

## STM8 + STVD + ASSEMBLER: Quick Start

(/2018/stm8\_stvd\_asm/)

**Forums: STM8 (/tags/stm8) , Assembler (/tags/assembler) , date: August 22, 2018.**

Again, I return to the proprietary development environment - ST Visual Develop, for which there are two reasons. Firstly, it turned out that it is impossible to write any complex firmware in assembler without a debugger, at least I did not succeed, because the program is anyway debugged using an LED or via UART anyway, it is simply done faster through the debug interface. Secondly, it seemed to me that studying architecture only by being guided by a datasheet is not entirely correct. Something may be misunderstood, something may be overlooked. Things like DMA, built-in RTC, or executing code from RAM will be easier to deal with with the debugger, while 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 STM8 assembler firmware from scratch using ST Visual Develop.

STVD - works in Windows operating systems starting from XP and higher. At the same time, it works great from under a virtual machine in Linux. For this article, I am using STVD 4.3.12, the latest version available at this point, and Windows XP SP3 as the guest OS. As a microcontroller, I will use a 20-pin STM8S103F3P6 (<http://4.bp.blogspot.com/-JB5qpbYEHlg/Vm2acOTEC4I/AAAAAAAAABWQ/eNTM-XclnY/s1600/stm8s103f3p6.jpg>) .

Below is the documentation I relied on when writing the 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/CD00161709.pdf/jcr:content/primaryPDFFileData](http://www.st.com/content/ccc/resource/technical/document/programming_manual/43/24/13/9a/89/df/45/ed/CD00161709.pdf/files/CD00161709.pdf/jcr:content/primaryPDFFileData))
- STM8 Assembler-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:content/primaryPDFFileData](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:content/primaryPDFFileData))
- 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/jcr:content/primaryPDFFileData](https://www.st.com/content/ccc/resource/technical/document/user_manual/e9/d3/c3/4b/2c/0b/46/f2/CD00004556.pdf/files/CD00004556.pdf/jcr:content/primaryPDFFileData))

As a disassembler, I will use the stm8-binutils (<https://stm8-binutils-gdb.sourceforge.io>) set of utilities . The binaries of this bundle for Windows are compiled to work in CYGWIN, i.e. they understand the unix forward slash delimited file path format. CYGWIN for Windows 7 and higher is installed without any problems by following the instructions on the site <https://cygwin.com/install.html> (<https://cygwin.com/install.html>) , as for Windows XP, the instructions are available at the office . site is outdated and no longer functional. The link from the instructions on which I put - died during the writing of the article. As an alternative link, I can specify - windows xp - Is it still possible to get Cygwin for XP? - Super User (<https://superuser.com/questions/1132000/is-it-still-possible-to-get-cygwin-for-xp>) . How long this link will live, I cannot say.

As an alternative to the binutils + cygwin bundle, you can use naken\_util from the naken\_asm bundle ([https://www.mikekohn.net/micro/naken\\_asm.php](https://www.mikekohn.net/micro/naken_asm.php)) .

### The content of the article:

#### I. Create a minimal Blink project

1. Opening a templated assembly project (/2018/stm8\_stvd\_asm/#1)
2. Adding a file with a vector table and interrupt handlers to the project  
(/2018/stm8\_stvd\_asm/#2)
3. Adding constants with peripheral addresses to the project (/2018/stm8\_stvd\_asm/#3)
4. Adding a file with a subroutine to the project (/2018/stm8\_stvd\_asm/#4)

#### II. Assembly language STVD

5. Understanding STVD Assembler (/2018/stm8\_stvd\_asm/#5)
6. Numeric constant format (/2018/stm8\_stvd\_asm/#6)
7. Label format (/2018/stm8\_stvd\_asm/#7)
8. Segmentation (/2018/stm8\_stvd\_asm/#8)
9. Basic assembler directives (/2018/stm8\_stvd\_asm/#9)

#### III. Debugging process

10. Copying the code into RAM and executing it from there (/2018/stm8\_stvd\_asm/#10)
11. STVD debug interface (/2018/stm8\_stvd\_asm/#11)
12. Debugging process in STVD (/2018/stm8\_stvd\_asm/#12)

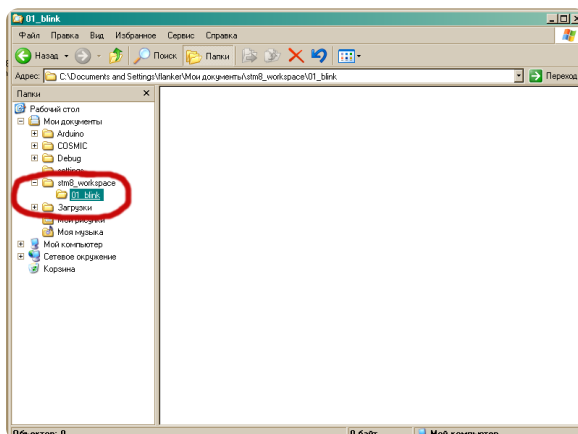
#### IV. Macro assembler

13. Introduction to the STVD Macro Assembler (/2018/stm8\_stvd\_asm/#13)
14. Delay\_ms delay macro (/2018/stm8\_stvd\_asm/#14)
15. A conditional case-to-number comparison operator macro (/2018/stm8\_stvd\_asm/#15)
16. 1 μs delay per conditional branch instruction (/2018/stm8\_stvd\_asm/#16)

## 1. Opening a template project in assembler

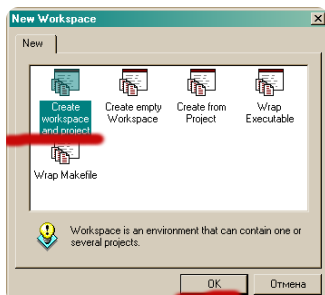
If the STVD assembler itself deserves all praise, then the project management system is made rather stupid. In STVD, there is the concept of Workspace, which stores the settings of the environment itself. A workspace can contain several projects.

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



([https://2.bp.blogspot.com/-7d\\_gzMI8EqY/W21clfd\\_0vI/AAAAAAAAADW0/wQALqE5IJ38Wxhd570UCYJb8NT\\_7WKcWACLcBGAs/s1600/stdv\\_01.png](https://2.bp.blogspot.com/-7d_gzMI8EqY/W21clfd_0vI/AAAAAAAAADW0/wQALqE5IJ38Wxhd570UCYJb8NT_7WKcWACLcBGAs/s1600/stdv_01.png))

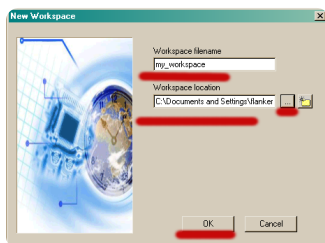
Now start STVD and select New Workspace from the menu:



(<https://2.bp.blogspot.com/->

[rERQdSJKrrs/W21clewSEDI/AAAAAAAAADWs/w62-xLQArYsV6pbOP6WtGrE1kd2xN2BAQCLcBGAs/s1600/stdv\\_02.png](https://2.bp.blogspot.com/-))

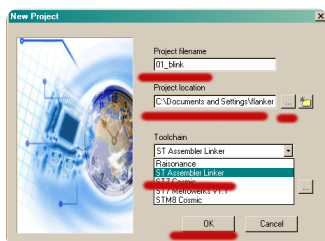
In the dialog box, set the name of the workspace and indicate the previously created folder for saving:



(<https://1.bp.blogspot.com/-kt8E9Om->

[8Eg/W21clCsZGI/AAAAAAAAADWw/I6E68s0RXDorBW7q\\_2tQ1mzz3ypEFqACLcBGAs/s1600/stdv\\_03.png](https://1.bp.blogspot.com/-kt8E9Om-))

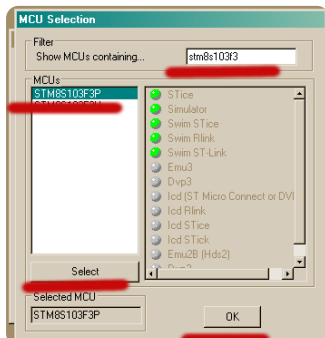
Similarly, for the project, select the name, specify the previously created project folder, and in the drop-down list, specify ST Assembler Linker as the Toolchain:



(<https://4.bp.blogspot.com/--knZp->

[JU9BM/W21cl7TBgYI/AAAAAAAAADW0/CjqlLezp3o\\_4gTJWSeIshZ5YI9XAVTrQCLcBGAs/s1600/stdv\\_04.png](https://4.bp.blogspot.com/--knZp-))

