Programming industrial embedded systems, 2014./2015.
Laboratory exercise 1

Contents

1. Required hardware, software and additional literature

For laboratory exercises you will need the following:

Hardware:

- STM32F4DISCOVERY development kit,
- USB-mini cable,
- UART/RS232 serial interface (with additional RS232/USB converter for computers without RS-232 port) or UART/USB bridge[1]

Software:

- Keil MDK-ARM uVision development environment[2] (version 4 (uV4) or version 5 (uV5))
  - integrated development environment (IDE) for ARM microcontrollers
- STM32F4 Standard Peripheral Library[3] (STSW-STM32065)
  - high-level library (hardware abstraction layer, HAL) for accessing on-chip microcontroller resources and peripherals

Additional literature:

- uVision 5 MDK getting started manual (*Getting Started - uVision 5 MDK.pdf*)
- STM32F4DISCOVERY board user manual (*UM1472 - Discovery kit for STM32F407.pdf*)
- STM32F4DISCOVERY board training materials (*STMicroelectronics Cortex-M4 Training STM32F407.pdf*)
- STM32F4 microcontroller family reference manual (*STM32F4 Reference manual.pdf*)
- STM32F4 microcontroller family programming manual (*STM32F3 and STM32F4 Series Cortex-M4 programming manual.pdf*)
- STM32F407VG microcontroller datasheet (*STM32F407VG.pdf*)
- STM32F2xx Standard Peripheral Library reference (*Description of STM32F2xx Standard Peripheral Library.pdf*)
- Keil application note 230 „*STM32F4 Lab for MDK v5.1*“ (*apnt_230.pdf*) + source code with example projects (*apnt_230.zip*)

Additional literature for lab exercises is packed conveniently in the archive *PIES_2014_Lab1_Pack.rar*, available from FER web page http://www.fer.unizg.hr/predmet/ppius, under the section „Laboratory exercises“. This lab guideline will address above references at some points and it is advisable to read referenced parts of the literature whenever further clarifications and instructions are needed.

[1] Needed for the part of lab related to serial communication with computer.
[2] Available from www.keil.com. Free version has 32kB limit on compiled code size (sufficient for lab exercises).
[3] Available from http://www.st.com/web/en/catalog/tools/PF257901.

2. Development environment installation

2.1. Keil MDK-ARM uVision installation

Laboratory exercises can be made either in Keil uV4 or Keil uV5 develpoment environments. Since the current official revision of Keil MDK-ARM uVision IDE is rev. 5 (uV5), these instructions will assume that you use uV5 IDE, although the uV4 can be used in a similar way, with some minor differences in procedures. To install Keil uV5 MDK you need to:

1. Obtain Keil MDK-ARM uVision installation media from www.keil.com (free edition)
2. Install Keil MDK-ARM uVision following the installation program instructions (we shall assume that you chose target installation folder *C:\Keil_v5*)

Two executables are important:

- *C:\Keil_v5\UV4\UV4.exe* - development environment executable (desktop shortcut is pointing to this file),
- *C:\Keil_v5\UV4\PackInstaller.exe* (*pack installer* utility that is needed to install additional software support and libraries for each microcontroller family).

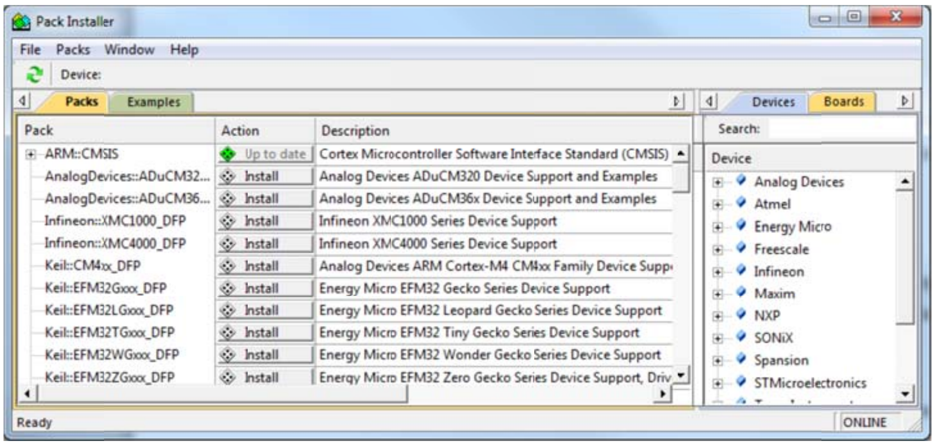
Note 1: Although the Keil MDK-ARM uVision is **version 5**, the executable is still named **uV4.exe** for some reason.

Note 2: Earlier revisions of Keil MDK-ARM uVision (4.xx and lower) **did not include** *PackInstaller* utility simply because they were already bundled with support for various microcontrollers and development boards out-of-the-box. With uV5 this approach changed and you have to add the support for at least one microcontroller family *after* the installation of Keil MDK-ARM uVision IDE, using *PackInstaller* utility. This utility is executed automatically at the end on uVision installation but it can be cancelled and re-run anytime after installation.

Additional information about Keil MDK-ARM uVision installation can be found in document „*Getting Started - uVision 5 MDK.pdf*“.

2.2. Adding support for STM32F4 family through the *Pack Installer* utility

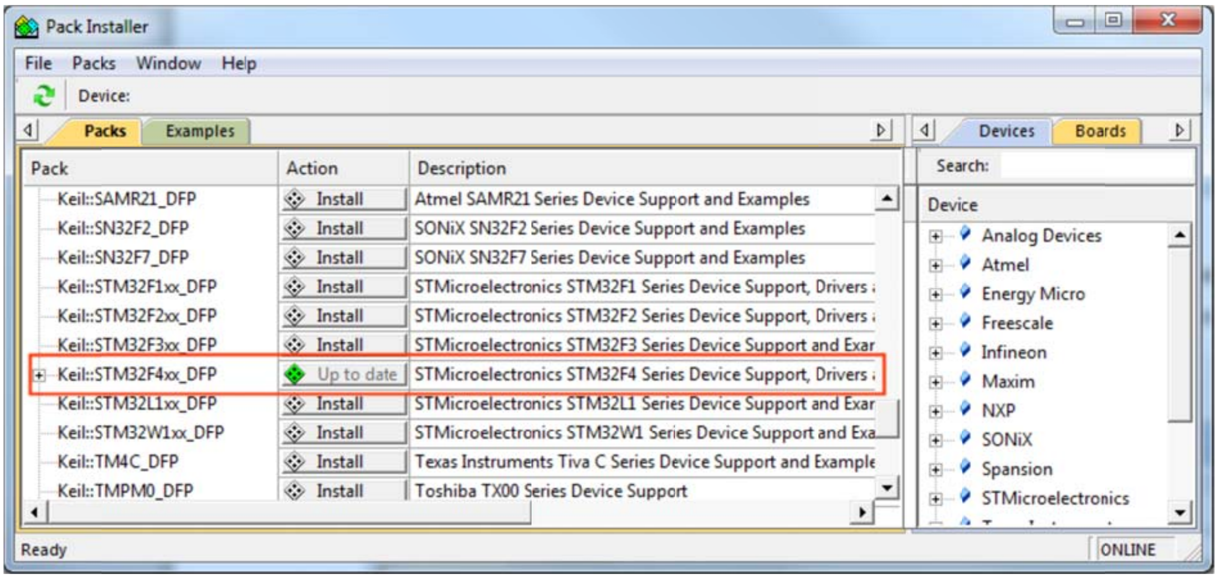
After the installation of Keil MDK-ARM uVision, the *Pack Installer* is automatically executed. It allows installation of additional software support (libraries, headers etc.), targeting specific silicon vendor ARM microcontroller family (so called "*Device Family Pack*" (DFP) packages). The *Pack Installer* main window in shown below:



Pack	Action	Description
ARM::CMSIS	Up to date	Cortex Microcontroller Software Interface Standard (CMSIS)
AnalogDevices::ADuCM32...	Install	Analog Devices ADuCM320 Device Support and Examples
AnalogDevices::ADuCM36...	Install	Analog Devices ADuCM36x Device Support and Examples
Infineon::XMC1000_DFP	Install	Infineon XMC1000 Series Device Support
Infineon::XMC4000_DFP	Install	Infineon XMC4000 Series Device Support
Keil::CM4xx_DFP	Install	Analog Devices ARM Cortex-M4 CM4xx Family Device Supp
Keil::EFM32Gxxx_DFP	Install	Energy Micro EFM32 Gecko Series Device Support
Keil::EFM32LGxxx_DFP	Install	Energy Micro EFM32 Leopard Gecko Series Device Support
Keil::EFM32TGxxx_DFP	Install	Energy Micro EFM32 Tiny Gecko Series Device Support
Keil::EFM32WGxxx_DFP	Install	Energy Micro EFM32 Wonder Gecko Series Device Support
Keil::EFM32ZGxxx_DFP	Install	Energy Micro EFM32 Zero Gecko Series Device Support, Driv

Device
Analog Devices
Atmel
Energy Micro
Freescale
Infineon
Maxim
NXP
SONiX
Spansion
STMicroelectronics

On the left side there is a list of supported microcontroller families, and indicators whether the support for each family has been installed (button label "*up to date*" means that support was installed, and "*install*" label means that it has still not been installed). The installation process is simple, it is sufficient to press *Install* button next to the desired microcontroller family, with an active Internet connection. For example, STM32F4 support will be installed by clicking the button in "*STM32F4*" table row ("*up to date*" indicates that microcontroller family support has been installed):



Pack	Action	Description
Keil::SAMR21_DFP	Install	Atmel SAMR21 Series Device Support and Examples
Keil::SN32F2_DFP	Install	SONiX SN32F2 Series Device Support and Examples
Keil::SN32F7_DFP	Install	SONiX SN32F7 Series Device Support and Examples
Keil::STM32F1xx_DFP	Install	STMicroelectronics STM32F1 Series Device Support, Drivers
Keil::STM32F2xx_DFP	Install	STMicroelectronics STM32F2 Series Device Support, Drivers
Keil::STM32F3xx_DFP	Install	STMicroelectronics STM32F3 Series Device Support and Exar
Keil::STM32F4xx_DFP	Up to date	STMicroelectronics STM32F4 Series Device Support, Drivers
Keil::STM32L1xx_DFP	Install	STMicroelectronics STM32L1 Series Device Support and Exar
Keil::STM32W1xx_DFP	Install	STMicroelectronics STM32W1 Series Device Support and Exa
Keil::TM4C_DFP	Install	Texas Instruments Tiva C Series Device Support and Example
Keil::TMPM0_DFP	Install	Toshiba TX00 Series Device Support

Device
Analog Devices
Atmel
Energy Micro
Freescale
Infineon
Maxim
NXP
SONiX
Spansion
STMicroelectronics

The DFP for microcontroller family is installed in *C:\Keil_v5\ARM\Pack* folder. The content of this folder should not be manually managed.

Note: It is possible to download a DFP package file to the local storage (HDD) and install it offline, without Internet connection. DFP packs can be downloaded from http://www.keil.com/dd2/pack/. DFP packs can be installed off-line by calling the option *File-Import* from the *Pack Installer* program main menu.

2.3. Installing drivers and upgrading STM32F4DISCOVERY board firmware

Next step is to install necessary drivers and software support for running STM32F4DISCOVERY under Windows environment[4]. More details about the board driver installation can be found in documents:

- *UM1472 - Discovery kit for STM32F407.pdf*
- *STMicroelectronics Cortex-M4 Training STM32F407.pdf*

[4] All instructions will refer to Windows 7, 64 bit edition, unless otherwise noted. Keil IDE is not supported under Linux or Mac OS X. It is advisable to use virtual machine with Windows OS if you use some of these operating systems.

Board drivers installation procedure is simple but care must be taken in the following steps:

1. Before plugging the STM32F4DISCOVERY board into computer for the first time, USB drivers must be installed by running the *stlink_winusb_install.bat* (in administrator mode) from *C:\Keil_v5\ARM\STLink\USBDriver* folder.
2. If you plug your board into computer *before* installing USB drivers as described in the previous step, Windows OS will recognize the board but fail to install drivers properly, indicating with message that drivers have not been installed properly; in this case you must do the following:
    - find improperly installed STM32F4DISCOVERY board drivers in Windows Device Manager (device called *ST-Link/V2*),
    - update USB driver pointing to the following location:

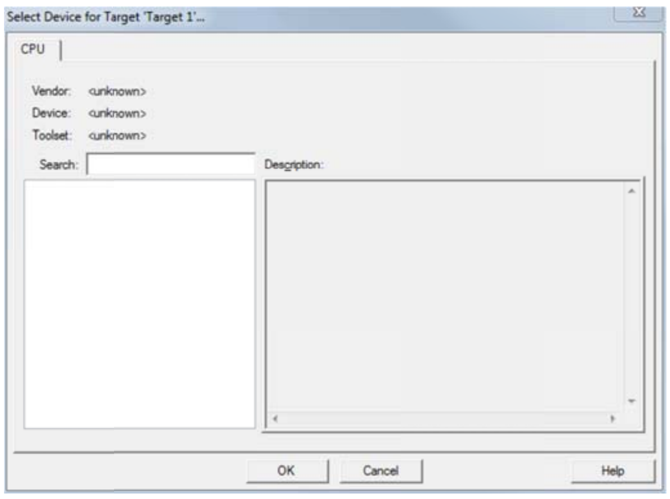
      C:\Keil_v5\ARM\STLink\USBDriver

3. After these steps USB drivers for STM32F4DISCOVERY board should work properly.
4. It is advisable to upgrade board firmware (with board plugged into computer) by using the utility *C:\Keil_v5\ARM\STLink\ST-LinkUpgrade.exe*; this step may be necessary in some cases because uVision IDE will not work with older board firmware revisions.

