STM8 + STVD + ASSEMBLER: Quick Start

(/2018/stm8_stvd_asm/)

sections: STM8 (/tags/stm8) , ASSEMBLYER (/tags/assembler) , date: August 22, 2018

I am returning to the proprietary development environment - ST Visual Develop, for two reasons. Firstly, it turned out that it is impossible to write any complex firmware in assembler without a debugger, at least I couldn't, because the program is debugged anyway using an LED or via UART, it is simply done faster through the debug interface. Secondly, it seemed to me that studying the architecture only using the datasheet is not quite right. Something may be misunderstood, something may be missed. With such things as DMA, built-in RTC or code execution from RAM, it will be easier to figure it out using a debugger, not forgetting to look at the datasheet.

STVD is a fairly simple development environment, I mastered it in an evening. In this article I want to tell you how to start writing and debugging firmware in STM8 assembler from scratch using ST Visual Develop.

STVD - works in Windows family OS, starting from XP and higher. At the same time, it works perfectly under a virtual machine in Linux. In this article, I use STVD 4.3.12, the latest available version at the moment, and Windows XP SP3 as a guest OS. As a microcontroller, I will use a 20-pin STM8S103F3P6 (http://4.bp.blogspot.com/-JB5qpbYEHIg/Vm2acOTEC4I/AAAAAAAABWQ/eNTM--XclnY/s1600/stm8s103f3p6.jpg) .

Below is the documentation I relied on when writing this article:

- STM8 Programming Guide - PM0044; (http://www.st.com/content/ccc/resource/technical/document/programming_manual/43/24/13/9a/89/df/45/ed/CD00161709.pdf/files/CD0016170
- STM8 Assembler and Linker User's Guide: ST Assembler-Linker UM0144 (https://www.st.com/content/ccc/resource/technical/document/user_manual/b8/61/d0/68/83/82/4f/2b/CD00056470.pdf/files/CD00056470.pdf/jc
- ST Visual Develop User Guide - UM0036 (https://www.st.com/content/ccc/resource/technical/document/user_manual/e9/d3/c3/4b/2c/0b/46/f2/CD00004556.pdf/files/CD00004556.pdf/jc

As a disassembler I will use the stm8-binutils (https://stm8-binutils-gdb.sourceforge.io) utility suite . The binaries of this suite for Windows are compiled to work in CYGWIN, i.e. they understand the Unix file path format with a forward slash as a separator. CYGWIN for Windows 7 and higher is installed without problems following the instructions on the site https://cygwin.com/install.html (https://cygwin.com/install.html) , for Windows XP you need to follow the instructions in this HowTo: windows xp - cygwin 2.5.2 mirror -- getting the latest XP release - Stack Overflow (https://stackoverflow.com/questions/39479826/cygwin-2-5-2-mirror-getting-the-last-xp-release) .

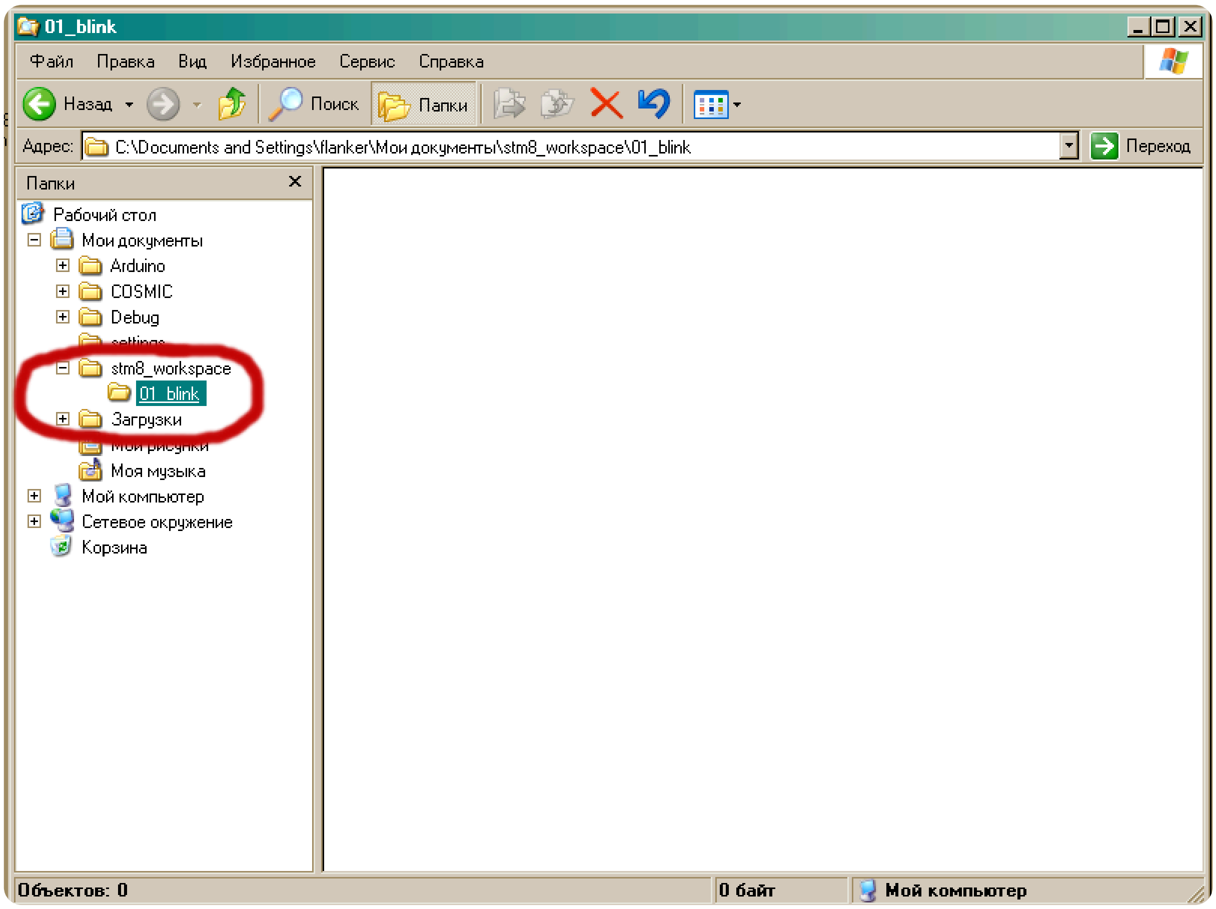
As an alternative to the binutils+cygwin bundle, you can use naken_util from the naken_asm (https://www.mikekohn.net/micro/naken_asm.php) package .

The content of the article:

1. Открытие шаблонного проекта на ассемблере

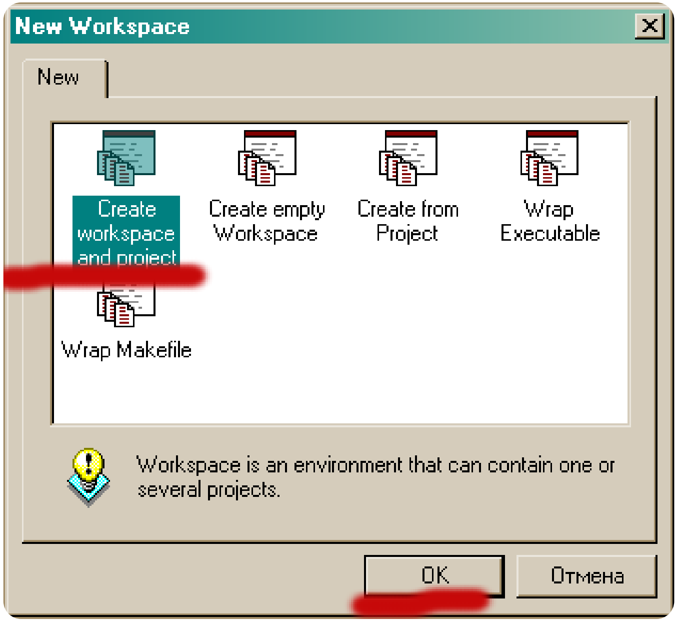
If the STVD assembler itself deserves all praise, then the project management system is made rather stupidly. In STVD there is a concept of Workspace - storing the settings of the environment itself. Workspace can contain several projects.

First of all, you will need to create an empty folder for the workspace and a nested folder for the future project. It will be difficult to do this using standard STVD tools, since there are no dialog forms for selecting a directory:



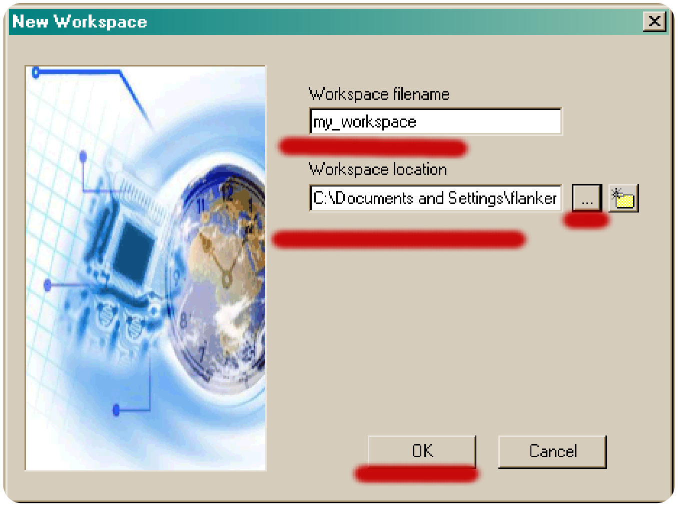
(/img/stm8/stvd/stvd_01.png)

Now launch STVD and select New Workspace from the menu:


New Workspace
New
Create workspace and project
Create empty Workspace
Create from Project
Wrap Executable
Wrap Makefile
Workspace is an environment that can contain one or several projects.
OK
Отмена

(/img/stm8/stvd/stvd_02.png)

In the dialog box, we specify the name of the workspace and indicate the previously created folder for saving:

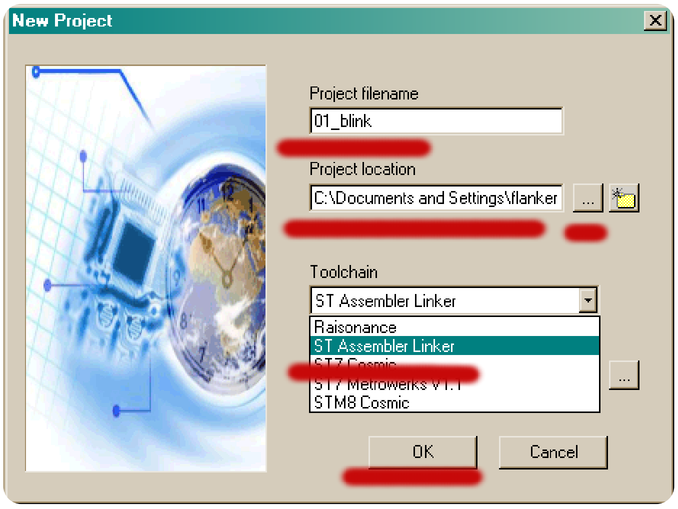

New Workspace

Workspace filename: my_workspace

Workspace location: C:\Documents and Settings\flanker

(/img/stm8/stvd/stvd_03.png)

Similarly, for the project, select a name, specify the previously created project folder, and in the drop-down list, specify ST Assembler Linker as the Toolchain:


New Project

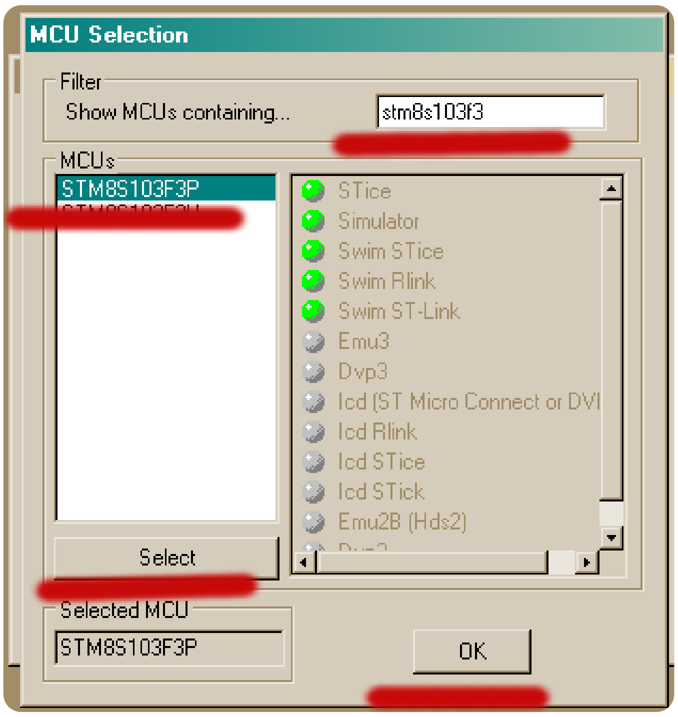
Project filename: 01_blink

Project location: C:\Documents and Settings\flanker

Toolchain: ST Assembler Linker

(/img/stm8/stvd/stvd_04.png)

It remains to specify the target microcontroller:


MCU Selection

Filter

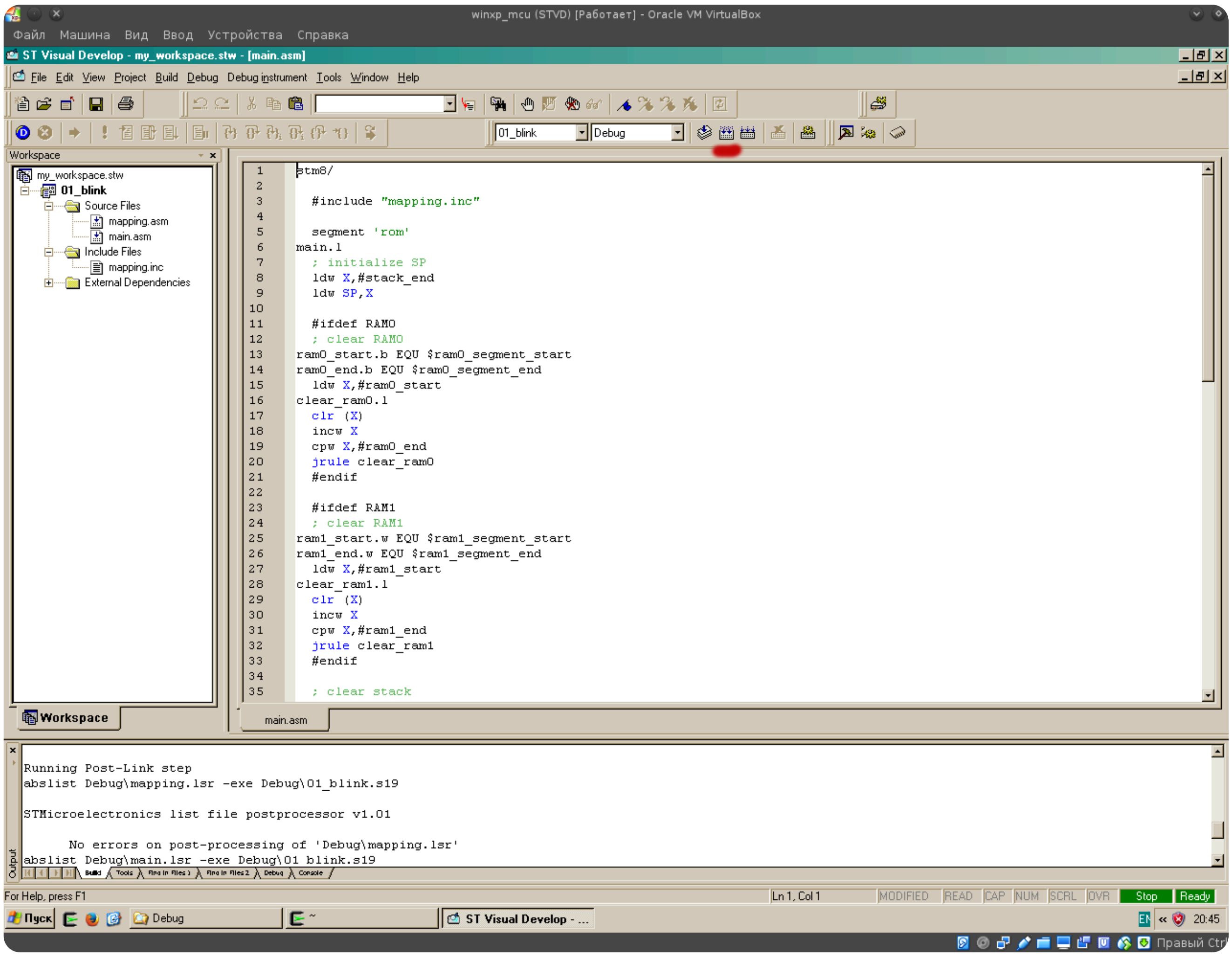
Show MCUs containing... stm8s103f3

MCUs

Selected MCU: STM8S103F3P

(/img/stm8/stvd/stvd_05.png)

A project template opens in front of us, we can compile it right away:

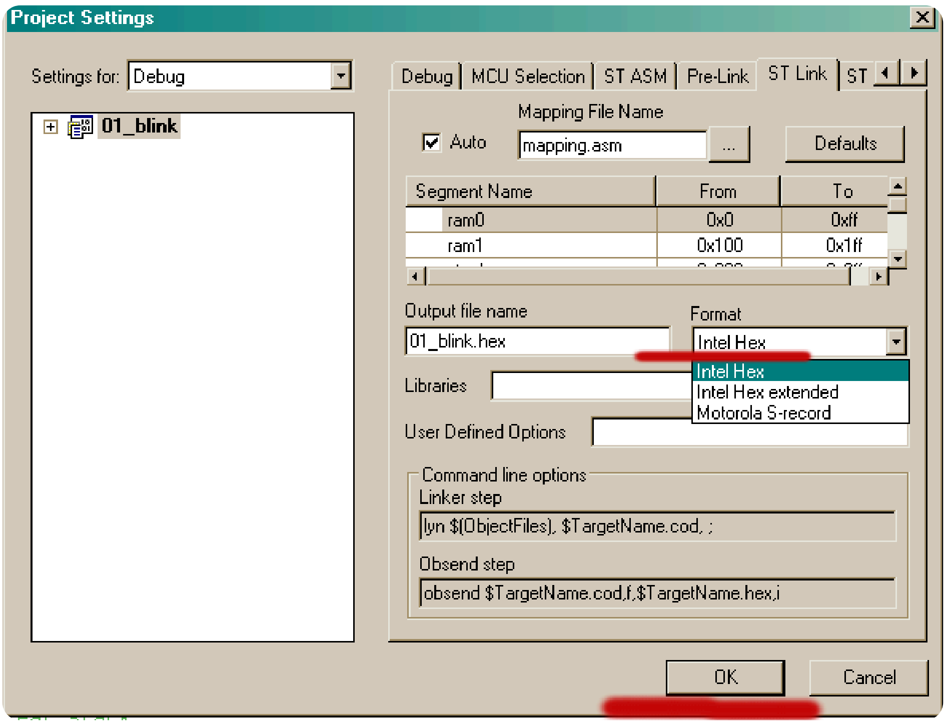

winxp_mcu (STVD) [Работает] - Oracle VM VirtualBox
Файл Машина Вид Ввод Устройства Справка
ST Visual Develop - my_workspace.stw - [main.asm]
File Edit View Project Build Debug Debug instrument Tools Window Help
01_blink
Debug
Workspace
my_workspace.stw
01_blink
Source Files
mapping.asm
main.asm
Include Files
mapping.inc
External Dependencies
main.asm
For Help, press F1
Ln 1, Col 1
MODIFIED READ CAP NUM SCRL OVR
Stop
Ready
Пуск
Debug
ST Visual Develop - ...
20:45
Правый Ctrl

```
stm8/

  #include "mapping.inc"

  segment 'rom'
main.l
  ; initialize SP
  ldw X,#stack_end
  ldw SP,X

  #ifdef RAM0
  ; clear RAM0
ram0_start.b EQU $ram0_segment_start
ram0_end.b EQU $ram0_segment_end
  ldw X,#ram0_start
clear_ram0.l
  clr (X)
  incw X
  cpw X,#ram0_end
  jrule clear_ram0
  #endif

  #ifdef RAM1
  ; clear RAM1
ram1_start.w EQU $ram1_segment_start
ram1_end.w EQU $ram1_segment_end
  ldw X,#ram1_start
clear_ram1.l
  clr (X)
  incw X
  cpw X,#ram1_end
  jrule clear_ram1
  #endif

  ; clear stack
```

```
Running Post-Link step
abslist Debug\mapping.lsr -exe Debug\01_blink.s19

STMicroelectronics list file postprocessor v1.01

      No errors on post-processing of 'Debug\mapping.lsr'
abslist Debug\main.lsr -exe Debug\01_blink.s19
```