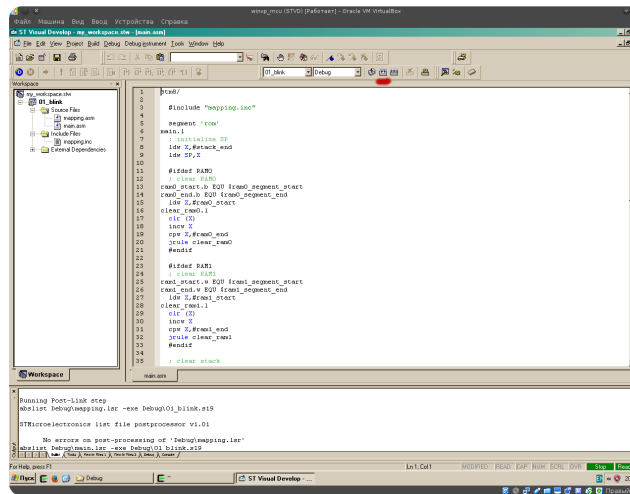
It remains to specify the target microcontroller:



(<https://2.bp.blogspot.com/->

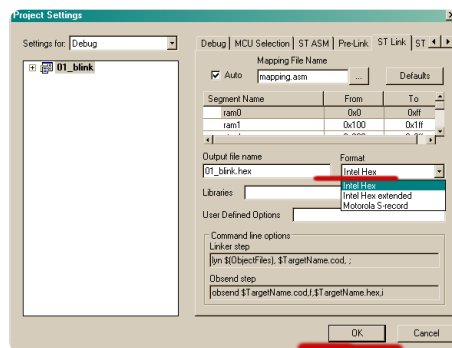
[XVLVMouzW9c/W21cJIWkml/AAAAAAAAADW4/P804AfBiiCozZDHSSy3Mpb0oMv\\_PiXwCLcBGAs/s1600/stdv\\_05.png](https://2.bp.blogspot.com/-))

A project template opens in front of us, you can immediately compile it:



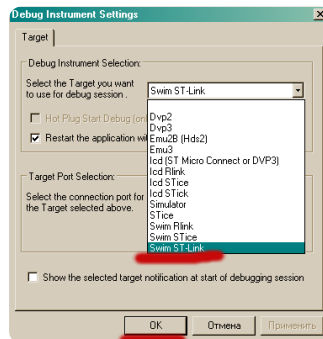
([https://2.bp.blogspot.com/-jUewpblCQrg/W21cJ5QMHYI/AAAAAAAAADW8/6ClgCwuWgN40E4Wt8o8xluUxiZKvfPQ-ACLcBGAs/s1600/stvd\\_06.png](https://2.bp.blogspot.com/-jUewpblCQrg/W21cJ5QMHYI/AAAAAAAAADW8/6ClgCwuWgN40E4Wt8o8xluUxiZKvfPQ-ACLcBGAs/s1600/stvd_06.png))

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



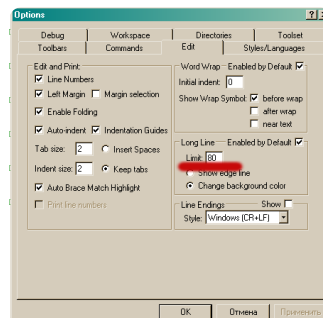
([https://4.bp.blogspot.com/-nk0bz8evMzM/W21cJfJDepl/AAAAAAAAADXA/DzKOK5i5pDwa5LQZ\\_iroCCRKsK\\_brAKwCLcBGAs/s1600/stvd\\_07.png](https://4.bp.blogspot.com/-nk0bz8evMzM/W21cJfJDepl/AAAAAAAAADXA/DzKOK5i5pDwa5LQZ_iroCCRKsK_brAKwCLcBGAs/s1600/stvd_07.png))

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



([https://1.bp.blogspot.com/-ytNqtCFRfI/W21cJi0gwI/AAAAAAAAADXE/esnjhz5urY8lbpEnkYtc1Geh9KlkgLTwCLcBGAs/s1600/stvd\\_08.png](https://1.bp.blogspot.com/-ytNqtCFRfI/W21cJi0gwI/AAAAAAAAADXE/esnjhz5urY8lbpEnkYtc1Geh9KlkgLTwCLcBGAs/s1600/stvd_08.png))

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



([https://4.bp.blogspot.com/-m1tnjUEJBBQ/W21cKEZUgCI/AAAAAAAAADXQ/dNoAu3-0UKUqm4u5CD7pMSUOhI\\_shx9DACLcBGAs/s1600/stvd\\_11.png](https://4.bp.blogspot.com/-m1tnjUEJBBQ/W21cKEZUgCI/AAAAAAAAADXQ/dNoAu3-0UKUqm4u5CD7pMSUOhI_shx9DACLcBGAs/s1600/stvd_11.png))

I will also draw your attention to the fact that there you can set the indentation width - Tab size. The default is two spaces, but I'm more used 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
│   ├── cba_err
│   ├── Debug
│   │   ├── 01_blink.cod
│   │   ├── 01_blink.gpr
│   │   ├── 01_blink.hex
│   │   ├── 01_blink.map
│   │   ├── 01_blink.s19
│   │   ├── 01_blink.sym
│   │   ├── main.lst
│   │   ├── main.obj
│   │   ├── mapping.lst
│   │   ├── mapping.obj
│   │   ├── main.asm
│   │   ├── mapping.asm
│   │   ├── mapping.inc
│   │   ├── my_workspace.stw
│   │   └── my_workspace.wed
└── 2 directories, 20 files
```

The firmware file can be disassembled from 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 the .sec1 section:
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 jrle 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 jrle 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 jrle 0x809b; 0x809b
80a2: 20 fe jra 0x80a2; 0x80a2
80a4: 80 ired

Disassembling the .sec2 section:
00008080 <.sec2>:
8080: 82 00 80 80 int 0x008080; 0x8080
8084: 82 00 80 a4 int 0x0080a4; 0x80a4
8088: 82 00 80 a4 int 0x0080a4; 0x80a4
808c: 82 00 80 a4 int 0x0080a4; 0x80a4
8090: 82 00 80 a4 int 0x0080a4; 0x80a4
8094: 82 00 80 a4 int 0x0080a4; 0x80a4
8098: 82 00 80 a4 int 0x0080a4; 0x80a4
809c: 82 00 80 a4 int 0x0080a4; 0x80a4
80a0: 82 00 80 a4 int 0x0080a4; 0x80a4
80a4: 82 00 80 a4 int 0x0080a4; 0x80a4
80a8: 82 00 80 a4 int 0x0080a4; 0x80a4
80ac: 82 00 80 a4 int 0x0080a4; 0x80a4
80b0: 82 00 80 a4 int 0x0080a4; 0x80a4
80b4: 82 00 80 a4 int 0x0080a4; 0x80a4
80b8: 82 00 80 a4 int 0x0080a4; 0x80a4
80bc: 82 00 80 a4 int 0x0080a4; 0x80a4
80c0: 82 00 80 a4 int 0x0080a4; 0x80a4
80c4: 82 00 80 a4 int 0x0080a4; 0x80a4
80c8: 82 00 80 a4 int 0x0080a4; 0x80a4
80cc: 82 00 80 a4 int 0x0080a4; 0x80a4
80d0: 82 00 80 a4 int 0x0080a4; 0x80a4
80d4: 82 00 80 a4 int 0x0080a4; 0x80a4
80d8: 82 00 80 a4 int 0x0080a4; 0x80a4
80dc: 82 00 80 a4 int 0x0080a4; 0x80a4
80e0: 82 00 80 a4 int 0x0080a4; 0x80a4
80e4: 82 00 80 a4 int 0x0080a4; 0x80a4
80e8: 82 00 80 a4 int 0x0080a4; 0x80a4
80ec: 82 00 80 a4 int 0x0080a4; 0x80a4
80f0: 82 00 80 a4 int 0x0080a4; 0x80a4
80f4: 82 00 80 a4 int 0x0080a4; 0x80a4
80f8: 82 00 80 a4 int 0x0080a4; 0x80a4
80fc: 82 00 80 a4 int 0x0080a4; 0x80a4
```

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

Loaded hexfile c:\documents and settings\flanker\myworkspace\01\_blink\Debug\01\_blink.hex from 0x8080 to 0x80a4  
Type help for a list of commands.