The STM32F4DISCOVERY board contains on-board *ST-Link/V2* programming and debugging interface, that can be used both for programming/debugging the on-board microcontroller and external target (through the expansion connector by setting the appropriate jumpers, refer to *UM1472 - Discovery kit for STM32F407.pdf* for more details about external target programming).

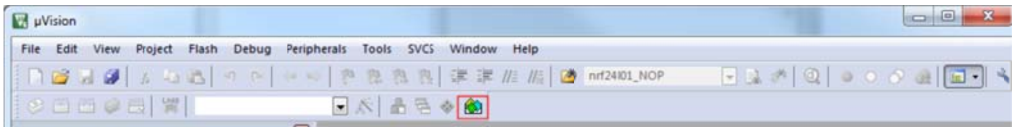
2.4. Verifying system installation and running an example

After following the steps above, we are ready to start developing firmware for STM32F4DISCOVERY board. Before proceeding, it is advisable to verify that toolchain and drivers are properly installed and the board is functional. First we have to check whether we have installed appropriate DFP for our microcontroller family. Right after installation, Keil MDK-ARM uVision does not support any microcontroller family out-of-the-box. In this case, when you call *Project - New μVision Project* option from the main menu you will see the following window:

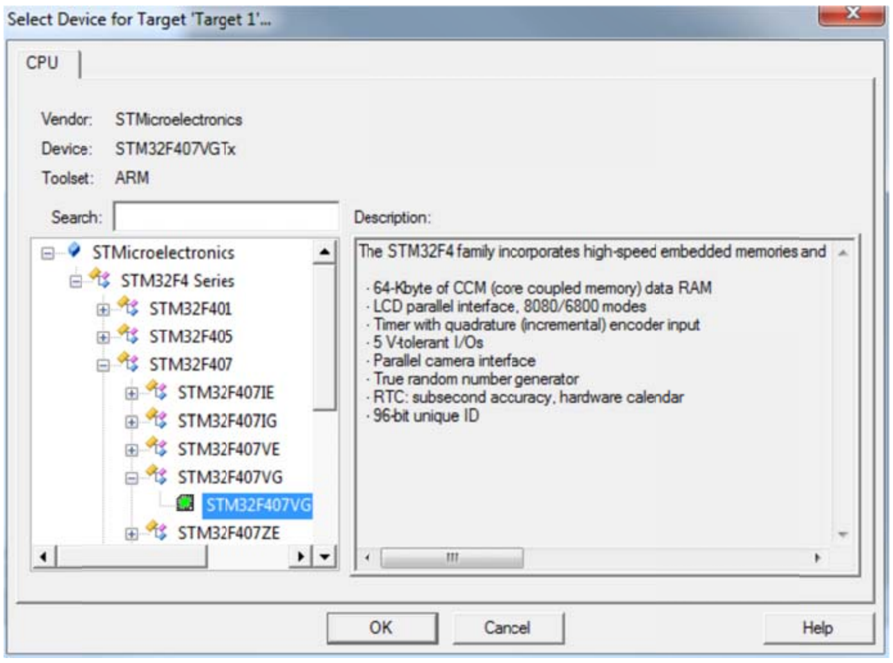


Vendor: <unknown>
Device: <unknown>
Toolset: <unknown>

On the left side there is a list of all installed supported microcontroller families. In this case the list is empty because no DFP support has been installed. It is necessary either to manually run *Pack Installer* from Keil MDK-ARM uVision installation folder or to press the icon (indicated by a red rectangle in a figure below) to call it from IDE:


µVision
File Edit View Project Flash Debug Peripherals Tools SVCS Window Help
nrf24l01_NOP

After installing DFP for STM32F4 family, the support for this type of microcontroller is added to Keil MDK-ARM uVision environment. When you try to create a new project, the following window will appear, showing that now we have support for the whole STM32F4 family (if we develop firmware for STM32F4DISCOVERY board, the part STM32F407VG must be chosen to match on-board processor):


Select Device for Target 'Target 1'...

CPU

Vendor: STMicroelectronics
Device: STM32F407VGTx
Toolset: ARM

Search:

Description:

The STM32F4 family incorporates high-speed embedded memories and

- 64-Kbyte of CCM (core coupled memory) data RAM
- LCD parallel interface, 8080/6800 modes
- Timer with quadrature (incremental) encoder input
- 5 V-tolerant I/Os
- Parallel camera interface
- True random number generator
- RTC: subsecond accuracy, hardware calendar
- 96-bit unique ID

However, instead of creating the new project at this point, we shall import an existing example just to verify that the board is functional. If we take a look on Keil uVision directory tree, we can see under \Pack\Keil folder that there is a subfolder *STM32F4xx_DFP* containing libraries, examples etc. for STM32F4 family. Some example projects are available from folders:

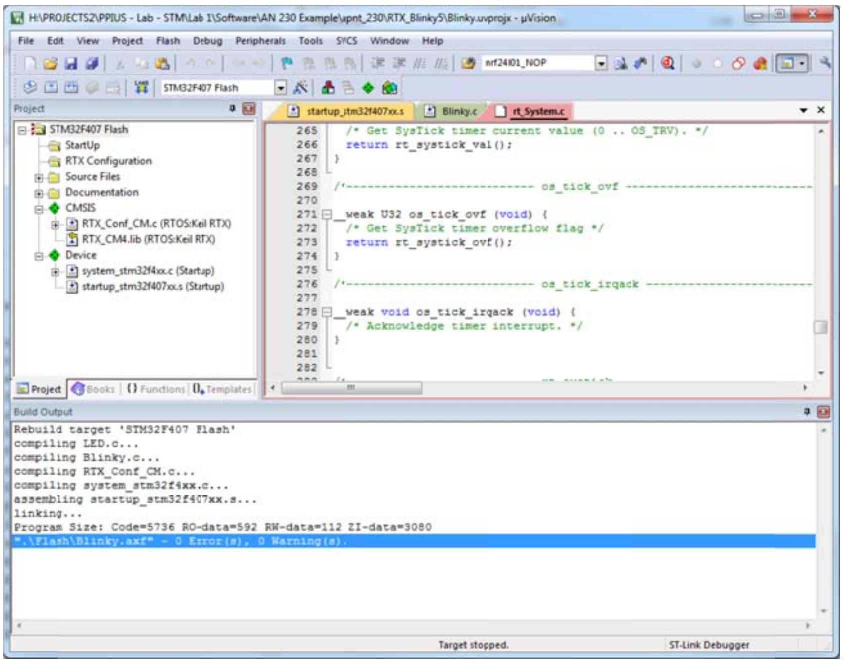
C:\Keil_v5\ARM\Pack\Keil\STM32F4xx_DFP\2.2.0\MDK\Boards\ST\STM32F4-Discovery
C:\Keil_v5\ARM\Pack\Keil\STM32F4xx_DFP\2.2.0\Projects\STM32F4-Discovery

However, for a quick high-level verification, it is better to download example projects for "*Keil Application note 230: MDK V5.10 Lab for the STM32F4 Discovery Board*", available from:

http://www.keil.com/appnotes/docs/apnt_230.asp
(also available from *apnt_230.zip* in *PIES_2014_Lab1_Pack.rar*).

Unpack *apnt_230.zip* to arbitrary location and open the project ***RTX_Blinky5*** (*Project - Open Project*) in uV5. After opening the project, call *Project - Rebuild all target files*. If compilation is successful, the following messages will show up in the *Build Output* window:

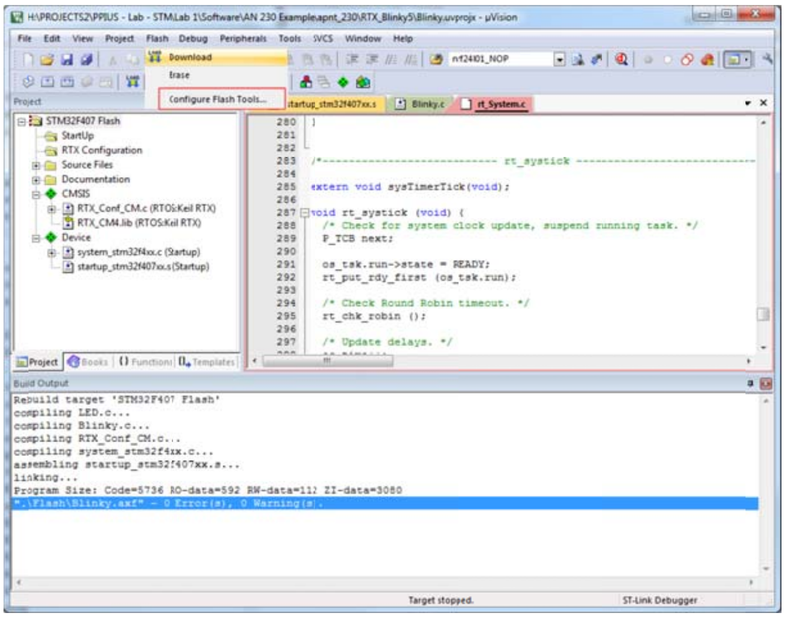
```
Rebuild target 'STM32F407 Flash'
compiling LED.c...
compiling Blinky.c...
compiling RTX_Conf_CM.c...
compiling system_stm32f4xx.c...
assembling startup_stm32f407xx.s...
linking...
Program Size: Code=5736 RO-data=592 RW-data=112 ZI-data=3080
".\Flash\Blinky.axf" - 0 Error(s), 0 Warning(s).
```


H:\PROJECTS2\PPIUS - Lab - STM\Lab 1\Software\AN 230 Example\apnt_230\RTX_Blinky5\Blinky.uvprojx - µVision
File Edit View Project Flash Debug Peripherals Tools SVCS Window Help
nrf24l01_NOP
STM32F407 Flash
Project
startup_stm32f407xx.s
Blinky.c
rt_System.c
STM32F407 Flash
StartUp
RTX Configuration
Source Files
Documentation
CMSIS
RTX_Conf_CM.c (RTOS:Keil RTX)
RTX_CM4.lib (RTOS:Keil RTX)
Device
system_stm32f4xx.c (Startup)
startup_stm32f407xx.s (Startup)
Project
Books
Functions
Templates
Build Output
Target stopped.
ST-Link Debugger

```
  /* Get SysTick timer current value (0 .. OS_TRV). */
  return rt_systick_val();
}

/*-------------------------- os_tick_ovf --------------------------

__weak U32 os_tick_ovf (void) {
  /* Get SysTick timer overflow flag */
  return rt_systick_ovf();
}

/*-------------------------- os_tick_irqack -----------------------

__weak void os_tick_irqack (void) {
  /* Acknowledge timer interrupt. */
}
```

```
Rebuild target 'STM32F407 Flash'
compiling LED.c...
compiling Blinky.c...
compiling RTX_Conf_CM.c...
compiling system_stm32f4xx.c...
assembling startup_stm32f407xx.s...
linking...
Program Size: Code=5736 RO-data=592 RW-data=112 ZI-data=3080
".\Flash\Blinky.axf" - 0 Error(s), 0 Warning(s).
```

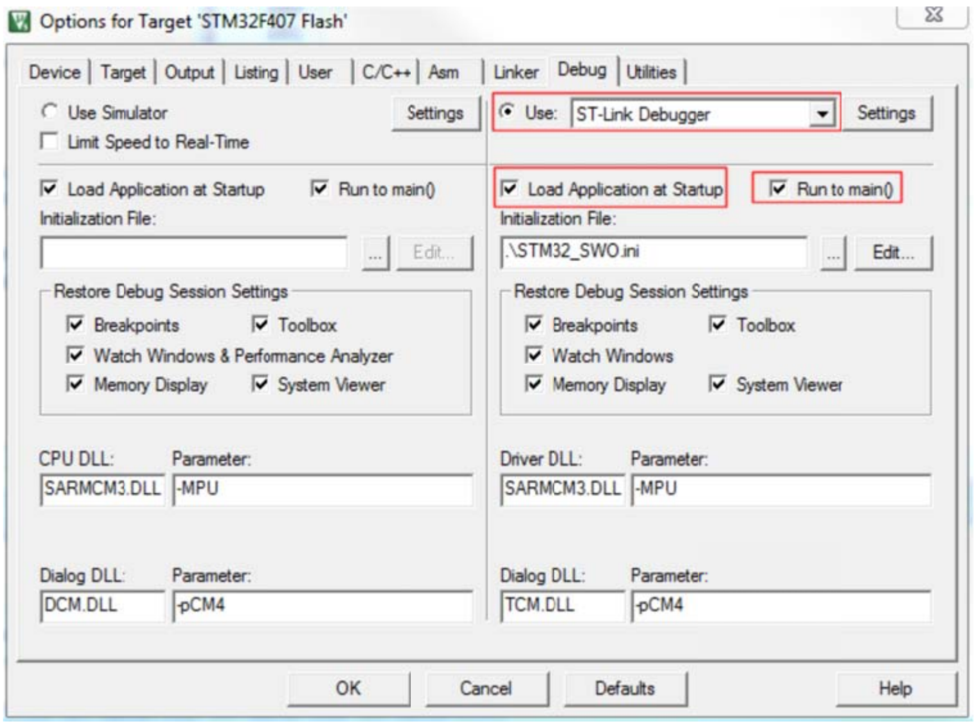
Upon successful project compilation, the target board must be programmed. Before programming, the programmer and debugger interface must be configured. Select *Flash - Configure Flash Tools...* option:


H:\PROJECTS2\PPIUS - Lab - STM\Lab 1\Software\AN 230 Example\apnt_230\RTX_Blinky5\Blinky.uvprojx - µVision
File Edit View Project Flash Debug Peripherals Tools SVCS Window Help
Download
Erase
Configure Flash Tools...
nrf24l01_NOP
Project
startup_stm32f407xx.s
Blinky.c
rt_System.c
STM32F407 Flash
StartUp
RTX Configuration
Source Files
Documentation
CMSIS
RTX_Conf_CM.c (RTOS:Keil RTX)
RTX_CM4.lib (RTOS:Keil RTX)
Device
system_stm32f4xx.c (Startup)
startup_stm32f407xx.s (Startup)
Project
Books
Functions
Templates
Build Output
Target stopped.
ST-Link Debugger

```
}


/*-------------------------- rt_systick ---------------------------

extern void sysTimerTick(void);

void rt_systick (void) {
  /* Check for system clock update, suspend running task. */
  P_TCB next;

  os_tsk.run->state = READY;
  rt_put_rdy_first (os_tsk.run);

  /* Check Round Robin timeout. */
  rt_chk_robin ();

  /* Update delays. */
```

```
Rebuild target 'STM32F407 Flash'
compiling LED.c...
compiling Blinky.c...
compiling RTX_Conf_CM.c...
compiling system_stm32f4xx.c...
assembling startup_stm32f407xx.s...
linking...
Program Size: Code=5736 RO-data=592 RW-data=112 ZI-data=3080
".\Flash\Blinky.axf" - 0 Error(s), 0 Warning(s).
```

Make sure these options are checked:


Options for Target 'STM32F407 Flash'

Device | Target | Output | Listing | User | C/C++ | Asm | Linker | Debug | Utilities

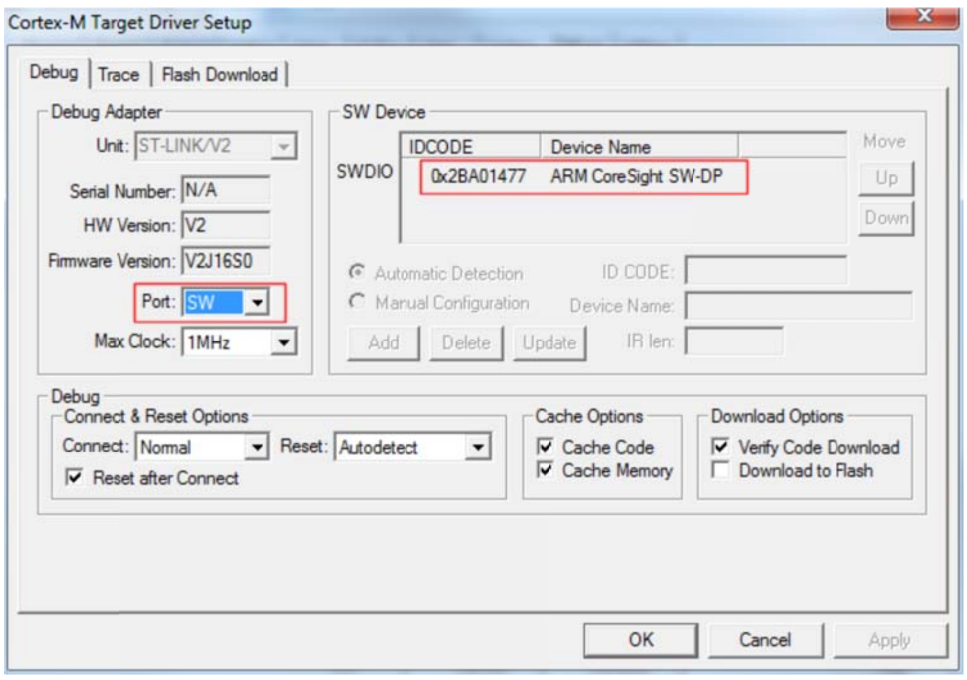
Use Simulator ☐ — Settings
☐ Limit Speed to Real-Time
☑ Load Application at Startup ☑ Run to main()
Initialization File:
Restore Debug Session Settings: ☑ Breakpoints ☑ Toolbox ☑ Watch Windows & Performance Analyzer ☑ Memory Display ☑ System Viewer
CPU DLL: SARMCM3.DLL Parameter: -MPU
Dialog DLL: DCM.DLL Parameter: -pCM4

● Use: ST-Link Debugger — Settings
☑ Load Application at Startup ☑ Run to main()
Initialization File: .\STM32_SWO.ini
Restore Debug Session Settings: ☑ Breakpoints ☑ Toolbox ☑ Watch Windows ☑ Memory Display ☑ System Viewer
Driver DLL: SARMCM3.DLL Parameter: -MPU
Dialog DLL: TCM.DLL Parameter: -pCM4

OK | Cancel | Defaults | Help

- *Use ST-Link debugger* - otherwise program will run in simulator; make sure that *ST-Link debugger* is chosen from list because this is the hardware interface available on board (it will not work with any other interface from the drop-down combo),
- *Load Application at Startup, Run to Main*

Next, click on *Settings* button and check "*Debug*" tab settings:


Cortex-M Target Driver Setup

Debug | Trace | Flash Download

Debug Adapter
Unit: ST-LINK/V2
Serial Number: N/A
HW Version: V2
Firmware Version: V2J16S0
Port: SW
Max Clock: 1MHz

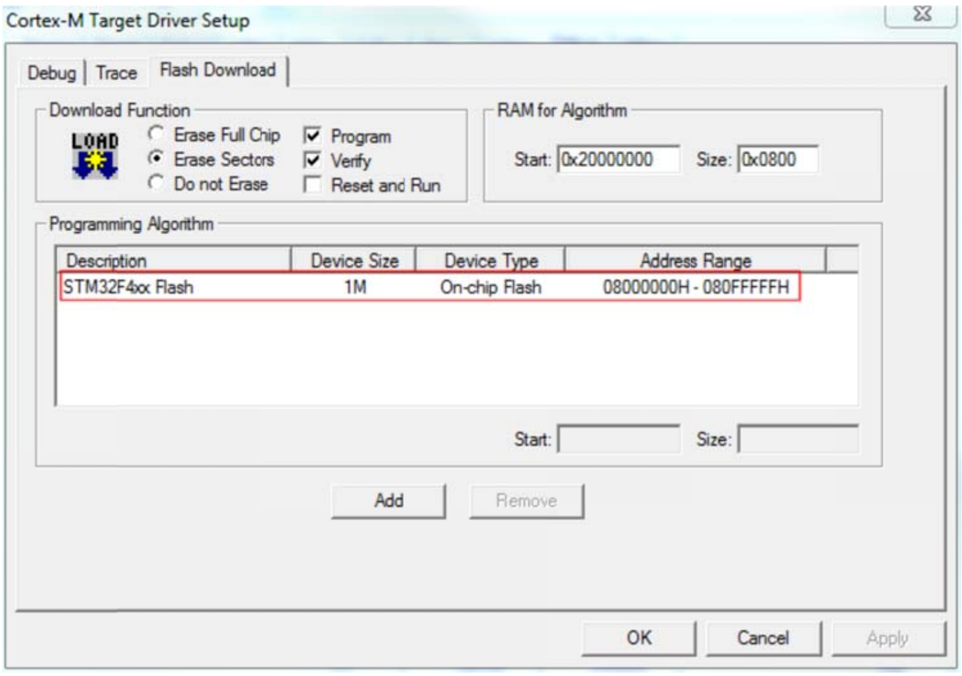
SW Device
	IDCODE	Device Name
SWDIO	0x2BA01477	ARM CoreSight SW-DP

● Automatic Detection ID CODE:
○ Manual Configuration Device Name:
Add | Delete | Update IR len:
Move Up | Down

Debug
Connect & Reset Options: Connect: Normal Reset: Autodetect
☑ Reset after Connect
Cache Options: ☑ Cache Code ☑ Cache Memory
Download Options: ☑ Verify Code Download ☐ Download to Flash

OK | Cancel | Apply

Make sure your device is recognized (code in *SW Device* window) and that you use *SW port* (not default *JTAG*!). Next check the setting on "*Flash Download*" tab:


Cortex-M Target Driver Setup

Debug | Trace | Flash Download

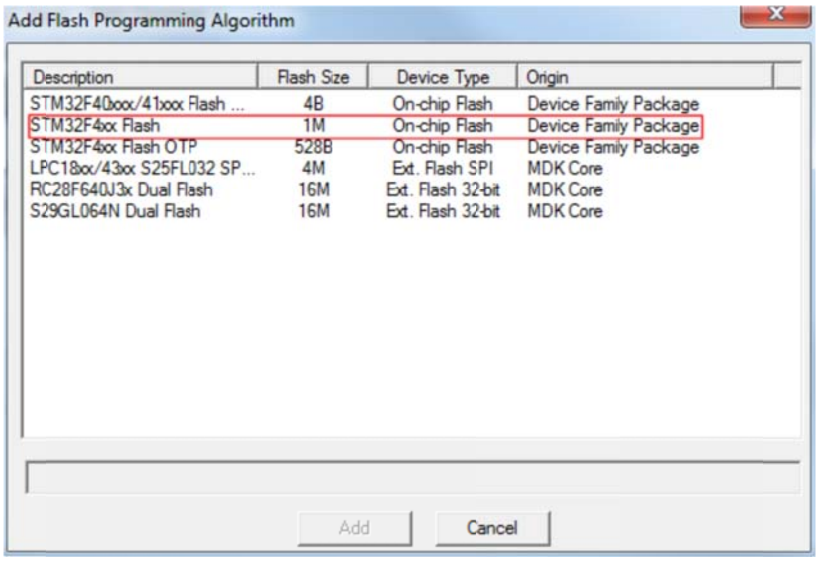
Download Function: ○ Erase Full Chip, ● Erase Sectors, ○ Do not Erase; ☑ Program, ☑ Verify, ☐ Reset and Run

RAM for Algorithm — Start: 0x20000000 Size: 0x0800

Programming Algorithm

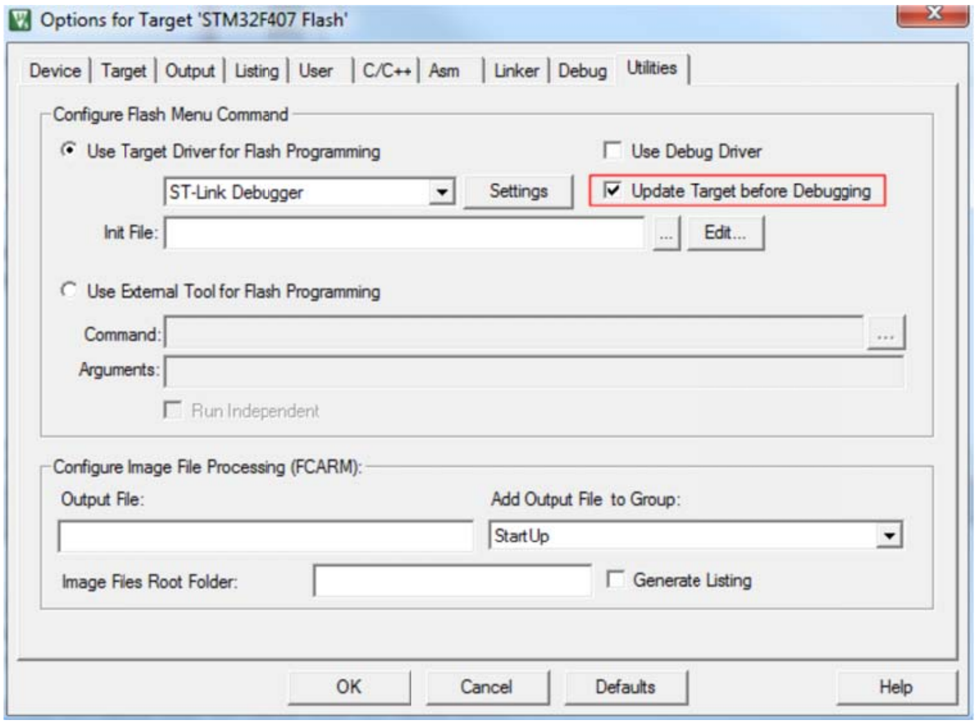
Description	Device Size	Device Type	Address Range
STM32F4xx Flash	1M	On-chip Flash	08000000H - 080FFFFFH

If *Programming Algorithm* list is empty, your board will not be programmed and you will receive an error message. In this case you must provide programming algorithm for your board by clicking *Add* button on *Flash Download* tab and selecting *STM32F4xx Flash/1M/On-chip Flash*:


Add Flash Programming Algorithm

Description	Flash Size	Device Type	Origin
STM32F40xxx/41xxx Flash ...	4B	On-chip Flash	Device Family Package
STM32F4xx Flash	1M	On-chip Flash	Device Family Package
STM32F4xx Flash OTP	528B	On-chip Flash	Device Family Package
LPC18xx/43xx S25FL032 SP...	4M	Ext. Flash SPI	MDK Core
RC28F640J3x Dual Flash	16M	Ext. Flash 32-bit	MDK Core
S29GL064N Dual Flash	16M	Ext. Flash 32-bit	MDK Core

Next, check the settings on *Utilites* tab of *Flash - Configure Flash Tools...* option:



Options for Target 'STM32F407 Flash'