(/img/stm8/stvd/stvd_06.png)

It remains to specify the firmware format in the project settings:



Settings for: Debug

01_blink

Mapping File Name

[✓] Auto mapping.asm

Segment Name	From	To
ram0	0x0	0xff
ram1	0x100	0x1ff

Output file name: 01_blink.hex

Format: Intel Hex

Intel Hex
Intel Hex extended
Motorola S-record

Libraries:

User Defined Options:

Command line options

Linker step

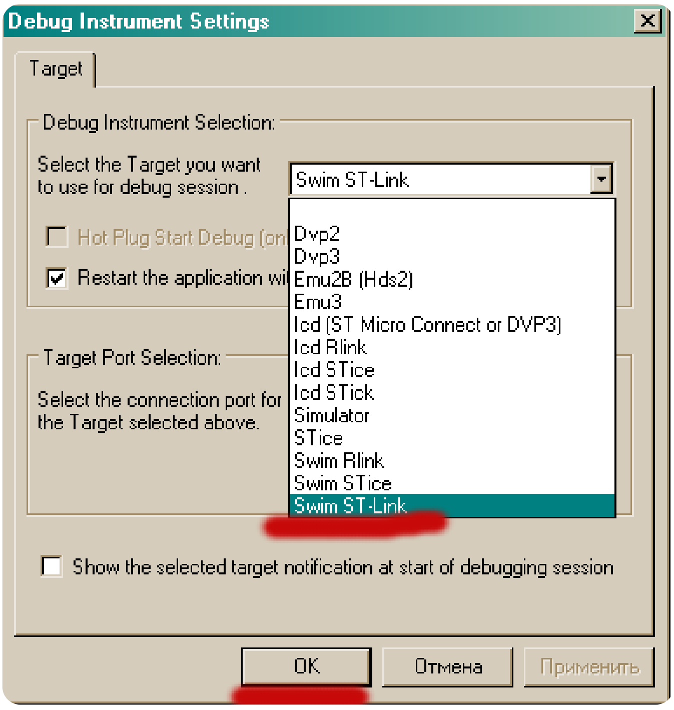
lyn $(ObjectFiles), $TargetName.cod, ;

Obsend step

obsend $TargetName.cod,f,$TargetName.hex,i

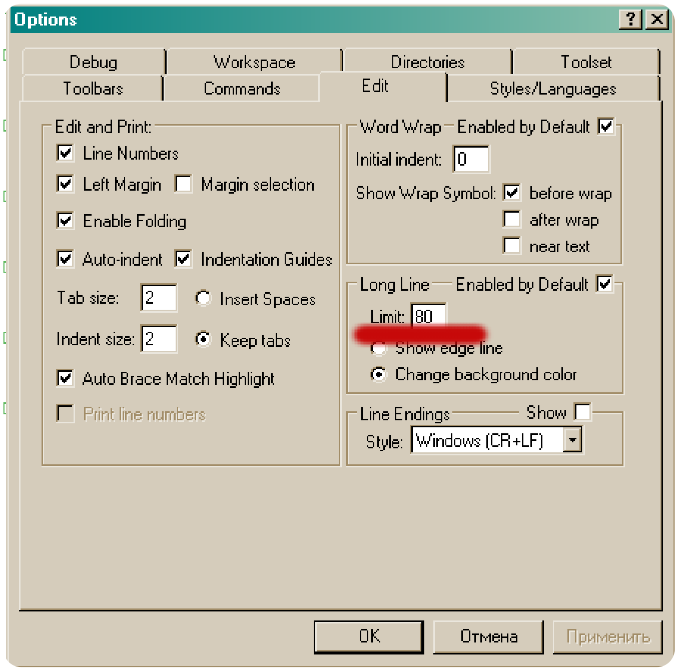
(/img/stm8/stvd/stvd_07.png)

As a debugger, you can choose Simulator or a hardware debugger on the SWIM interface. Algorithms can be debugged on the simulator, but when working with external interfaces, you will no longer be able to do without hardware:


Debug Instrument Settings
Target
Debug Instrument Selection:
Select the Target you want to use for debug session .
Swim ST-Link
Hot Plug Start Debug (on
Restart the application wi
Dvp2
Dvp3
Emu2B (Hds2)
Emu3
Icd (ST Micro Connect or DVP3)
Icd Rlink
Icd STice
Icd STick
Simulator
STice
Swim Rlink
Swim STice
Swim ST-Link
Target Port Selection:
Select the connection port for the Target selected above.
Show the selected target notification at start of debugging session
OK
Отмена
Применить

(/img/stm8/stvd/stvd_08.png)

In the STVD options you can configure the environment for yourself, for example you can set the line length after which the line will be highlighted with a red background:



(/img/stm8/stvd/stvd_11.png)

I would also like to draw your attention to the fact that you can set the indent width there - Tab size. By default, two spaces are used, but I am more accustomed to using four spaces.

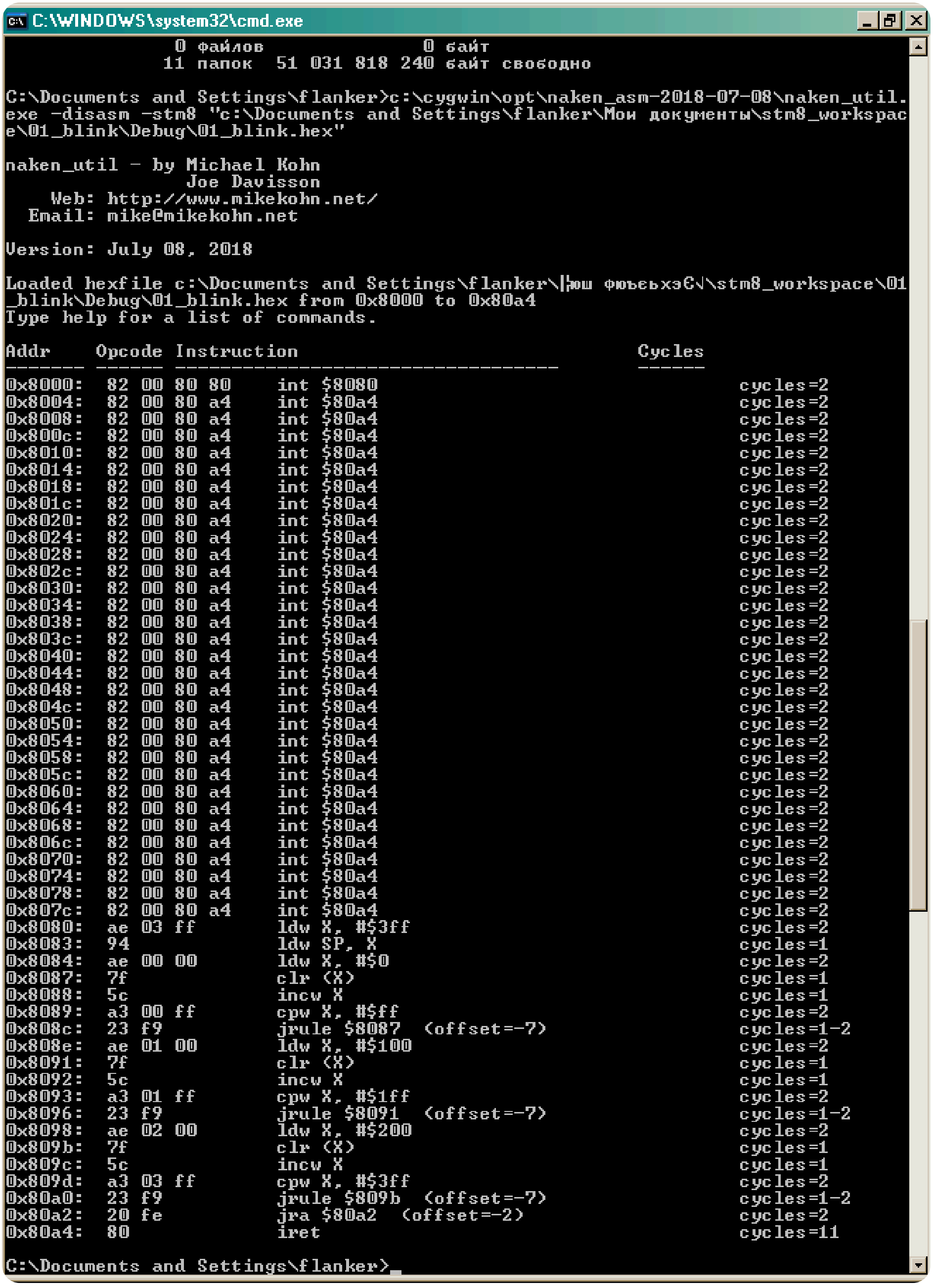
After recompiling and saving the workspace, the project file structure will look like this:

```
$ tree ~/docs/stm8_workspace
/home/flanker/docs/stm8_workspace
├── 01_blink
│   ├── 01_blink.dep
│   ├── 01_blink.stp
│   ├── cbe.err
│   ├── Debug
│   │   ├── 01_blink.cod
│   │   ├── 01_blink.grp
│   │   ├── 01_blink.hex
│   │   ├── 01_blink.map
│   │   ├── 01_blink.s19
│   │   ├── 01_blink.sym
│   │   ├── main.lsr
│   │   ├── main.lst
│   │   ├── main.obj
│   │   ├── mapping.lsr
│   │   ├── mapping.lst
│   │   └── mapping.obj
│   ├── main.asm
│   ├── mapping.asm
│   └── mapping.inc
├── my_workspace.stw
└── my_workspace.wed

2 directories, 20 files
```

Файл прошивки можно дизассемблировать из CYGWIN:

```
$ stm8-objdump.exe -m stm8 -D ~/docs/stm8_workspace/01_blink/Debug/01_blink.hex

/home/flanker/docs/stm8_workspace/01_blink/Debug/01_blink.hex: ihex file format


Disassembling section .sec1:

00008080 <.sec1>:
    8080:       ae 03 ff        ldw X,#0x03ff ;0x3ff
    8083:       94              ldw SP,X
    8084: ae 00 00 ldw X,#0x0000
    8087:       7f              clr (X)
    8088:       5c              incw X
    8089:       a3 00 ff        cpw X,#0x00ff ;0xff
    808c:       23 f9           jrule 0x8087 ;0x8087
    808e:       ae 01 00        ldw X,#0x0100 ;0x100
    8091:       7f              clr (X)
    8092:       5c              incw X
    8093:       a3 01 ff        cpw X,#0x01ff ;0x1ff
    8096:       23 f9           jrule 0x8091 ;0x8091
    8098: ae 02 00 ldw X,#0x0200 ;0x200
    809b:       7f              clr (X)
    809c:       5c              incw X
    809d:       a3 03 ff        cpw X,#0x03ff ;0x3ff
    80a0:       23 f9           jrule 0x809b ;0x809b
    80a2:       20 fe           jra 0x80a2 ;0x80a2
    80a4: 80 would go

Disassembling section .sec2:

00008000 <.sec2>:
    8000:       82 00 80 80     int 0x008080 ;0x8080
    8004:       82 00 80 a4     int 0x0080a4 ;0x80a4
    8008:       82 00 80 a4     int 0x0080a4 ;0x80a4
    800c:       82 00 80 a4     int 0x0080a4 ;0x80a4
    8010:       82 00 80 a4     int 0x0080a4 ;0x80a4
    8014:       82 00 80 a4     int 0x0080a4 ;0x80a4
    8018:       82 00 80 a4     int 0x0080a4 ;0x80a4
    801c:       82 00 80 a4     int 0x0080a4 ;0x80a4
    8020:       82 00 80 a4     int 0x0080a4 ;0x80a4
    8024:       82 00 80 a4     int 0x0080a4 ;0x80a4
    8028:       82 00 80 a4     int 0x0080a4 ;0x80a4
    802c:       82 00 80 a4     int 0x0080a4 ;0x80a4
    8030:       82 00 80 a4     int 0x0080a4 ;0x80a4
    8034:       82 00 80 a4     int 0x0080a4 ;0x80a4
    8038:       82 00 80 a4     int 0x0080a4 ;0x80a4
    803c:       82 00 80 a4     int 0x0080a4 ;0x80a4
    8040:       82 00 80 a4     int 0x0080a4 ;0x80a4
    8044:       82 00 80 a4     int 0x0080a4 ;0x80a4
    8048:       82 00 80 a4     int 0x0080a4 ;0x80a4
    804c:       82 00 80 a4     int 0x0080a4 ;0x80a4
    8050:       82 00 80 a4     int 0x0080a4 ;0x80a4
    8054:       82 00 80 a4     int 0x0080a4 ;0x80a4
    8058:       82 00 80 a4     int 0x0080a4 ;0x80a4
    805c:       82 00 80 a4     int 0x0080a4 ;0x80a4
    8060:       82 00 80 a4     int 0x0080a4 ;0x80a4
    8064:       82 00 80 a4     int 0x0080a4 ;0x80a4
    8068:       82 00 80 a4     int 0x0080a4 ;0x80a4
    806c:       82 00 80 a4     int 0x0080a4 ;0x80a4
    8070:       82 00 80 a4     int 0x0080a4 ;0x80a4
    8074:       82 00 80 a4     int 0x0080a4 ;0x80a4
    8078:       82 00 80 a4     int 0x0080a4 ;0x80a4
    807c:       82 00 80 a4     int 0x0080a4 ;0x80a4
```

Or the firmware can be disassembled from the Windows command line using the naken_utils disassembler:



```
               0 файлов              0 байт
              11 папок  51 031 818 240 байт свободно

C:\Documents and Settings\flanker>c:\cygwin\opt\naken_asm-2018-07-08\naken_util.
exe -disasm -stm8 "c:\Documents and Settings\flanker\Мои документы\stm8_workspac
e\01_blink\Debug\01_blink.hex"

naken_util - by Michael Kohn
                Joe Davisson
    Web: http://www.mikekohn.net/
  Email: mike@mikekohn.net

Version: July 08, 2018

Loaded hexfile c:\Documents and Settings\flanker\|мои фюьеьхэЄ√\stm8_workspace\01
_blink\Debug\01_blink.hex from 0x8000 to 0x80a4
Type help for a list of commands.

Addr    Opcode Instruction                              Cycles
------- ------ ----------------------------------       ------
0x8000:  82 00 80 80    int $8080                                cycles=2
0x8004:  82 00 80 a4    int $80a4                                cycles=2
0x8008:  82 00 80 a4    int $80a4                                cycles=2
0x800c:  82 00 80 a4    int $80a4                                cycles=2
0x8010:  82 00 80 a4    int $80a4                                cycles=2
0x8014:  82 00 80 a4    int $80a4                                cycles=2
0x8018:  82 00 80 a4    int $80a4                                cycles=2
0x801c:  82 00 80 a4    int $80a4                                cycles=2
0x8020:  82 00 80 a4    int $80a4                                cycles=2
0x8024:  82 00 80 a4    int $80a4                                cycles=2
0x8028:  82 00 80 a4    int $80a4                                cycles=2
0x802c:  82 00 80 a4    int $80a4                                cycles=2
0x8030:  82 00 80 a4    int $80a4                                cycles=2
0x8034:  82 00 80 a4    int $80a4                                cycles=2
0x8038:  82 00 80 a4    int $80a4                                cycles=2
0x803c:  82 00 80 a4    int $80a4                                cycles=2
0x8040:  82 00 80 a4    int $80a4                                cycles=2
0x8044:  82 00 80 a4    int $80a4                                cycles=2
0x8048:  82 00 80 a4    int $80a4                                cycles=2
0x804c:  82 00 80 a4    int $80a4                                cycles=2
0x8050:  82 00 80 a4    int $80a4                                cycles=2
0x8054:  82 00 80 a4    int $80a4                                cycles=2
0x8058:  82 00 80 a4    int $80a4                                cycles=2
0x805c:  82 00 80 a4    int $80a4                                cycles=2
0x8060:  82 00 80 a4    int $80a4                                cycles=2
0x8064:  82 00 80 a4    int $80a4                                cycles=2
0x8068:  82 00 80 a4    int $80a4                                cycles=2
0x806c:  82 00 80 a4    int $80a4                                cycles=2
0x8070:  82 00 80 a4    int $80a4                                cycles=2
0x8074:  82 00 80 a4    int $80a4                                cycles=2
0x8078:  82 00 80 a4    int $80a4                                cycles=2
0x807c:  82 00 80 a4    int $80a4                                cycles=2
0x8080:  ae 03 ff       ldw X, #$3ff                             cycles=2
0x8083:  94             ldw SP, X                                cycles=1
0x8084:  ae 00 00       ldw X, #$0                               cycles=2
0x8087:  7f             clr (X)                                  cycles=1
0x8088:  5c             incw X                                   cycles=1
0x8089:  a3 00 ff       cpw X, #$ff                              cycles=2
0x808c:  23 f9          jrule $8087  (offset=-7)                 cycles=1-2
0x808e:  ae 01 00       ldw X, #$100                             cycles=2
0x8091:  7f             clr (X)                                  cycles=1
0x8092:  5c             incw X                                   cycles=1
0x8093:  a3 01 ff       cpw X, #$1ff                             cycles=2
0x8096:  23 f9          jrule $8091  (offset=-7)                 cycles=1-2
0x8098:  ae 02 00       ldw X, #$200                             cycles=2
0x809b:  7f             clr (X)                                  cycles=1
0x809c:  5c             incw X                                   cycles=1
0x809d:  a3 03 ff       cpw X, #$3ff                             cycles=2
0x80a0:  23 f9          jrule $809b  (offset=-7)                 cycles=1-2
0x80a2:  20 fe          jra $80a2  (offset=-2)                   cycles=2
0x80a4:  80             iret                                     cycles=11

C:\Documents and Settings\flanker>_
```

(/img/stm8/stvd/stvd_09.png)

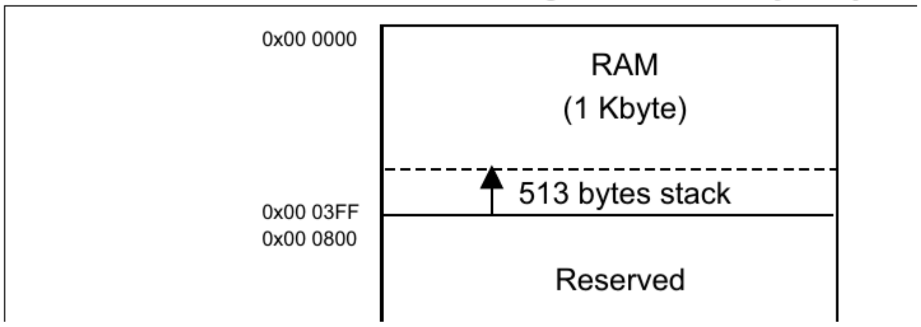
The project template consists of the following files: main.asm, mapping.inc and mapping.asm. The mapping.inc file contains constants for dividing RAM into segments:

```
;------------------------------------------------------
; SEGMENT MAPPING FILE AUTOMATICALLY GENERATED BY STVD
; SHOULD NOT BE MANUALLY MODIFIED.
; CHANGES WILL BE LOST WHEN FILE IS REGENERATED.
;------------------------------------------------------
#define RAM0 1
#define ram0_segment_start 0
#define ram0_segment_end FF
#define RAM1 1
#define ram1_segment_start 100
#define ram1_segment_end 1FF
#define stack_segment_start 200
#define stack_segment_end 3FF
```

Here the first two pages of RAM are allocated to the RAM0 and RAM1 segments, the remaining 513 bytes are given to the stack segment. That is, everything is according to the datasheet:

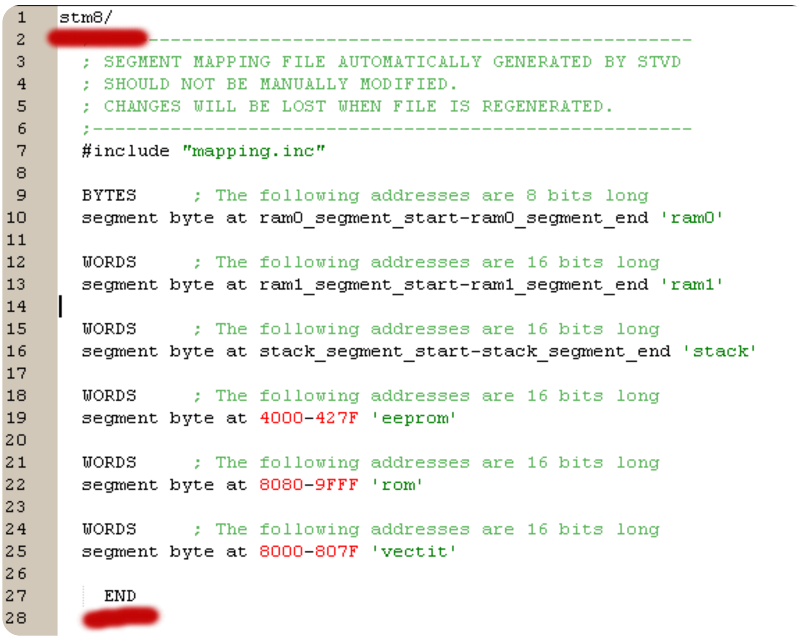
Memory map

Figure 7. Memory map



(/img/stm8/stvd/stvd_10.png)

The mapping.asm file defines the segments:



```
stm8/
	;------------------------------------------------------
	; SEGMENT MAPPING FILE AUTOMATICALLY GENERATED BY STVD
	; SHOULD NOT BE MANUALLY MODIFIED.
	; CHANGES WILL BE LOST WHEN FILE IS REGENERATED.
	;------------------------------------------------------
	#include "mapping.inc"

	BYTES      ; The following addresses are 8 bits long
	segment byte at ram0_segment_start-ram0_segment_end 'ram0'

	WORDS      ; The following addresses are 16 bits long
	segment byte at ram1_segment_start-ram1_segment_end 'ram1'

	WORDS      ; The following addresses are 16 bits long
	segment byte at stack_segment_start-stack_segment_end 'stack'

	WORDS      ; The following addresses are 16 bits long
	segment byte at 4000-427F 'eeprom'

	WORDS      ; The following addresses are 16 bits long
	segment byte at 8080-9FFF 'rom'

	WORDS      ; The following addresses are 16 bits long
	segment byte at 8000-807F 'vectit'

		END
```

(/img/stm8/stvd/stvd_12.png)

I would like to draw attention to the fact that **the assembler file in STVD begins with the line 'stm8/' and ends with the line 'end'** . Instead of 'stm8' you can specify 'st7', but this is for those who write for the stm7 architecture.

Now let's look at the main.asm file

```
stm8 /

    #include "mapping.inc"

    segment 'rom'
main.l
    ; initialize SP
    ldw X,#stack_end
    ldw SP,X

    #ifdef RAM0
    ; clear RAM0
ram0_start.b EQU $ram0_segment_start
ram0_end.b EQU $ram0_segment_end
    ldw X,#ram0_start
clear_ram0.l
    clr (X)
    incw X
    cpw X,#ram0_end
    jrule clear_ram0
    #endif

    #ifdef RAM1
    ; clear RAM1
ram1_start.w EQU $ram1_segment_start
ram1_end.w EQU $ram1_segment_end
    ldw X,#ram1_start
clear_ram1.l
    clr (X)
    incw X
    cpw X,#ram1_end
    jrule clear_ram1
    #endif

    ; clear stack
stack_start.w EQU $stack_segment_start
stack_end.w EQU $stack_segment_end
    ldw X,#stack_start
clear_stack.l
    clr (X)
    incw X
    cpw X,#stack_end
    jrule clear_stack

infinite_loop.l
    jra infinite_loop

    interrupt NonHandledInterrupt
NonHandledInterrupt.l
    iret

    segment 'vectit'
    dc.l {$82000000+main}                                   ; reset
    dc.l {$82000000+NonHandledInterrupt}    ; trap
    dc.l {$82000000+NonHandledInterrupt}    ; irq0
    dc.l {$82000000+NonHandledInterrupt}    ; irq1
    dc.l {$82000000+NonHandledInterrupt}    ; irq2
    dc.l {$82000000+NonHandledInterrupt}    ; irq3
    dc.l {$82000000+NonHandledInterrupt}    ; irq4
    dc.l {$82000000+NonHandledInterrupt}    ; irq5
    dc.l {$82000000+NonHandledInterrupt}    ; irq6
    dc.l {$82000000+NonHandledInterrupt}    ; irq7
    dc.l {$82000000+NonHandledInterrupt}    ; irq8
    dc.l {$82000000+NonHandledInterrupt}    ; irq9
    dc.l {$82000000+NonHandledInterrupt}    ; irq10
    dc.l {$82000000+NonHandledInterrupt}    ; irq11
    dc.l {$82000000+NonHandledInterrupt}    ; irq12
    dc.l {$82000000+NonHandledInterrupt}    ; irq13
    dc.l {$82000000+NonHandledInterrupt}    ; irq14
    dc.l {$82000000+NonHandledInterrupt}    ; irq15
    dc.l {$82000000+NonHandledInterrupt}    ; irq16
    dc.l {$82000000+NonHandledInterrupt}    ; irq17
    dc.l {$82000000+NonHandledInterrupt}    ; irq18
    dc.l {$82000000+NonHandledInterrupt}    ; irq19
    dc.l {$82000000+NonHandledInterrupt}    ; irq20
    dc.l {$82000000+NonHandledInterrupt}    ; irq21
    dc.l {$82000000+NonHandledInterrupt}    ; irq22
    dc.l {$82000000+NonHandledInterrupt}    ; irq23
    dc.l {$82000000+NonHandledInterrupt}    ; irq24
    dc.l {$82000000+NonHandledInterrupt}    ; irq25
    dc.l {$82000000+NonHandledInterrupt}    ; irq26
    dc.l {$82000000+NonHandledInterrupt}    ; irq27
    dc.l {$82000000+NonHandledInterrupt}    ; irq28
    dc.l {$82000000+NonHandledInterrupt}    ; irq29

    end
```

There is an interrupt table and a Reset interrupt handler that ends the main loop. In the Reset handler, the stack pointer is set and the RAM is cleared.

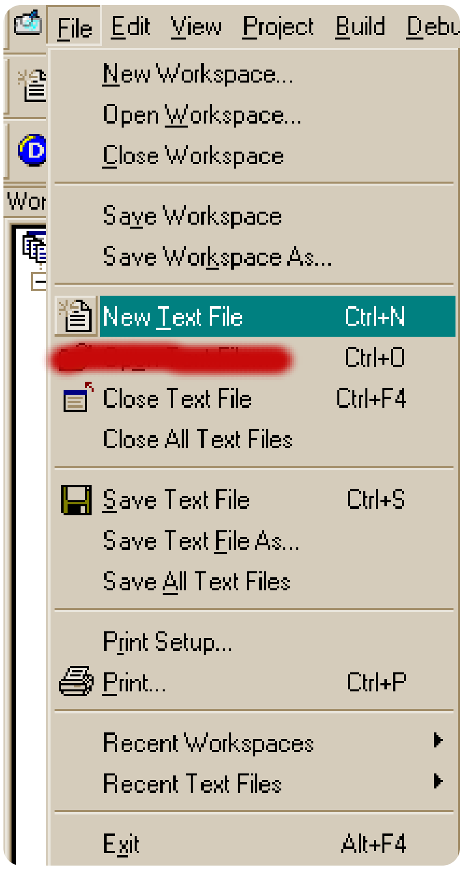
The first thing that catches your eye in the above listing is the original assignment of the vector table in the form of the INT instruction opcode and the label address. If you try to replace this construction with, for example, the following line: "INT main", you will get an error: "Unknown opcode". It is difficult to say why this is done, but the fact is a fact. There is no INT instruction in the STVD assembler.

2. Adding a file with a vector table and interrupt handlers to the project

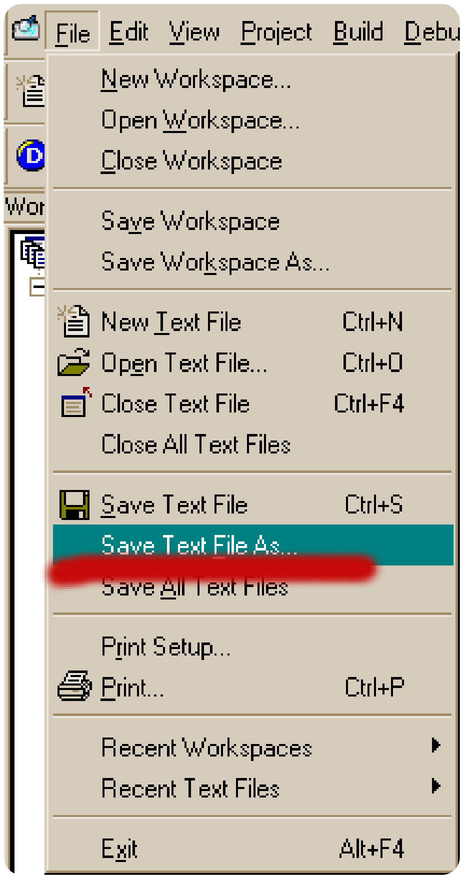
Now let's try to transfer the vector table with interrupt handlers to a separate file irq.asm, leaving only the main loop in main.asm.

1) To do this, you first need to create a new file:

2) After creation, the file will need to be saved to the current project directory:

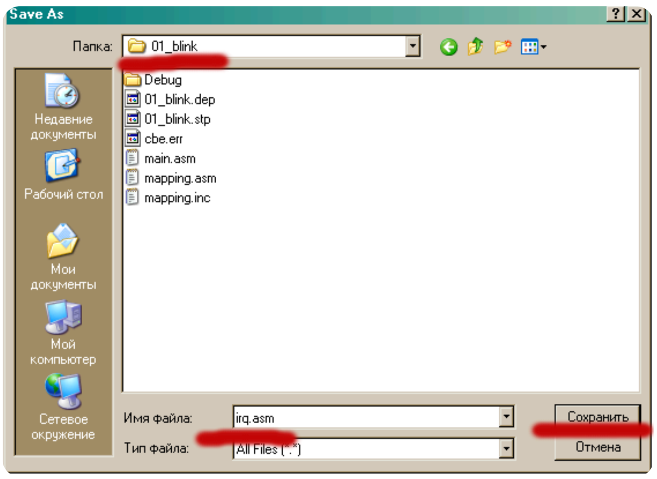


(/img/stm8/stvd/stvd_13.png)



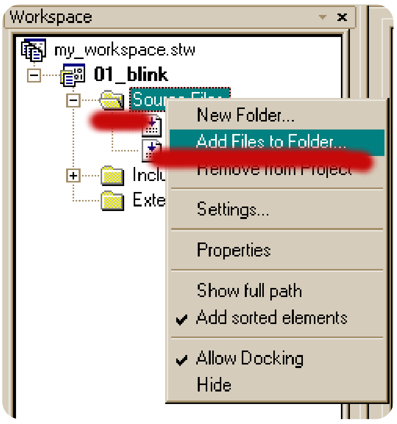
(/img/stm8/stvd/stvd_14.png)

3) In the dialog box, select the directory, file name, and save it:



(/img/stm8/stvd/stvd_15.png)

4) After that, you will need to add the file to the project. To do this, right-click on Source Files and select adding a file, after which the new file will appear in the project file structure:



(/img/stm8/stvd/stvd_16.png)

We'll drop the interrupt table and all their handlers into the irq.asm file:

```
stm8/
    extern main
    #include "mapping.inc"

    segment 'rom'
reset.l
    ; initialize SP
    ldw X,#$03ff
    ldw SP,X
    jp main
    jra reset

    interrupt NonHandledInterrupt
NonHandledInterrupt.l
    iret

    segment 'vectit'
    dc.l {$82000000+reset}                  ; reset
    dc.l {$82000000+NonHandledInterrupt}    ; trap
    dc.l {$82000000+NonHandledInterrupt}    ; irq0
    dc.l {$82000000+NonHandledInterrupt}    ; irq1
    dc.l {$82000000+NonHandledInterrupt}    ; irq2
    dc.l {$82000000+NonHandledInterrupt}    ; irq3
    dc.l {$82000000+NonHandledInterrupt}    ; irq4
    dc.l {$82000000+NonHandledInterrupt}    ; irq5
    dc.l {$82000000+NonHandledInterrupt}    ; irq6
    dc.l {$82000000+NonHandledInterrupt}    ; irq7
    dc.l {$82000000+NonHandledInterrupt}    ; irq8
    dc.l {$82000000+NonHandledInterrupt}    ; irq9
    dc.l {$82000000+NonHandledInterrupt}    ; irq10
    dc.l {$82000000+NonHandledInterrupt}    ; irq11
    dc.l {$82000000+NonHandledInterrupt}    ; irq12
    dc.l {$82000000+NonHandledInterrupt}    ; irq13
    dc.l {$82000000+NonHandledInterrupt}    ; irq14
    dc.l {$82000000+NonHandledInterrupt}    ; irq15
    dc.l {$82000000+NonHandledInterrupt}    ; irq16
    dc.l {$82000000+NonHandledInterrupt}    ; irq17
    dc.l {$82000000+NonHandledInterrupt}    ; irq18
    dc.l {$82000000+NonHandledInterrupt}    ; irq19
    dc.l {$82000000+NonHandledInterrupt}    ; irq20
    dc.l {$82000000+NonHandledInterrupt}    ; irq21
    dc.l {$82000000+NonHandledInterrupt}    ; irq22
    dc.l {$82000000+NonHandledInterrupt}    ; irq23
    dc.l {$82000000+NonHandledInterrupt}    ; irq24
    dc.l {$82000000+NonHandledInterrupt}    ; irq25
    dc.l {$82000000+NonHandledInterrupt}    ; irq26
    dc.l {$82000000+NonHandledInterrupt}    ; irq27
    dc.l {$82000000+NonHandledInterrupt}    ; irq28
    dc.l {$82000000+NonHandledInterrupt}    ; irq29

        END
```

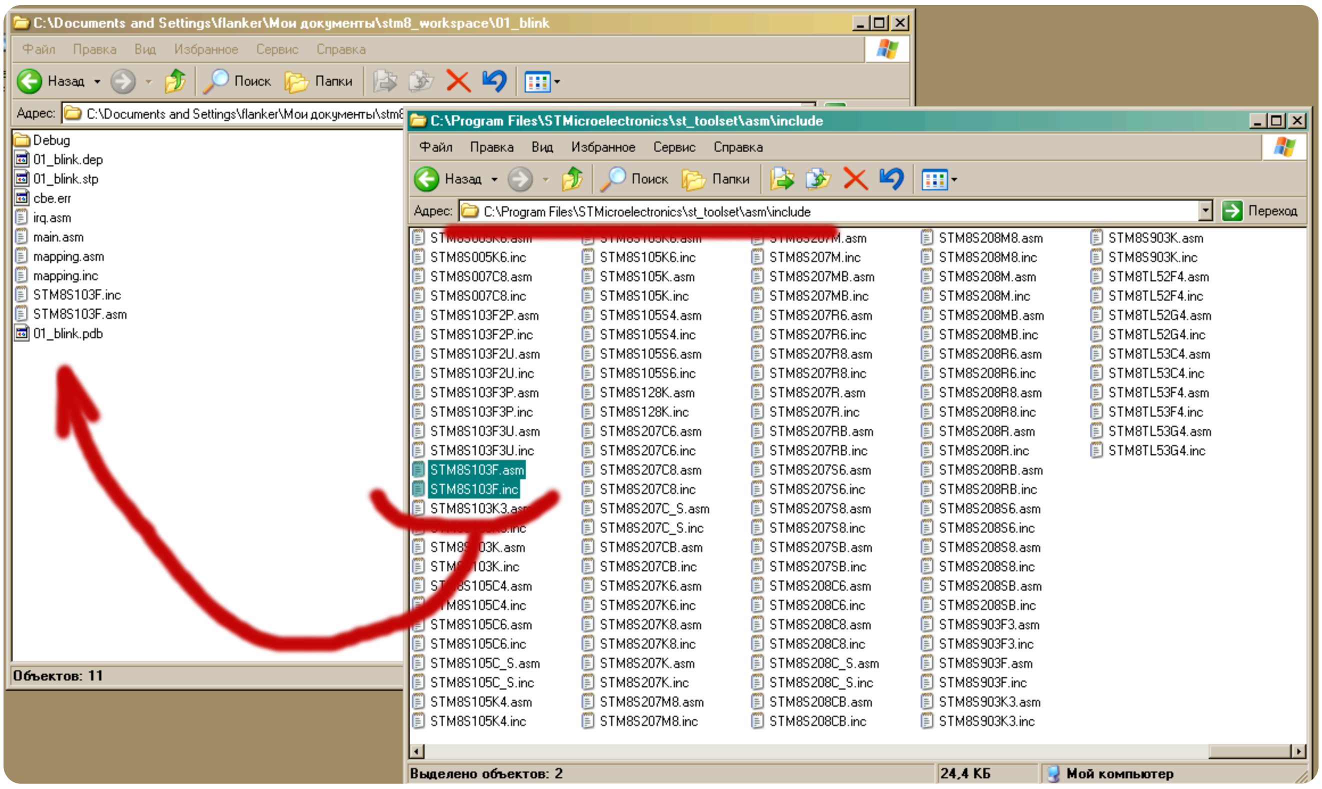
As a result, only the main loop will remain in main.asm:

```
stm8 /

    segment 'rom'
.main
    jp main

    end
```

When compiling, an error may be generated about an incorrect EOF, but this can be corrected in the editor itself, for this you need to press enter after end.

