a speaker, a Hall sensor, 7 segment display, ... to enable us to design a wealth of interesting programming problems.

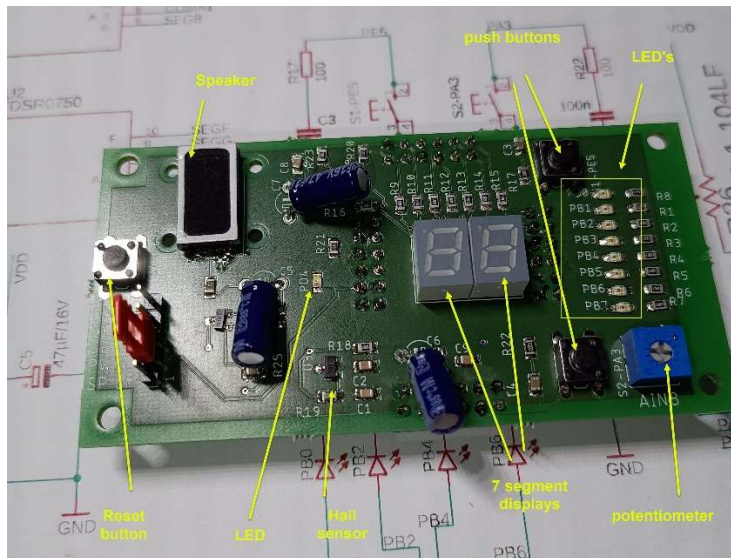


Figure 1-2: the daughter board.

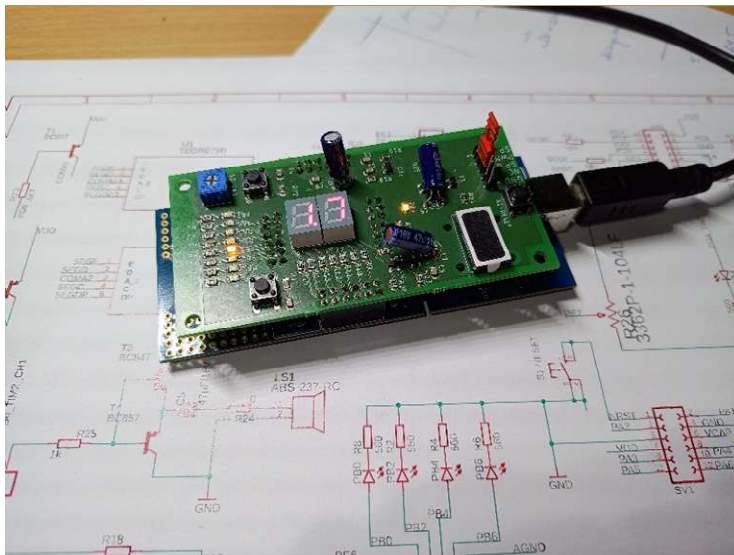


Figure 1-3: 2 boards connected.

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:

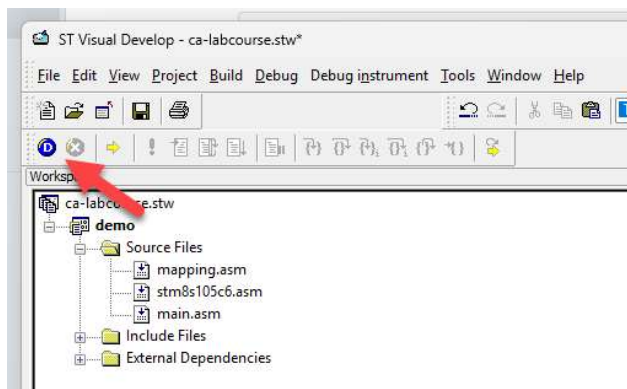


Figure 2-1: Start debugging.

- Meet the debugger
 - Start/pause the program

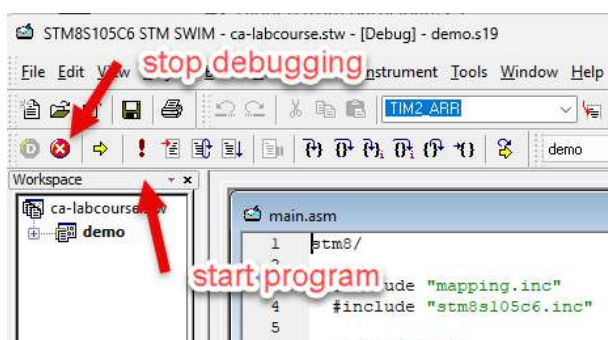


Figure 2-2: start and stop the program.

- Set a breakpoint and step through the program.

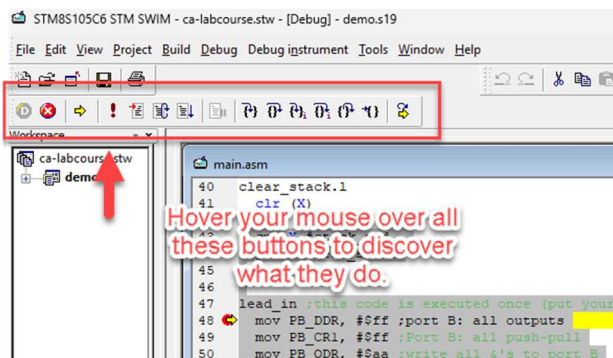


Figure 2-3: debugging commands

- Inspect the registers:

On the menu: View -> Core registers

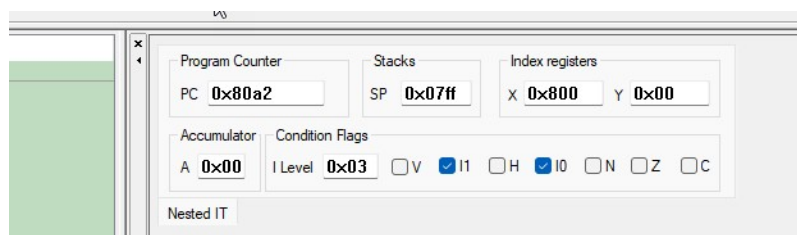


Figure 2 4: View core registers.

- 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.