Addr	Opcode	Instruction	Cycles
0x8080	ae 03 ff	ldw X, #03ff	cycle=2
0x8083	94	ldw SP, X	cycle=2
0x8084	ae 00 00	ldw X, #0000	cycle=2
0x8087	7f	clr (X)	cycle=2
0x8088	5c	incw X	cycle=2
0x8089	a3 00 ff	cpw X, #00ff	cycle=2
0x808c	23 f9	jrle 0x8087 (offset=-7)	cycle=1-2
0x808e	ae 01 00	ldw X, #0100	cycle=2
0x8091	7f	clr (X)	cycle=2
0x8092	5c	incw X	cycle=1
0x8093	a3 01 ff	cpw X, #01ff	cycle=1
0x8096	23 f9	jrle 0x8091 (offset=-7)	cycle=1-2
0x8098	ae 02 00	ldw X, #0200	cycle=2
0x809b	7f	clr (X)	cycle=1
0x809c	5c	incw X	cycle=1
0x809d	a3 03 ff	cpw X, #03ff	cycle=2
0x809d	23 f9	jrle 0x809b (offset=-7)	cycle=1-2
0x80a2	20 fe	jra 0x80a2 (offset=2)	cycle=2
0x80a4	80	ired	cycle=11

(<https://2.bp.blogspot.com/>-

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

```

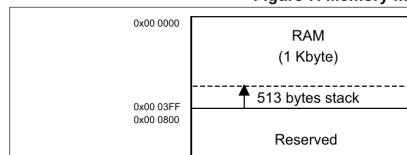
;-----
; SEGMENT MAPPING FILE AUTOMATICALLY GENERATED BY STVD
; SHOULD NOT BE MANUALLY MODIFIED.
; CHANGES WILL BE LOST WHEN FILE IS REGENERATED.
;-----
#define RAM0 1
#define ram0_segment_start 0
#define ram0_segment_end FF
#define RAM1 1
#define ram1_segment_start 100
#define ram1_segment_end 1FF
#define 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 allocated to the stack segment. Those. everything according to the datasheet:

## Memory map

Figure 7. Memory map



([https://4.bp.blogspot.com/-oKUfh6B\\_Sn4/W21cKJ6EDmI/AAAAAAAAADXM/0gAbxsYZx-MLeXPtaVT3wUtsRT3fpLDgCLcBGAs/s1600/stvd\\_10.png](https://4.bp.blogspot.com/-oKUfh6B_Sn4/W21cKJ6EDmI/AAAAAAAAADXM/0gAbxsYZx-MLeXPtaVT3wUtsRT3fpLDgCLcBGAs/s1600/stvd_10.png))

The mapping.asm file defines the segments:

```

1  stm8/
2
3  ; SEGMENT MAPPING FILE AUTOMATICALLY GENERATED BY STVD
4  ; SHOULD NOT BE MANUALLY MODIFIED.
5  ; CHANGES WILL BE LOST WHEN FILE IS REGENERATED.
6  ;-----
7  #include "mapping.inc"
8
9  BYTES ; The following addresses are 8 bits long
10 segment byte at ram0_segment_start-ram0_segment_end 'ram0'
11
12 WORDS ; The following addresses are 16 bits long
13 segment byte at ram1_segment_start-ram1_segment_end 'ram1'
14
15 WORDS ; The following addresses are 16 bits long
16 segment byte at stack_segment_start-stack_segment_end 'stack'
17
18 WORDS ; The following addresses are 16 bits long
19 segment byte at 4000-427F 'eeprom'
20
21 WORDS ; The following addresses are 16 bits long
22 segment byte at 8080-9FFF 'rom'
23
24 WORDS ; The following addresses are 16 bits long
25 segment byte at 8000-807F 'vecttbl'
26
27 END
28

```

([https://3.bp.blogspot.com/-9DMj1JfKHTw/W21cKWA7\\_-I/AAAAAAAAADXU/iFiwC\\_kHoV4uEy1FHSiJ6iDwsUVAvemJgCLcBGAs/s1600/stvd\\_12.png](https://3.bp.blogspot.com/-9DMj1JfKHTw/W21cKWA7_-I/AAAAAAAAADXU/iFiwC_kHoV4uEy1FHSiJ6iDwsUVAvemJgCLcBGAs/s1600/stvd_12.png))

I would like to draw your attention to the fact that the **assembler file in STVD begins with the string 'stm8 /' and ends with the string '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:
; 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, #ram_0_start
clear_ram0:
clr (X)
incw X
cpw X, #ram_0_end
jrle clear_ram0
#endif

#ifdef RAM1
; clear RAM1
ram1_start.w EQU $ ram1_segment_start
ram1_end.w EQU $ ram1_segment_end
ldw X, #ram_1_start
clear_ram1:
clr (X)
incw X
cpw X, #ram_1_end
jrle 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:
clr (X)
incw X
cpw X, #stack_end
jrle clear_stack

infinite_loop:
jra infinite_loop

interrupt NonHandledInterrupt
NonHandledInterrupt:
iret

segment 'vectit'
dc.l { $ 82000000 + main}
dc.l { $ 82000000 + NonHandledInterrupt} ; trap ; reset
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, which ends with a main loop. In the Reset handler, the value of the stack pointer is set and the RAM is cleared.

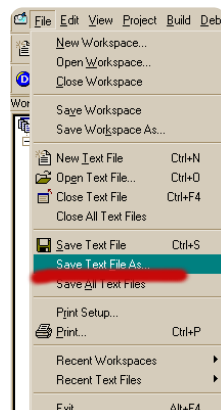
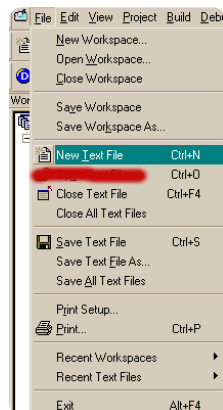
The first thing that catches your eye in the above listing is the original definition of the vector table in the form of an INT instruction opcode and a label address. If you try to replace this construction for example with the following line: "INT main", you will get an error: "Unknown opcode". It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

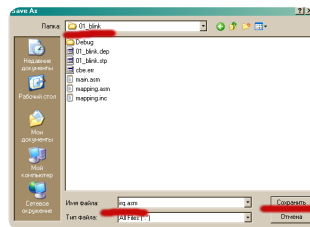


([https://4.bp.blogspot.com/-](https://4.bp.blogspot.com/-EN32UzWBA1A/W21cKsrs7NI/AAAAAAAAADXY/S6w4wMgzhC0/W21cKo0Fc9/AAAAAAAAADXC/zKlvrUfxRLozSDToOvjvg6y216_pApE8mdrjHOvRTIYH4sZnFK3BwCkLBGAs/s1600/stvd_14.png)

([https://2.bp.blogspot.com/-](https://2.bp.blogspot.com/-EN32UzWBA1A/W21cKsrs7NI/AAAAAAAAADXY/S6w4wMgzhC0/W21cKo0Fc9/AAAAAAAAADXC/zKlvrUfxRLozSDToOvjvg6y216_pApE8mdrjHOvRTIYH4sZnFK3BwCkLBGAs/s1600/stvd_14.png)

EN32UzWBA1A/W21cKsrs7NI/AAAAAAAAADXY/S6w4wMgzhC0/W21cKo0Fc9/AAAAAAAAADXC/zKlvrUfxRLozSDToOvjvg6y216\_pApE8mdrjHOvRTIYH4sZnFK3BwCkLBGAs/s1600/stvd\_14.png)

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

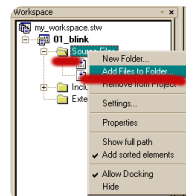


(<https://4.bp.blogspot.com/-WTAr-kUicil/W21cLAO9c->

I/AAAAAAADXg/J5tsdBxKZQMw5OgCPkGIOOHz6FIwGQJQ1dBCAs/1600/std\_15.png)

AYUvO7bEfig/W21cLe\_dycl/AAAAAADXk/PxLQiY\_JPT0erA3r8VOI-  
K5AiA8JO2smQCLcBGAs/s1600/stdv\_16.png)

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



Let's 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 $0, $ 0fff
ldw SP, $ X
jp main
jra reset

interrupt NonHandledInterrupt
NonHandledInterrupt.l
iret

segment 'vectir'
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