3. Adding constants with peripheral addresses to the project

Now we need to add the addresses of the peripheral I/O registers. To do this, find the STM8S103F.asm and STM8S103F.inc files in the STVD installation directory and copy them to the project folder:


C:\Documents and Settings\flanker\Мои документы\stm8_workspace\01_blink

Адрес: C:\Documents and Settings\flanker\Мои документы\stm8

Debug
01_blink.dep
01_blink.stp
cbe.err
irq.asm
main.asm
mapping.asm
mapping.inc
STM8S103F.inc
STM8S103F.asm
01_blink.pdb

Объектов: 11

C:\Program Files\STMicroelectronics\st_toolset\asm\include

Адрес: C:\Program Files\STMicroelectronics\st_toolset\asm\include

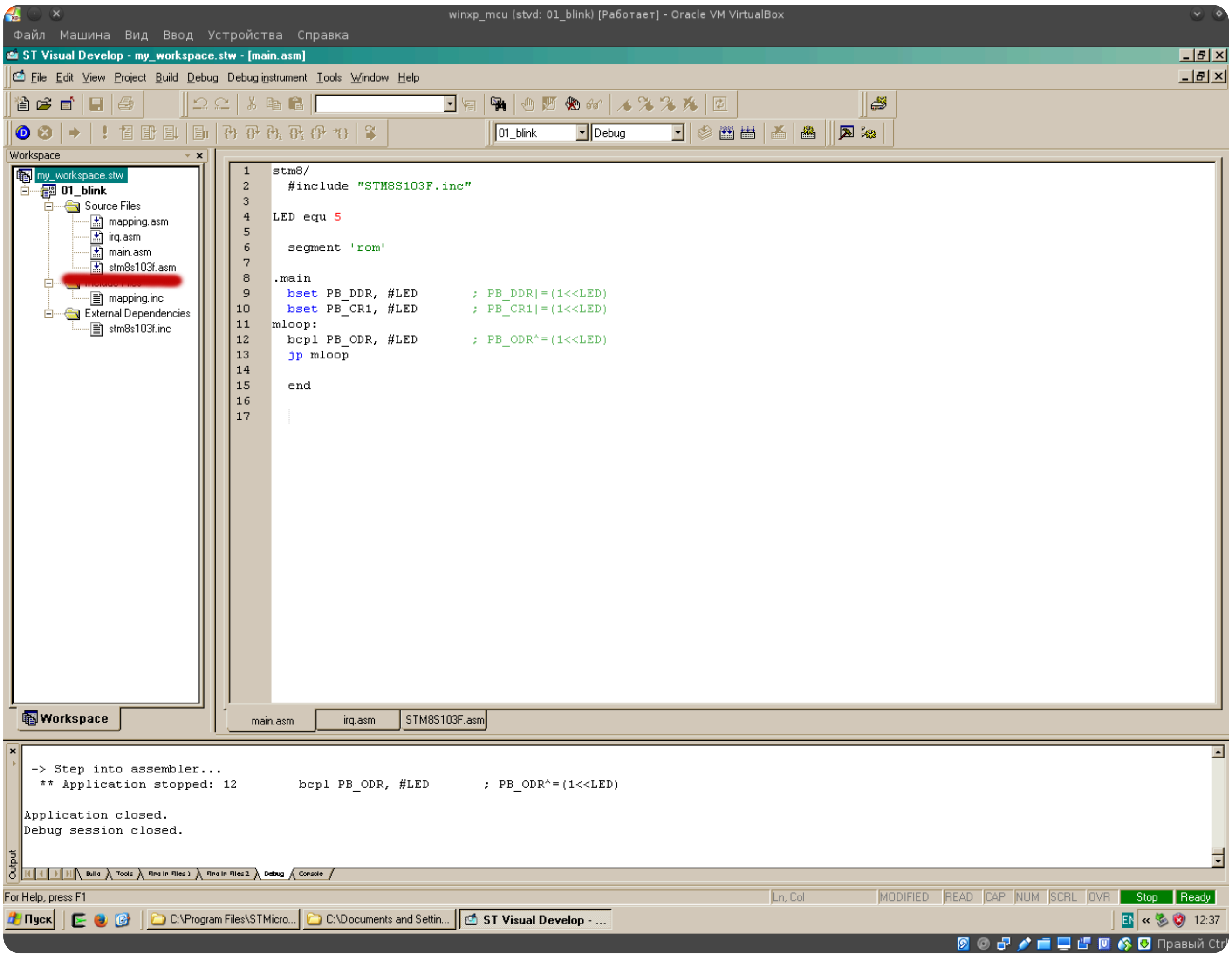
STM8S005K6.asm
STM8S005K6.inc
STM8S007C8.asm
STM8S007C8.inc
STM8S103F2P.asm
STM8S103F2P.inc
STM8S103F2U.asm
STM8S103F2U.inc
STM8S103F3P.asm
STM8S103F3P.inc
STM8S103F3U.asm
STM8S103F3U.inc
STM8S103F.asm
STM8S103F.inc
STM8S103K3.asm
STM8S103K3.inc
STM8S103K.asm
STM8S103K.inc
STM8S105C4.asm
STM8S105C4.inc
STM8S105C6.asm
STM8S105C6.inc
STM8S105C_S.asm
STM8S105C_S.inc
STM8S105K4.asm
STM8S105K4.inc
STM8S105K6.asm
STM8S105K6.inc
STM8S105K.asm
STM8S105K.inc
STM8S105S4.asm
STM8S105S4.inc
STM8S105S6.asm
STM8S105S6.inc
STM8S128K.asm
STM8S128K.inc
STM8S207C6.asm
STM8S207C6.inc
STM8S207C8.asm
STM8S207C8.inc
STM8S207C_S.asm
STM8S207C_S.inc
STM8S207CB.asm
STM8S207CB.inc
STM8S207K6.asm
STM8S207K6.inc
STM8S207K8.asm
STM8S207K8.inc
STM8S207K.asm
STM8S207K.inc
STM8S207M8.asm
STM8S207M8.inc
STM8S207M.asm
STM8S207M.inc
STM8S207MB.asm
STM8S207MB.inc
STM8S207R6.asm
STM8S207R6.inc
STM8S207R8.asm
STM8S207R8.inc
STM8S207R.asm
STM8S207R.inc
STM8S207RB.asm
STM8S207RB.inc
STM8S207S6.asm
STM8S207S6.inc
STM8S207S8.asm
STM8S207S8.inc
STM8S207SB.asm
STM8S207SB.inc
STM8S208C6.asm
STM8S208C6.inc
STM8S208C8.asm
STM8S208C8.inc
STM8S208C_S.asm
STM8S208C_S.inc
STM8S208CB.asm
STM8S208CB.inc
STM8S208M8.asm
STM8S208M8.inc
STM8S208M.asm
STM8S208M.inc
STM8S208MB.asm
STM8S208MB.inc
STM8S208R6.asm
STM8S208R6.inc
STM8S208R8.asm
STM8S208R8.inc
STM8S208R.asm
STM8S208R.inc
STM8S208RB.asm
STM8S208RB.inc
STM8S208S6.asm
STM8S208S6.inc
STM8S208S8.asm
STM8S208S8.inc
STM8S208SB.asm
STM8S208SB.inc
STM8S903F3.asm
STM8S903F3.inc
STM8S903F.asm
STM8S903F.inc
STM8S903K3.asm
STM8S903K3.inc
STM8S903K.asm
STM8S903K.inc
STM8TL52F4.asm
STM8TL52F4.inc
STM8TL52G4.asm
STM8TL52G4.inc
STM8TL53C4.asm
STM8TL53C4.inc
STM8TL53F4.asm
STM8TL53F4.inc
STM8TL53G4.asm
STM8TL53G4.inc

Выделено объектов: 2 | 24,4 КБ | Мой компьютер

(/img/stm8/stvd/stvd_19.png)

After that, you need to add the STM8S103F.asm file to the project, and change the contents of main.asm to look like this:

```
stm8/
    #include "STM8S103F.inc"

LED equ 5

    segment 'rom'

.main
    bset PB_DDR, #LED       ; PB_DDR|=(1<<LED)
    bset PB_CR1, #LED       ; PB_CR1|=(1<<LED)
mloop:
    bcpl PB_ODR, #LED       ; PB_ODR^=(1<<LED)
    jp mloop

    end
```

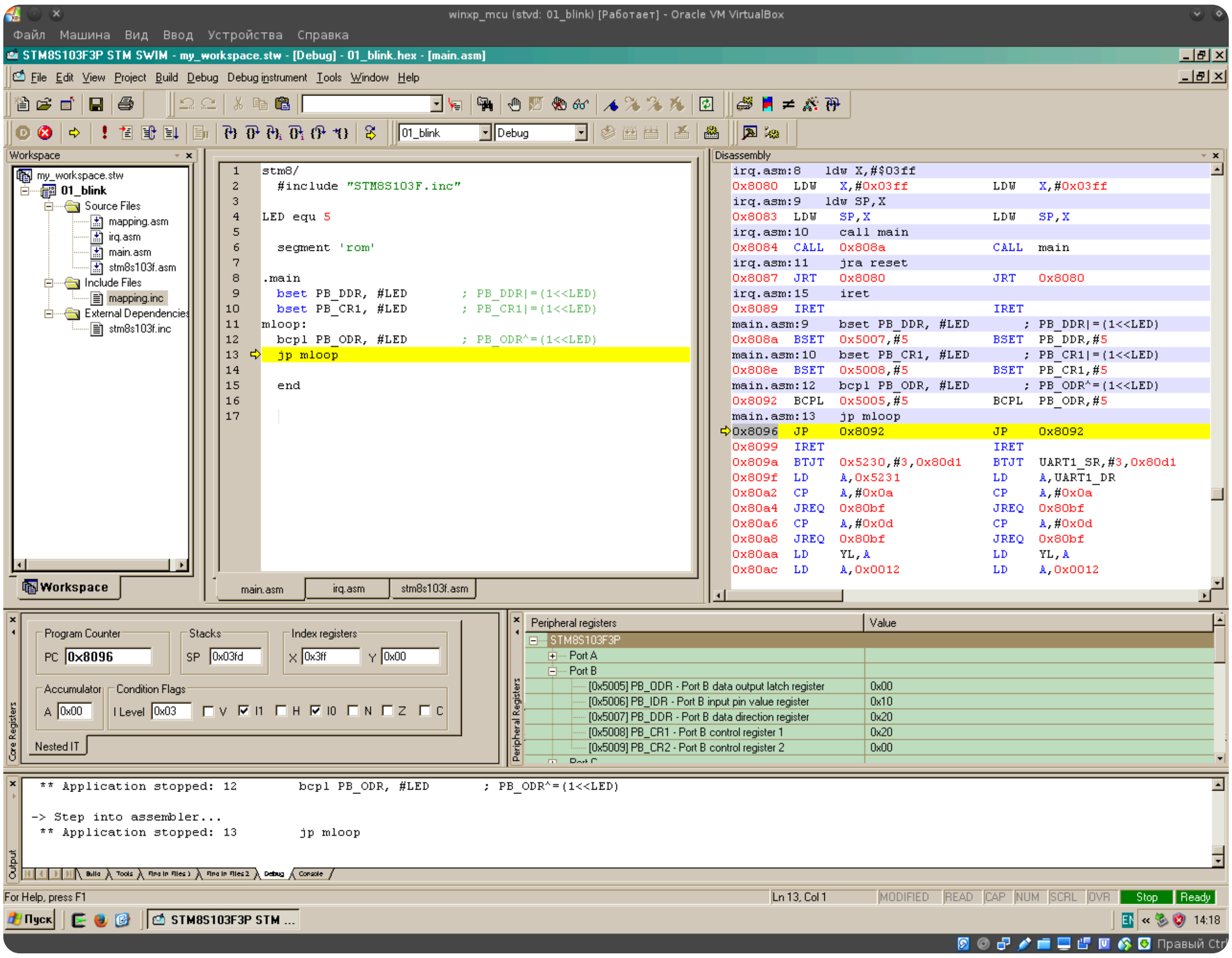


```
stm8/
  #include "STM8S103F.inc"

LED equ 5

  segment 'rom'

.main
  bset PB_DDR, #LED        ; PB_DDR|=(1<<LED)
  bset PB_CR1, #LED        ; PB_CR1|=(1<<LED)
mloop:
  bcpl PB_ODR, #LED        ; PB_ODR^=(1<<LED)
  jp mloop

  end
```

```
 -> Step into assembler...
  ** Application stopped: 12        bcpl PB_ODR, #LED        ; PB_ODR^=(1<<LED)

Application closed.
Debug session closed.
```

(/img/stm8/stvd/stvd_20.png)

After compilation, such a program can be loaded into the microcontroller and run in the debugger:


STM8S103F3P STM SWIM - my_workspace.stw - [Debug] - 01_blink.hex - [main.asm]

```
stm8/
  #include "STM8S103F.inc"

LED equ 5

  segment 'rom'

.main
  bset PB_DDR, #LED        ; PB_DDR|=(1<<LED)
  bset PB_CR1, #LED        ; PB_CR1|=(1<<LED)
mloop:
  bcpl PB_ODR, #LED        ; PB_ODR^=(1<<LED)
  jp mloop

  end
```

Disassembly

```
irq.asm:8   ldw X,#$03ff
0x8080  LDW   X,#0x03ff              LDW   X,#0x03ff
irq.asm:9   ldw SP,X
0x8083  LDW   SP,X                   LDW   SP,X
irq.asm:10    call main
0x8084  CALL  0x808a                 CALL  main
irq.asm:11    jra reset
0x8087  JRT   0x8080                 JRT   0x8080
irq.asm:15    iret
0x8089  IRET                         IRET
main.asm:9    bset PB_DDR, #LED          ; PB_DDR|=(1<<LED)
0x808a  BSET  0x5007,#5              BSET  PB_DDR,#5
main.asm:10   bset PB_CR1, #LED          ; PB_CR1|=(1<<LED)
0x808e  BSET  0x5008,#5              BSET  PB_CR1,#5
main.asm:12   bcpl PB_ODR, #LED          ; PB_ODR^=(1<<LED)
0x8092  BCPL  0x5005,#5              BCPL  PB_ODR,#5
main.asm:13   jp mloop
0x8096  JP    0x8092                 JP    0x8092
0x8099  IRET                         IRET
0x809a  BTJT  0x5230,#3,0x80d1       BTJT  UART1_SR,#3,0x80d1
0x809f  LD    A,0x5231               LD    A,UART1_DR
0x80a2  CP    A,#0x0a                CP    A,#0x0a
0x80a4  JREQ  0x80bf                 JREQ  0x80bf
0x80a6  CP    A,#0x0d                CP    A,#0x0d
0x80a8  JREQ  0x80bf                 JREQ  0x80bf
0x80aa  LD    YL,A                   LD    YL,A
0x80ac  LD    A,0x0012               LD    A,0x0012
```

Program Counter: PC 0x8096 · Stacks: SP 0x03fd · Index registers: X 0x3ff, Y 0x00 · Accumulator: A 0x00 · Condition Flags: I Level 0x03, ☐ V, ☑ I1, ☐ H, ☑ I0, ☐ N, ☐ Z, ☐ C

Peripheral registers	Value
STM8S103F3P	
Port A	
Port B	
[0x5005] PB_ODR - Port B data output latch register	0x00
[0x5006] PB_IDR - Port B input pin value register	0x10
[0x5007] PB_DDR - Port B data direction register	0x20
[0x5008] PB_CR1 - Port B control register 1	0x20
[0x5009] PB_CR2 - Port B control register 2	0x00

```
 ** Application stopped: 12       bcpl PB_ODR, #LED       ; PB_ODR^=(1<<LED)

-> Step into assembler...
 ** Application stopped: 13       jp mloop
```

For Help, press F1 · Ln 13, Col 1 · Stop · Ready

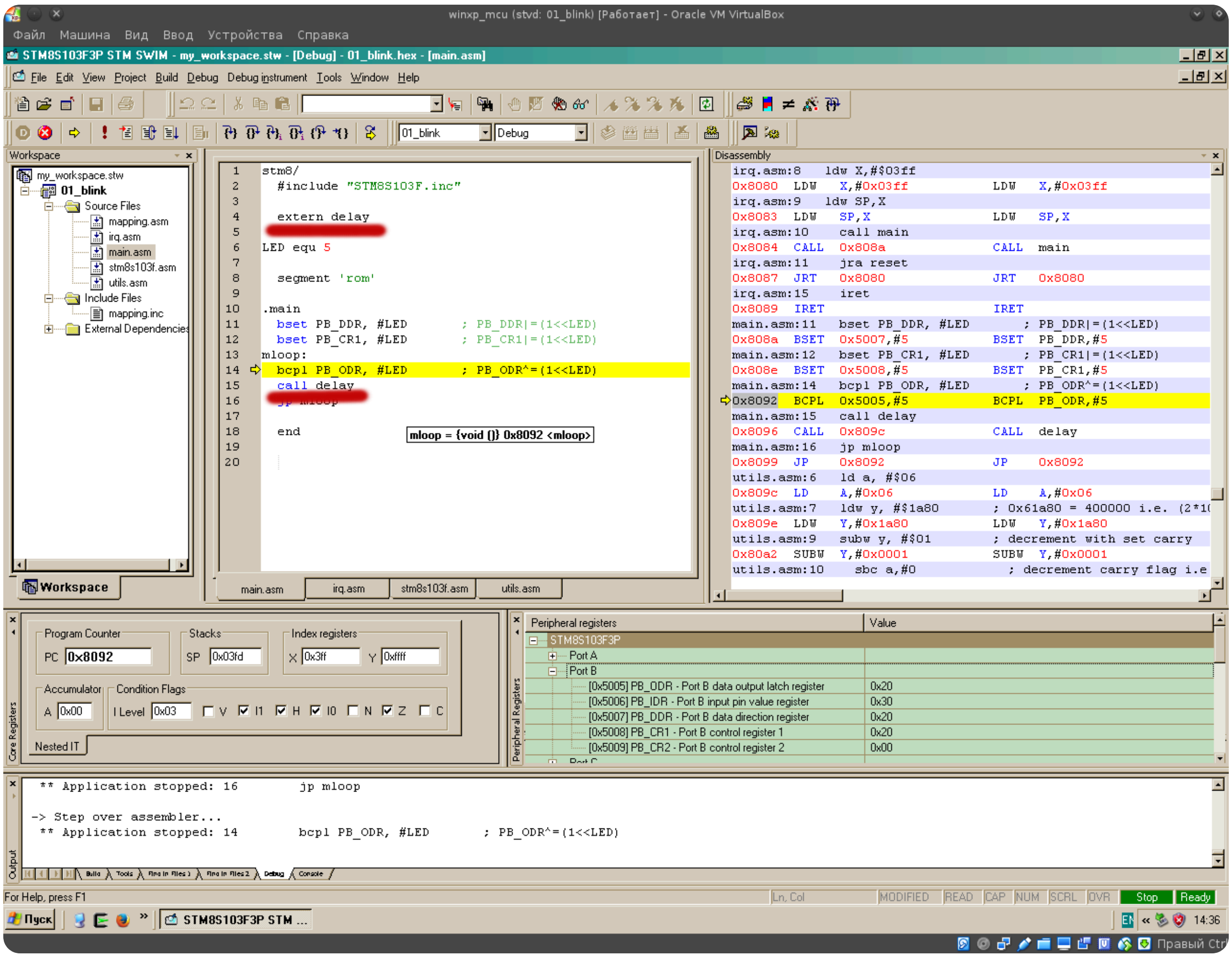
(/img/stm8/stvd/stvd_21.png)

4. Adding a file with a subroutine to the project

It remains to add a delay for the main loop so that the program works correctly. To do this, similar to how we added the irq.asm file to the project, we will add another utils.asm file with the following content:

```
stm8 /

    segment 'rom'

.delay:
    ld a, #$06
    ldw y, #$1a80               ; 0x61a80 = 400000 i.e. (2*10^6 MHz)/5cycles
loop:
    subw y, #$01                ; decrement with set carry
    sbc a,#0                    ; decrement carry flag i.e. a = a - carry_flag
    jrne loop
    ret
    end
```

Now in the main.asm file after #include "STM8S103F.inc" you need to add: "extern delay", and in the main loop you can insert a call to the subroutine: "call delay":



```
stm8/
    #include "STM8S103F.inc"

    extern delay
[redacted]
LED equ 5

    segment 'rom'

.main
    bset PB_DDR, #LED       ; PB_DDR|=(1<<LED)
    bset PB_CR1, #LED       ; PB_CR1|=(1<<LED)
mloop:
    bcpl PB_ODR, #LED       ; PB_ODR^=(1<<LED)
    call delay
[redacted]

    end
```

mloop = {void (}} 0x8092 <mloop>

```
irq.asm:8    ldw X,#$03ff
0x8080  LDW   X,#0x03ff                  LDW   X,#0x03ff
irq.asm:9    ldw SP,X
0x8083  LDW   SP,X                       LDW   SP,X
irq.asm:10    call main
0x8084  CALL  0x808a                     CALL  main
irq.asm:11    jra reset
0x8087  JRT   0x8080                     JRT   0x8080
irq.asm:15    iret
0x8089  IRET                             IRET
main.asm:11  bset PB_DDR, #LED            ; PB_DDR|=(1<<LED)
0x808a  BSET  0x5007,#5                  BSET  PB_DDR,#5
main.asm:12  bset PB_CR1, #LED            ; PB_CR1|=(1<<LED)
0x808e  BSET  0x5008,#5                  BSET  PB_CR1,#5
main.asm:14  bcpl PB_ODR, #LED            ; PB_ODR^=(1<<LED)
0x8092  BCPL  0x5005,#5                  BCPL  PB_ODR,#5
main.asm:15  call delay
0x8096  CALL  0x809c                     CALL  delay
main.asm:16  jp mloop
0x8099  JP    0x8092                     JP    0x8092
utils.asm:6  ld a, #$06
0x809c  LD    A,#0x06                    LD    A,#0x06
utils.asm:7  ldw y, #$1a80               ; 0x61a80 = 400000 i.e. (2*1
0x809e  LDW   Y,#0x1a80                  LDW   Y,#0x1a80
utils.asm:9  subw y, #$01                ; decrement with set carry
0x80a2  SUBW  Y,#0x0001                  SUBW  Y,#0x0001
utils.asm:10   sbc a,#0                   ; decrement carry flag i.e
```