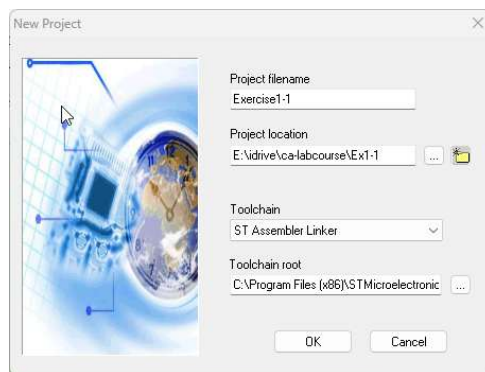
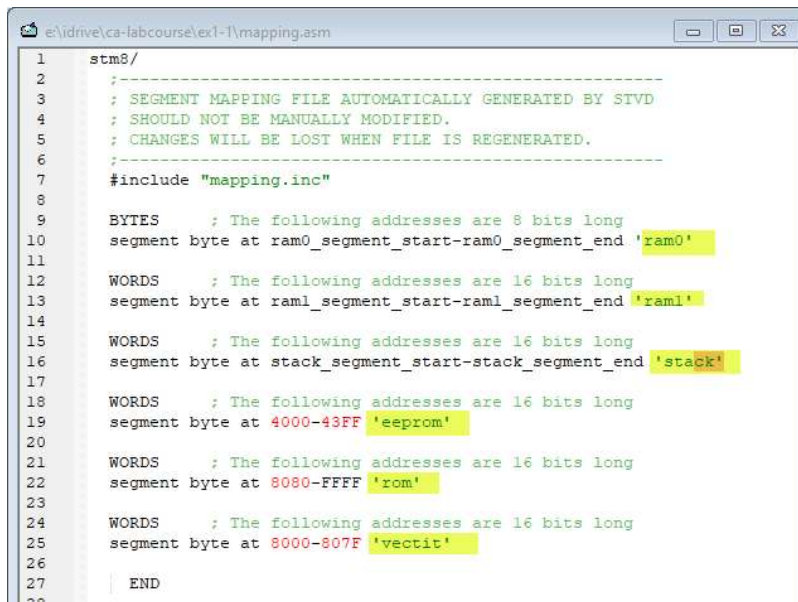


Figure 2-5: Create a new project.

- 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 SIVD
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 be 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.

2.4.2Example:

```
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?

2.4.3How 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)

2.4.4Exercise 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	I1	H	I0	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

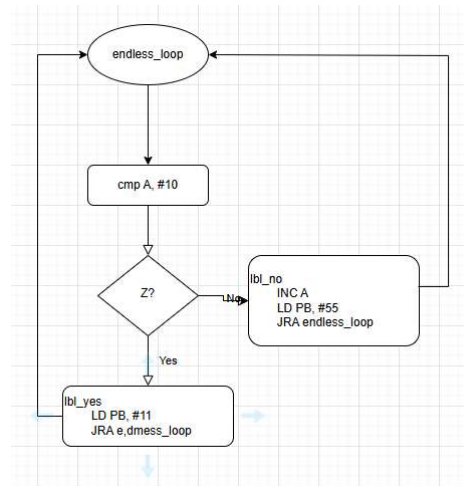


Figure2.15: example of flowchart
(created with app.diagrams.net)

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, ...

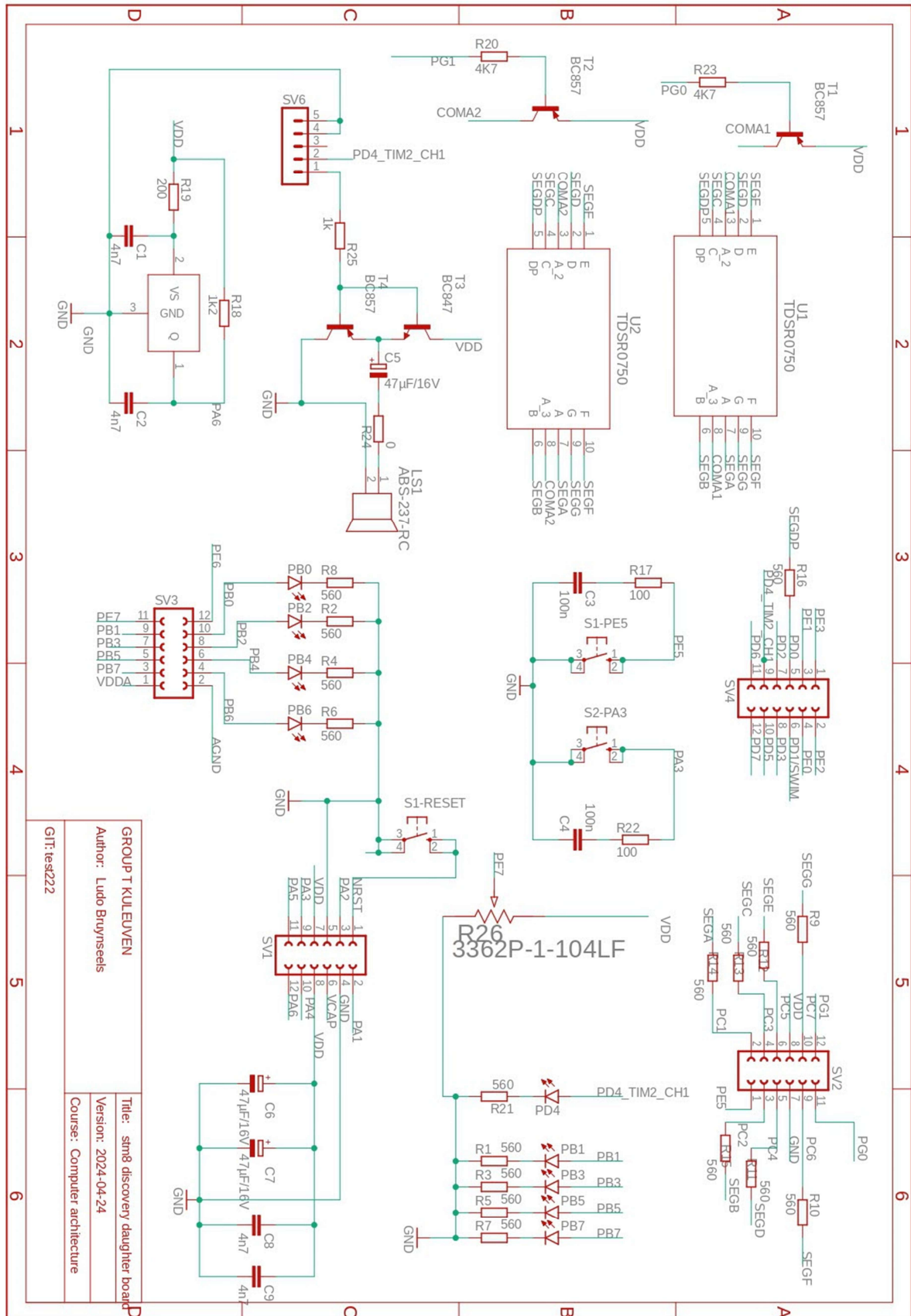
2.8.3 Programmers manual

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



3 Variables, interrupts and timers

3.1 Variables

Every programming language needs variables. Variables live somewhere in memory and can be written and read whenever the program has the need.

In this environment we were working on 'bare metal' and have also access to ROM memory when we develop the application. This is useful to store constants that must be available after a power on reset.

As you already know, each memory cell has an address. To store 1 byte, we need 1 memory cell. To store a word, we need 2 memory cells. So, if there is 1 byte at address 0 and a word at address 1, the next byte would go to address 3.

By consequence, as soon as you declare more than 2 variables, the bookkeeping of these variables becomes a nightmare. If you change the size of 1 variable from byte to word (2 bytes), a can of worms opens and spread all over the place.

But Hooray, here comes the linker!

The assembler translates all instructions to binary opcodes but does not assign addresses to variables and labels. The assembler cannot do that because a whole project can consist of many files and variables and labels can be defined in one file and used in other files. The assembler treats the source files one by one and writes out object files.

The linker reads all the object files (generated by the assembler) and assigns addresses. This is not just a concatenation of the object modules. It resolves all the external references and if a referenced label is not defined as PUBLIC an error is detected. The linker also places the segments according to the mapping and checks if any of them is overrun.

3.2 Example

The keywords ds and dc are explained later. We ask a bit of patience.