```

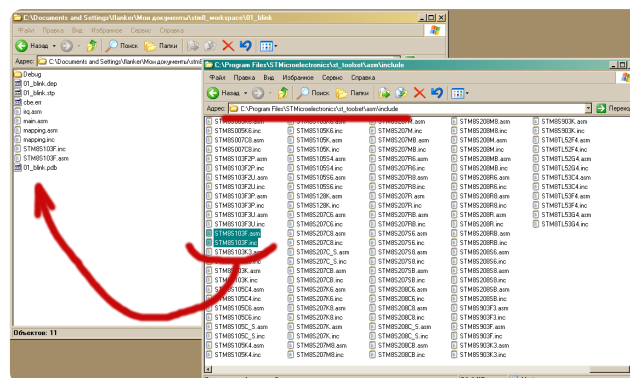
As a result, only the main loop will remain in main.asm:

```
stm8 /
    segment 'rom'
.main
    jp main
end
```

When compiling, it may give an error for an incorrect EOF, then it is 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, you need to find the STM8S103F.asm and STM8S103F.inc files in the directory from the STVD installation and copy them to the project folder:



(<https://2.bp.blogspot.com/->

MTNVF7hJk3U/W21cLx93aGI/AAAAAAAAADXw/ksF1Bay2acsAGHt0tUtkTrNtS E8Ha4ZwCLcBGAs/s1600/styld 19.png)

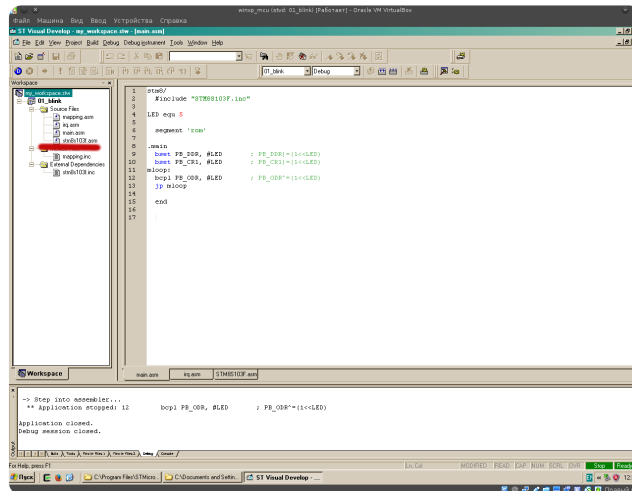
Then you need to add the STM8S103F.asm file to the project, and bring the content of main.asm to the following form:

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

LED equ 5

segment 'rom'

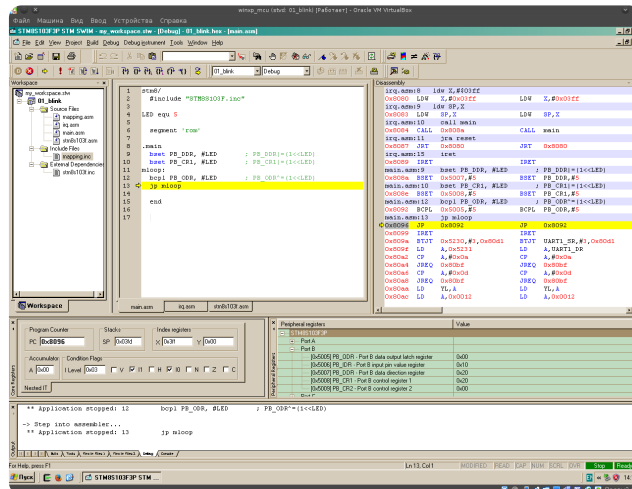
.main
bset PB_DDR, #LED      ; PB_DDR |= (1 << LED)
bset PB_CR1, #LED      ; PB_CR1 |= (1 << LED)
mloop :
    bcp1 PB_ODR, #LED    ; PB_ODR ^= (1 << LED)
    jp mloop
end
```



([https://1.bp.blogspot.com/-](https://1.bp.blogspot.com/-wGFze8e1fMQ/W21cMDmRaEI/AAAAAAAAADX0/SBf5lpG1RJ4GIWqDZ6QaCYIOZkBkWO9DQCLcBGAs/s1600/stdv_20.png)

[wGFze8e1fMQ/W21cMDmRaEI/AAAAAAAAADX0/SBf5lpG1RJ4GIWqDZ6QaCYIOZkBkWO9DQCLcBGAs/s1600/stdv\\_20.png](https://1.bp.blogspot.com/-wGFze8e1fMQ/W21cMDmRaEI/AAAAAAAAADX0/SBf5lpG1RJ4GIWqDZ6QaCYIOZkBkWO9DQCLcBGAs/s1600/stdv_20.png))

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



([https://2.bp.blogspot.com/-](https://2.bp.blogspot.com/-rtgkiDIHlg/W21cMX34YOI/AAAAAAAAADX4/SaUdUlpkTciS6XNFsySp3EcwH9Uq9sgCLcBGAs/s1600/stdv_21.png)

[rtgkiDIHlg/W21cMX34YOI/AAAAAAAAADX4/SaUdUlpkTciS6XNFsySp3EcwH9Uq9sgCLcBGAs/s1600/stdv\\_21.png](https://2.bp.blogspot.com/-rtgkiDIHlg/W21cMX34YOI/AAAAAAAAADX4/SaUdUlpkTciS6XNFsySp3EcwH9Uq9sgCLcBGAs/s1600/stdv_21.png))

#### 4. Adding a file with a subroutine to the project

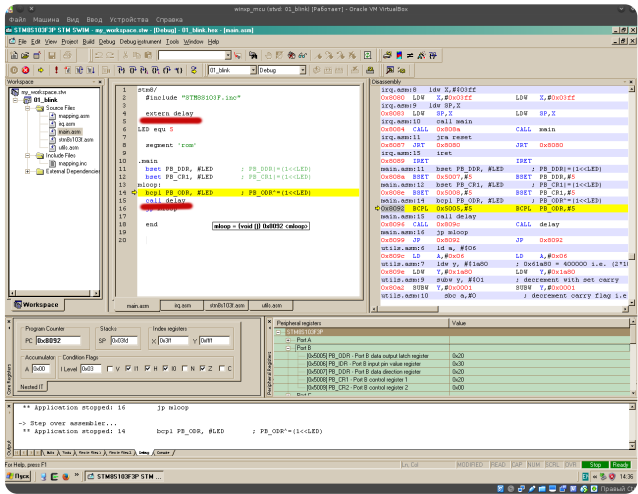
It remains to add a delay for the main loop for the program to work correctly. To do this, in the same way as the irq.asm file was added to the project, add the utils.asm file to it with the following content:

```
stm8 /
segment 'rom'

.delay :
    lda a, # $ 06
    ldw y, # $ 1a80                ; 0x61a80 = 400000 ie (2 * 10 ^ 6 MHz) / 5cycles
loop :
    subw y, # $ 01                ; decrement with set carry
    sbc a, # 0                    ; decrement carry flag ie 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 you can insert a subroutine call into the main loop: "call delay":





(https://4.bp.blogspot.com/-Dx0RpbJrD38/W21cMW5KtYI/AAAAAAAAADx8/Ee-CYotS4v8PGBfn8Iya4RJ1ZU1GJNmTQCLcBGAs/s1600/stdv\_22.png)