Program Counter: PC 0x8092
Stacks: SP 0x03fd
Index registers: X 0x3ff Y 0xffff
Accumulator: A 0x00
Condition Flags: I Level 0x03 ☐ V ☑ I1 ☑ H ☑ I0 ☐ N ☑ Z ☐ C
Nested IT

Peripheral registers	Value
STM8S103F3P	
Port A	
Port B	
[0x5005] PB_ODR - Port B data output latch register	0x20
[0x5006] PB_IDR - Port B input pin value register	0x30
[0x5007] PB_DDR - Port B data direction register	0x20
[0x5008] PB_CR1 - Port B control register 1	0x20
[0x5009] PB_CR2 - Port B control register 2	0x00

```
 ** Application stopped: 16         jp mloop

-> Step over assembler...
 ** Application stopped: 14         bcpl PB_ODR, #LED       ; PB_ODR^=(1<<LED)
```

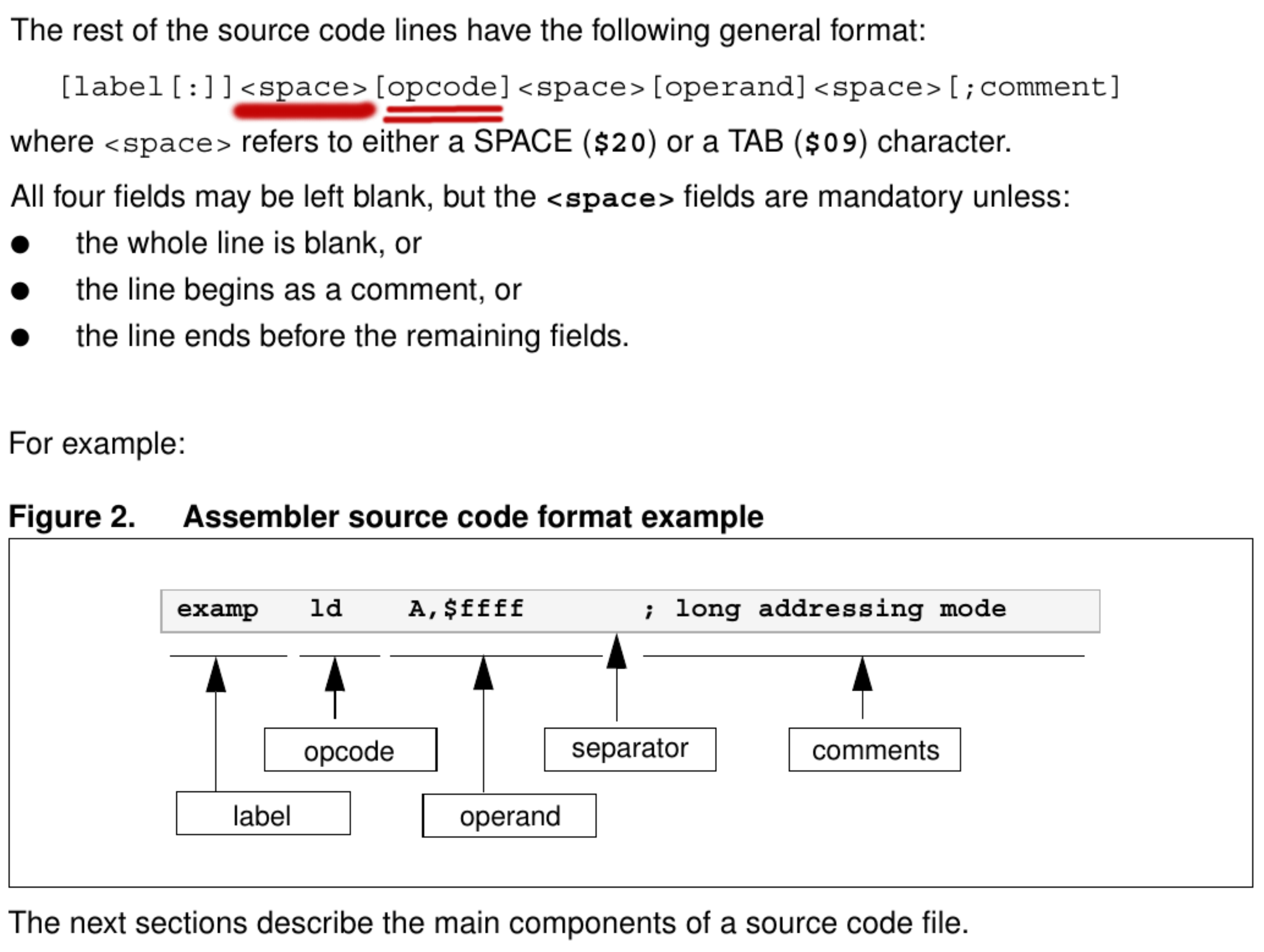
(/img/stm8/stvd/stvd_22.png)

The minimal program in STVD assembler is ready. Now we will deal with what we have programmed.

5. Basic information about STVD assembler

The assembler language consists of assembler instructions, assembler directives, preprocessor directives and macro definition language. Previously, all this together was called a macro assembler, and technically, a macro assembler is very close to a HLL in terms of capabilities. At least, this was considered earlier, when these HLLs were much simpler, like BASIC.

The STVD assembler line has the following format:



The rest of the source code lines have the following general format:

`[label[:]]<space>[opcode]<space>[operand]<space>[;comment]`

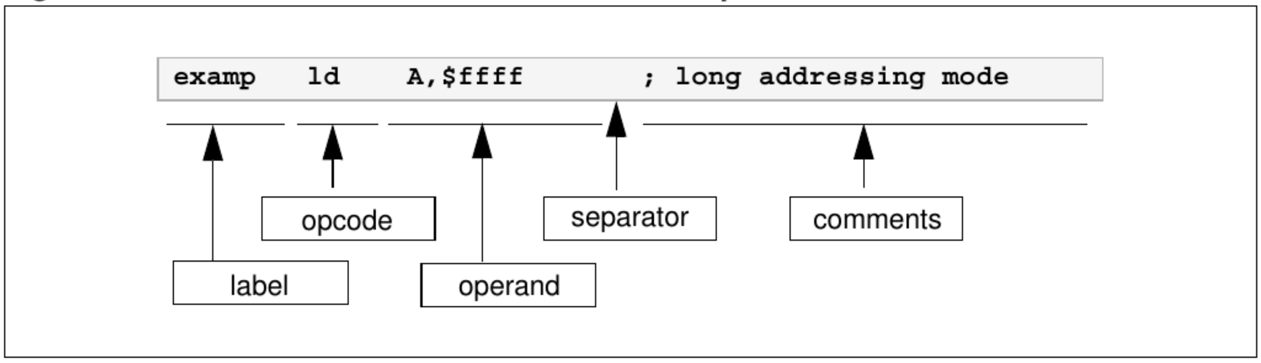
where `<space>` refers to either a SPACE (**$20**) or a TAB (**$09**) character.

All four fields may be left blank, but the **`<space>`** fields are mandatory unless:

- the whole line is blank, or
- the line begins as a comment, or
- the line ends before the remaining fields.

For example:

Figure 2. Assembler source code format example



The next sections describe the main components of a source code file.

(/img/stm8/stvd/stvd_23.png)

First comes an optional label, then comes an optional opcode, then comes the operands (if any), and ends the line with an optional comment.

Note that even if the label is omitted, **an assembler instruction or directive MUST be preceded by a SPACE or TAB. ONLY THE LABEL COMES AT THE BEGINNING OF THE LINE!** The exception to this rule is the "stm8/" directive at the beginning of the source code.

Accordingly, if you write "segment 'rom'" or "#include "STM8S103F.inc"" at the beginning of the line, then during compilation it will give an error. Apparently, the STVD developers were not indifferent to Python ;)

6. Format of numeric constants

By default, STVD uses the Motorola format for numeric constants, where decimal numbers are written as is, hexadecimal numbers are preceded by a dollar sign, binary numbers by a percent sign, and octal numbers by a tilde.

Using the directives: MOTOROLA, INTEL, TEXAS, ZILOG you can change the format of a numeric constant:

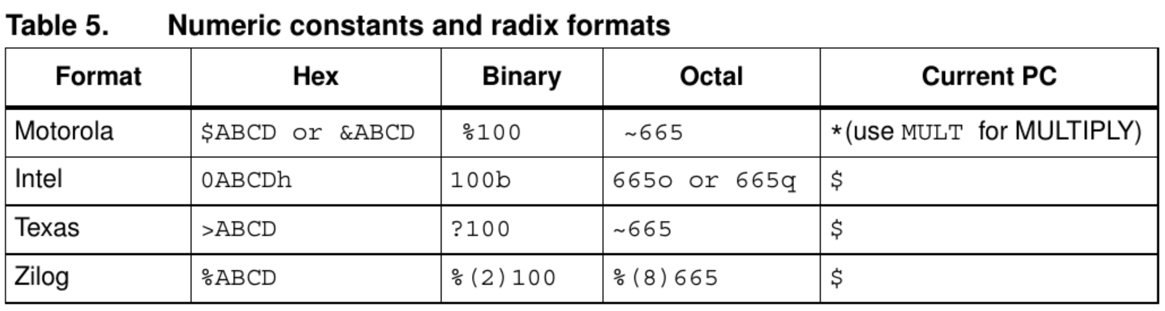


Table 5. Numeric constants and radix formats

Format	Hex	Binary	Octal	Current PC
Motorola	`$ABCD or &ABCD`	`%100`	`~665`	*(use `MULT` for MULTIPLY)
Intel	`0ABCDh`	`100b`	`665o or 665q`	$
Texas	`>ABCD`	`?100`	`~665`	$
Zilog	`%ABCD`	`%(2)100`	`%(8)665`	$

(/img/stm8/stvd/stvd_24.png)

The new format will be applied to the program text after the directive.

7. Label format

The label must start from the beginning of the line and may contain: uppercase and lowercase letters of the English alphabet, numbers and the underscore symbol. The label must start with a letter or an underscore symbol, i.e. not with a number. The label can contain up to 30 characters.

The label may end with a colon or not at all. The colon is ignored.

The label may contain a suffix of the form: label[.b|.w|.l], which consists of a dot and additional letters: b, w, l.

Using the suffixes: b, w, l, the label can be one-, two-, or four-byte. Thus, the label can be used for one-, two-, or three-byte addressing.

As an example, we can use the following source code:

```
stm8/
    segment byte at 10 'ram0'
var1.b ds.b
    segment byte at 100 'ram1'
var2.w ds.b

    segment 'rom'
.main
    ld a,var1
    ld a,var2
    ld a, const
mloop.l
    jpf mloop

const:
    dc.b $aa
    end
```

Which will be compiled into the following mashcode:

```
    808a:       b6 10           ld A,0x10 ;0x10
    808c:       c6 01 00        ld A,0x0100 ;0x100
    808f:       c6 80 96        ld A,0x8096 ;0x8096
    8092:       ac 00 80 92     jpf 0x008092 ;0x8092
    8096:       aa 00           or
```

Here two variables are specified: var1 in the zero page of RAM, var2 in the first page of RAM, and the constant const in the flash memory area. When accessing the first variable, short addressing (shortmem) is used, when accessing the second variable and the constant, long addressing (longmem) is used, and the jpf instruction jumps to the label with extended addressing (extmem).

By default, the label size is two bytes (one word), but it can be changed using the directives: BYTES, WORDS, LONGS.

The label can be absolute or relative. The relative label is calculated by the linker at the firmware compilation stage.

By default, all labels are local. Using the PUBLIC and EXTERN directives, you can exchange labels between individual program modules. The PUBLIC directive makes the label visible to the linker. The EXTERN directive makes an external label visible to this module. As an alternative to the PUBLIC directive, a dot at the beginning of the label can be used.

8. Segmentation

Segmentation allows you to control from the project what the linker script does in GCC, i.e. using the segmentation directive, you can tell the linker where a particular code or data should be located. One program module (file) can contain up to 128 segments.

The segment is set using the segmentation directive. Its format is shown below:

```
    [<name>] SEGMENT [<align>] [<combine>] '<class>' [cod]
```

First comes the optional parameter - name. It can be up to 12 characters long. Segments of the same class, but with different names, are grouped by the linker one after another in the order of definition.

The alignment parameter is "align", it can take the following values:

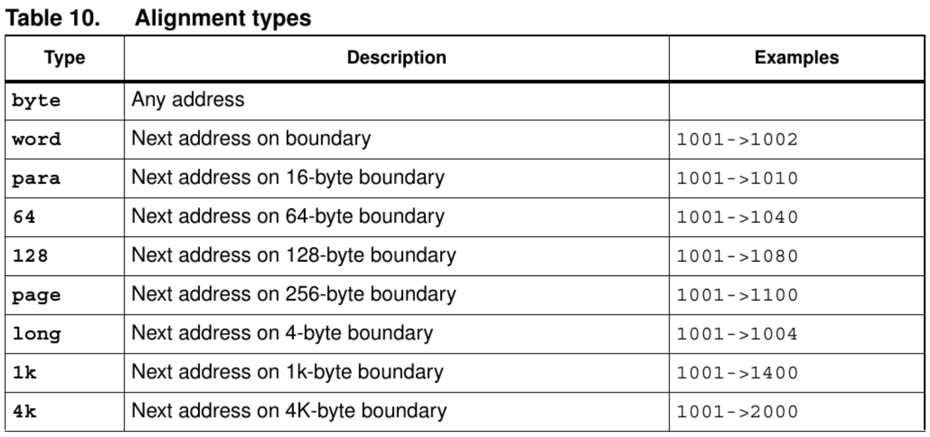
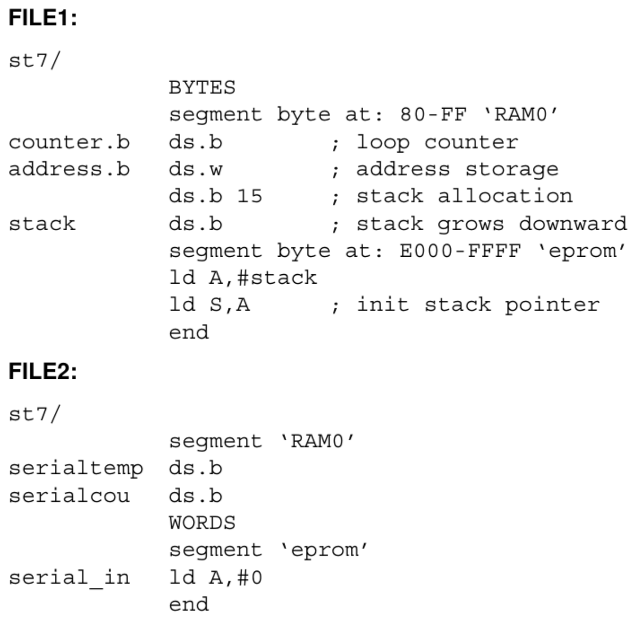


Table 10. Alignment types

Type	Description	Examples
byte	Any address	
word	Next address on boundary	1001->1002
para	Next address on 16-byte boundary	1001->1010
64	Next address on 64-byte boundary	1001->1040
128	Next address on 128-byte boundary	1001->1080
page	Next address on 256-byte boundary	1001->1100
long	Next address on 4-byte boundary	1001->1004
1k	Next address on 1k-byte boundary	1001->1400
4k	Next address on 4K-byte boundary	1001->2000

(/img/stm8/stvd/stvd_26.png)

Above I gave an example of how to use segments to set variables in the RAM area. Below are more examples from the company manual:



FILE1:

```
st7/
            BYTES
            segment byte at: 80-FF ‘RAM0’
counter.b   ds.b        ; loop counter
address.b   ds.w        ; address storage
            ds.b 15     ; stack allocation
stack       ds.b        ; stack grows downward
            segment byte at: E000-FFFF ‘eprom’
            ld A,#stack
            ld S,A      ; init stack pointer
            end
```

FILE2:

```
st7/
            segment ‘RAM0’
serialtemp  ds.b
serialcou   ds.b
            WORDS
            segment ‘eprom’
serial_in   ld A,#0
            end
```

(/img/stm8/stvd/stvd_25.png)

Here the counter label corresponds to address 0x80, the address label to 0x81, stack to 0x92. In the second file, the serialtemp label has address 0x93, and serialcou to 0x94. Since both files write to the "eprom" segment, the second file writes after the first.

The next parameter of the segment directive is <combine> , which tells the linker where to place the segment. There are three options for this parameter:

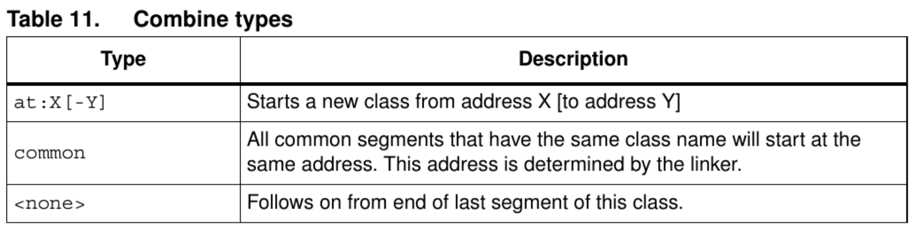


Table 11. Combine types

Type	Description
`at:X[-Y]`	Starts a new class from address X [to address Y]
`common`	All common segments that have the same class name will start at the same address. This address is determined by the linker.
`<none>`	Follows on from end of last segment of this class.

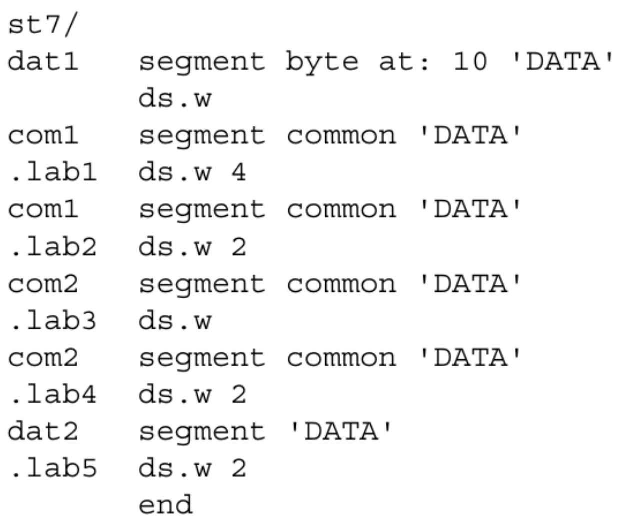
(/img/stm8/stvd/stvd_27.png)

The at- option MUST be used when declaring a class, and must be used ONLY ONCE .

Also in case of at the starting segment address must be specified, and optionally the last segment address or its size. All addresses are written in HEXADECIMAL form AS IS without prefixes.

The common option allows you to define common data areas. Areas of this type with the same class name will have the same starting address. This allows you to use such segments for data exchange.

Let's consider this example:

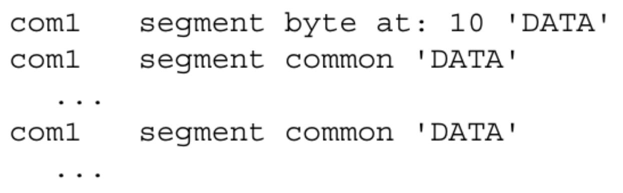


```
st7/
dat1   segment byte at: 10 'DATA'
       ds.w
com1   segment common 'DATA'
.lab1  ds.w 4
com1   segment common 'DATA'
.lab2  ds.w 2
com2   segment common 'DATA'
.lab3  ds.w
com2   segment common 'DATA'
.lab4  ds.w 2
dat2   segment 'DATA'
.lab5  ds.w 2
       end
```

(/img/stm8/stvd/stvd_28.png)

Here, labels lab1 and lab2 will have address 0x12, lab3 and lab4 will have address 0x1a, and lab5 will have address 0x1e.