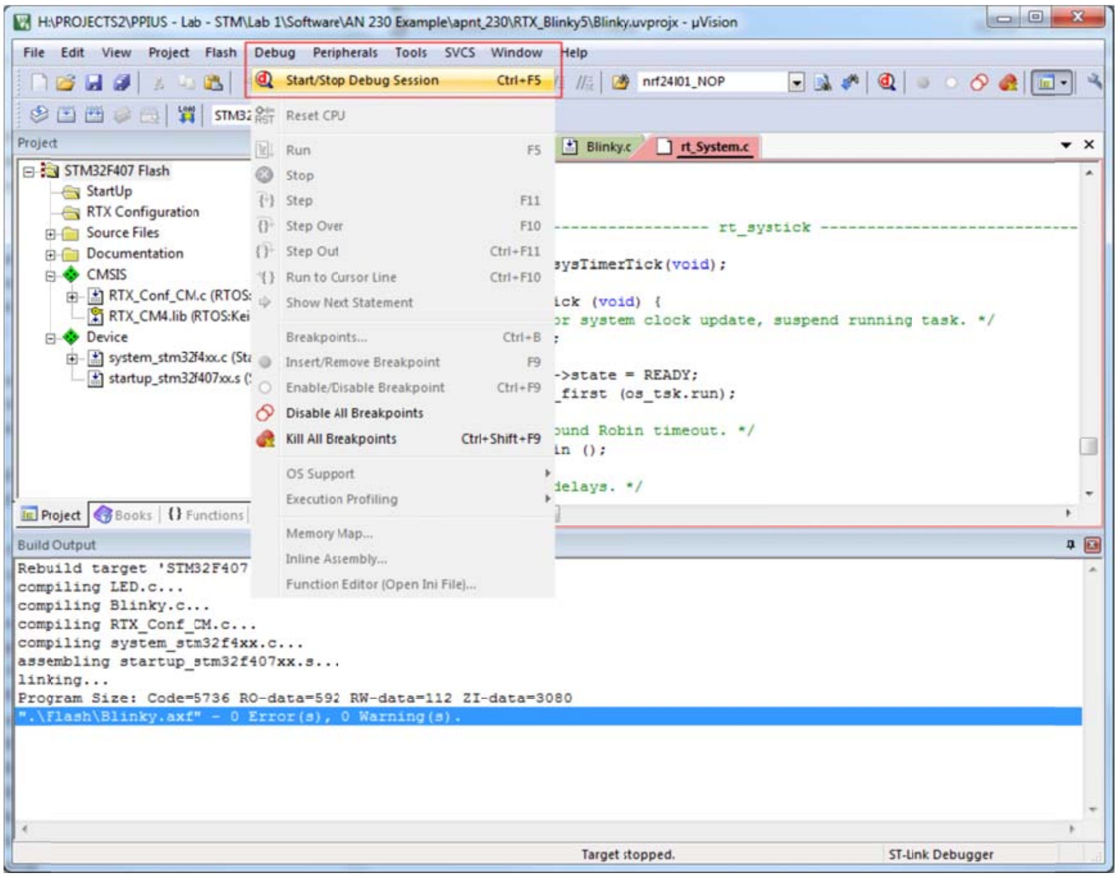
Device | Target | Output | Listing | User | C/C++ | Asm | Linker | Debug | Utilities

Configure Flash Menu Command
- (●) Use Target Driver for Flash Programming — ST-Link Debugger — Settings
- [ ] Use Debug Driver
- [✓] Update Target before Debugging
- Init File:
- ( ) Use External Tool for Flash Programming
- Command:
- Arguments:
- [ ] Run Independent

Configure Image File Processing (FCARM):
- Output File:
- Add Output File to Group: StartUp
- Image Files Root Folder:
- [ ] Generate Listing

OK | Cancel | Defaults | Help

Make sure the *Update Target before Debugging* option is checked, otherwise your target board will not automatically be updated (reprogrammed) before every debugging session. Call *File - Save All* option to ensure everything is saved. Now run option *Debug - Start/Stop Debug Session* (Ctrl + F5):



H:\PROJECTS2\PPIUS - Lab - STM\Lab 1\Software\AN 230 Example\apnt_230\RTX_Blinky5\Blinky.uvprojx - µVision

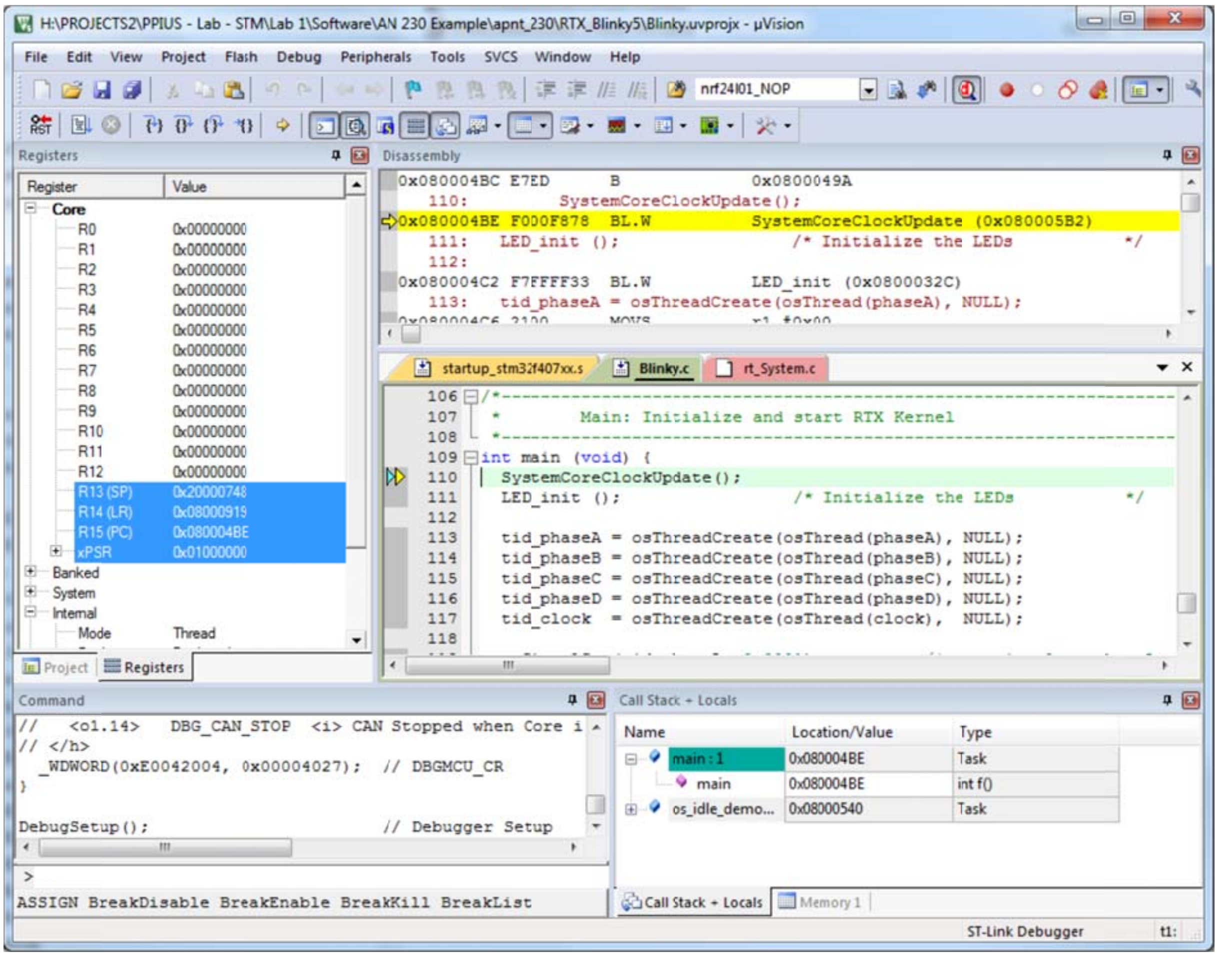
```
------------------ rt_systick ----------------------------
sysTimerTick(void);
ick (void) {
or system clock update, suspend running task. */
->state = READY;
_first (os_tsk.run);
ound Robin timeout. */
in ();
delays. */
```

```
Rebuild target 'STM32F407
compiling LED.c...
compiling Blinky.c...
compiling RTX_Conf_CM.c...
compiling system_stm32f4xx.c...
assembling startup_stm32f407xx.s...
linking...
Program Size: Code=5736 RO-data=592 RW-data=112 ZI-data=3080
".\Flash\Blinky.axf" - 0 Error(s), 0 Warning(s).
```

In Build Output window you should see the following output:

```
Rebuild target 'STM32F407 Flash'
compiling LED.c...
compiling Blinky.c...
compiling RTX_Conf_CM.c...
compiling system_stm32f4xx.c...
assembling startup_stm32f407xx.s...
linking...
Program Size: Code=5736 RO-data=592 RW-data=112 ZI-data=3080
".\Flash\Blinky.axf" - 0 Error(s), 0 Warning(s).
Load      "H:\\PROJECTS2\\PPIUS      -      Lab      -      STM\\Lab      1\\Software\\AN      230
Example\\apnt_230\\RTX_Blinky5\\Flash\\Blinky.axf"
Erase Done.
Programming Done.
Verify OK.
```

After running the program, it will pause at the first statement in *main()* function:



H:\PROJECTS2\PPIUS - Lab - STM\Lab 1\Software\AN 230 Example\apnt_230\RTX_Blinky5\Blinky.uvprojx - µVision

Registers

Register	Value
Core	
R0	0x00000000
R1	0x00000000
R2	0x00000000
R3	0x00000000
R4	0x00000000
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x20000748
R14 (LR)	0x08000919
R15 (PC)	0x080004BE
xPSR	0x01000000
Banked	
System	
Internal	
Mode	Thread

Disassembly

```
0x080004BC E7ED      B                 0x0800049A
   110:         SystemCoreClockUpdate();
0x080004BE F000F878  BL.W              SystemCoreClockUpdate (0x080005B2)
   111:   LED_init ();                     /* Initialize the LEDs         */
   112:
0x080004C2 F7FFFF33  BL.W              LED_init (0x0800032C)
   113:   tid_phaseA = osThreadCreate(osThread(phaseA), NULL);
```

startup_stm32f407xx.s | Blinky.c | rt_System.c

```
/*----------------------------------------------------------------------------
 *      Main: Initialize and start RTX Kernel
 *---------------------------------------------------------------------------*/
int main (void) {
  SystemCoreClockUpdate();
  LED_init ();                          /* Initialize the LEDs         */

  tid_phaseA = osThreadCreate(osThread(phaseA), NULL);
  tid_phaseB = osThreadCreate(osThread(phaseB), NULL);
  tid_phaseC = osThreadCreate(osThread(phaseC), NULL);
  tid_phaseD = osThreadCreate(osThread(phaseD), NULL);
  tid_clock  = osThreadCreate(osThread(clock),  NULL);
```

Command

```
//   <o1.14>   DBG_CAN_STOP  <i> CAN Stopped when Core i
// </h>
  _WDWORD(0xE0042004, 0x00004027);  // DBGMCU_CR
}

DebugSetup();                       // Debugger Setup
>
ASSIGN BreakDisable BreakEnable BreakKill BreakList
```

Call Stack + Locals

Name	Location/Value	Type
main : 1	0x080004BE	Task
main	0x080004BE	int f()
os_idle_demo...	0x08000540	Task

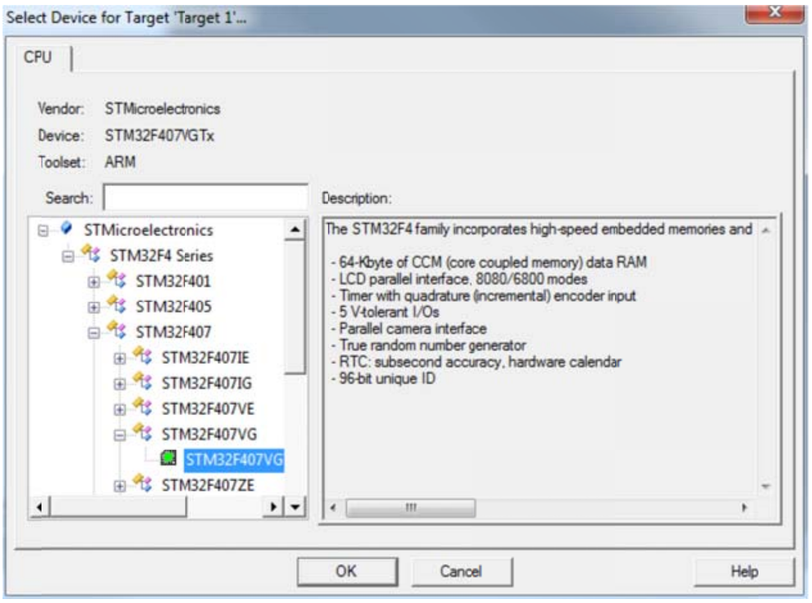
At this point everything works but program is halted. Press F5 to run program execution and observe the board. You can use breakpoints to stop program execution, watch variable values etc.

If LEDs are flashing and rotating - congratulations, you may proceed to lab exercises!