```

main-ex1.asm
1  stm8/
2
3  #include "mapping.inc"
4  segment 'ram0'
5  myarray ds.w 10
6  mycounter ds.b
7
8  segment 'rom'
9  mystring dc.b "Hello World!"
10
11 main.l
12 ; initialize SP
13 ldw X,#stack_end
14 ldw SP,X
15
16 ;
17
18 ;
19
20 ;
21
22 ;
23
24 ;
25
26 ;
27
28 ;
29
30 ;
31
32 ;
33
34 ;
35
36 ;
37
38 ;
39
40 ;
41
42 ;
43
44 ;
45 ;
46 ;
47 ;
48 ;
49 ;
50 mov mycounter , # 87
51 infinite_loop.l
52 inc mycounter
53 jra infinite_loop
54
55 interrupt NonHandledInterrupt
56 NonHandledInterrupt.l
57 iret
58
59 segment 'vectit'
60 dc.l {$82000000+main} ; reset
61 dc.l {$82000000+NonHandledInterrupt} ; trap
62 dc.l {$82000000+NonHandledInterrupt} ; irq0
63 dc.l {$82000000+NonHandledInterrupt} ; irq1
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91 dc.l {$82000000+NonHandledInterrupt} , irq29
92
93 end
94

```

Line 4 is inserted because we want to declare variables in the ram0 segment.

Line 5 reserves 10 words (20 bytes) for a variable named myarray.

Line 6 reserves 1 byte for a variable named mycounter.

DS means Define Space. Initializing ram doesn't make much sense.

On line 8, we switch to the rom segment.

Line 9 defines a string in rom. Assembler is not aware of the type 'string' but it initializes the following memory cells with bytes. These bytes happen to be printable characters that form the words "Hello World!".

Line 50 initializes mycounter to 87

In the main loop, we increment mycounter.

3.3 Run the example

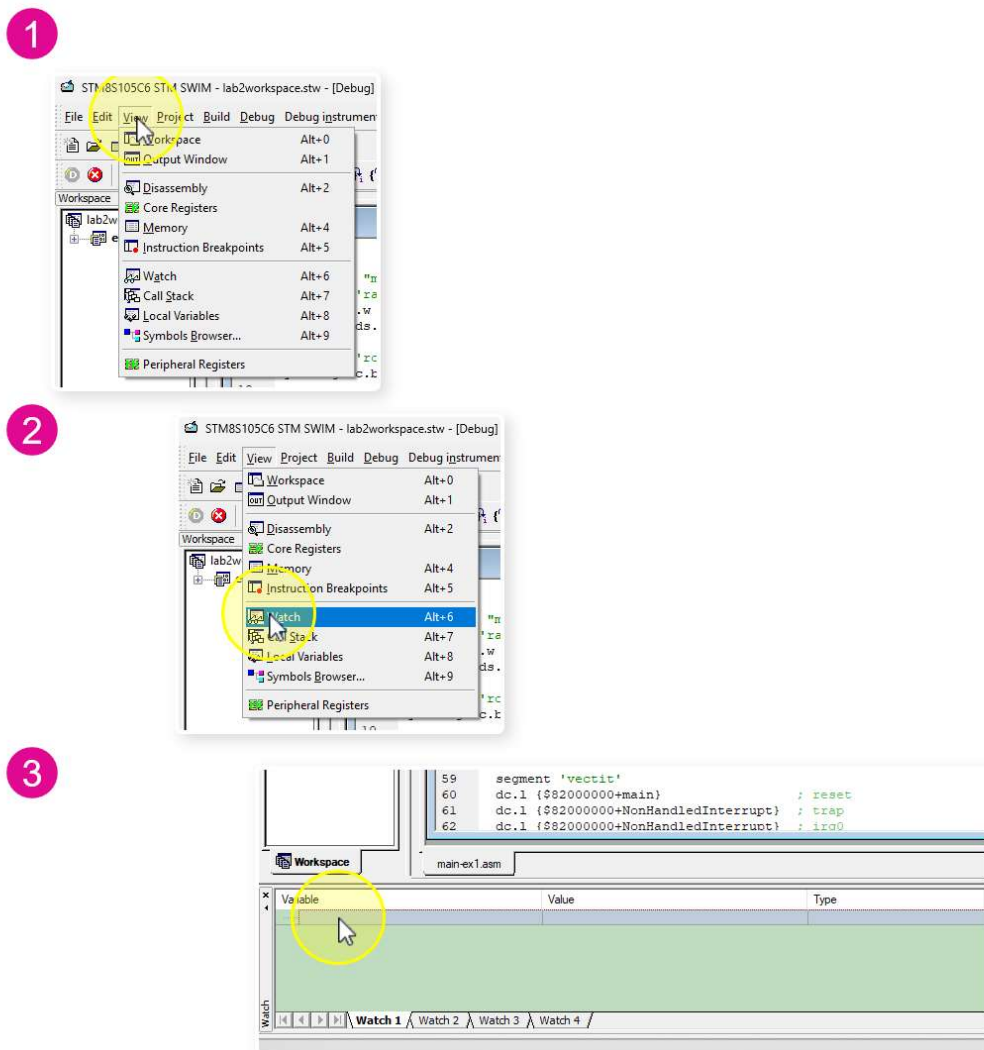
Download the workspace ca-labcourse2.zip

Unzip the file, open the workspace with STVD and make example1 the active project.

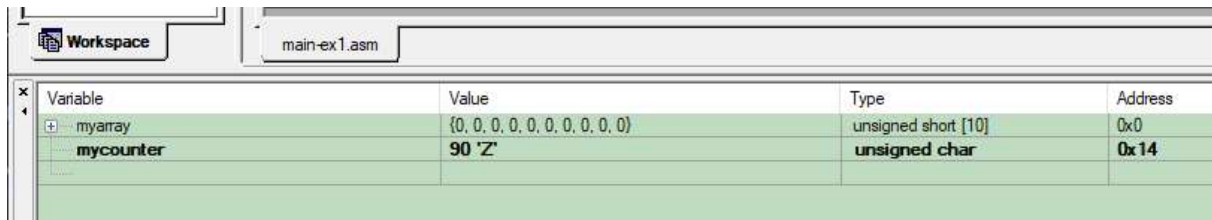
Set a breakpoint at line 52 and start the program.

Now, the demo program should be halted at the “inc mycounter” instruction.

We will open the watch window:



In the watch window, enter the name of the variables:



The screenshot shows a debugger's watch window with two variables: 'myarray' and 'mycounter'. 'myarray' is of type 'unsigned short [10]' and contains the value '{0, 0, 0, 0, 0, 0, 0, 0, 0, 0}' at address '0x0'. 'mycounter' is of type 'unsigned char' and contains the value '90 'Z'' at address '0x14'.

Variable	Value	Type	Address
myarray	{0, 0, 0, 0, 0, 0, 0, 0, 0, 0}	unsigned short [10]	0x0
mycounter	90 'Z'	unsigned char	0x14

myarray lives at address 0x0 and is initialized to all 0.

mycounter lives at address 0x14 and contains the value 90. $0x14 = 20$ decimal. This is what we expected: The first array contains 10 words which is 20 bytes. These are addresses 0..19. The next available address is 20 (0x14). So the linker does it right!

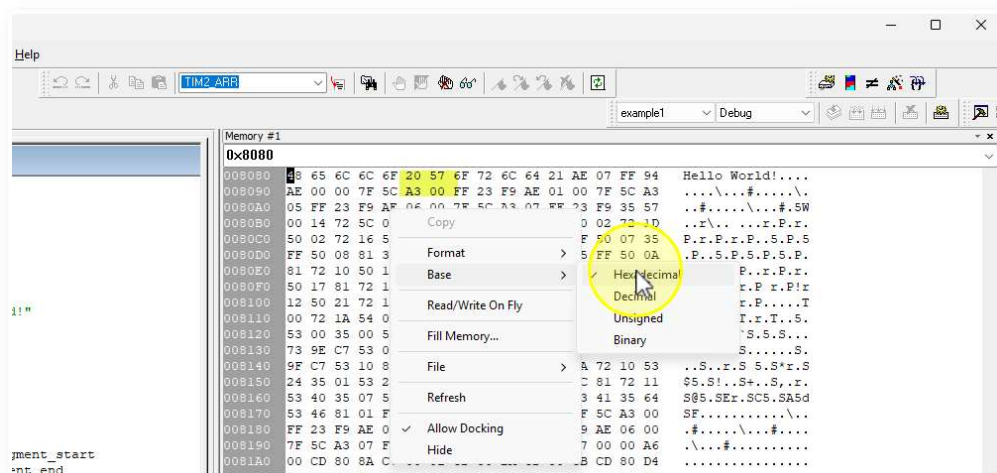
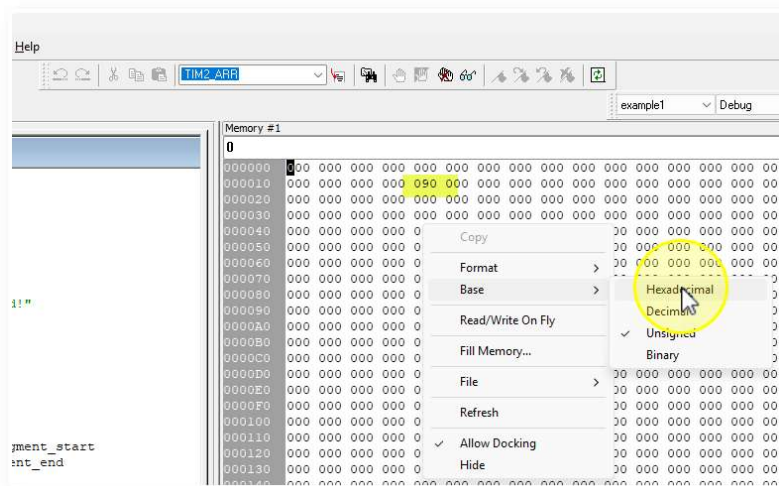
Step through the program step by step and see how the value changes.

Right click on the watch window and examine how the presentation can be changed in hex, decimal, unsigned, binary

Add 'mystring' to the watch window. At which address lives this variable?

With the program still halted, examine the contents of the memory:

The same as above, click view -> memory and arrange the windows properly.



- Bookkeeping of addresses is done by the linker
- Use symbolic names for variables
- In RAM segments:
 - ds.b (define space). Reserves 1 byte.
 - ds.w (define space). Reserves 1 word.
 - ds.b 10 reserves 10 bytes.

In ram segments, do not initialize. Value will be reset to 0.
- In ROM segments
 - dc.b (define byte)
 - dc.w (define word):

In rom segments, do initialize. Your program cannot write to

rom!

- Examine contents in the Watch window. Also find the physical addresses in this window.
- Examine the contents of the memory in the memory window.

3.4 Interrupts