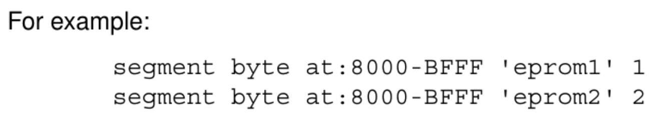
You cannot combine common and at- segments with the same name. The following example demonstrates the error:



```
com1   segment byte at: 10 'DATA'
com1   segment common 'DATA'
   ...
com1   segment common 'DATA'
   ...
```

(/img/stm8/stvd/stvd_29.png)

The last parameter of the segment directive is cod, which allows you to control the output file obtained during linking, in which the segment data will be placed. If the cod parameter is omitted, then everything will be compiled into a single default.cod. If you specify a number from zero to nine as the cod parameter, then the linker will place all segments of this class in the prog_x.cod file, where x is the code number. This can be useful, for example, when forming different eeproms for different devices. For example:



For example:

```
        segment byte at:8000-BFFF 'eprom1' 1
        segment byte at:8000-BFFF 'eprom2' 2
```

(/img/stm8/stvd/stvd_30.png)

9. Basic assembler directives

A complete list of directives with detailed descriptions is given in the STM8 Assembler and Linker User's Guide: ST Assembler-Linker UM0144 (https://www.st.com/content/ccc/resource/technical/document/user_manual/b8/61/d0/68/83/82/4f/2b/CD00056470.pdf/files/CD00056470.pdf/jcr:cont , Appendix A. I would like to mention the most requested directives.

EQU - substitution or conformity directive. Probably exists in all assembly languages. Format:

```
label EQU expression
```

When compiling a program, the label is replaced by its expression. Example:

```
var1    equ 5
```

#define is a directive similar to EQU, but has a significant difference. In EQU, a number is assigned to the label, its dimension: byte, word, double word - is determined at the time of assignment. In the case of using #define, a string is assigned to the label, which is converted to a number during compilation, and its dimension is determined in the program itself. Format:

```
    #define <alias> <string>
```

Example of use:

```
    #define value 5
    ld a,#value    ; ld a,#5
```

CEQU - a directive similar to EQU but allows you to change its contents:

```
lab1    CEQU {lab1+1} ; inc lab1
```

Used in combination with the REPEAT and UNTIL directives.

Directives: **DC.B, DC.W, DS.L** - allow you to write a constant or an array. It is also possible to write an ASCII string.

Directives: **DS.B, DS.W, DS.L** - allow you to reserve space. The number of reserved cells is indicated after a comma.

The STRING directive allows you to reserve an ASCII string. It is a synonym for the DC.B directive. Examples of use:

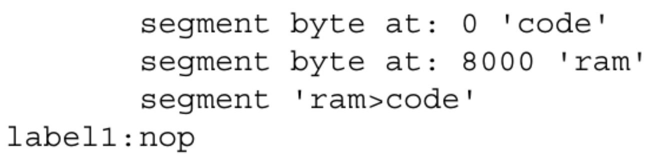
```
    STRING 1,2,3 ;    generates 01,02,03
    STRING “HELLO”    ; generates 48,45,4C,4C,4F
    STRING “HELLO”,0 ;    generates 48,45,4C,4C,4F,00
```

I would like to consider the directives intended for writing macros in the corresponding chapter.

10. Copying code into RAM and executing it from there

Executing code from RAM makes sense in the STM8L series, there is a power saving mode WFE that allows working with the periphery without interruptions, which eliminates the use of flash memory during the main cycle. Refusal to use flash memory allows reducing energy consumption, but it should not be forgotten that a program from RAM takes longer to execute than from flash. Now we will see for ourselves, but before running a program from RAM, it must be copied there. In this case, all absolute addresses must be indexed somehow.

The STM8 Assembler and Linker User's Guide: ST Assembler-Linker UM0144 (https://www.st.com/content/ccc/resource/technical/document/user_manual/b8/61/d0/68/83/82/4f/2b/CD00056470.pdf/files/CD00056470.pdf/jcr:cont describes a mechanism for writing code specifically designed for copying to RAM. To do this, two segments are declared, where one will be used for execution and the other for storing code. For example:



```
        segment byte at: 0 'code'
        segment byte at: 8000 'ram'
        segment 'ram>code'
label1:nop
```

(/img/stm8/stvd/stvd_31.png)

here the code starting with label1 will be stored in 'code' segment, but all labels will be recalculated relative to 'ram' segment. In addition, the address space of 'ram' segment will be reserved for this code.

All this is good, but I couldn't compile this example. However, it is not so difficult to create relocatable code manually. For the experiment, let's take the example of the basic project from the first part. From it, you will need to delete the utils.asm file, and then main.asm will look like this:

```
stm8/
    #include "STM8S103F.inc"

LED equ 5
offset equ $4000

    segment  'rom'
.main :
     clrw x
lp
    ld a , xl
    ; 11/18/2022 fixed a bug, missing hash sign below
    cp a , # $20                    ; 32 bytes length of ram_loader
    jreq jump
    ld a , (code, x )
     ld ( $100 , x ), a
    incw x
    jp lp
jump :
     jp $100

    segment 'eeprom'
code
    bset PB_DDR, #LED           ; PB_DDR|=(1<<LED)
    bset PB_CR1, #LED           ; PB_CR1|=(1<<LED)
mloop:
    bcpl PB_ODR, #LED           ; PB_ODR^=(1<<LED)
    ; --- delay 1 sec ----
    ld a, #$06
    ldw y, #$1a80               ; 0x61a80 = 400000 i.e. (2*10^6 MHz)/5cycles
loop:
    subw y, #$01                ; decrement with set carry
    sbc a,#0                    ; decrement carry flag i.e. a = a - carry_flag
    jrne loop
    ;----------------------
    ;jra mloop                  ; relative label
    jp {mloop-offset+$100}      ; absolute label

    end
```

Here the 'eeprom' segment contains the program placed in the RAM, and the 'rom' segment contains the bootloader. The program is placed in the first page, i.e. from address 0x100, while the zero page is allocated for variables. In the program, all relative labels are left as is, while absolute ones are written with an offset: address-(starting address of eeprom)+(address of the beginning of the first page of RAM).

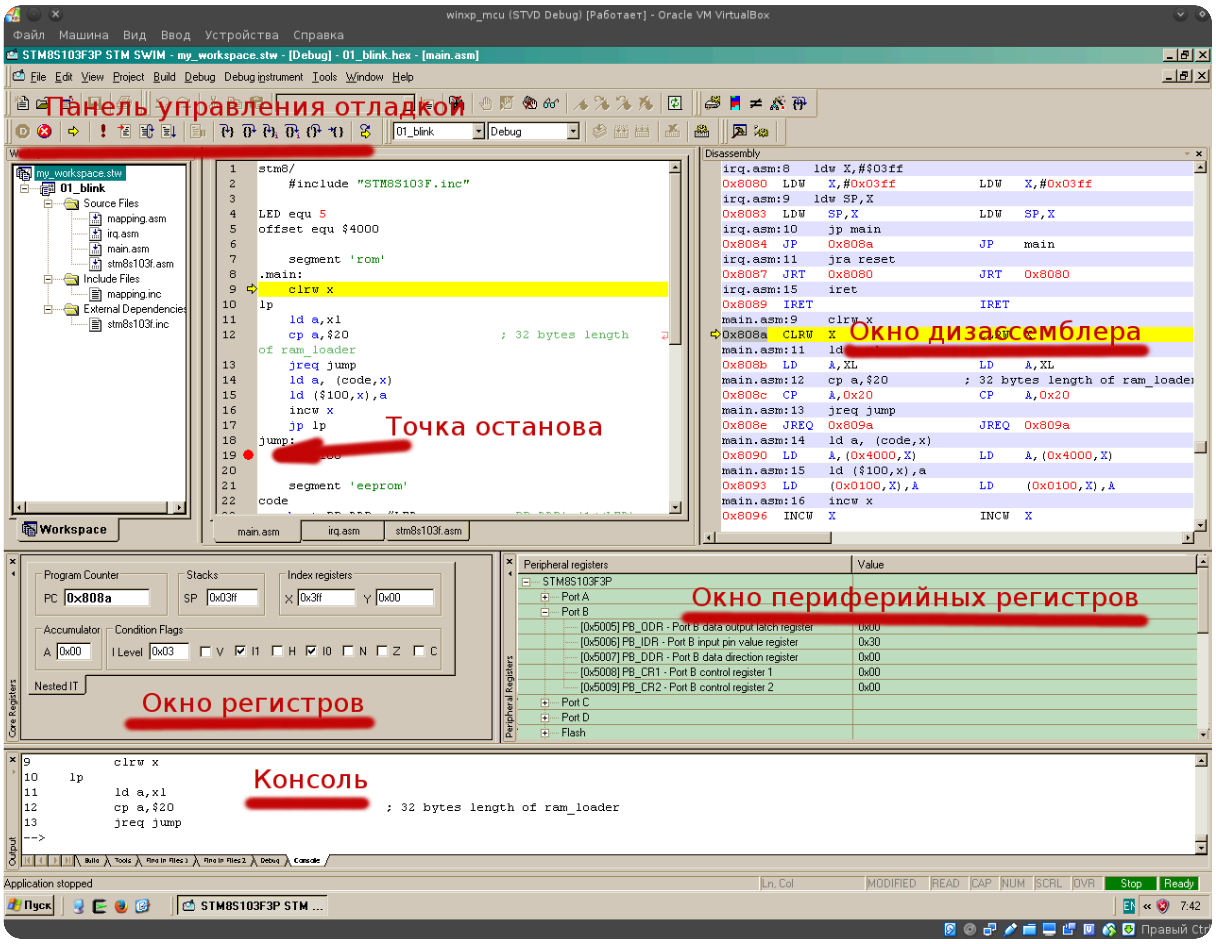
After flashing the microcontroller, it will be seen that the LED has a blinking half-period much longer than one second. Determining "by eye", I got about a threefold decrease in performance, which is actually what was said in the documentation: 3-level STM8 pipeline: translation of chapters 3, 4, 5 of the STM8 microcontroller programming manual (PM0044) (/2018/stm8_manual/#10)

```
Reading an instruction from RAM is similar to reading from ROM. However, since the RAM bus is only 8-bit wide, it takes
4 consecutive reads to fill one Fx word. As a result, code from RAM executes slower than from flash memory.
```

In the future, I will use this program to demonstrate the debugging process in STVD.

11. STVD debug interface

STVD uses the powerful GDB as a debugger, its full capabilities are available through the "Console" tab in the "Output Window". STVD acts as a frontend to GDB:



```
stm8/
    #include "STM8S103F.inc"

LED equ 5
offset equ $4000

    segment 'rom'
.main:
    clrw x
lp
    ld a,xl
    cp a,$20                    ; 32 bytes length
of ram_loader
    jreq jump
    ld a, (code,x)
    ld ($100,x),a
    incw x
    jp lp
jump:
    [illegible]

    segment 'eeprom'
code
```

```
irq.asm:8   ldw X,#$03ff
0x8080  LDW   X,#0x03ff                LDW   X,#0x03ff
irq.asm:9   ldw SP,X
0x8083  LDW   SP,X                     LDW   SP,X
irq.asm:10   jp main
0x8084  JP    0x808a                   JP    main
irq.asm:11   jra reset
0x8087  JRT   0x8080                   JRT   0x8080
irq.asm:15   iret
0x8089  IRET                           IRET
main.asm:9   clrw x
0x808a  CLRW  X
main.asm:11   ld
0x808b  LD    A,XL                     LD    A,XL
main.asm:12   cp a,$20           ; 32 bytes length of ram_loader
0x808c  CP    A,0x20                   CP    A,0x20
main.asm:13   jreq jump
0x808e  JREQ  0x809a                   JREQ  0x809a
main.asm:14   ld a, (code,x)
0x8090  LD    A,(0x4000,X)             LD    A,(0x4000,X)
main.asm:15   ld ($100,x),a
0x8093  LD    (0x0100,X),A             LD    (0x0100,X),A
main.asm:16   incw x
0x8096  INCW  X                        INCW  X
```

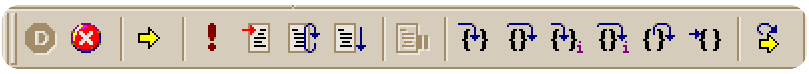
Peripheral registers	Value
STM8S103F3P	
Port A	
Port B	
[0x5005] PB_ODR - Port B data output latch register	0x00
[0x5006] PB_IDR - Port B input pin value register	0x30
[0x5007] PB_DDR - Port B data direction register	0x00
[0x5008] PB_CR1 - Port B control register 1	0x00
[0x5009] PB_CR2 - Port B control register 2	0x00
Port C	
Port D	
Flash	

```
9           clrw x
10     lp
11          ld a,xl
12          cp a,$20                          ; 32 bytes length of ram_loader
13          jreq jump
-->
```

(/img/stm8/stvd/stvd_32.png)

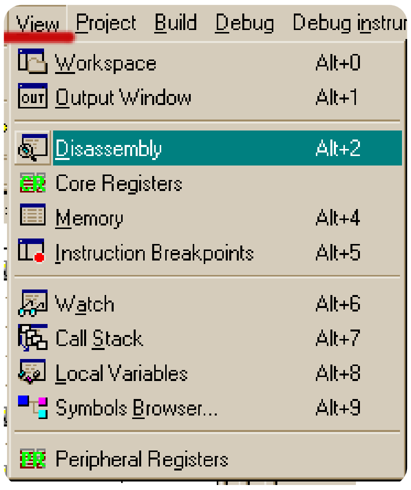
I don't have much experience with STVD yet, but from all the GDB features, I only printed memory dumps via console. Everything else I did via GUI, Working with it saves a lot of time.

So, after debugging is started, the debugging control panel becomes available:

 (/img/stm8/stvd/stvd_33.png)

It implements the tracing mode and contains the following operations: start debugging, stop debugging, GDB command - Run, GDB command - Continue, GDB command - next, GDB command - nexti, GDB command - step, GDB command - stepi, etc. All icons have tooltips that explain their functions.

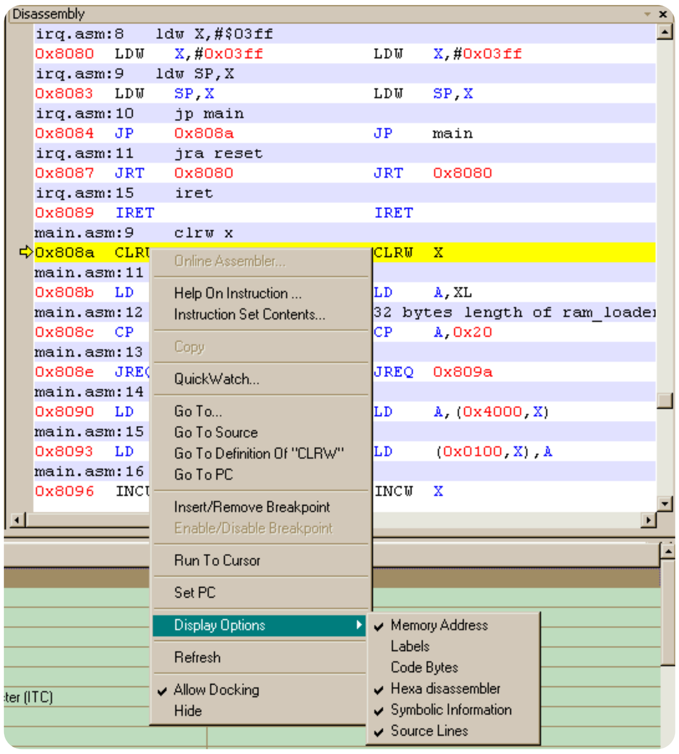
You can open a particular window via menu->View:



(/img/stm8/stvd/stvd_34.png)

All windows are stackable and can be closed with a cross. Window settings are saved in the workspace. The most useful windows in my opinion are the following:

1. Disassembler window. It is probably most useful when debugging a C program, it allows you to see what code a particular program construct has been converted into. For an assembler, this window is useful because it allows you to see the addresses and dimensions of labels, as well as the numeric address of the input-output port. You can configure the displayed columns via "Display Options". The Refresh option is needed when you enter debugging commands via the gdb console. Then you have to refresh to see the changes.

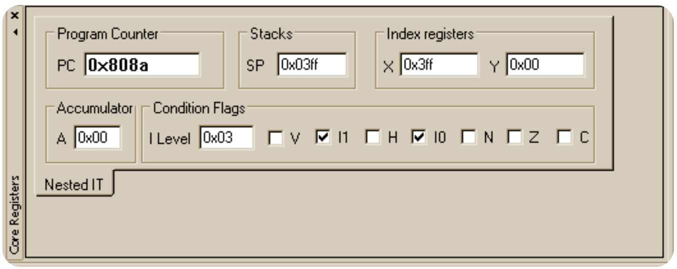


Disassembly

```
irq.asm:8    ldw X,#$03ff
0x8080  LDW    X,#0x03ff                 LDW    X,#0x03ff
irq.asm:9    ldw SP,X
0x8083  LDW    SP,X                      LDW    SP,X
irq.asm:10     jp main
0x8084  JP     0x808a                    JP     main
irq.asm:11     jra reset
0x8087  JRT    0x8080                    JRT    0x8080
irq.asm:15     iret
0x8089  IRET                             IRET
main.asm:9     clrw x
0x808a  CLRU                             CLRW   X
main.asm:11
0x808b  LD                               LD     A,XL
main.asm:12                              32 bytes length of ram_loader
0x808c  CP                               CP     A,0x20
main.asm:13
0x808e  JRE(                             JREQ   0x809a
main.asm:14
0x8090  LD                               LD     A,(0x4000,X)
main.asm:15
0x8093  LD                               LD     (0x0100,X),A
main.asm:16
0x8096  INCU                             INCW   X
```

(/img/stm8/stvd/stvd_35.png)

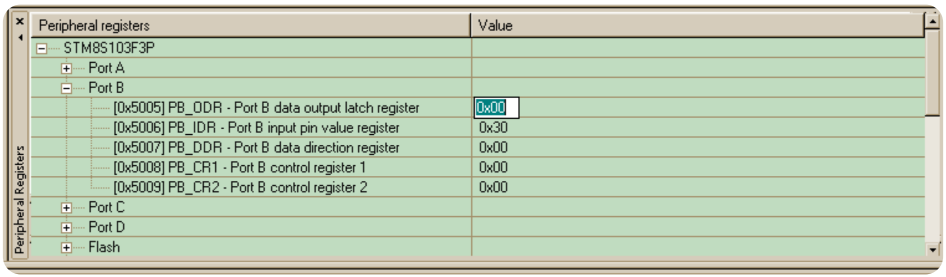
2. The next window "Core Registers" shows the contents of the microcontroller registers and status flags. You can change the value of the registers, remove and set the necessary flags directly in the window.