3. Creating a new project

3.1. Creating a new project in uV5 using run-time environment manager

A new project is created by clicking *Project - New µVision Project* and providing the project name and folder. In a process of project file creation, you must choose the exact microcontroller part you will use (STM32F407VG[5] in case of STM32F4DISCOVERY board):

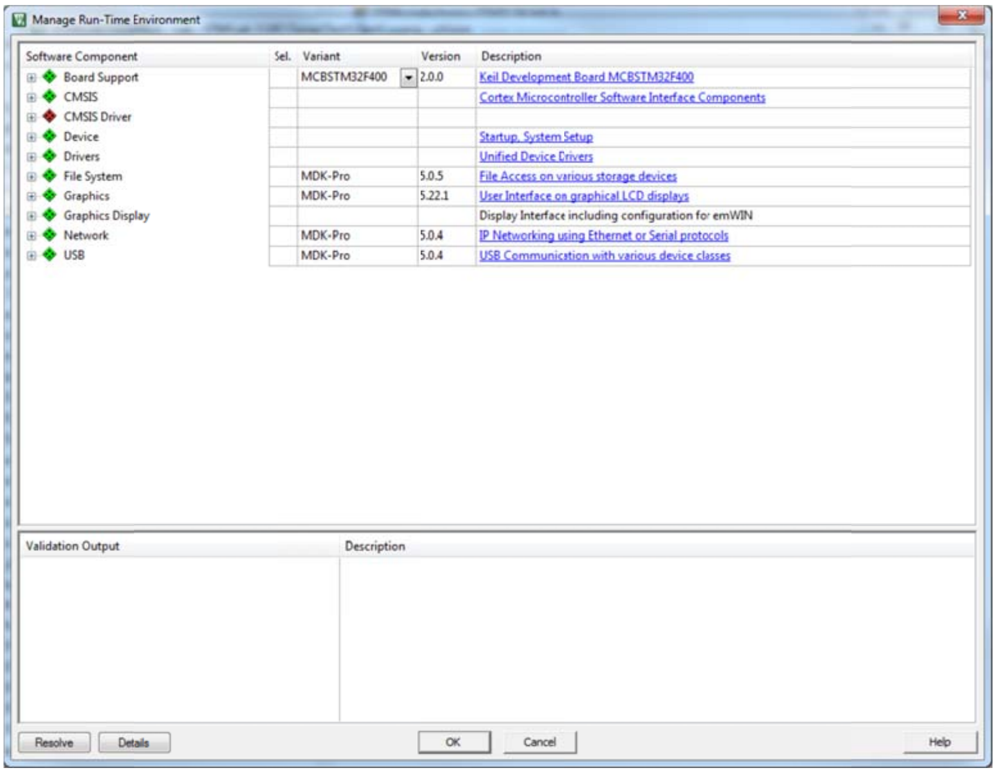

Select Device for Target 'Target 1'...

Vendor: STMicroelectronics
Device: STM32F407VGTx
Toolset: ARM

Description:
The STM32F4 family incorporates high-speed embedded memories and

- 64-Kbyte of CCM (core coupled memory) data RAM
- LCD parallel interface, 8080/6800 modes
- Timer with quadrature (incremental) encoder input
- 5 V-tolerant I/Os
- Parallel camera interface
- True random number generator
- RTC: subsecond accuracy, hardware calendar
- 96-bit unique ID

Then the next window appears where you can select various program components:


Manage Run-Time Environment

Software Component	Sel.	Variant	Version	Description
Board Support		MCBSTM32F400	2.0.0	Keil Development Board MCBSTM32F400
CMSIS				Cortex Microcontroller Software Interface Components
CMSIS Driver				
Device				Startup, System Setup
Drivers				Unified Device Drivers
File System		MDK-Pro	5.0.5	File Access on various storage devices
Graphics		MDK-Pro	5.22.1	User Interface on graphical LCD displays
Graphics Display				Display Interface including configuration for emWIN
Network		MDK-Pro	5.0.4	IP Networking using Ethernet or Serial protocols
USB		MDK-Pro	5.0.4	USB Communication with various device classes

[5] Microcontroller part can be chosen only if an appropriate DFP package has been installed

Unlike the previous revisions, Keil uV5 provides managed run-time environment where you can select what software components and libraries you need for development of your application. Some of components that you may include are the following:

- *Board Support* - program support for peripherals, LEDs, buttons etc. for various development boards (BSP (*board support package*) level libraries),
- *CMSIS - Cortex® Microcontroller Software Interface Standard* - vendor-independent hardware abstraction layer (HAL) for Cortex-M microcontrollers; it simplifies and unifies software development cycle with an emphasis on cross-platform portability of code among different silicon vendors; this is high-level library on a layer above vendor-specific HAL libraries; the CMSIS layer will not be used in this laboratory exercise,
- *Device / Startup / STM32Cube Framework (API) / STM32Cube HAL* - hardware abstraction layer (HAL) targeting STM32F4 family; this layer is vendor-specific (STMicroelectronics) and provides high-level access to microcontroller resources and peripherals without a need for individual register access programming[6]; this laboratory exercise will focus on vendor-specific HAL-level ARM programming (but provided from different library, as explained later),
- *Drivers, File System, Graphics, Network, USB* - advanced libraries and services that will not be covered in this lab.

Run-time environment manager enables easy inclusion of various libraries and services, while taking care of all dependancies and helping user to select all needed libraries for the solution.

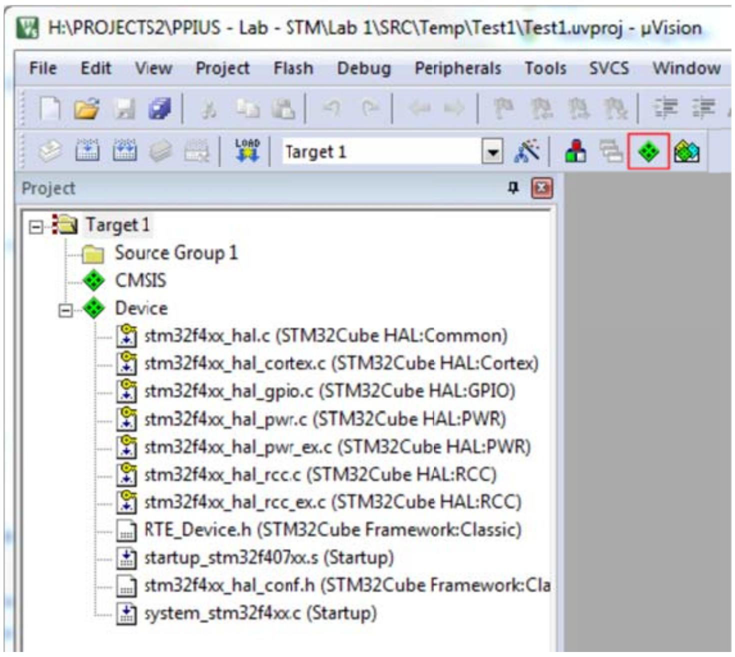
This laboratory exercise will introduce HAL programming paradigm for STM32F4 microcontroller family[7]. In order to prepare project for using the STMCubeF4 HAL libraries, the following components must be selected:

```
CMSIS
       Core
Device
       Startup
Device
       STM32Cube Framework (API)
              Classic
Device
       STM32Cube HAL
              Common
              Cortex
              GPIO
              PWR
              RCC
```

[6] *HAL-level programming paradigm* (using high-level C peripheral driver functions) provides an extra layer of abstraction compared to the *register-level programming paradigm* (using direct hardware register access, that is commonly encountered in firmware development for microcontrollers, especially for 8- and 16-bit families); register-level approach results in smaller and more optimized code, but HAL-level approach speeds up the development and reduces chances for introducing bugs; these paradigms are not mutually exclusive and they can be freely mixed within the same project as necessary

[7] Although the rest of the chapter shows how to setup *STMCubeF4 firmware package HAL* in the last revision of Keil MDK-ARM uVision (to demonstrate how to start a new project in uV5), this lab exercise will not use *STMCubeF4 HAL*, and the guidelines will be based on an older HAL library (*STM32F4xx Standard Peripheral Library*), for the reasons explained later

These components are required to write simple minimum program example, e.g. one that use only GPIO pins. After clicking OK button, the following files are shown in *Project* window:


H:\PROJECTS2\PPIUS - Lab - STM\Lab 1\SRC\Temp\Test1\Test1.uvproj - µVision
File Edit View Project Flash Debug Peripherals Tools SVCS Window
Target 1
Project

Target 1
- Source Group 1
- CMSIS
- Device
  - stm32f4xx_hal.c (STM32Cube HAL:Common)
  - stm32f4xx_hal_cortex.c (STM32Cube HAL:Cortex)
  - stm32f4xx_hal_gpio.c (STM32Cube HAL:GPIO)
  - stm32f4xx_hal_pwr.c (STM32Cube HAL:PWR)
  - stm32f4xx_hal_pwr_ex.c (STM32Cube HAL:PWR)
  - stm32f4xx_hal_rcc.c (STM32Cube HAL:RCC)
  - stm32f4xx_hal_rcc_ex.c (STM32Cube HAL:RCC)
  - RTE_Device.h (STM32Cube Framework:Classic)
  - startup_stm32f407xx.s (Startup)
  - stm32f4xx_hal_conf.h (STM32Cube Framework:Cla
  - system_stm32f4xx.c (Startup)

Files marked with small yellow mark are linked files in Keil uVision IDE installation (DFP pack), while other files

```
RTE_Device.h
startup_stm32f407xx.s
stm32f4xx_hal_conf.h
system_stm32f4xx.c
```

are copied locally in a project subfolder *\RTE\Device\STM32F407VGTx*. The run-time environment manager takes care to track all dependancies. For example, by clicking on "*Target 1*" node (right-click) + *Options*, on C/C++ tab, under *Compiler control string* there are directives for compiler how to find all necessary files:

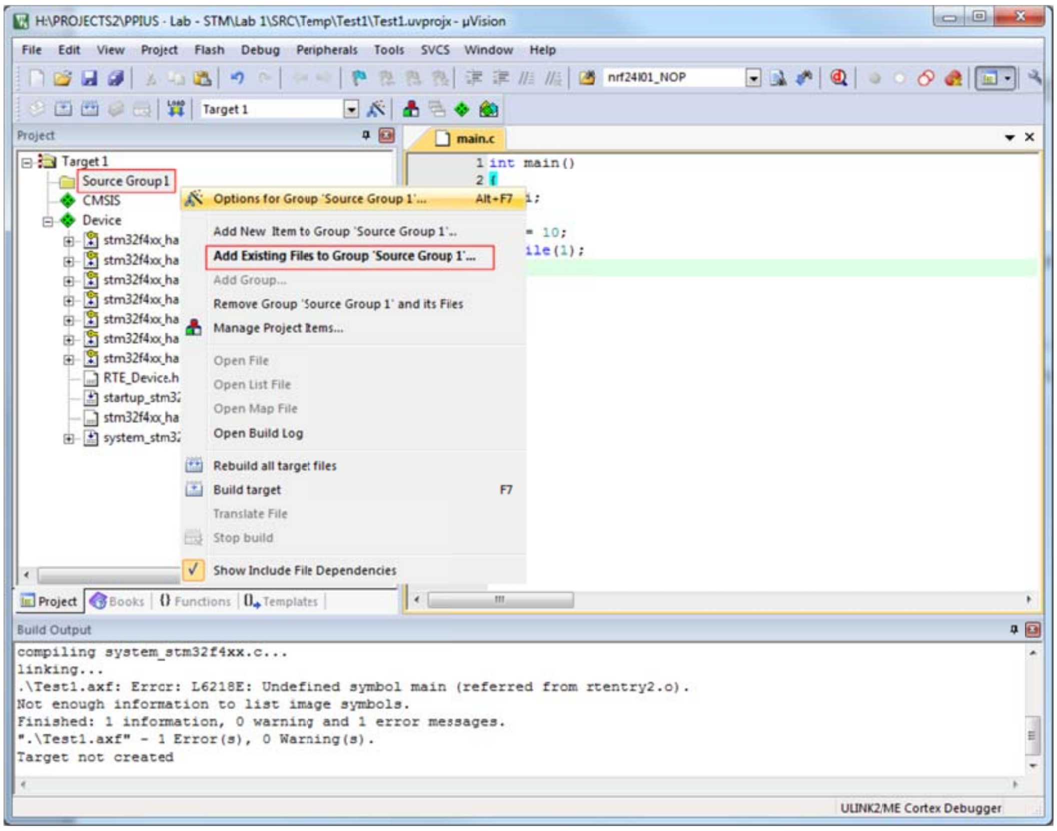
```
-c --cpu Cortex-M4.fp -g -O0 --apcs=interwork
-I "H:\PROJECTS2\PPIUS - Lab - STM\Lab 1\SRC\Temp\Test1\RTE"
-I "H:\PROJECTS2\PPIUS - Lab - STM\Lab 1\SRC\Temp\Test1\RTE\Device\STM32F407VGTx"
-I D:\Keil_v5\ARM\PACK\ARM\CMSIS\3.20.4\CMSIS\Include
-I
D:\Keil_v5\ARM\PACK\Keil\STM32F4xx_DFP\2.2.0\Drivers\CMSIS\Device\ST\STM32F4xx\Incl
ude
-I D:\Keil_v5\ARM\PACK\Keil\STM32F4xx_DFP\2.2.0\Drivers\STM32F4xx_HAL_Driver\Inc
-D_RTE_ -DSTM32F407xx -o "*.o" --omf_browse "*.crf" --depend "*.d"
```

This kind of information is usually managed by makefiles or command-line arguments, but run-time manager helps developers to track all dependancies more easily. On this tab one can e.g. define preprocessor symbols, add additional include paths (for libraries not controlled by run-time manager) etc.