3.4.1 Introduction: polling vs interrupts

A MCU has many peripherals: digital inputs, timers, ADC, different communication systems like UART, SPI, TWI, ... If you want to query the status of all those subsystems, you could do so by checking them one by one in an endless loop. You could check if an AD conversion has finished, whether data has come in on SPI or whether your data has been send on the UART,...

This technique is called '**polling**' and has some drawbacks.

- The polling period is not exactly known: it depends on the total amount of work that has to be done in the endless loop. For instance, when data has come in your program, you have to process them and the loop takes more time.
- Most of the time, nothing happens.
- You might miss events that last shorter than the interval between 2 polling cycles. Consider the program we developed in session1: we made the main loop deliberately very slow. It is imaginable that a user presses an input button so short that the main loop misses it completely.
- It is a lot of work to program all the querying and get the if/then/else statements right.

To overcome this all the peripherals can generate **interrupts**. This means that a peripheral calls a special subroutine when an event occurs.

This kind of subroutine has a name: it is called an Interrupt Services Routine (ISR). And that is exactly what it does: it serves an interrupt.

Analogy from real life: you did an audition in a great movie studio and you return home full of hope. When you left they told you not to call. "Don't call us, we 'll call you". Of course, when you get home, you are very impatient and you are very tempted to call anyway. After 50 or so calls, you know all the excuses: the manager is not in, I left a message, they haven't decided yet, ...

- In s/w development they call this the Hollywood principle: don't call us, we 'll call you.

- In our case, when a timer expires, when somebody presses a switch, when new data come in, ... a special routine is called: the ISR. This is handled in hardware, you don't have to do anything. As soon as the interrupt occurs, the PC is loaded with the appropriate address and execution continues where it should.
- In higher level languages like C and Java this mechanism is implemented using callback functions. It is not a language feature, it is a service of the operating system, which we don't have in small embedded systems.

3.4.2 Interrupts on digital inputs

3.4.2.1 Step 1: To enable interrupts on digital inputs, we need to configure the CR2 (control register 2).

11.9.5 Port x control register 2 (Px_CR2)

Address offset: 0x04
Reset value: 0x00

7	6	5	4	3	2	1	0
C27	C26	C25	C24	C23	C22	C21	C20
rw	rw	rw	rw	rw	rw	rw	rw

Bits 7:0 **C2[7:0]**: Control bits

These bits are set and cleared by software. They select different functions in input mode and output mode. In input mode, the CR2 bit enables the interrupt capability if available. If the I/O does not have interrupt capability, setting the CR2 bit has no effect. In output mode, setting the bit increases the speed of the I/O. This applies to ports with O3 and O4 output types (see pin description table).

– In input mode (DDR = 0):

0: External interrupt disabled

1: External interrupt enabled

– In output mode (DDR = 1):

0: Output speed up to 2 MHz

1: Output speed up to 10 MHz

3.4.2.2 Step 2 set up an interrupt service routing (ISR)

Every interrupt is coupled with an address in memory. Should the interrupt occur, the program counter is loaded with the contents of this address

These addresses are called 'interrupt vectors'. The interrupt vectors are flashed in the MCU at a fixed location. (See memory map in the datasheet and see lecture notes).