Program Counter: PC 0x808a
Stacks: SP 0x03ff
Index registers: X 0x3ff Y 0x00
Accumulator: A 0x00
Condition Flags: I Level 0x03 ☐ V [✓ I1] ☐ H [✓ I0] ☐ N ☐ Z ☐ C
Nested IT

(/img/stm8/stvd/stvd_36.png)

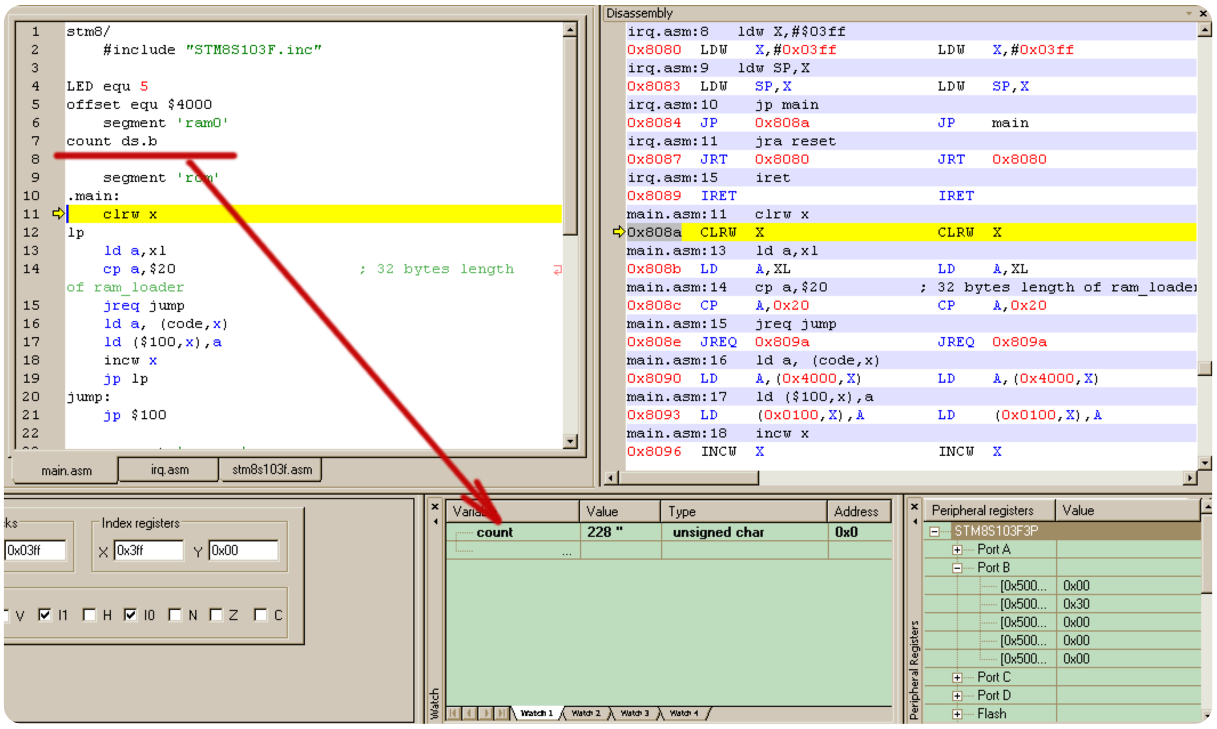
3. The peripheral registers window is a tree-like list grouped by hardware modules. Here you can also change the contents of the RVV:



Peripheral registers	Value
⊟ STM8S103F3P	
⊞ Port A	
⊟ Port B	
[0x5005] PB_ODR - Port B data output latch register	0x00
[0x5006] PB_IDR - Port B input pin value register	0x30
[0x5007] PB_DDR - Port B data direction register	0x00
[0x5008] PB_CR1 - Port B control register 1	0x00
[0x5009] PB_CR2 - Port B control register 2	0x00
⊞ Port C	
⊞ Port D	
⊞ Flash	

(/img/stm8/stvd/stvd_37.png)

4. The next useful window is the variables window. A variable can be entered by name or by address. You can change the value of a variable.



```
stm8/
    #include "STM8S103F.inc"

LED equ 5
offset equ $4000
    segment 'ram0'
count ds.b

    segment 'rom'
.main:
    clrw x
lp
    ld a,xl
    cp a,$20                          ; 32 bytes length
of ram_loader
    jreq jump
    ld a, (code,x)
    ld ($100,x),a
    incw x
    jp lp
jump:
    jp $100
```

```
Disassembly
irq.asm:8    ldw X,#$03ff
0x8080  LDW   X,#0x03ff              LDW   X,#0x03ff
irq.asm:9    ldw SP,X
0x8083  LDW   SP,X                   LDW   SP,X
irq.asm:10   jp main
0x8084  JP    0x808a                 JP    main
irq.asm:11   jra reset
0x8087  JRT   0x8080                 JRT   0x8080
irq.asm:15   iret
0x8089  IRET                         IRET
main.asm:11  clrw x
0x808a  CLRW  X                      CLRW  X
main.asm:13  ld a,xl
0x808b  LD    A,XL                   LD    A,XL
main.asm:14  cp a,$20              ; 32 bytes length of ram_loade
0x808c  CP    A,0x20                 CP    A,0x20
main.asm:15  jreq jump
0x808e  JREQ  0x809a                 JREQ  0x809a
main.asm:16  ld a, (code,x)
0x8090  LD    A,(0x4000,X)           LD    A,(0x4000,X)
main.asm:17  ld ($100,x),a
0x8093  LD    (0x0100,X),A           LD    (0x0100,X),A
main.asm:18  incw x
0x8096  INCW  X                      INCW  X
```

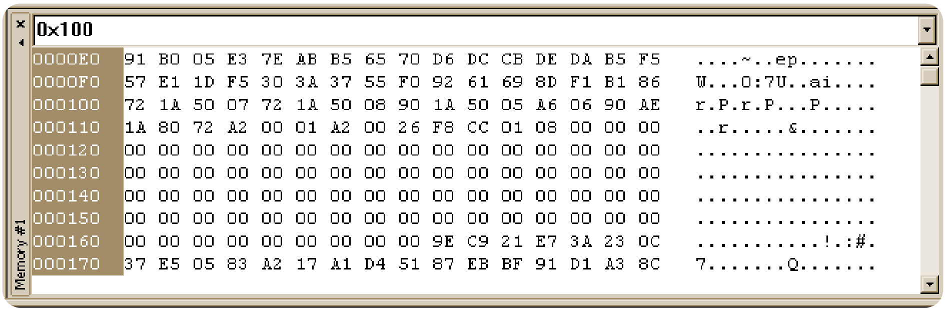
Index registers: X 0x3ff, Y 0x00; 0x03ff

Variable	Value	Type	Address
count	228 "	unsigned char	0x0

Peripheral registers	Value
STM8S103F3P	
Port A	
Port B	
[0x500...	0x00
[0x500...	0x30
[0x500...	0x00
[0x500...	0x00
[0x500...	0x00
Port C	
Port D	
Flash	

(/img/stm8/stvd/stvd_38.png)

5. And the last window is the memory dump window. Personally, I find it easier to enter the command "x/10xb" in the gdb console, but this window can also be useful. At least because you can change the contents of memory cells in interactive mode.

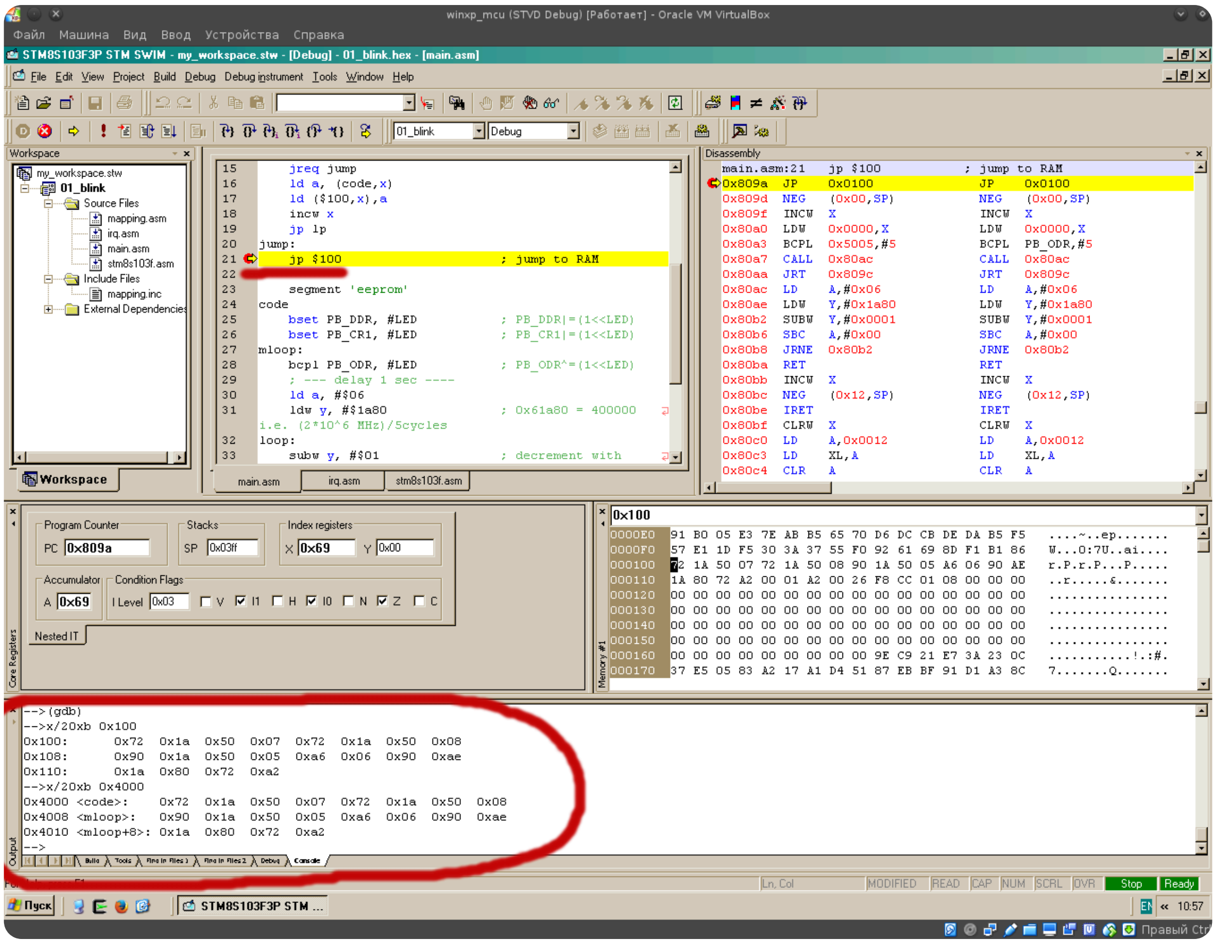


```
0x100
0000E0  91 B0 05 E3 7E AB B5 65 70 D6 DC CB DE DA B5 F5   ....~..ep.......
0000F0  57 E1 1D F5 30 3A 37 55 F0 92 61 69 8D F1 B1 86   W...0:7U..ai....
000100  72 1A 50 07 72 1A 50 08 90 1A 50 05 A6 06 90 AE   r.P.r.P...P.....
000110  1A 80 72 A2 00 01 A2 00 26 F8 CC 01 08 00 00 00   ..r.....&.......
000120  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
000130  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
000140  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
000150  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
000160  00 00 00 00 00 00 00 00 00 9E C9 21 E7 3A 23 0C   ...........!.:#.
000170  37 E5 05 83 A2 17 A1 D4 51 87 EB BF 91 D1 A3 8C   7.......Q.......
```

(/img/stm8/stvd/stvd_39.png)

12. Debugging process in STVD

To debug the algorithm, we set a breakpoint at the jp $100 instruction, i.e. after the copy operation from eeprom to ram is completed, and before transferring control to the copied code. Click the Run icon (looks like an exclamation mark) and after stopping, compare both segments: ram1 and eeprom:



```
    jreq jump
    ld a, (code,x)
    ld ($100,x),a
    incw x
    jp lp
jump:
    jp $100                        ; jump to RAM

    segment 'eeprom'
code
    bset PB_DDR, #LED              ; PB_DDR|=(1<<LED)
    bset PB_CR1, #LED              ; PB_CR1|=(1<<LED)
mloop:
    bcpl PB_ODR, #LED              ; PB_ODR^=(1<<LED)
    ; --- delay 1 sec ----
    ld a, #$06
    ldw y, #$1a80                  ; 0x61a80 = 400000
i.e. (2*10^6 MHz)/5cycles
loop:
    subw y, #$01                   ; decrement with
```

```
main.asm:21   jp $100                    ; jump to RAM
0x809a  JP     0x0100                 JP     0x0100
0x809d  NEG    (0x00,SP)              NEG    (0x00,SP)
0x809f  INCW   X                      INCW   X
0x80a0  LDW    0x0000,X               LDW    0x0000,X
0x80a3  BCPL   0x5005,#5              BCPL   PB_ODR,#5
0x80a7  CALL   0x80ac                 CALL   0x80ac
0x80aa  JRT    0x809c                 JRT    0x809c
0x80ac  LD     A,#0x06                LD     A,#0x06
0x80ae  LDW    Y,#0x1a80              LDW    Y,#0x1a80
0x80b2  SUBW   Y,#0x0001              SUBW   Y,#0x0001
0x80b6  SBC    A,#0x00                SBC    A,#0x00
0x80b8  JRNE   0x80b2                 JRNE   0x80b2
0x80ba  RET                           RET
0x80bb  INCW   X                      INCW   X
0x80bc  NEG    (0x12,SP)              NEG    (0x12,SP)
0x80be  IRET                          IRET
0x80bf  CLRW   X                      CLRW   X
0x80c0  LD     A,0x0012               LD     A,0x0012
0x80c3  LD     XL,A                   LD     XL,A
0x80c4  CLR    A                      CLR    A
```

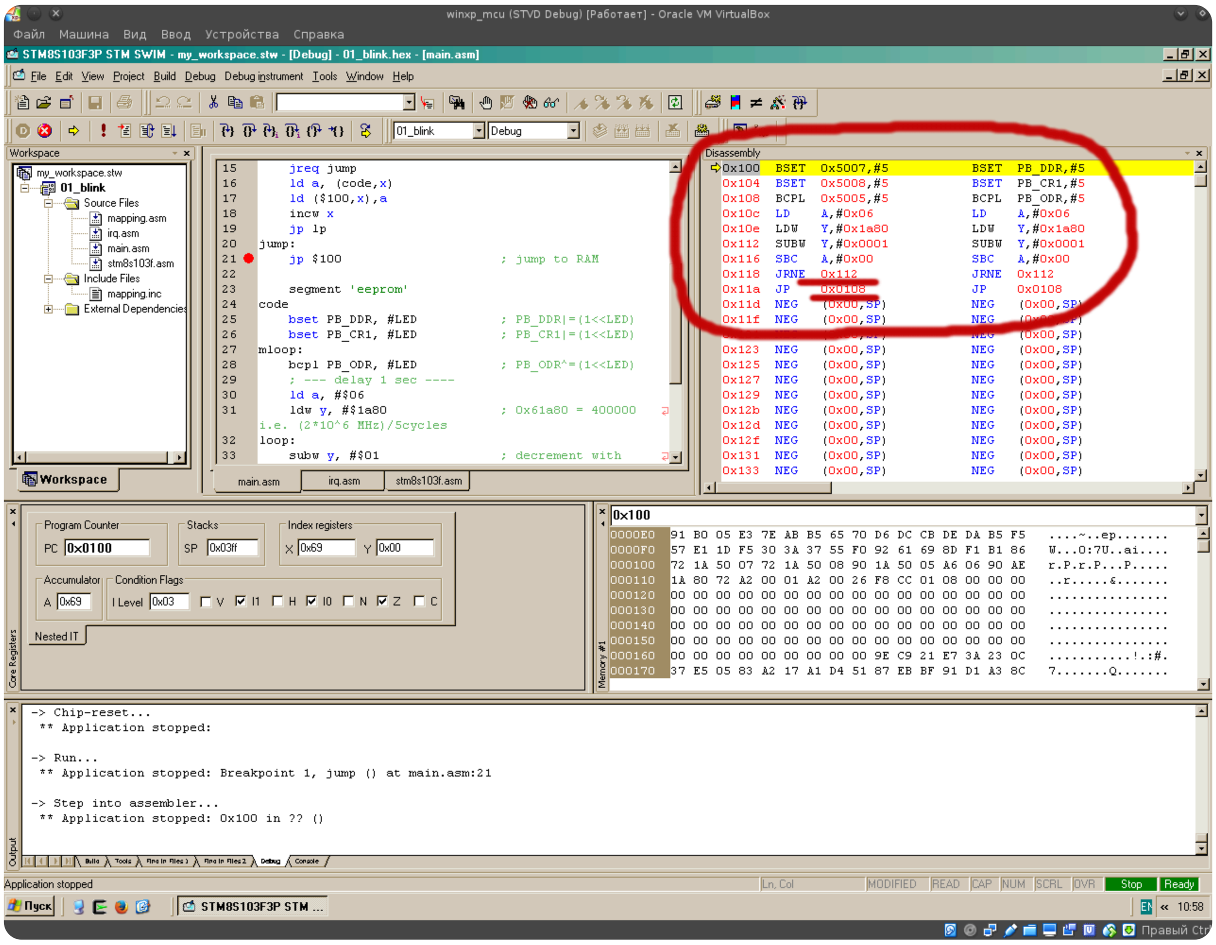
PC 0x809a SP 0x03ff X 0x69 Y 0x00 A 0x69 I Level 0x03

```
0x100
0000E0 91 B0 05 E3 7E AB B5 65 70 D6 DC CB DE DA B5 F5   ....~..ep.......
0000F0 57 E1 1D F5 30 3A 37 55 F0 92 61 69 8D F1 B1 86   W...0:7U..ai....
000100 72 1A 50 07 72 1A 50 08 90 1A 50 05 A6 06 90 AE   r.P.r.P...P.....
000110 1A 80 72 A2 00 01 A2 00 26 F8 CC 01 08 00 00 00   ..r.....&.......
000120 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
000130 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
000140 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
000150 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
000160 00 00 00 00 00 00 00 00 00 9E C9 21 E7 3A 23 0C   ..........!.:#.
000170 37 E5 05 83 A2 17 A1 D4 51 87 EB BF 91 D1 A3 8C   7.......Q.......
```

```
--> (gdb)
-->x/20xb 0x100
0x100:        0x72  0x1a  0x50  0x07  0x72  0x1a  0x50  0x08
0x108:        0x90  0x1a  0x50  0x05  0xa6  0x06  0x90  0xae
0x110:        0x1a  0x80  0x72  0xa2
-->x/20xb 0x4000
0x4000 <code>:    0x72  0x1a  0x50  0x07  0x72  0x1a  0x50  0x08
0x4008 <mloop>:   0x90  0x1a  0x50  0x05  0xa6  0x06  0x90  0xae
0x4010 <mloop+8>: 0x1a  0x80  0x72  0xa2
-->
```

(/img/stm8/stvd/stvd_40.png)

If the code has been copied successfully, we jump to address 0x100, after which we check the copied code in the disassembler window. Particular attention should be paid to the jump addresses.



```
    jreq jump
    ld a, (code,x)
    ld ($100,x),a
    incw x
    jp lp
jump:
    jp $100                        ; jump to RAM

    segment 'eeprom'
code
    bset PB_DDR, #LED              ; PB_DDR|=(1<<LED)
    bset PB_CR1, #LED              ; PB_CR1|=(1<<LED)
mloop:
    bcpl PB_ODR, #LED              ; PB_ODR^=(1<<LED)
    ; --- delay 1 sec ----
    ld a, #$06
    ldw y, #$1a80                  ; 0x61a80 = 400000
i.e. (2*10^6 MHz)/5cycles
loop:
    subw y, #$01                   ; decrement with
```

```
0x100  BSET   0x5007,#5              BSET   PB_DDR,#5
0x104  BSET   0x5008,#5              BSET   PB_CR1,#5
0x108  BCPL   0x5005,#5              BCPL   PB_ODR,#5
0x10c  LD     A,#0x06                LD     A,#0x06
0x10e  LDW    Y,#0x1a80              LDW    Y,#0x1a80
0x112  SUBW   Y,#0x0001              SUBW   Y,#0x0001
0x116  SBC    A,#0x00                SBC    A,#0x00
0x118  JRNE   0x112                  JRNE   0x112
0x11a  JP     0x0108                 JP     0x0108
0x11d  NEG    (0x00,SP)              NEG    (0x00,SP)
0x11f  NEG    (0x00,SP)              NEG    (0x00,SP)
                                     NEG    (0x00,SP)
0x123  NEG    (0x00,SP)              NEG    (0x00,SP)
0x125  NEG    (0x00,SP)              NEG    (0x00,SP)
0x127  NEG    (0x00,SP)              NEG    (0x00,SP)
0x129  NEG    (0x00,SP)              NEG    (0x00,SP)
0x12b  NEG    (0x00,SP)              NEG    (0x00,SP)
0x12d  NEG    (0x00,SP)              NEG    (0x00,SP)
0x12f  NEG    (0x00,SP)              NEG    (0x00,SP)
0x131  NEG    (0x00,SP)              NEG    (0x00,SP)
0x133  NEG    (0x00,SP)              NEG    (0x00,SP)
```