After selecting the appropriate components, programming and debugging interface must be configured through the *Flash - Configure Flash Tools...* option (refer to notes given in chapter *Verifying system installation and running an example*).

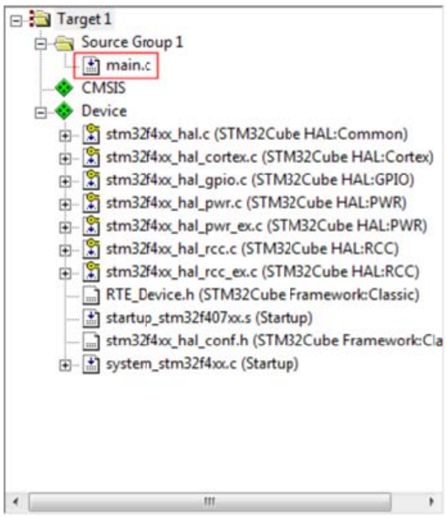
At this point, we still cannot build an executable because we do not have minimum amount of application code (C source file with *main()* function - predefined entry point symbol needed by linker). So the next step is to add *main.c* to the project.

Create new empty file (*File - New*), click *File - Save As...* and name the file *main.c* (save the file in project root folder - same location as *.uvprojx). Please note that newly created *.c file **will not be automatically included** in your project build (if you try building the project at this point linker will complain that it cannot find *main()* function**)**. You need to add *main.c* by right-clicking *Source Group 1* node and selecting *Add Existing Files to Group...* option:


H:\PROJECTS2\PPIUS - Lab - STM\Lab 1\SRC\Temp\Test1\Test1.uvprojx - µVision
File Edit View Project Flash Debug Peripherals Tools SVCS Window Help
Target 1
Project
main.c
Target 1
Source Group 1
CMSIS
Device
RTE_Device.h
Options for Group 'Source Group 1'... Alt+F7
Add New Item to Group 'Source Group 1'...
Add Existing Files to Group 'Source Group 1'...
Add Group...
Remove Group 'Source Group 1' and its Files
Manage Project Items...
Open File
Open List File
Open Map File
Open Build Log
Rebuild all target files
Build target F7
Translate File
Stop build
Show Include File Dependencies
Project
Books
Functions
Templates
Build Output
ULINK2/ME Cortex Debugger

```
compiling system_stm32f4xx.c...
linking...
.\Test1.axf: Error: L6218E: Undefined symbol main (referred from rtentry2.o).
Not enough information to list image symbols.
Finished: 1 information, 0 warning and 1 error messages.
".\Test1.axf" - 1 Error(s), 0 Warning(s).
Target not created
```

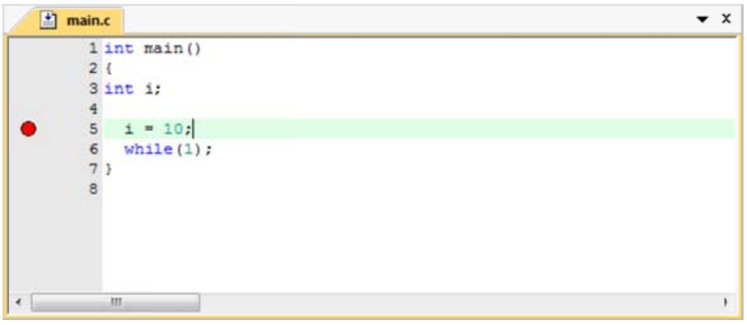
After adding the file it will appear in the source tree:


Target 1
Source Group 1
main.c
CMSIS
Device
stm32f4xx_hal.c (STM32Cube HAL:Common)
stm32f4xx_hal_cortex.c (STM32Cube HAL:Cortex)
stm32f4xx_hal_gpio.c (STM32Cube HAL:GPIO)
stm32f4xx_hal_pwr.c (STM32Cube HAL:PWR)
stm32f4xx_hal_pwr_ex.c (STM32Cube HAL:PWR)
stm32f4xx_hal_rcc.c (STM32Cube HAL:RCC)
stm32f4xx_hal_rcc_ex.c (STM32Cube HAL:RCC)
RTE_Device.h (STM32Cube Framework:Classic)
startup_stm32f407xx.s (Startup)
stm32f4xx_hal_conf.h (STM32Cube Framework:Cla
system_stm32f4xx.c (Startup)

Write the following code in *main.c*:

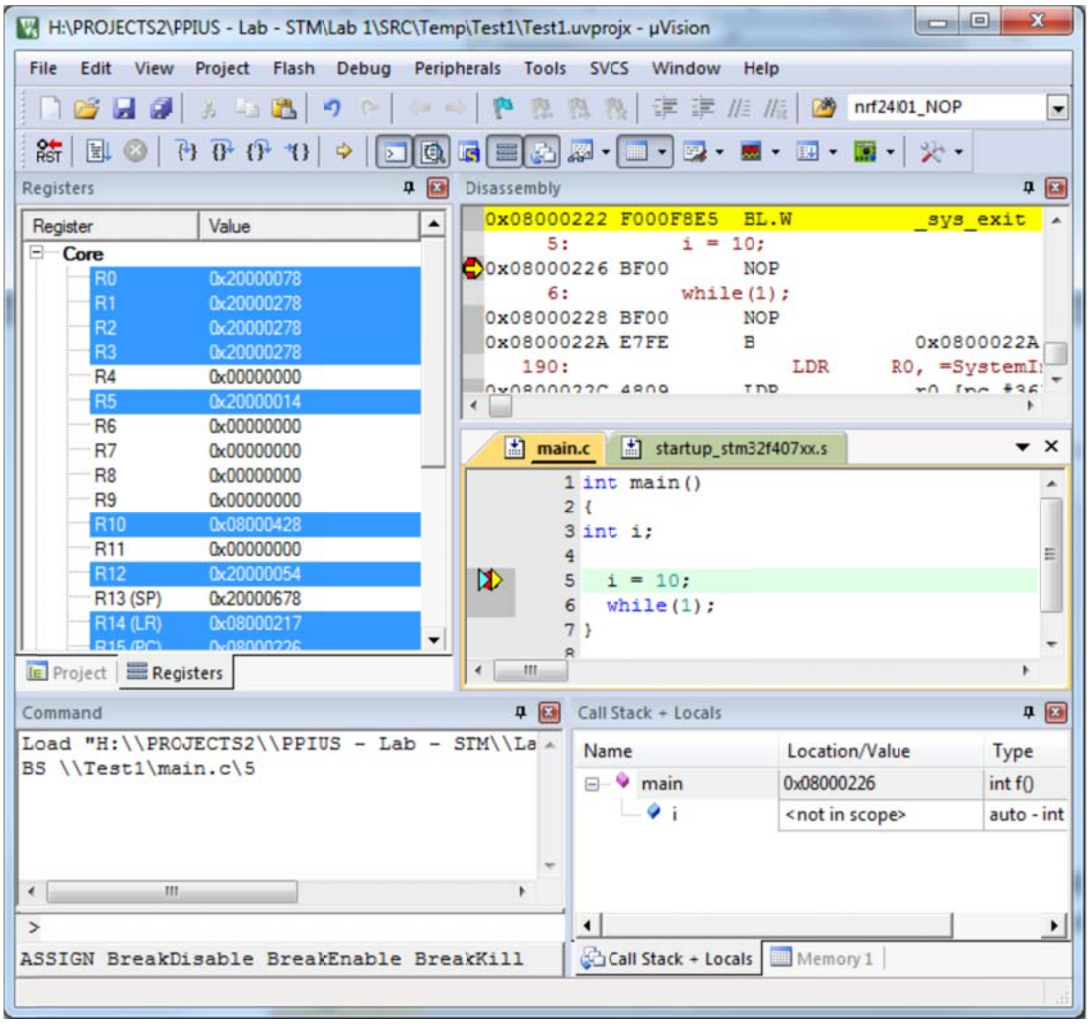
```c
int main()
{
int i;

        i = 10;
        while(1);
}
```

Press F7 to Build target. Set breakpoint to line "*i = 10;*" (either by positioning cursor in that line and pressing F9 or double clicking on pane on the left side of the code window - red dot should appear):


main.c

```c
int main()
{
int i;

  i = 10;
  while(1);
}
```

Press Ctrl + F5 to program target and run program - program should run and hit the breakpoint.


H:\PROJECTS2\PPIUS - Lab - STM\Lab 1\SRC\Temp\Test1\Test1.uvprojx - µVision

Register	Value
Core	
R0	0x20000078
R1	0x20000278
R2	0x20000278
R3	0x20000278
R4	0x00000000
R5	0x20000014
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x08000428
R11	0x00000000
R12	0x20000054
R13 (SP)	0x20000678
R14 (LR)	0x08000217
R15 (PC)	0x08000226

```
0x08000222 F000F8E5  BL.W              _sys_exit
     5:          i = 10;
0x08000226 BF00      NOP
     6:          while(1);
0x08000228 BF00      NOP
0x0800022A E7FE      B                 0x0800022A
   190:                LDR       R0, =SystemI
```

```c
int main()
{
int i;

  i = 10;
  while(1);
}
```

```
Load "H:\\PROJECTS2\\PPIUS - Lab - STM\\La
BS \\Test1\main.c\5

>
ASSIGN BreakDisable BreakEnable BreakKill
```

Name	Location/Value	Type
main	0x08000226	int f()
i	<not in scope>	auto - int

At this point we have functional project that compiles, produces executable code, successfully programss FLASH memory of microcontroller, and has the hardware debug capabilities checked by hitting the breakpoint. This is the minimum environment setup that you need in order to start making some useful programs that will use microcontroller peripherals to access external world.

3.2. STM32F4 Standard Peripheral Library

Unlike the earlier revisions, Keil MDK-ARM uVision 5 (uV5) requires user first to install the appropriate microcontroller family DFP in order to use certain microcontroller in a project. For case of STM32F4 microcontroller family, the official Keil DFP comes bundled with *STM32CubeF4 firmware package HAL library*. This is the latest revision of HAL library provided by STMicroelectronics for STM32F4 line of microcontrollers.

However, although the *STM32CubeF4 firmware package HAL* is well integrated into the latest Keil MDK-ARM uVision release through the run-time environment manager, there are some drawbacks for using both this library and run-time environment manager at the moment:

- STM32CubeF4 library is still not as mature as the previous HAL library version,
- examples and community code base support is in the early stage,
- the uVision run-time environment manager performs many „behind the scenes magic“ and generating intermediate files that should not be altered by user, thus imposing non-standard and proprietary workflow that is not compatible with other C compilers and integrated development environments,
- this vendor imposed workflow is also toolset version dependant and „magic“ behind run-time environment manager hides the mechanisms of building component-based solutions composed of loosely-coupled libraries with large number of C files (students should learn how to make this without a help of run-time environment manager).

Therefore, in this lab exercise we shall use an older, stable and better supported HAL layer library (*STM32F4 Standard Peripheral Library*) instead of *STM32CubeF4 firmware package HAL* library, due to codebase maturity, documentation, large number of community-provided examples, etc. We shall not rely on "magic" behind the run-time environment manager, and learn how to setup project environment, resolving all inter-dependancies from the scratch. The *STM32F4 Standard Peripheral Library* provides support for all currently available STM32F4 family microcontrollers.

The library can be downloaded from http://www.st.com/web/en/catalog/tools/PF257901. The *STM32F4 Standard Peripheral Library* is distributed as an open source package.

The easiest way to start using *STM32F4 Standard Peripheral Library* is to use project template contained in

PIES_2014_Lab1_Pack.rar | uVision project templates.rar | uV5.rar | Empty Template - Src

Extract this folder from archive and load the project from *\uVision\Template.uvproj* project file. This project template was built from scratch using the source code provided from the library source and it is portable among machines, without requiring to install the *STM32F4 Standard Peripheral Library* to your local Keil MDK-ARM uVision installation folder. The next chapter will show how to build such project template from the scratch.

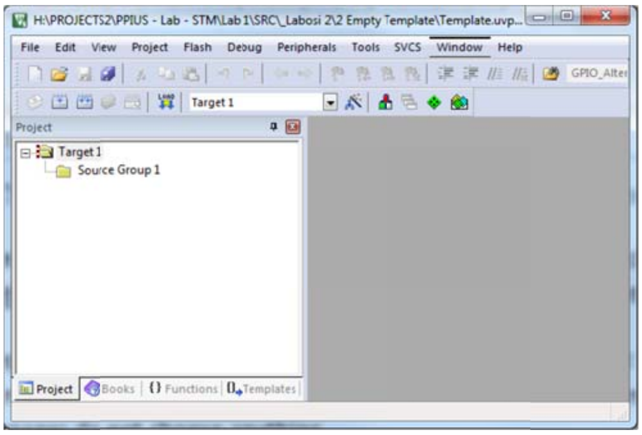
3.3. Creating STM32F4 Standard Peripheral Library based project from source

The first step is to create a new **empty project** based on the chosen STM32F4 microcontroller, but **without including any components** from the run-time environment manager.

Step 1 - Creating empty project for STM32F407VG microcontoller

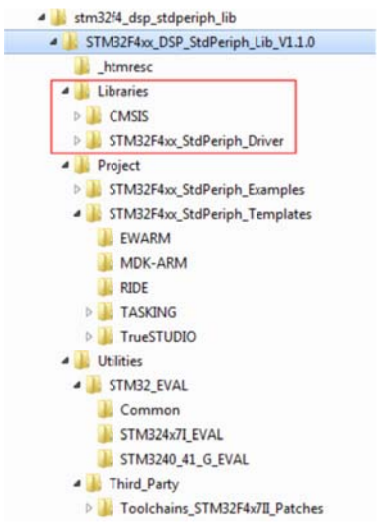
Make a new folder \Template at some arbitrary path, with two subfolders \Template\Libraries and \Template\uVision.