```

60
61 segment 'vectit'
62 dc.l {$82000000+main} ; reset
63 dc.l {$82000000+NonHandledInterrupt} ; trap
64 dc.l {$82000000+NonHandledInterrupt} ; irq0
65 dc.l {$82000000+NonHandledInterrupt} ; irq1
66 dc.l {$82000000+NonHandledInterrupt} ; irq2
67 dc.l {$82000000+NonHandledInterrupt} ; irq3
68 dc.l {$82000000+NonHandledInterrupt} ; irq4
69 dc.l {$82000000+NonHandledInterrupt} ; irq5
70 dc.l {$82000000+NonHandledInterrupt} ; irq6

```


This default configuration make every interrupt jump to the NonHandledInterrupt label. In the next exercise, we will make an interrupt for the PA3 and PE5 ports. So we must find out which interrupt level is appropriate for these ports.

Consult Table 1: Interrupt Vector mapping in the appendices

This table comes from the datasheet of the STMs8105. The values may be different for other family members.

The first one (which has ;reset as comment is the entry point of our program. This means that the MCU will start executing at this point upon power reset. Execution in our program starts at the 'main' label.

We see that PortA external interrupts are 'EXTI0' interrupts aka IRQ n°3. Find the values for the port E.

By default, all interrupts are handled at the NonHandledInterrupt label: should they occur, they do nothing.

3.4.2.3 The Interrupt service routine

The last instruction of an ISR is IRET.

What is the difference between RET and IRET?

3.4.2.4 Step 3: sensitivity of interrupts.

See Table 2: External Interrupt control register 1 and Table 3: External Interrupt Control Register 2.

Step 1: enable interrupts on the port

Step 2: implement an ISR and update the interrupt vector table.

Step 3: set the sensitivity on which the interrupt should fire.

Step 4: do RIM just before the main loop. RIM enables all interrupts. If you omit this, you will not see any interrupt.

3.5 Exercise 1

- Create a new project in the workspace you downloaded.
- Configure port B as output (all 8 bits)
- Enable interrupts on port A and on port B
- Configure a variable in ram. Name it myCounter

- Initialize this counter to 100 after every reset.
- When you press PA3, the counter should increment and the LED's should reflect the value of the counter in binary.
- When you press PE5, the counter should decrement and the LED's should reflect the value of the counter in binary.
- Implement the ISR's and configure the table.
- Which sensitivity are you using?

3.6 Timers

Configuration of timers is difficult in every MCU. The reason for this is because they are so versatile. If you have some spare time, have a look at the different timers in the STM8 Reference manual.

To make your life easier, we will not use all possibilities and give you the configuration of the timers for free.

3.6.1 Timer 2 as PWM generator

In our configuration, the output of Timer2 is directly connected to PD4. But we still have to configure PD4 as output and, on our board as push-pull output. But by now you should now how to do that.

To see how it works, study following figure from the reference manual.

Figure 2. PWM generation principle

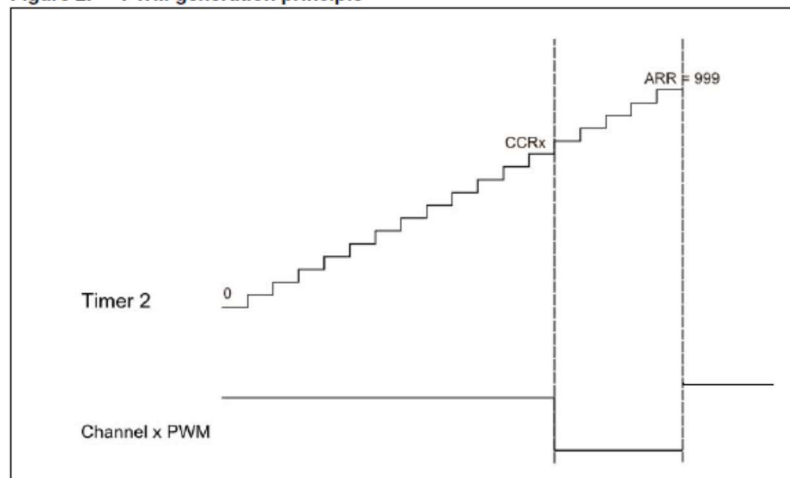


Figure 16: PWM generation principle

At T_0 , the timer starts counting. When the value CCR2 is reached, the output is toggled but the counting goes on.

When the ARR value is reached, the counter is reset to 0, the output toggles again and the process restarts.