PC 0x0100 SP 0x03ff X 0x69 Y 0x00 A 0x69 I Level 0x03

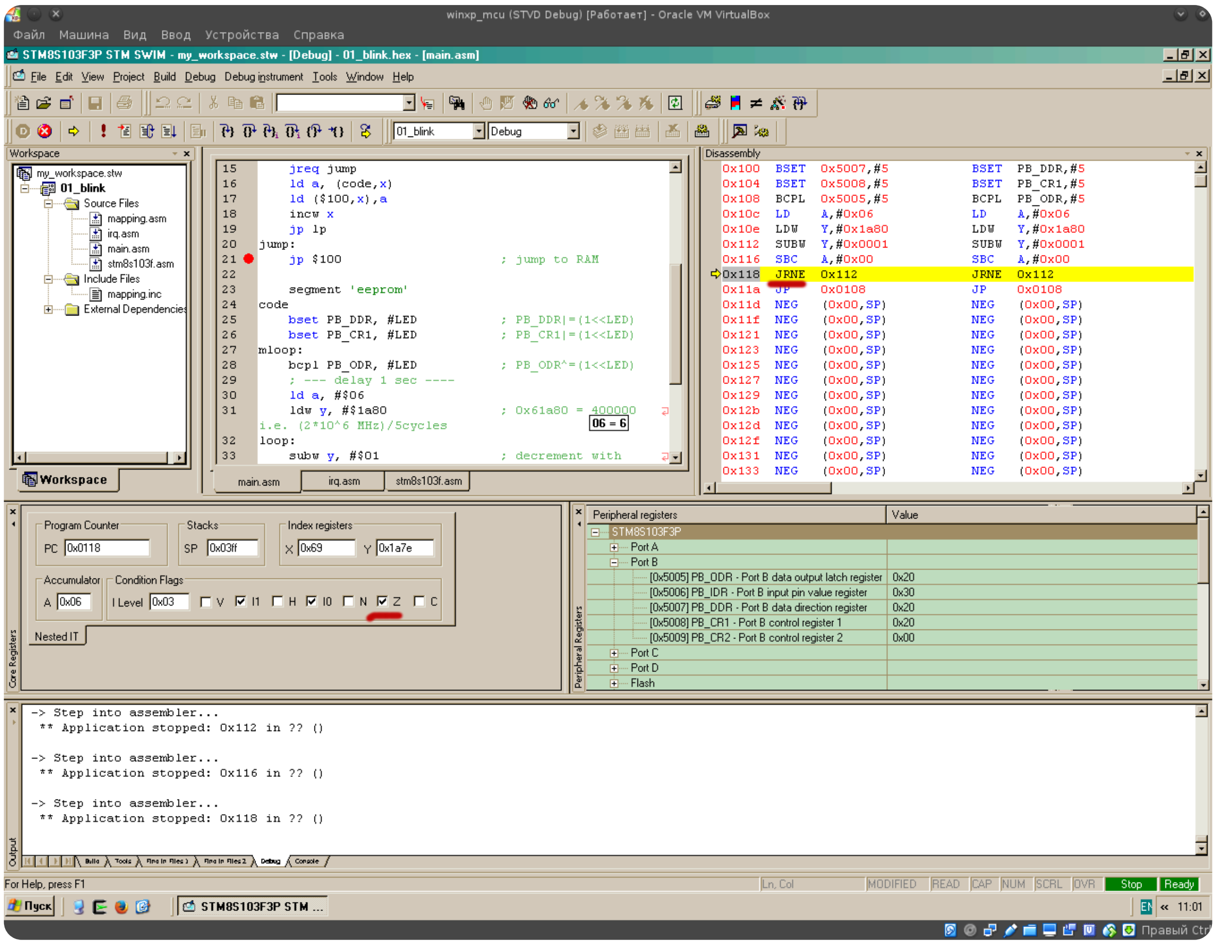
```
0x100
0000E0 91 B0 05 E3 7E AB B5 65 70 D6 DC CB DE DA B5 F5   ....~..ep.......
0000F0 57 E1 1D F5 30 3A 37 55 F0 92 61 69 8D F1 B1 86   W...0:7U..ai....
000100 72 1A 50 07 72 1A 50 08 90 1A 50 05 A6 06 90 AE   r.P.r.P...P.....
000110 1A 80 72 A2 00 01 A2 00 26 F8 CC 01 08 00 00 00   ..r.....&.......
000120 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
000130 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
000140 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
000150 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
000160 00 00 00 00 00 00 00 00 00 9E C9 21 E7 3A 23 0C   ..........!.:#.
000170 37 E5 05 83 A2 17 A1 D4 51 87 EB BF 91 D1 A3 8C   7.......Q.......
```

```
-> Chip-reset...
 ** Application stopped:

-> Run...
 ** Application stopped: Breakpoint 1, jump () at main.asm:21

-> Step into assembler...
 ** Application stopped: 0x100 in ?? ()
```

Application stopped

(/img/stm8/stvd/stvd_41.png)

If the algorithm is copied successfully, you can proceed to the usual debugging of the algorithm. It is convenient to set the necessary flags on the conditional jump instructions in order to pass through certain branches of the algorithm without delay.



```
    jreq jump
    ld a, (code,x)
    ld ($100,x),a
    incw x
    jp lp
jump:
    jp $100                        ; jump to RAM

    segment 'eeprom'
code
    bset PB_DDR, #LED              ; PB_DDR|=(1<<LED)
    bset PB_CR1, #LED              ; PB_CR1|=(1<<LED)
mloop:
    bcpl PB_ODR, #LED              ; PB_ODR^=(1<<LED)
    ; --- delay 1 sec ----
    ld a, #$06
    ldw y, #$1a80                  ; 0x61a80 = 400000
i.e. (2*10^6 MHz)/5cycles
loop:
    subw y, #$01                   ; decrement with
```

```
0x100   BSET   0x5007,#5           BSET   PB_DDR,#5
0x104   BSET   0x5008,#5           BSET   PB_CR1,#5
0x108   BCPL   0x5005,#5           BCPL   PB_ODR,#5
0x10c   LD     A,#0x06             LD     A,#0x06
0x10e   LDW    Y,#0x1a80           LDW    Y,#0x1a80
0x112   SUBW   Y,#0x0001           SUBW   Y,#0x0001
0x116   SBC    A,#0x00             SBC    A,#0x00
0x118   JRNE   0x112               JRNE   0x112
0x11a   JP     0x0108              JP     0x0108
0x11d   NEG    (0x00,SP)           NEG    (0x00,SP)
0x11f   NEG    (0x00,SP)           NEG    (0x00,SP)
0x121   NEG    (0x00,SP)           NEG    (0x00,SP)
0x123   NEG    (0x00,SP)           NEG    (0x00,SP)
0x125   NEG    (0x00,SP)           NEG    (0x00,SP)
0x127   NEG    (0x00,SP)           NEG    (0x00,SP)
0x129   NEG    (0x00,SP)           NEG    (0x00,SP)
0x12b   NEG    (0x00,SP)           NEG    (0x00,SP)
0x12d   NEG    (0x00,SP)           NEG    (0x00,SP)
0x12f   NEG    (0x00,SP)           NEG    (0x00,SP)
0x131   NEG    (0x00,SP)           NEG    (0x00,SP)
0x133   NEG    (0x00,SP)           NEG    (0x00,SP)
```

PC 0x0118, SP 0x03ff, X 0x69, Y 0x1a7e, A 0x06, I Level 0x03; Condition Flags: V [ ], I1 [✓], H [ ], I0 [✓], N [ ], Z [✓], C [ ]

Peripheral registers	Value
STM8S103F3P	
Port A	
Port B	
[0x5005] PB_ODR - Port B data output latch register	0x20
[0x5006] PB_IDR - Port B input pin value register	0x30
[0x5007] PB_DDR - Port B data direction register	0x20
[0x5008] PB_CR1 - Port B control register 1	0x20
[0x5009] PB_CR2 - Port B control register 2	0x00
Port C	
Port D	
Flash	

```
-> Step into assembler...
 ** Application stopped: 0x112 in ?? ()

-> Step into assembler...
 ** Application stopped: 0x116 in ?? ()

-> Step into assembler...
 ** Application stopped: 0x118 in ?? ()
```

(/img/stm8/stvd/stvd_42.png)

In my opinion, it’s not complicated at all.

13. Introduction to STVD Macro Assembler

I am not an expert on macro assembler, frankly speaking, I have never used it and therefore have only a general idea about it. I thought that this was a good opportunity to fill an unfortunate gap in my education, however, the following text should not be treated as some kind of manual.

Macro assembler allows you to reduce the routine when writing assembler programs, as well as reduce your time debugging them. The principle is to combine a group of assembler instructions into a single macro, which, like a function or subroutine, can be set parameters and controlled using a preprocessor. Once you write and debug such a macro, you will not have to do it again. Using macros can bring the readability of a program closer to high-level languages like BASIC. At the same time, you will still have all the power of assembler, and you will not be limited by the framework of any programming language.

The main problem with macros is the complexity of debugging them at the preprocessor stage. When you get an error on a macro, you get an error on the line where it is called, and not on the problematic construction in the macro body. To be fair, I should say that STVD has a %OUT directive that outputs a line to the compilation log, but you can't output a number through it.

The macro assembler directives and the macro definition format are described in the manual: STM8 Assembler and Linker User's Guide: ST Assembler-Linker UM0144 (https://www.st.com/content/ccc/resource/technical/document/user_manual/b8/61/d0/68/83/82/4f/2b/CD00056470.pdf/files/CD00056470.pdf/jcr:cont . I will not repeat what is written there, instead I will give several examples of using macros.

To begin with, a very simple case, let's try to replace the line of transition to an absolute address with a macro:

```
    jp {mloop-offset+$100}
```

Here it is necessary to put the address calculation into a macro, so as not to write it every time. The macro will be like this:

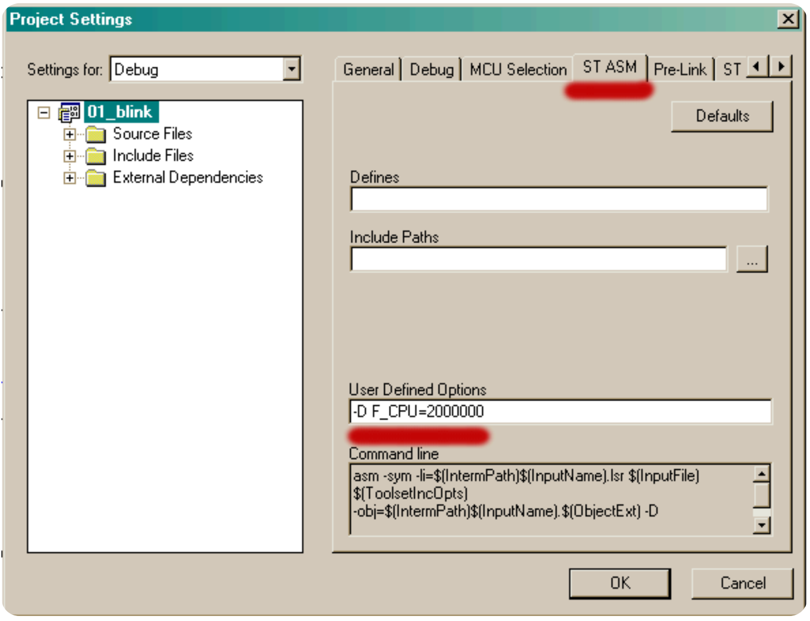
```
jp_ram MACRO adr
    jp {adr-offset+$100}
    MEND
```

Then the use of the macro will be reduced to the line:

```
    jp_ram mloop
```

14. Delay macro delay_ms

Now a more complicated example, we will try to repeat the well-known macro _delay_ms which is available in gcc-avr and is used quite often. To do this, first in the project settings you will need to specify the F_CPU frequency using a named constant:



Settings for: Debug

Defines

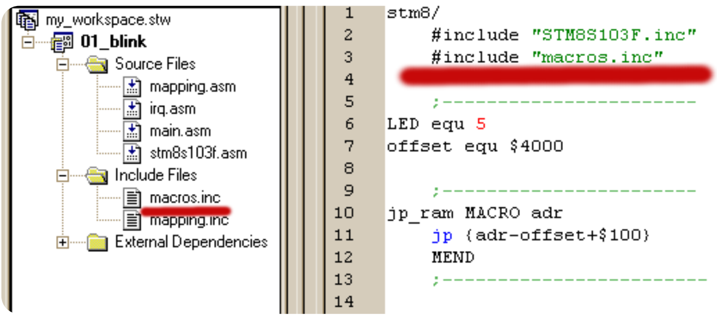
Include Paths

User Defined Options
-D F_CPU=2000000

Command line
```
asm -sym -li=$(IntermPath)$(InputName).lsr $(InputFile)
$(ToolsetIncOpts)
-obj=$(IntermPath)$(InputName).$(ObjectExt) -D
```

(/img/stm8/stvd/stvd_43.png)

In addition, we will need a separate file macros.inc, into which we will dump the macros. This file will then need to be included via the #include directive:



```
stm8/
    #include "STM8S103F.inc"
    #include "macros.inc"

    ;------------------------
LED equ 5
offset equ $4000

    ;------------------------
jp_ram MACRO adr
    jp (adr-offset+$100)
    MEND
    ;------------------------

```

(/img/stm8/stvd/stvd_44.png)

My macro turned out like this:

```
delay_ms MACRO ms
    #IFDEF F_CPU
    #ELSE
    #define F_CPU 2000000
    #ENDIF
    ld a, #{{SEG {ms mult {F_CPU div 1000}}} div 5}
    ldw y, #{{OFFSET {ms mult {F_CPU div 1000}}} div 5}
    LOCAL loop
loop:
    subw y, #$01    ; decrement with set carry
    sbc a,#0        ; decrement carry flag i.e. a = a - carry_flag
    jrne loop
    MEND
```

Then the code in the main.asm module will look like this:

```
stm8/
    #include "STM8S103F.inc"
    #include "macros.inc"

LED equ 5
offset equ $4000

jp_ram MACRO adr
    jp {adr-offset+$100}
    MEND

    segment 'rom'
.main:
    ; --- GPIO Setup ---------
    bset PB_DDR, #LED           ; PB_DDR|=(1<<LED)
    bset PB_CR1, #LED           ; PB_CR1|=(1<<LED)
    ; --- Copy to RAM ---------
    clrw x
lp
    ld a,xl
    cp a,$20                    ; 32 bytes length of ram_loader
    jreq jump
    ld a, (code,x)
    ld ($100,x),a
    incw x
    jra lp
jump:
    jp $100

    segment 'eeprom'
code
mloop:
    bcpl PB_ODR, #LED           ; PB_ODR^=(1<<LED)
    delay_ms 1000
    jp_ram mloop

    end
```

15. Macro of conditional operator of comparison of a register with a number

As I have already said, macros can be used to create something similar to operators in JVU. For example, in the code copyer from eeprom to ram there is a comparison of a number with a register. You can write a macro that compares a register with a number and performs a transition by a label in case of a match.

I got two macros. One for 16-bit registers, the other for 8-bit ones:

```
if_reg_eq8 MACRO reg value label
    #IFIDN reg yh
    ld a, y
    #ENDIF
    #IFIDN reg yl
    ld a,yl
    #ENDIF
    #IFIDN reg xh
    ld a, xh
    #ENDIF
    #IFIDN reg xl
    ld a,xl
    #ENDIF
    cp a,value
    jreq label
    MEND
    ;--------------------------
if_reg_eq16 MACRO reg value label
    #IFIDN reg x
        pushw x
        ldw x,#{value}
        cpw x,($01,sp)
        popw x
        jreq label
    #ELSE
        pushw x
        pushw y
        ldw x,#{value}
        cpw x,($01,sp)
        popw y
        popw x
        jreq label
    #ENDIF
    MEND
```

main.c in this case takes the form:

```
stm8/
    #include "STM8S103F.inc"
    #include "macros.inc"

    ;-----------------------
LED equ 5
offset equ $4000

    ;-----------------------
jp_ram MACRO adr
    jp {adr-offset+$100}
    MEND
    ;------------------------

    segment 'rom'
.main:
    ; GPIO SETUP
    bset PB_DDR, #LED           ; PB_DDR|=(1<<LED)
    bset PB_CR1, #LED           ; PB_CR1|=(1<<LED)
    ; Copy EEPROM to RAM1
    clrw x
lp
    if_reg_eq8 xl $20 jump
    ld a, (code,x)
    ld ($100,x),a
    incw x
    jra lp
jump:
    jp $100                     ; go to RAM

    ;-----------------------
    segment 'eeprom'
code:
mloop:
    bcpl PB_ODR, #LED           ; PB_ODR^=(1<<LED)
    delay_ms 1000
    jp_ram mloop

    end
```

Similarly, you can write macros to compare a number with a memory cell, or compare based on whether it is greater than, less than, less than or equal to, etc.

16. 1 μs delay on conditional branch instructions

If it is not a problem to organize a timer delay of 1 ms or more, then with a delay of 1 μs, not everything is so simple. Timer interrupt handlers will not have time to process incoming interrupts. The only way out is to make a delay on instructions. But counting cycles with the pipeline running is a thankless task. The number of cycles will depend on the previous instructions and the addressing method. In addition, I came across information that at different clock frequencies the same instructions will be executed in a different number of cycles. And it seems that I have encountered this in practice. As a way to solve this problem, it is recommended to block the pipeline. This is done using instructions that reset the pipeline queue - FLUSH. Such instructions are conditional jump instructions. Thus, the delay can be organized by sequentially executing conditional jump operations with a zero offset. One such instruction should ALWAYS be executed in 2 cycles.

The REPEAT and UNTIL directives will help you record a sequence of instructions. Unfortunately, they cannot be used in macros, so their use looks like this:

```
    scf                         ; Set Carry Flag
us  CEQU 5
    REPEAT
    dc.b $25, $0                ; JRC label
us  CEQU {us-1}
    UNTIL {us eq 0}
```

Here it is assumed that fCPU = 2MHz and since the conditional jump instruction in case of a jump is executed in two clock cycles, then each JRC instruction takes 1 μs to execute. The REPEAT/UNTIL directives insert five such instructions into the program text, therefore a delay of 5 μs is formed.

For delays of 10 μs and more, you can use TIM4/TIM6 timers, and for smaller delays, I think this method will do.

(/)

ALSO ON **COUNT-ZERO**

ATtiny13a: использование …
6 лет назад • 8 комментариев
ATtiny13a: использование ассемблера GNU-AS в программах на Си …

ATmega8 + PCD8544: работа с …
7 лет назад • 4 комментариев
ATmega8 + PCD8544: работа с графическим дисплеем от телефонов …

Использование Qbs для работы с STM8
2 года назад • 6 комментариев
Использование Qbs для работы с STM8/SDCC в QtCreator разделы: …

STM8S + SDCC: …
6 лет назад • 5 комментариев
STM8S + SDCC: Программирование БЕЗ SPL. Интерфейсы: UART …

STM8S + SDCC: …
6 лет назад • 5 комментар
STM8S + SDCC: Программирование на связке языков Си и …

3 Комментариев 1 **Войти ▾**

G Присоединиться к обсуждению...

ВОЙТИ С ПОМОЩЬЮ **ИЛИ ЧЕРЕЗ DISQUS** ?

Имя

♡ **Поделиться** **Лучшие** **Новые** **Старые**

P **Prism**
3 года назад

А вот у меня вопрос: а из eeprom можно код выполнять? не всегда же используется

0 0 Ответить Поделиться ›

B **Василий**
5 лет назад

Доброго времени.
Объясните пожалуйста почему стек 513 байт?
Автор: "Здесь первые две страницы ОЗУ выделены в сегменты RAM0 и RAM1, остальные 513 байт...".
Как бы логично что:
-с 0x000 по 0x0FF RAM0 - 256 байт
-с 0x100 по 0x1FF RAM1 - 256 байт
= 512 байт
-с 0x200 по 0x3FF STACK - 512 байт, откуда лишний байт для стека?

0 0 Ответить Поделиться ›

flanker Модератор ➦ Василий
4 года назад

странно, только сейчас увидел коммент. по вопросу. скорее всего 513-й байт стека "виртуальный". т.к. указатель стека всегда указывает на свободную ячейку памяти, то если полностью забить стек, указатель стека будет указывать на 513-й байт.

0 0 Ответить Поделиться ›

Подписаться **О защите персональных данных** **Не продавайте мои данные**