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

5. Basic information about the STVD assembler

The assembly language consists of assembly instructions, assembly directives, preprocessor directives, and a macro language. Previously, all this together was called a macroassembler, and technically, a macroassembler comes close to the HLU in terms of capabilities. At least, it was considered so before, when these JAVUs were many times 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]
```

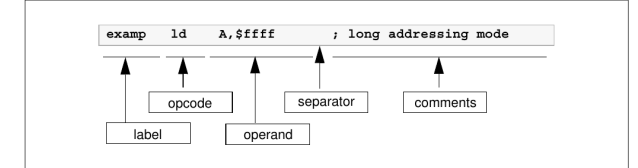
where <space> refers to either a SPACE (\$20) or a TAB (\$09) character.

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.

(https://3.bp.blogspot.com/-kJVfUWTJw9l/W21cMoWzPul/AAAAAAAAADYA/buo\_PeXw3rQ0PJGSOZFNi7vDxySNhKTNgCLcBGAs/s1600/stdv\_23.png)

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

Note that even if a label, **assembly instruction or directive is omitted, a SPACE or TABULATION MUST be preceded by a SPACEBAR or TABULATION. AT THE BEGINNING OF A LINE IS ONLY A LABEL!** An exception to this rule is the "stm8 /" directive at the beginning of the source code.

Соответственно, если написать "segment 'rom'" или "#include "STM8S103F.inc"" сначала строки то, при компиляции выдаст ошибку. Видимо разработчики STVD были неравнодушны к питону ;)

6. Формат числовых констант

По умолчанию, в STVD установлен мотороловский формат числовых констант, когда десятичное число пишется как есть, перед шестнадцатеричным числом ставится значок доллара, перед двоичным - знак процента и перед восьмеричным - тильда.

С помощью директив: MOTOROLA, INTEL, TEXAS, ZILOG можно менять формат числовой константы:

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	\$

(https://4.bp.blogspot.com/-jxjsbNYi8ww/W21cMwpgI7I/AAAAAAAAADYE/S8VqjxmchgR\_vk1gLBxe4YAPpVR9FWACLcBGAs/s1600/stdv\_24.png)

Новый формат будет применим к тексту программы после директивы.

7. Формат метки

Метка обязательно должна начинаться с начала строки, и может содержать в себе: заглавные и строчные буквы английского алфавита, цифры и символ подчёркивания. Начинаться метка должна с буквы или символа подчёркивания, т.е. никак не с цифры. Метка может содержать до 30 символов.

Метка может заканчиваться символом двоеточия или не содержать его вообще. Символ двоеточия игнорируется.

Метка может содержать суффикс вида: label[b|w|l], который состоит из точки и добавочных букв: b, w, l.

С помощью суффиксов: b, w, l, метка может быть одно-, двух-, или четырёхбайтной. Т.о. метка может быть использована для одно-,двух или трехбайтной адресации.

В качестве примера можно привести такой исходник:

Который будет скомпилирован в следующий машинный код:

```
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:
jpf mloop

const:
dc.b $aa
end
```

888a:	b6 10	ld A,0x10
;0x10		
888c:	c6 01 00	ld A,0x010
0 ;0x100		
888f:	c6 80 96	ld A,0x809
6 ;0x8096		
8092:	ac 00 80 92	jpf 0x0080
92 ;0x8092		
8096:	aa 00	or

Здесь задаётся две переменные: var1 в нулевой странице ОЗУ, var2 в первой странице ОЗУ, и константа const в области флеш-памяти. При обращении к первой переменной используется короткая адресация (shortmem), при обращении к второй переменной и к константе используется длинная адресация (longmem), и инструкция jpf осуществляет переход метке с расширенной адресацией (extmem).

По умолчанию размер метки равен двум байтам (одному слову), но он может быть изменён директивами: BYTES, WORDS, LONGS.

Метка может быть абсолютной и относительной. Относительную метку вычисляет компоновщик на этапе компиляции прошивки.

По умолчанию, все метки локальные. С помощью директив PUBLIC и EXTERN можно обмениваться метками между отдельными модулями программы. Директива PUBLIC делает метку видимой компоновщику. Директива EXTERN делает внешнюю метку видимой данному модулю. В качестве альтернативы директиве PUBLIC, может выступать точка в начале метки.

8. Сегментация

Сегментация позволяет управлять из проекта тем, что в GCC делает скрипт компоновщика, т.е. с помощью директивы сегментации можно указывать компоновщику, где должны располагаться тот или иной код или данные. Один программный модуль (файл) может содержать до 128 сегментов.

Сегмент устанавливается с помощью директивы сегментации. Ее формат представлен ниже:

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

Директива может быть знакома по NASM, только там, она управляет размещением кода в объектном файле, а здесь указывает физическую память.

Вначале идет необязательный параметр - имя (name). Он может иметь длину до 12 символов. Сегменты одного класса, но с разными именами группируются компоновщиком друг за другом в порядке определения.

Параметр выравнивания - "align", может принимать следующие значения:

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

zHRtsfOxNBM/W21cNZzNFKI/AAAAAAAAADYM/7aoqWFA9fn8FLBUynOQp4qQYIzID0gJQLcBGAs/s1600/stdv\_26.png

Выше я приводил пример, как с помощью сегментов задавать переменные в области оперативной памяти. Ниже приведены еще примеры из фирменного руководства:

```
FILE1:
st7/
    BYTES
    segment byte at: 80-FF 'RAM0'
counter.b ds.b ; loop counter
address.b ds.w ; address storage
stack ds.b 15 ; stack allocation
        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
```

IBM7AQ/W21cNOzBjll/AAAAAAAAADYI/VvPrLGs680YIqBAPklXatgn7Bo3p2QpsgCLcBGAs/s1600/stdv\_25.png)

Здесь метке counter соответствует адрес 0x80, метке address - 0x81, stack - 0x92. Во втором файле, метка serialtemp имеет адрес 0x93, а serialcou - 0x94. Т.к. оба файла пишут в сегмент "eprom", то второй файл пишет следом за первым.

Следующий параметр директивы segment - <combine> указывает компоновщику где следует располагать сегмент. Имеется три варианта для этого параметра:

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.

(https://4.bp.blogspot.com/-

StfMSo0tDiU/W21cNsD3Oql/AAAAAAAAADYQ/hSDbWP\_sJgkUDgblBwWIB-CV-4VkdFpAClCBGAs/s1600/stdv\_27.png)

**Вариант at- ДОЛЖЕН использоваться при объявлении класса, и использоваться ТОЛЬКО ОДИН РАЗ.**

Также в случае at должен обязательно задан начальный адрес сегмента, и опционально последний адрес сегмента или его размер. Все адреса пишутся в ШЕСТНАДЦАТЕРИЧНОМ виде КАК ЕСТЬ без префиксов.

Вариант common позволяет определить общие области данных. Области этого типа с одинаковым именем класса, будут иметь одинаковый стартовый адрес. Это позволяет использовать такие сегменты для обмена данными.

Рассмотрим такой пример:

```
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
```

(https://1.bp.blogspot.com/-997GISG0Wuc/W21cN-

7iKRl/AAAAAAAAADYU/Lh\_xVXeR3\_EoWMC7YJse-bGjEDXtc1f5wCLcBGAs/s1600/stdv\_28.png)

Здесь метки lab1 и lab2 будут иметь адрес 0x12, lab3 и lab4 будут иметь адрес 0x1a, a lab5 - 0x1e.

Нельзя комбинировать вместе common и at- сегменты под одинаковыми именами. Следующий пример демонстрирует ошибку:

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

(https://4.bp.blogspot.com/-75V96IKMGmo/W21cODtgH0l/AAAAAAAAADYY/WmnP9Qlcppsi3NX3UyzGiBLLsBORqdi0QCLcBGAs/s1600/stdv\_29.png)

Последний параметр директивы segment - cod, позволяет управлять получаемым при линковке выходным файлом, в котором будут размещены данные сегмента. Если параметр cod опущен, то все скомпируется в единый default.cod. Если же в качестве параметра cod задать число от нуля до девяти, то компоновщик все сегменты данного класса разместит в файле prog\_x.cod, где x - номер кода. Это может быть полезно, например при формировании разных еергом для разных устройств. Например:

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

kOpzw/W21cORPcfHl/AAAAAAAAADYc/-

v5wr2LQGK0dChu9RhPcmkTBjfh4osUHAClCBGAs/s1600/stdv\_30.png)

9. Основные директивы ассемблера

Полный перечень директив с подробным описанием изложен в Руководство пользователя по ассемблеру и компоновщику STM8: 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:content/translator приложение A. Я бы хотел упомянуть наиболее востребованные директивы.

**EQU** - директива подмены или соответствия. Имеется наверно во всех ассемблерных языках. Формат:

метка	EQU выражение
-------	---------------

При компиляции программы, метка заменяется своим выражением. Пример:

var1	equ 5
------	-------

**#define** - директива похожая на EQU, но имеющая существенное отличие. В EQU метке присваивается число, его размерность: байт, слово, двойное слово - определяется в момент присваивания. В случае использования **#define**, метке присваивается строка которая уже при компиляции преобразуется в число, и его размерность определяется в самой программе. Формат:

#define <псевдоним> <строка>
------------------------------

Пример использования:

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

**CEQU** - директива похожая на EQU но позволяет менять свое содержимое:

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

Используется в сочетании с директивами REPEAT и UNTIL.

Директивы: **DC.B, DC.W, DS.L** - позволяют записать константу, или массив. Также возможна запись строки ASCII.

Директивы: **DS.B, DS.W, DS.L** - позволяют зарезервировать место. Через запятую указывается количество резервируемых ячеек.

Директива **STRING** позволяет зарезервировать ASCII строку. Является синонимом директивы DC.B. Примеры использования:

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

Директивы предназначенные для написания макросов хотелось бы рассмотреть в соответствующей главе.

## 10. Копирование кода в ОЗУ и выполнение его оттуда

Выполнение кода из ОЗУ имеет смысл в STM8L - серии, там есть режим энергосбережения WFE который позволяет работать с периферией без прерываний, что позволяет исключить использование флеш-памяти при работе главного цикла. Отказ от использования флеш-памяти позволяет снизить энергопотребление, но при этом не следует забывать, что программа из оперативной памяти выполняется дольше нежели с флеша. Сейчас мы в этом убедимся, но перед тем как запустить программу из ОЗУ, ее нужно туда скопировать. При этом все абсолютные адреса надо будет как-то проиндексировать.

В Руководство пользователя по ассемблеру и компоновщику STM8: 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:content/translation](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:content/translation))

описывается механизм написания кода специально предназначенный для копирования в ОЗУ. Для этого декларируется два сегмента, где один будет использоваться для выполнения, а другой для хранения кода. Например:

```
segment byte at: 0 'code'
segment byte at: 8000 'ram' (https://1.bp.blogspot.com/-e-
segment 'ram>code'
label1:nop
nHzPnBWSI/W21cOrtsa-
//AAAAAAAAADYg/iKq_C3SnkJAHaB4m0pZsFY4YQUSqda1WACLcBGAs/s1600/stdv_31.png)
```

здесь код начинающийся с метки label1 будет сохранен в 'code' сегменте, но все метки будут пересчитаны относительно 'ram' сегмента. Кроме того, адресное пространство 'ram' сегмента будет зарезервировано под этот код.

Все это хорошо, но вот скомпилировать у меня этот пример никак не получалось. Впрочем, создать перемещаемый код вручную не так сложно. Для эксперимента возьмем пример базового проекта из первой части. Из него потребуется удалить файл utils.asm, и тогда main.asm будет выглядеть так:

```

stm8/      #include "STM8S103F.inc"

LED equ 5
offset equ $4000

        segment 'rom'
.main:
        clrw x

lp:
        ld a,x1
        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, #506
        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
        jrnc loop
        ;-----
        jra mloop               ; relative label
        jp [mloop-offset+$100]  ; absolute label

end

```

Здесь сегмент 'eeprom' содержит программу размещаемую в оперативной памяти, а сегмент 'rom' содержит загрузчик. Программа размещается в первой странице т.е. с адреса 0x100, тогда как нулевая страница отведена на переменные. В программе все относительные метки оставлены как есть, в то время как абсолютные записываются со сдвижением: адрес=(начальный адрес eeprom)+(адрес начала первой страницы ОЗУ).

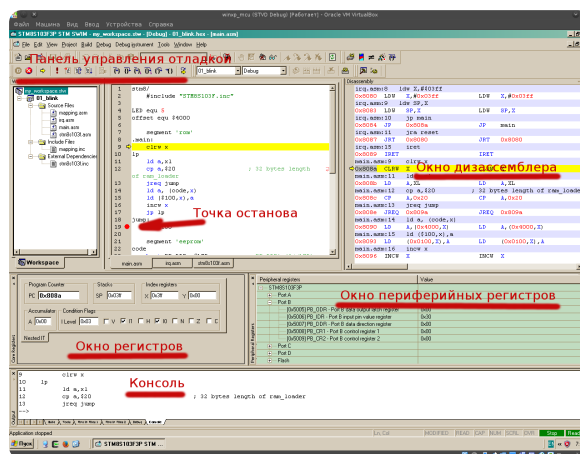
После прошивки микроконтроллера будет видно, что светодиод имеет полупериод мигания гораздо больше чем одна секунда. Определяя "на глаз", у меня получилось где-то трёхкратное снижение быстродействия, о чем собственно и говорилось в документации: 3-х уровневый конвейер STM8: перевод глав 3, 4, 5 руководства по программированию микроконтроллеров STM8 (PM00044) (/2018/stm8\_manual/#10)

Чтение инструкции из ОЗУ походит на чтение из ПЗУ. Однако, вследствие того, что шина ОЗУ является лишь 8-битной, потребуется 4 последовательных операции чтения, чтобы заполнить одно Fx слово. Вследствие этого код из ОЗУ выполняется медленнее, нежели с флеш-памяти.

В дальнейшем, эту программу я буду использовать для демонстрации процесса отладки в STVP.

## 11. Интерфейс отладки STVD

В качестве отладчика в STVD используется мощнейший GDB, полные его возможности доступны через вкладку "Консоль" в окне "Output Window". STVD выступает в качестве фроненда к GDB:



(<https://3.bp.blogspot.com/-->

HDGm C57is/W21cO6HzOrl/AAAAAAYDYk/iclXANWqFOwiU8ntV8akluObFsQhosEGACLCBGAs/s1600/stdv 32.png)

У меня пока не очень большой опыт использования STVD, но из всех возможностей GDB через консоль я печатал только дампы памяти. Все остальное я делал через графический интерфейс, Работа с ним экономит много времени.

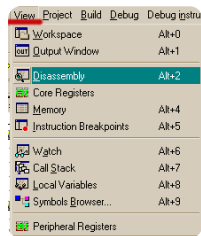
Итак, после после запуска отладки, становится доступна панель управления отладкой:



([https://1.bp.blogspot.com/-1JKHlnS1xNs/W21cPlzMmeI/AAAAAAAAADYs/KnhTmyLYXRM8wp6mPACBask\\_Ep0X040EACLcBGAs/s1600/std\\_33.png](https://1.bp.blogspot.com/-1JKHlnS1xNs/W21cPlzMmeI/AAAAAAAAADYs/KnhTmyLYXRM8wp6mPACBask_Ep0X040EACLcBGAs/s1600/std_33.png))

Она реализует режим трассировки и содержит операции: начать отладку, завершить отладку, команда GDB - Run, команда GDB - Continue, команда GDB - next, команда GDB - nexti, команда GDB - step, команда GDB - stepi и пр. У всех иконок есть всплывающие подсказки, раскрывающие их функции.

Открыть то или иное окно можно через меню->View:



(<https://2.bp.blogspot.com/-uzSYsDBw->

y8/W21cPAm14I/AAAAAAADYo/WBroH6Ba\_Ag6fWfTz32xV6mmmkMYGa1VwCLcBGAs/s1600/stdv\_34.png)

Все окна стекабельны, закрываются крестиком. Настройки окон сохраняются в workspace.

Самые полезные на мой взгляд следующие окна:

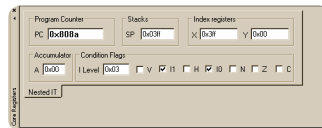
1. Окно дизассемблера. Наиболее полезно оно наверное при отладке Си программы, позволяет увидеть, в какой код преобразовалась та или иная программная конструкция. Для ассемблера это окно полезно тем, что позволяет увидеть адреса и размерность меток, а также числовой адрес порта ввода-вывода. Через "Display Options" можно настроить отображаемые столбцы. Опция Refresh нужна когда вводишь отладочные команды через консоль gdb. Потом чтобы увидеть изменения приходится обновлять.



(<https://3.bp.blogspot.com/->

Y0gnw758SF1/W21cPdOicll/AAAAAAADYw/wqdyOzrm-w8xLacvXOY-PVLSdgsjy92wCLcBGAs/s1600/stdv\_35.png)

2. Следующее окно "Core Registers", показывает содержимое регистров микроконтроллера и флагов состояния. Прямо в окне можно менять значение регистров, снимать и устанавливать нужные флаги.



([https://3.bp.blogspot.com/-b4Be\\_7jr-](https://3.bp.blogspot.com/-b4Be_7jr-)

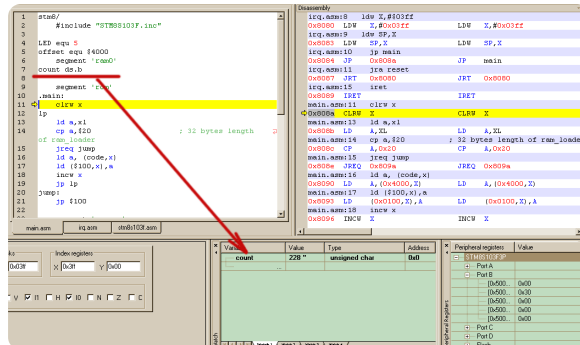
aE/W21cP7XmIXI/AAAAAAADY0/MR9c0VCr-QlIm-mwIVcUgCtVdkf-Ff\_cawCLcBGAs/s1600/stdv\_36.png)