The configuration of TIM2 is given.

The value of ARR (Automatic Reload Register) sets the frequency. The clock of the counter has a value of 2 MHz.

The value of CCR sets the duty cycle. If CCR is 50% of ARR, the duty cycle is 50%

CCR and AAR cannot be written as 16 bit register.

You have to do:

```
ldw x, #some value
```

```
ld a, xh
```

```
ld TIM2_ARRH, a      or ld, TIM2_CCR1H, a
```

```
...
```

3.7 Exercise 2

- Configure D4 as output
- Configure TIM2 to run at 5 kHz, duty cycle is 25%
- Measure the result on an oscilloscope.
- Change the value of the duty cycle at will.
- Use PA3 and PE5 to start/stop the timer.

3.8 TIM3 as timer

We will use TIM3 to generate a periodic interrupt. The configuration of the timer is given.

We will use the prescaler to generate longer periods.

The concept of duty cycle of meaningless in this context.

3.9 Exercise 3

Make one LED on the B port blink at a rate of twice a second (2 Hz, T = 500 ms)

You must clear the flag in SR1, otherwise the program will get stuck in the interrupt!

```
BRES TIM3_SR1,#0 ; clear flag
```

This must be the last instruction in your ISR, just before the IRET

3.10 Challenge

(Re)do the Knight rider but do all the work in the ISR not in the main loop.

4 Addressing modes



4.1 What have we learned so far

4.1.1 Error with dll registration

STVD Connection error (usb://usb): gdi-error [40201]: can't access configuration database STVD
DAO PROBLEM

Run following commands as administrator:

```
Regsvr32 /u "C:\Program Files (x86)\Common Files\Microsoft Shared\DAO\DAO350.DLL"
```

```
Regsvr32 "C:\Program Files (x86)\Common Files\Microsoft Shared\DAO\DAO350.DLL"
```

4.1.2 If then else and loops

- Draw flowcharts: they might help to get the logic right.
- Almost every instruction affect the Condition Code (CC) flags:
 - o CP sets the Z flag if the operands are equal
 - o INC and DEC set the Z flag if the result is 0
 - o Check the programmers manual to find out what your code does with the CC flags.
 - o The state of a flag is only valid till the next instruction is executed.

- The JRxx instructions check the appropriate CC flag. For instance JREQ jumps if the Z bit is set. JRNE jumps if the Z bit is cleared.

4.1.3 *Digital I/O*

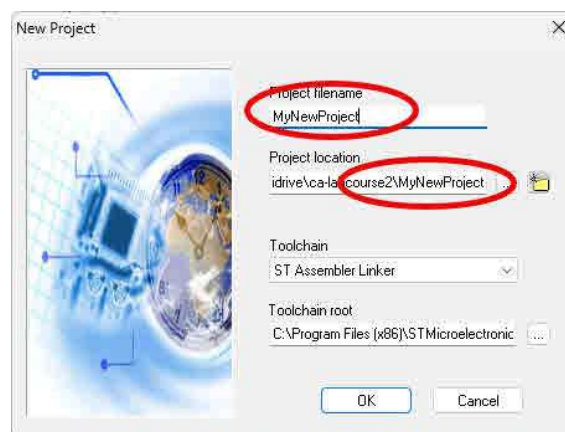
- 5 registers:
 - DDR
 - IDR (for inputs)
 - ODR (for outputs)
 - CR1
 - CR2

4.1.4 *Set up a workspace*

When you set up your own workspaces and projects, STVD is not very helpful in choosing the right directories.

So

- Create your workspace somewhere in a place where Windows allows you to. This means: not somewhere in c:\program files
- New projects must go in a new folder



4.1.5 TIM2

We always use following config for TIM2, although it has much more possibilities:

```
MOV  TIM2_CR1, #00000001 ; counter enable ON
MOV  TIM2_IER, #000      ; no interrupts
MOV  TIM2_CCMR1, #01100000 ; PWM mode 1 + CC1 as output

MOV  TIM2_CCER1, #00000001 ; enable CC1 output
```

In this configuration, TIM2 works as a PWM modulator. If CC1 is enabled, it is directly connected to PD4.

We can turn the timer on or off with bit 0 in TIM2_CR1.

The frequency is configured in TIM2_ARR:

```
ldw x, #12000
ld a, xh
ld TIM2_ARRH, a
ld a, xl
ld TIM2_ARRL, a
```

The dutycyle is configured in TIM2_CCR1

```
ldw x, #3000
ld a, xh
ld TIM2_CCR1H, a
ld a, xl
ld TIM2_CCR1L, a
```

4.1.6 TIM3

TIM3 is configured to generate a period interrupt.