Run option *Project - New μVision Project*; save the project file to *Template*\uVision and name it *Template.uvproj*. Choose STM32F407VG microcontoller. From run-time environment manager **do not choose anything**. Your project should look like this:



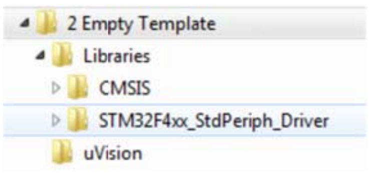
Step 2 - Adding STM32F4 Standard Peripheral Library

Now extract the content of *stm32f4_dsp_stdperiph_lib.zip*[8] to some location (e.g. *C:*\Temp).

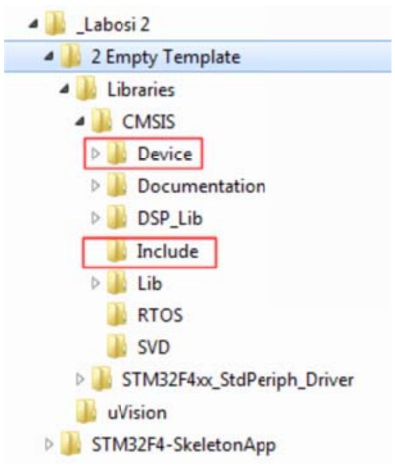

stm32f4_dsp_stdperiph_lib
STM32F4xx_DSP_StdPeriph_Lib_V1.1.0
_htmresc
Libraries
CMSIS
STM32F4xx_StdPeriph_Driver
Project
STM32F4xx_StdPeriph_Examples
STM32F4xx_StdPeriph_Templates
EWARM
MDK-ARM
RIDE
TASKING
TrueSTUDIO
Utilities
STM32_EVAL
Common
STM324x7I_EVAL
STM3240_41_G_EVAL
Third_Party
Toolchains_STM32F4x7II_Patches

[8] Available from http://www.st.com/web/en/catalog/tools/PF257901

Copy folders *CMSIS* and *STM32F4xx_StdPeriph_Driver* from *\Libraries* folder to your project *\Template\Libraries* folder. Now your project folder should look like this:



If you want to minimize the size of your project template to keep only the minimum necessary things for HAL, delete everything from CMSIS folder, except *Device* and *Include* subfolders as noted below:



<u>*Step 3* - Adding necessary files to your project build</u>

Before adding *main.c* you need to copy some additional files.

First copy the file *startup_stm32f40xx.s* from

\Template\Libraries\CMSIS\Device\ST\STM32F4xx\Source\Templates\arm

to your root project folder *\Template*. This file is needed for proper initialization of microcontroller before entering the *main()* function.

Then go to the folder where you have extracted your original archive and copy some files from subfolders in extracted library as follows. If you extracted your library zip file to e.g. from *C:\Temp*, then go to folder

C:\Temp\stm32f4_dsp_stdperiph_lib\STM32F4xx_DSP_StdPeriph_Lib_V1.1.0\Project\STM32F4xx_StdPeriph_Templates

and copy the files:

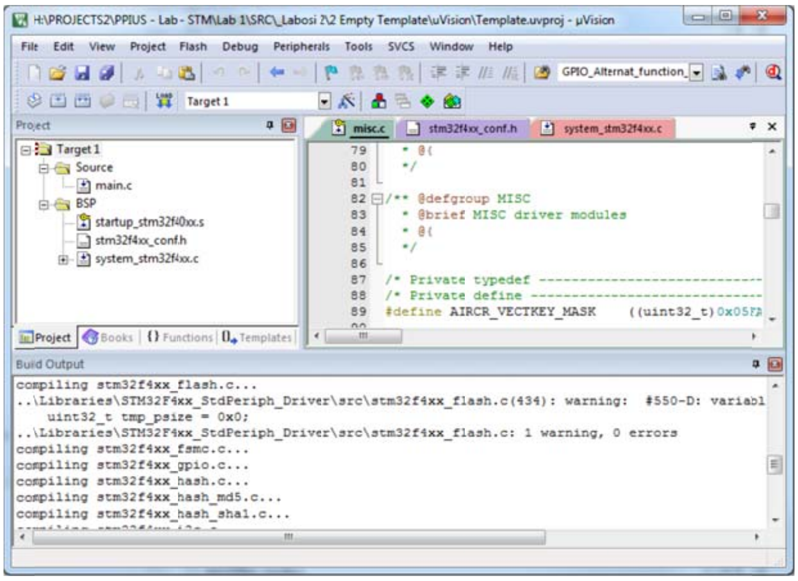
```
stm32f4xx_conf.h
system_stm32f4xx.c
```

to your project root folder *\Template*. Then add empty file *main.c* to root folder *\Template*.

The next step is to include files

```
startup_stm32f40xx.s
stm32f4xx_conf.h
system_stm32f4xx.c
main.c
```

to your project (right-click on node in Project tree, "*Add Existing Files*"). You can group files like this:

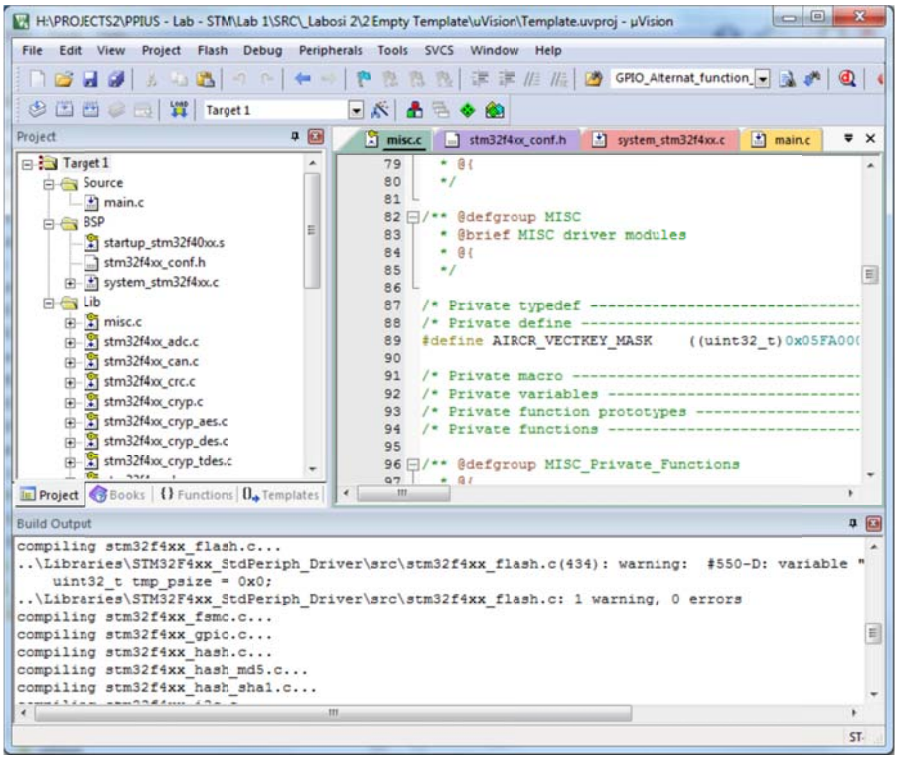


```
    * @{
    */

/** @defgroup MISC
  * @brief MISC driver modules
  * @{
  */

/* Private typedef -----------------------------
/* Private define ------------------------------
#define AIRCR_VECTKEY_MASK    ((uint32_t)0x05FA
```

```
compiling stm32f4xx_flash.c...
..\Libraries\STM32F4xx_StdPeriph_Driver\src\stm32f4xx_flash.c(434): warning:  #550-D: variabl
    uint32_t tmp_psize = 0x0;
..\Libraries\STM32F4xx_StdPeriph_Driver\src\stm32f4xx_flash.c: 1 warning, 0 errors
compiling stm32f4xx_fsmc.c...
compiling stm32f4xx_gpio.c...
compiling stm32f4xx_hash.c...
compiling stm32f4xx_hash_md5.c...
compiling stm32f4xx_hash_sha1.c...
```

Source group contains your application files while *BSP* group contains additional files.

Now you need to add source files of *STM32F4 Standard Peripheral Library*. Make a new group in a tree (e.g. *Lib*) and add all files from folder *\Libraries\STM32F4xx_StdPeriph_Driver\src*:

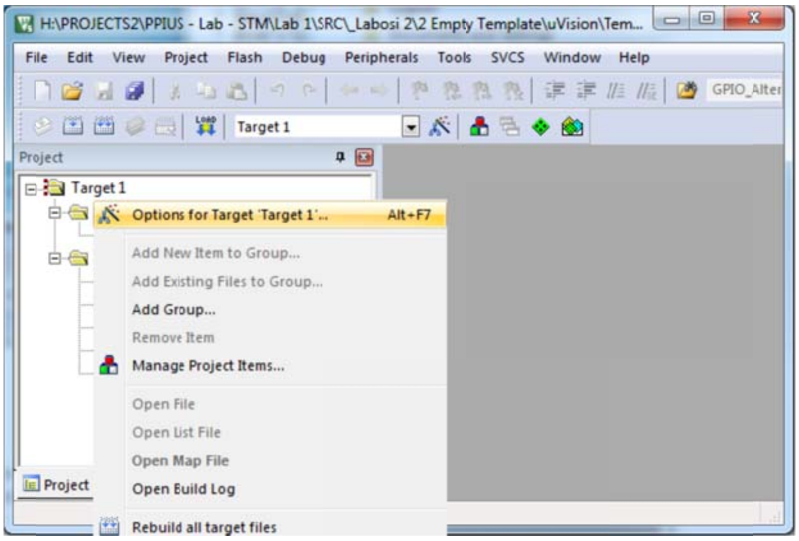


```
    * @{
    */

/** @defgroup MISC
  * @brief MISC driver modules
  * @{
  */

/* Private typedef ---------------------------------
/* Private define ----------------------------------
#define AIRCR_VECTKEY_MASK    ((uint32_t)0x05FA00(

/* Private macro -----------------------------------
/* Private variables -------------------------------
/* Private function prototypes ---------------------
/* Private functions -------------------------------

/** @defgroup MISC_Private_Functions
```

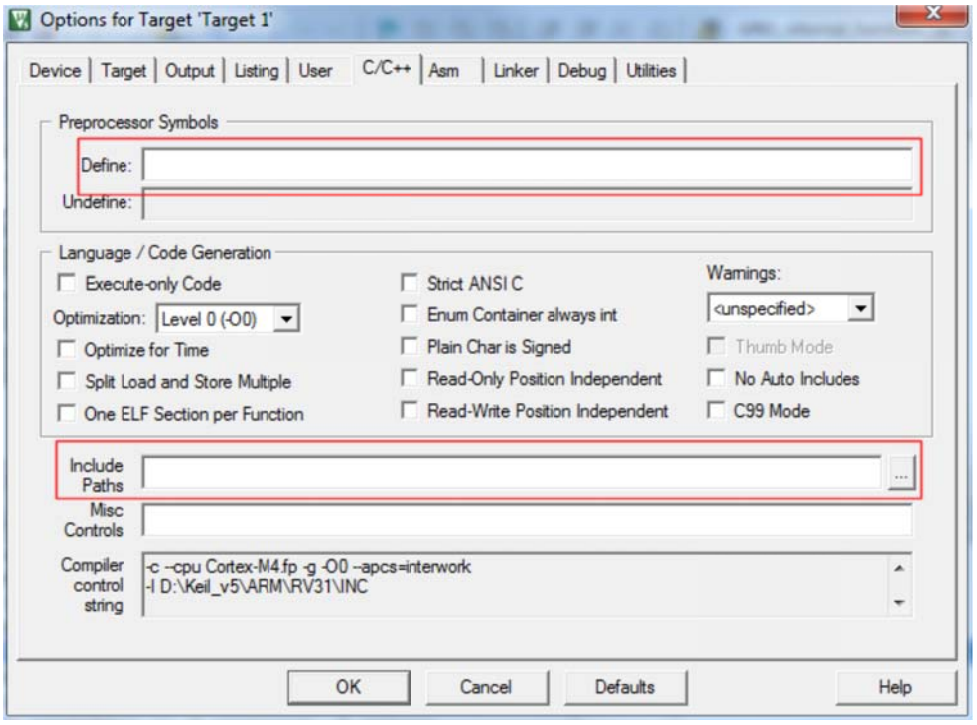
```
compiling stm32f4xx_flash.c...
..\Libraries\STM32F4xx_StdPeriph_Driver\src\stm32f4xx_flash.c(434): warning:  #550-D: variable "
    uint32_t tmp_psize = 0x0;
..\Libraries\STM32F4xx_StdPeriph_Driver\src\stm32f4xx_flash.c: 1 warning, 0 errors
compiling stm32f4xx_fsmc.c...
compiling stm32f4xx_gpio.c...
compiling stm32f4xx_hash.c...
compiling stm32f4xx_hash_md5.c...
compiling stm32f4xx_hash_sha1.c...
```

Step 4 - Setting up build options

At this point your project will not build properly until you configure some additional settings. Right click on *Target 1*, choose *Options for Target*:


H:\PROJECTS2\PPIUS - Lab - STM\Lab 1\SRC_Labosi 2\2 Empty Template\uVision\Tem...
File Edit View Project Flash Debug Peripherals Tools SVCS Window Help
Target 1
Project
Target 1
Options for Target 'Target 1'... Alt+F7
Add New Item to Group...
Add Existing Files to Group...
Add Group...
Remove Item
Manage Project Items...
Open File
Open List File
Open Map File
Open Build Log
Rebuild all target files

You need to set up two fields:


Options for Target 'Target 1'
Device Target Output Listing User C/C++ Asm Linker Debug Utilities
Preprocessor Symbols
Define:
Undefine:
Language / Code Generation
Execute-only Code
Optimization: Level 0 (-O0)
Optimize for Time
Split Load and Store Multiple
One ELF Section per Function
Strict ANSI C
Enum Container always int
Plain Char is Signed
Read-Only Position Independent
Read-Write Position Independent
Warnings: <unspecified>
Thumb Mode
No Auto Includes
C99 Mode
Include Paths
Misc Controls
Compiler control string
OK Cancel Defaults Help

```
-c --cpu Cortex-M4.fp -g -O0 --apcs=interwork
-I D:\Keil_v5\ARM\RV31\INC
```

In *Preprocessor Symbols* add the following definitions:

```
USE_STDPERIPH_DRIVER, STM32F4XX
```

should be
STM32F40_41xxx

These preprocessor symbols are needed to properly compile *Peripheral Library*.