3. Окно периферийных регистров - это древовидный список сгруппированный по аппаратным модулям. Здесь также можно менять содержимое PBB:



([https://3.bp.blogspot.com/-bsMRxR4YdNo/W21cQD-Bp-I/AAAAAAADY4/Ht7R0hemIYgmrQgHptOZYgY95C\\_PNOoDgCLcBGAs/s1600/stdv\\_37.png](https://3.bp.blogspot.com/-bsMRxR4YdNo/W21cQD-Bp-I/AAAAAAADY4/Ht7R0hemIYgmrQgHptOZYgY95C_PNOoDgCLcBGAs/s1600/stdv_37.png))

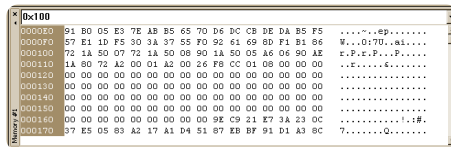
4. Следующее полезное окошко - окно переменных. Переменную можно ввести по имени или по адресу. Можно менять значение переменной.



(<https://3.bp.blogspot.com/->

XvqW2RTM\_z8/W21cQRJzZDI/AAAAAAADY8/xtCJ00ZkREM0oqNOKRZv8XAWurDa7cQPQCLcBGAs/s1600/stdv\_38.png)

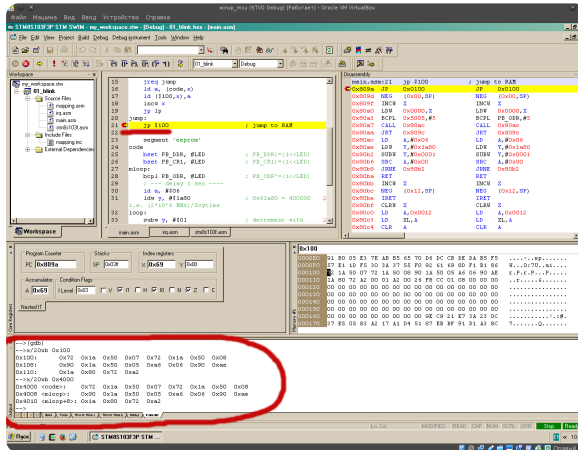
5. И последнее окно - окно с дампом памяти. Лично мне как-то проще вводить в консоли gdb команду: "x/10xb", но данное окно тоже может быть полезно. Хотя бы тем, что там в интерактивном режиме можно менять содержимое ячеек памяти.



([https://2.bp.blogspot.com/-oH2-57MhVnQ/W21cQaUJHml/AAAAAAAAADZA/0x9LvSR29KuU9wcJcNPK8T4JBKvN5EfWCLcBGAs/s1600/stdv\\_39.png](https://2.bp.blogspot.com/-oH2-57MhVnQ/W21cQaUJHml/AAAAAAAAADZA/0x9LvSR29KuU9wcJcNPK8T4JBKvN5EfWCLcBGAs/s1600/stdv_39.png))

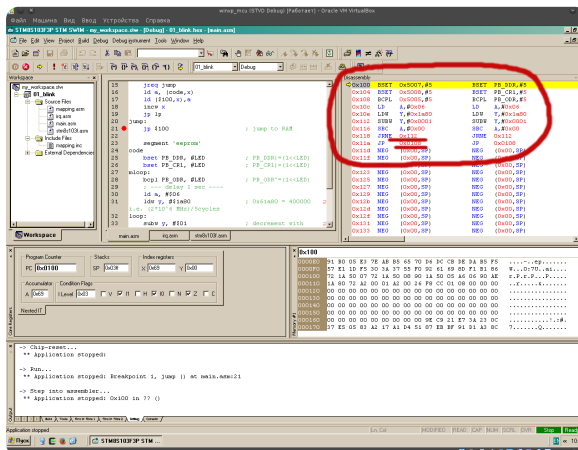
## 12. Процесс отладки в STVD

Для отладки алгоритма, поставим точку остановки на инструкции `jr $100`, т.е. после завершения операции копирования из `eeorgm` в `ram`, и перед передачей управлению скопированному коду. Жмём иконку Run (выглядит как восклицательный знак) и после остановки свернем оба сегмента: `ram1` и `eeorgm`:



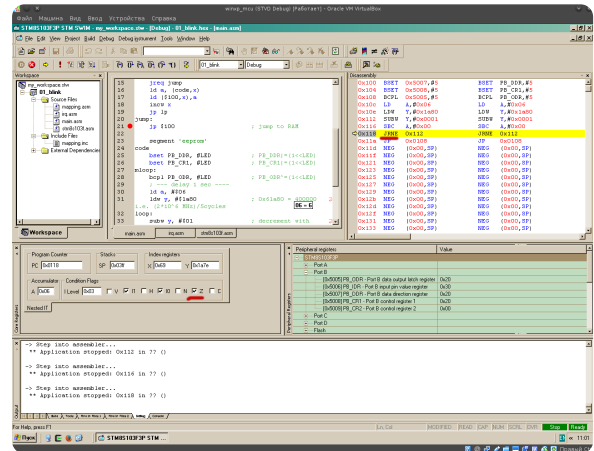
([https://1.bp.blogspot.com/-0Vq6c7oPqK4/W21cQrSgbHI/AAAAAAAAADZE/rONWmqGxOQ00he1HvT8J460a-jiacsVJACLcBGAs/s1600/stdv\\_40.png](https://1.bp.blogspot.com/-0Vq6c7oPqK4/W21cQrSgbHI/AAAAAAAAADZE/rONWmqGxOQ00he1HvT8J460a-jiacsVJACLcBGAs/s1600/stdv_40.png))

Если код скопировался успешно, делаем прыжок на адрес `0x100`, после чего в окне дизассемблера проверяем скопированный код. Особое внимание нужно обратить на адреса переходов.



([https://4.bp.blogspot.com/-nbs\\_4hvKTQA/W21cQqU7oul/AAAAAAAAADZI/99ece2-Qu9EQGvUa9glw-qfoz40oixJwCLcBGAs/s1600/stdv\\_41.png](https://4.bp.blogspot.com/-nbs_4hvKTQA/W21cQqU7oul/AAAAAAAAADZI/99ece2-Qu9EQGvUa9glw-qfoz40oixJwCLcBGAs/s1600/stdv_41.png))

Если алгоритм скопирован успешно, можно приступать к обычной отладке алгоритма. На инструкции условных переходов бывает удобно выставлять нужные флаги, чтобы без проволочек проходить по тем или иным веткам алгоритма.



([https://1.bp.blogspot.com/-O3sZ03VqaOM/W21cQ3HepLI/AAAAAAAAADZM/EK88yq-NTLcSR8BD9pX8vIBRG05IHjEkgCLcBGA/s1600/stdv\\_42.png](https://1.bp.blogspot.com/-O3sZ03VqaOM/W21cQ3HepLI/AAAAAAAAADZM/EK88yq-NTLcSR8BD9pX8vIBRG05IHjEkgCLcBGA/s1600/stdv_42.png))

По-моему, ничего сложного.

13. Введение в макроассемблер STVD

Я не спец по макроассемблеру, прямо говоря, я им никогда не пользовался и поэтому имею о нем только общее представление. Мне показалось, что это хороший повод заполнить досадный пробел в образовании, тем не менее, к нижеизложенному тексту не следует относиться как какому-то руководству.

Макроассемблер позволяет сократить рутину при написании ассемблерных программ, а также уменьшить ваше время на отладку оных. Принцип заключается в объединении группы ассемблерных инструкций в единый макрос которому, навроде функции или подпрограмме, можно задавать параметры, и управлять с помощью препроцессора. Однажды написав и отладив такой макрос вам не придётся делать это снова. Использование макросов может приблизить читаемость программы к языкам высокого уровня навроде бейсика. При этом у вас по прежнему будет вся мощь ассемблера, и вы не будете ограничены рамками какого-либо языка программирования.

Главной проблемой макросов считается трудоёмкость их отладки на этапе работы препроцессора. Когда вам выдаст ошибку на макрос, вам выдаст ошибку на ту строку, где он вызывается, а не на проблемную конструкцию в теле макроса. Справедливости ради должен сказать, что в STVD имеется директива %OUT которая выводит какую либо строку в лог компиляции, но вот вывести число через нее не получится.

Директивы макроассемблера и формат задания макроса описан в руководстве: Руководство пользователя по ассемблеру и компоновщику STM8: 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:content/translation](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:content/translation)) Я не буду повторять написанное там, вместо этого я приведу несколько примеров использования макросов.

Для начала совсем простой случай, попробуем заменить на макрос строку перехода по абсолютному адресу:

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

Здесь необходимо вычисление адреса записать в макрос, чтобы не писать это каждый раз. Макрос будет таким:

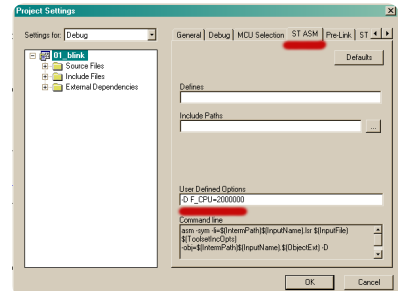
```
jp_ram MACRO adr
    jp {adr-offset+$100}
MEND
```

Тогда использование макроса сведётся к строке:

```
jp_ram mloop
```

14. Макрос задержки delay\_ms

Теперь пример посложнее, постараемся повторить широко известный макрос \_delay\_ms который имеется в gcc-avr и используется довольно часто. Для этого, сначала в настройках проекта нужно будет с помощью именованной константы указать частоту F\_CPU:

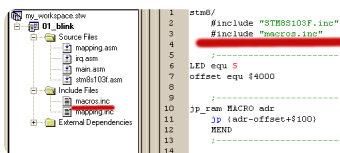


(<https://2.bp.blogspot.com/->

N48FFZe06F0/W21cRMPjPol/AAAAAAAAADZQ/NKZLFATzN8YBn7QZxVDXuiIH7hGnjMSQCLcBGA/s1600/stdv\_43.png)



Кроме этого, нам понадобится отдельный файл `macros.inc`, в который мы будем скидывать макросы. Этот файл потом нужно будет включить через директиву `#include`:



([https://2.bp.blogspot.com/-](https://2.bp.blogspot.com/-TuE6OTPooNI/W21cRReM02I/AAAAAAADZU/Ir3PhI8DtI8I4rv_OyaeD4PnVTHi_WprQCLcBGAs/s1600/stdvd_44.png)

[TuE6OTPooNI/W21cRReM02I/AAAAAAADZU/Ir3PhI8DtI8I4rv\\_OyaeD4PnVTHi\\_WprQCLcBGAs/s1600/stdvd\\_44.png](https://2.bp.blogspot.com/-TuE6OTPooNI/W21cRReM02I/AAAAAAADZU/Ir3PhI8DtI8I4rv_OyaeD4PnVTHi_WprQCLcBGAs/s1600/stdvd_44.png))

Макрос у меня получился таким:

```
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
jne loop
MEND
```

Тогда код в модуле `main.asm` примет следующий вид:

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

LED equ 5
offset equ $4000

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

segment 'rom'
; --- GPIO Setup -----
bset PB_DDR, #LED ; PB_DDR|=(1<<LED)
bset PB_CR1, #LED ; PB_CR1|=(1<<LED)
; --- Copy to RAM -----
clr x
lp
ld a, x1
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 'eprom'
code
mloop:
bcpl PB_ODR, #LED ; PB_ODR^=(1<<LED)
delay_ms 1000
jp_ram mloop
end
```

## 15. Макрос условного оператора сравнения регистра с числом

Как я уже говорил, на макросах можно сделать подобие операторов в ЯВУ. Например, в копирушке кода из `еергот` в `гат` имеется сравнение числа с регистром. Можно написать макрос который сравнивает регистр с числом и осуществляет переход по метке в случае совпадения.

У меня получилось два макроса. Один для 16-битных регистров, другой для 8-битных:

```
if_reg_eq8 MACRO reg value label
#IFDEF reg yh
ld a, y
#ENDIF
#IFDEF reg yl
ld a, yl
#ENDIF
#IFDEF reg xh
ld a, xh
#ENDIF
#IFDEF reg xl
ld a, xl
#ENDIF
cp a, value
jreq label
MEND
;-----
if_reg_eq16 MACRO reg value label
#IFDEF 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` в таком случае принимает вид:

```
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
clr w x
lp
if_reg_eq8 x1 $20 jump
ld a, (code,x)
ld ($100,x),a
incw x
jra lp
jump:
jp $100                    ; go to RAM
;-----
segment 'eeprom'
code:
mloop:
bcl pb_odr, #LED          ; PB_ODR^=(1<<LED)
delay_ms 1000
jp_ram mloop
end
```

Аналогично можно написать макросы для сравнения числа с ячейкой памяти, или сравнения по признаку больше, меньше, меньше или равно и т.д.

16. Задержка длительностью 1 мкс на инструкции условного перехода

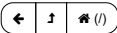
Если организовать задержку по таймеру на 1 мс и более не составляет проблем, то с задержкой на 1мкс не все так просто. Обработчики прерываний таймера будут не успевать обрабатывать поступающие прерывания. Единственным выходом будет сделать задержку на инструкциях. Но считать циклы при работающем конвейере, дело неблагодарное. Количество циклов будет зависеть от предыдущих инструкций и от способа адресации. Кроме того, я наткнулся на информацию, что на разных тактовых частотах одни и те же инструкции будут выполняться за разное количество циклов. И вроде бы я сталкивался с этим на практике. В качестве способа решения этой проблемы рекомендуют заблокировать работу конвейера. Делается это с помощью инструкций выполняющих сброс очереди конвейера - FLUSH. Такими инструкциями являются инструкции условного перехода. Т.о. задержку можно организовать последовательном выполнении операций условного перехода с нулевым смещением. Одна такая инструкция должна выполняться за 2 цикла ВСЕГДА.

Записать последовательность инструкций помогут директивы REPEAT и UNTIL. К сожалению их нельзя использовать в макросах поэтому их использование выглядит так:

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

Здесь предполагается, что fCPU = 2MHz и т.к. инструкция условного перехода с случае перехода выполняется за два такта, то на выполнении каждой инструкции JRC уходит 1 мкс. Директивы REPEAT/UNTIL вставляют в текст программы пять таких инструкций, следовательно формируется задержка длительностью 5мкс.

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




share:

- (https://vk.com/share.php?url=http%3A%2F%2Fwww.count-zero.ru%2F2018%2Fstm8\_stvd\_asm%2F&title=STM8%20%2B%20STVD%20%2B%20ASSEMBLER%3A%20)
- (https://www.facebook.com/sharer.php?src=sp&u=http%3A%2F%2Fwww.count-zero.ru%2F2018%2Fstm8\_stvd\_asm%2F&title=STM8%20%2B%20STVD%20%2B%20ASSEMBLER%3A%20)
- (https://connect.ok.ru/offer?url=http%3A%2F%2Fwww.count-zero.ru%2F2018%2Fstm8\_stvd\_asm%2F&title=STM8%20%2B%20STVD%20%2B%20ASSEMBLER%3A%20)
- (https://connect.mail.ru/share?url=http%3A%2F%2Fwww.count-zero.ru%2F2018%2Fstm8\_stvd\_asm%2F&title=STM8%20%2B%20STVD%20%2B%20ASSEMBLER%3A%20)
- (https://twitter.com/intent/tweet?text=STM8%20%2B%20STVD%20%2B%20ASSEMBLER%3A%20%D0%91%D1%8B%D1%81%D1%82%D1%80%D1%8B%D0%B9%20%D1%)

ТАКЖЕ НА COUNT-ZERO

<div>Настройка: Eclipse, SW4STM32, ...</div> <div>2 года назад · 10 комментарий...</div> <div>Настройка: Eclipse, SW4STM32, STM32CubeIDE и Qt ...</div>	<div>STM8 + ASSEMBLER: ...</div> <div>2 года назад · 5 комментарий...</div> <div>STM8 + ASSEMBLER: Драйвер FM-приемника RDA5807m для ...</div>	<div>ATtiny13a: использование ...</div> <div>4 года назад · 8 комментарий...</div> <div>ATtiny13a: использование ассемблера GNU-AS в программах на Си ...</div>	<div>Связь /</div> <div>4 года на</div> <div>Связь дж микро примере</div>
---	--	---	---

3 Комментариев count-zero  Политика конфиденциальности Disqus Войти ▾

 Рекомендовать  Твитнуть  Поделиться Лучшее в начале ▾



Присоединиться к обсуждению...

войти с помощью

или через DISQUS 

Имя

**Prism** • 2 месяца назад

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

^ | ▾ • Ответить • Поделиться ▾

**Василий** • 2 года 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?

^ | ▾ • Ответить • Поделиться ▾

**flanker** Модератор → Василий • 2 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.

^ | ▾ • Ответить • Поделиться ▾