```
MOV  TIM3_CR1, #00000001 ; timer on
MOV  TIM3_PSCR, #007     ; prescaler x128
BSET TIM3_EGR, #0        ; force UEV to update prescaler
MOV  TIM3_IER, #001      ; TIM3 interrupt on update
                             enabled
```

The period can be configured by supplying values to TIM3_ARRH and TIM3_ARRL. Eventually the value of the prescaler must be changed as well.

Do not forget to put

```
BRES TIM3_SR1, #0 ; clear flag
```

as last instruction on your ISR.

4.1 Addressing modes

4.1.1 Immediate addressing

```
LD A, #1
```

4.1.2 Direct addressing

```
LD A, 1
```

Or

```
Segment ram0
Count ds.b
Segment rom
...
...
LD A, Count
```

4.1.3 Indexed addressing

=> retrieve data at a **base address + an offset**.

eg: base = 0x0800, index = 0x10, indexed addressing would retrieve data at address 0x8010

example application: **arrays**.

- **Name of the array equals the address of the array.**
- **Indexed values are at the base address + index.**
- **Example:**
 - o If the array myValues is at address 0x13
 - o myValues[0] is at address 0x13
 - o myValues[1] is at address 0x14
 - o myValues[5] is at address 0x18
- **Red flag: assembler is stupid and does not know how long your array is.**
- **Red flag 2: if myValues is an array of words**
 - o myValues[0] is at address 0x13
 - o myValues[1] is at address 0x15
 - o myValues[5] is at address 0x1D
 - o ...
- **Red flag 3: the C language (3th year, operating systems) is stupid and does not know how long your array is. However, C knows the difference between bytes, integer and long integer, ...**

assembler:

```

segment 'ram0'
myValues ds.b 10 ; allocates 10 bytes for array myValues

LDW X, #5
LD A, (myValues,X) ; loads myValues[5] in the
accumulator

INCW X
LD a, #10
LD (myValues,X), a ; loads the accumulator to myValues[6]

```

- **Notes:**

- NO SPACES in brackets E.G. LD A, (tones,X)!
- A strange bug is that the text editor doesn't allow you to write the left squared bracket on a Belgian period AZERTY keyboard. An easy workaround is to copy paste it from another location/file. Alt 91 also works. (alt 93 =])
- X is a 16 bit register

- More info can be found in the Programming Manual.

4.1.4 Exercises

Exercise 1: set up a new workspace and project

Exercise 2: arrays part 1

- Allocate 8 bytes in ram1 segment. (array1 ds.b 8)
- Write a loop that is executed **only once** that fills this ram locations with all powers of 2 under 256 (1, 2, 4, ..., 128)
 - o The base address is the label that you assigned to your array
 - o X or Y is your loop counter
 - o Use incw and the CC flags to control your loop.
 - o Carefully study the SLL instruction. How does it affect the bits in CC? Which JRxx instruction is the most beneficial for you?

Skeleton:

```

segment 'ram0'
myArray ds.b 8
counter ds.b

segment 'rom'

main

```

```

...
...
ldw x, #0 ;start at index 0
ld a, #1      ; initialize a

... do something with a to get the next
number

...
ld (myarray,x), a ; store the contents of
a at address myaddress + x
               in this case this is myaddress[0]
incw x
...
    
```

Exercise 3 Arrays part 2

Do the same exercise, but now with an array of words instead of bytes.

Exercise 4 Arrays part 3

- Allocate an array in the rom segment that contains all the bit patterns that occur in the Knight Rider (3, 6 12, 24 ...).
 - o Do not forget to put a label!
 - o Use dc.b directive!
- Allocate a variable that holds the number of values in that array.
- Write a program that is executed repeatedly. Every 500 ms it should read a value from you array (index 0 ... n where n-1 is the number of values in your array. The value read must be copied immediately to PD_ODR.
- Do not forget to configure port D as outputs, push-pull.

Exercise 5 Arrays part 3: play a scale

- We will use TIM2 as tone synthesizer. Configure D4 as push-pull output and place the jumper to connect to the speaker.
- This table holds the frequencies of a scale over 1 octave.
- Calculate the ARR value for TIM2 for each frequency.
- Your program can calculate the duty cycle (divide by 2 = shift right)
- Program the 8 values in table in the rom segment. Label this array 'pitch'.

segment 'rom'
 pitch dc.w 880, 988,
- Write a program that plays the scale 1 time. Every note should play for 500 ms. The scale should play A -> A', G -> A.

tone	f	counts	hex
------	---	--------	-----

A	880
B	988
C	1047
D	1175
E	1319
F	1397
G	1568
A'	1760

Exercise 6: play a tune.

- Make an array in the rom segment met following values:
2-3-4-2-4-5-6-6
- Label this array 'song'
- Program a loop that uses these values as indexes to access the array 'pitch' Play that note for 500 ms.
- Play the tune once.
- Extension: after playing it once, wait 5 seconds and play it again.

Exercise 7: PWM

- Configure TIM2 for 100 Hz
- Set the jumper to D4
- Watch LED D4 for different values of the duty cycle : 10%, 20%, 50%, 100% do you notice a difference in brightness?

5 Appendices

Interrupt vector mapping

STM8S105x4/6

7 Interrupt vector mapping

Table 10. Interrupt mapping

IRQ no.	Source block	Description	Wakeup from halt mode	Wakeup from active-halt mode	Vector address
-	RESET	Reset	Yes	Yes	0x00 8000
-	TRAP	Software interrupt	-	-	0x00 8004
0	TLI	External top level interrupt	-	-	0x00 8008
1	AWU	Auto wake up from halt	-	Yes	0x00 800C
2	CLK	Clock controller	-	-	0x00 8010
3	EXTI0	Port A external interrupts	Yes ⁽¹⁾	Yes ⁽¹⁾	0x00 8014
4	EXTI1	Port B external interrupts	Yes	Yes	0x00 8018
5	EXTI2	Port C external interrupts	Yes	Yes	0x00 801C
6	EXTI3	Port D external interrupts	Yes	Yes	0x00 8020
7	EXTI4	Port E external interrupts	Yes	Yes	0x00 8024
8	Reserved	-	-	-	0x00 8028
9	Reserved	-	-	-	0x00 802C
10	SPI	End of transfer	Yes	Yes	0x00 8030
11	TIM1	TIM1 update/overflow/underflow/trigger/break	-	-	0x00 8034
12	TIM1	TIM1 capture/compare	-	-	0x00 8038
13	TIM2	TIM2 update/overflow	-	-	0x00 803C
14	TIM2	TIM2 capture/compare	-	-	0x00 8040
15	TIM3	TIM3 update/overflow	-	-	0x00 8044
16	TIM3	TIM3 capture/compare	-	-	0x00 8048
17	Reserved	-	-	-	0x00 804C
18	Reserved	-	-	-	0x00 8050
19	I2C	I2C interrupt	Yes	Yes	0x00 8054

44/121

DocID14771 Rev 15



Table 1: Interrupt Vector mapping

Interrupt controller (ITC)

RM0016

6.9.3 External interrupt control register 1 (EXTI_CR1)

Address offset: 0x00

Reset value: 0x00

7	6	5	4	3	2	1	0
PDIS[1:0]		PCIS[1:0]		PBIS[1:0]		PAIS[1:0]	
rw		rw		rw		rw	

Bits 7:6 **PDIS[1:0]**: Port D external interrupt sensitivity bits

These bits can only be written when I1 and I0 in the CCR register are both set to 1 (level 3). They define the sensitivity of Port D external interrupts.

- 00: Falling edge and low level
- 01: Rising edge only
- 10: Falling edge only
- 11: Rising and falling edge

Bits 5:4 **PCIS[1:0]**: Port C external interrupt sensitivity bits

These bits can only be written when I1 and I0 in the CCR register are both set to 1 (level 3). They define the sensitivity of Port C external interrupts.

- 00: Falling edge and low level
- 01: Rising edge only
- 10: Falling edge only
- 11: Rising and falling edge

Bits 3:2 **PBIS[1:0]**: Port B external interrupt sensitivity bits

These bits can only be written when I1 and I0 in the CCR register are both set to 1 (level 3). They define the sensitivity of Port B external interrupts.

- 00: Falling edge and low level
- 01: Rising edge only
- 10: Falling edge only
- 11: Rising and falling edge

Bits 1:0 **PAIS[1:0]**: Port A external interrupt sensitivity bits

These bits can only be written when I1 and I0 in the CCR register are both set to 1 (level 3). They define the sensitivity of Port A external interrupts.

- 00: Falling edge and low level
- 01: Rising edge only
- 10: Falling edge only
- 11: Rising and falling edge

70/467

DocID14587 Rev 14



Table 2: External Interrupt control register 1

RM0016

Interrupt controller (ITC)

6.9.4 External interrupt control register 1 (EXTI_CR2)

Address offset: 0x01

Reset value: 0x00

7	6	5	4	3	2	1	0
Reserved					TLIS	PEIS[1:0]	
r					rw	rw	

Bits 7:3 Reserved.

Bit 2 **TLIS**: Top level interrupt sensitivity

This bit is set and cleared by software. This bit can be written only when external interrupt is disabled on the corresponding GPIO port (PD7 or PC3, refer to [Section 6.6: External interrupts on page 66](#)).

0: Falling edge

1: Rising edge

Bits 1:0 **PEIS[1:0]**: Port E external interrupt sensitivity bits

These bits can only be written when I1 and I0 in the CCR register are both set to 1 (level 3). They define the sensitivity of the Port E external interrupts.

00: Falling edge and low level

01: Rising edge only

10: Falling edge only

11: Rising and falling edge



DocID14587 Rev 14

71/467



Table 3: External Interrupt Control Register 2

6 Documentation

6.1 The instruction set

All STM8 devices share the same instruction set. This is documented in **pm0044-stm8-cpu-programming-manual-stmicroelectronics.pdf**. This document is available on Toledo/Ultra and can be downloaded from STM's website. We will need chapter 7: all details on all instructions. This is the reference information that documents all effects of an instruction.

6.2 The reference manual

The next document is **RM0016-stm8s-and-stm8af-reference-manual.pdf** (available on Toledo/Ultra). This document explains how all the building block of the MCU work. Here you can find all details on the configuration of digital I/O, interrupts, timers, clock control, ...

To make your life a bit easier, we highlighted the sections you need in the table of contents.

After this series of lab exercise, you should be able to use this document.

6.3 The datasheet

All parts in the STM8 family share the same instruction set and all peripherals work the same way if present.

Document name: **Datasheet-stm8s105.pdf**. As you might guess, this document is available on Toledo/Ultra + STM website.

This document describes in detail all details that are specific for STM8S105K4. Eg: addresses of registers, memory map, interrupt vector table, ...

Peripherals and available memory, number of timers + specific functionality are documented here.

We will only need chapter 7, the Interrupt vector mapping.

6.4 The IDE

We use STVD. You can download en.stvd-stm8.zip from Toledo/Ultra.

Installation should be straightforward. To solve common problems that might occur: see appendix 1.

Important : always run this software as administrator. The program does not behave like a modern windows program and accesses folders that require elevated privileges in the latest versions of MS-windows. The easiest way to do this is change the properties in the shortcut on your desktop and launch the program with this shortcut.