GPIO, Timer and USART project here should be included path that leads to the gpio.h, timer.h and usart.h files

add this folder to the path: DEVICE\

In *Include Paths* add the following string:

```
..\;..\Libraries\CMSIS\Include;..\Libraries\CMSIS\ST\STM32F4xx\Include;..\Libraries
\STM32F4xx_StdPeriph_Driver\inc
```

This will add the following (relative) folders where compiler will look for include files:

```
..\
..\Libraries\CMSIS\Include
..\Libraries\CMSIS\Device\ST\STM32F4xx\Include
..\Libraries\STM32F4xx_StdPeriph_Driver\inc
```

If include files search path is not specified, the compiler will not know where to look for header files.

Adjust setting on *Debug* and *Utilities* tabs to use STLink2 debuger interface as explained before.

Add some minimum code to *main.c*:

```
int main()
{
	int i = 0;
	while(1);
}
```

Build the project. If you properly followed the instructions, the project should build without errors (except for few non-critical warnings):

fmc.c file should be deleted before Building

```
linking...
Program Size: Code=880 RO-data=424 RW-data=20 ZI-data=1636
".\Template.axf" - 0 Error(s), 3 Warning(s).
```

Congratulations! At this point you have your own fully functional project template!

Note: During the development, sometimes it is needed to execute the full rebuild of all library project source files. This can be avoided if you build *STM32F4 Standard Peripheral Library* from source to C library (*.lib) file (by changing the output from *Executable* to *Library* on *Output* tab for *Target Options*). In this case library (lib file) contains relocatable object code and it is not necessary to keep original C source files in your source tree. The linker will use only header files and link functions against object code precompiled in library, without a need for rebuilding all library source files. The version of project template that uses precompiled library is given in

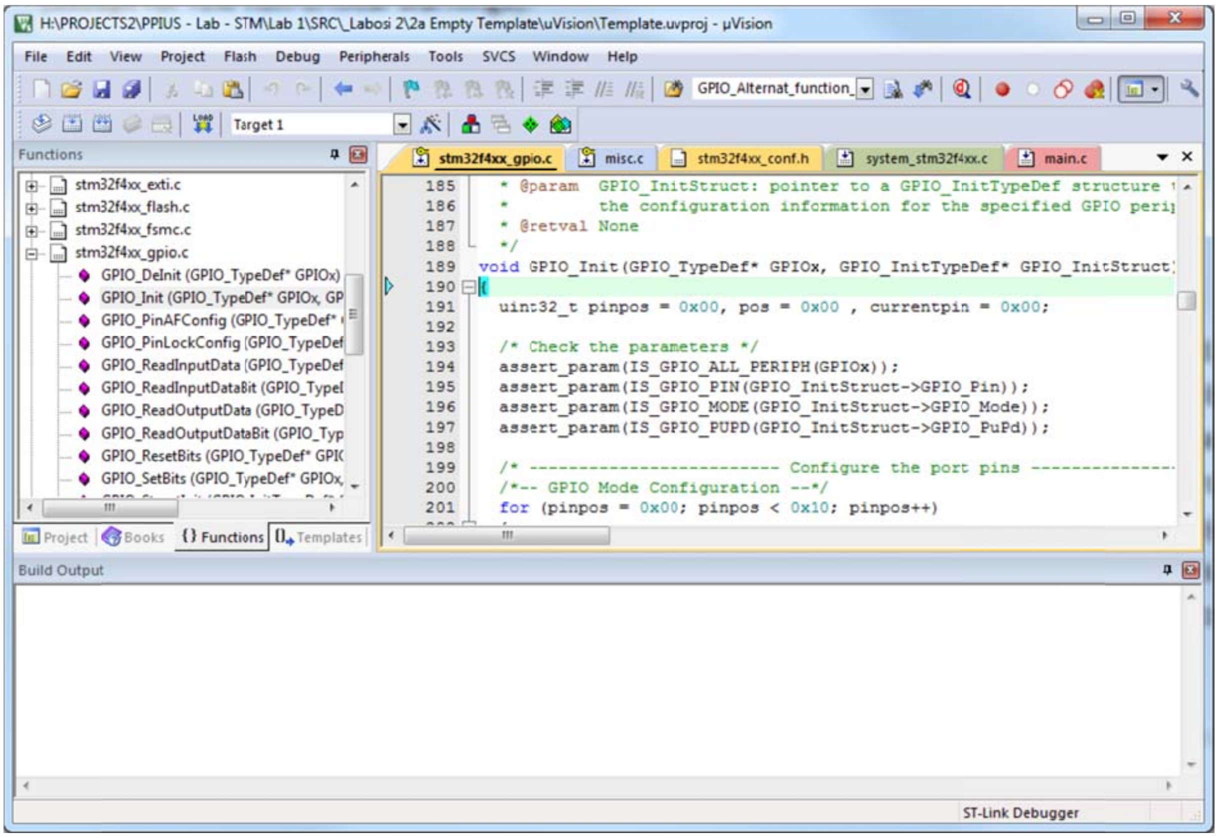
PIES_2014_Lab1_Pack.rar | uVision project templates.rar | uV5.rar | Empty Template - Lib

and it follows exactly the same structure as described above, except for using lib file instead of individual C sources.

Tip: If you use the template with full HAL library source code, given in

PIES_2014_Lab1_Pack.rar | uVision project templates.rar | uV5.rar | Empty Template - Src

you will have an easy access to whole HAL API through Keil MDK-ARM uVision IDE:


H:\PROJECTS2\PPIUS - Lab - STM\Lab 1\SRC_Labosi 2\2a Empty Template\uVision\Template.uvproj - µVision
File
Edit
View
Project
Flash
Debug
Peripherals
Tools
SVCS
Window
Help
GPIO_Alternat_function_
Target 1
Functions
stm32f4xx_exti.c
stm32f4xx_flash.c
stm32f4xx_fsmc.c
stm32f4xx_gpio.c
GPIO_DeInit (GPIO_TypeDef* GPIOx)
GPIO_Init (GPIO_TypeDef* GPIOx, GP
GPIO_PinAFConfig (GPIO_TypeDef*
GPIO_PinLockConfig (GPIO_TypeDef
GPIO_ReadInputData (GPIO_TypeDef
GPIO_ReadInputDataBit (GPIO_TypeI
GPIO_ReadOutputData (GPIO_TypeD
GPIO_ReadOutputDataBit (GPIO_Typ
GPIO_ResetBits (GPIO_TypeDef* GPIC
GPIO_SetBits (GPIO_TypeDef* GPIOx,
Project
Books
Functions
Templates
stm32f4xx_gpio.c
misc.c
stm32f4xx_conf.h
system_stm32f4xx.c
main.c
Build Output
ST-Link Debugger

```
  * @param  GPIO_InitStruct: pointer to a GPIO_InitTypeDef structure t
  *         the configuration information for the specified GPIO peri
  * @retval None
  */
void GPIO_Init(GPIO_TypeDef* GPIOx, GPIO_InitTypeDef* GPIO_InitStruct)
{
  uint32_t pinpos = 0x00, pos = 0x00 , currentpin = 0x00;

  /* Check the parameters */
  assert_param(IS_GPIO_ALL_PERIPH(GPIOx));
  assert_param(IS_GPIO_PIN(GPIO_InitStruct->GPIO_Pin));
  assert_param(IS_GPIO_MODE(GPIO_InitStruct->GPIO_Mode));
  assert_param(IS_GPIO_PUPD(GPIO_InitStruct->GPIO_PuPd));

  /* ------------------------ Configure the port pins ---------------
  /*-- GPIO Mode Configuration --*/
  for (pinpos = 0x00; pinpos < 0x10; pinpos++)
```

If you click on *Functions* tab in a *Project* window, the tree structure (kind of "*function browser*") will be shown, with list of all functions contained in each C source file. By expanding each node (C source) you can see all functions in that file, and access the source code of each function by double-clicking on it. *Functions* tab provides quick and convenient way to learn library API or just to quickly browse your application code.

In all subsequent examples, it is assumed that you use either one of the project templates described above, or you make your own based on the same procedure.

4. General-purpose input/output (GPIO) ports

The introductory "*Hello World*" programming example counterpart in a microcontroller world is usually called "*Blinky*". Considering that there is no readily available console output for your *printf* statements to check the program behaviour (until you make youself one), it is usually the first step to light some LED(s) on and off, by changing the logic levels at GPIO pins. In this part of exercise we shall learn how to control LED states (GPIO levels) by using HAL-level STM32F4 library.

Assignment:

Make Blinky program that will use HAL-level library (*STM32F4 Standard Peripheral Library*) to blink on-board LEDs.

Guidelines:

4.1. Step 1: Create a new project

The easiest way to create a new project is to copy one of project templates described in previous chapter to some folder (e.g. ..\Lab_GPIO). Add an empty *main.c* file to your project root folder.

4.2. Step 2: Determine the GPIO pins connected to on-board LEDs

In order to control the on-board LEDs, it is first necessary to determine to which GPIO pins these LEDs are connected. STM32F4DISCOVERY board user manual[9] defines LEDs positions on page 16. We are concerned with four user-controlled LEDs:

User LED ID	color	Port pin
LD3	orange	PD13
LD4	green	PD12
LD5	red	PD14
LD6	blue	PD15

4.3. Step 3: Initialization of GPIO interface

In *main.c* add the following lines:

```
int main(void)
{
   while(1);
}
```

Each microcontroller program must not exit the *main()* function because there is no other program environment (e.g. OS) where this function should "return". Moreover, the microcontroller program without interaction with peripherals rarely can serve some useful purpose so the next step would be to add support for GPIO peripheral, as an example of the simplest peripheral.

[9] *UM1472 - Discovery kit for STM32F407.pdf*

Programming each microcontroller requires reading a great deal of literature before even getting started. Relevant datasheets are usually organized in a way that there is (a) user guide covering the microcontroller family[10], (b) datasheet for specific microcontroller part[11], and (c) the programming manual[12]. These manuals and datasheets describe microcontroller hardware and low-level programming interface, without any references to HAL library. This is the prerequisite to understand how to use the HAL library, which is described in a separate manual[13].

Although the amount of information in all mentioned relevant documents can be intimidating on a first sight, these materials do not have to be read completely before getting started. A good way to speed-up the initial application development is to study examples provided on installation path

x:\Keil_v5\ARM\Pack\Keil\STM32F4xx_DFP\2.2.0\Projects

It is also useful to take a look on *Functions* tab in a project template with the whole HAL library included as a C source, to quickly learn the most important API functions.

In our new project, first we have to include HAL support for GPIO configuration. We shall do GPIO initialization in *gpio_init()* function:

```
#include <stm32f4xx.h>       // common stuff
#include <stm32f4xx_gpio.h>  // gpio control
#include <stm32f4xx_rcc.h>   // reset and clocking

void gpio_init()
{
}

int main(void)
{
        gpio_init();
        while(1);
}
```

It is intuitive and reasonable to assume that the header file *stm32f4xx_gpio.h* will contain all function prototypes needed for GPIO control. However, as we shall see soon, it is also necessary to include *stm32f4xx_rcc.h* header (*Reset and clock control* (RCC) peripheral), because each GPIO port (more generally, every peripheral) has to be explicitely assigned a clock to be functional. Moreover, a common header containing important definitions *stm32f4xx.h* should always be included.

Most of simple microcontrollers (such as 8051, AVR, PIC, MSP430 etc.) are easily configured simply by writing hexadecimal values into corresponding hardware registers. Such approach is possible with ARM microcontrollers too, but since ARM microcontrollers are much more complex, the register-level application programming tends to be tedious, time-consuming and error-prone process. Thus, HAL-level library is a preferred way of application development that provides faster development with less bugs. STM32F4 HAL libraries (both *STM32 Cube* and *STM32F4 Standard Peripheral Library*)

[10] RM0090 Reference manual STM32F405xx/07xx, STM32F415xx/17xx, STM32F42xxx and STM32F43xxx advanced ARM®-based 32-bit MCUs (*STM32F4 Reference manual.pdf*)
[11] STM32F405xx / STM32F407xx datasheet (*STM32F407VG.pdf*)
[12] PM0214 Programming manual STM32F3 and STM32F4 Series Cortex®-M4 programming manual (*STM32F3 and STM32F4 Series Cortex-M4 programming manual.pdf*).
[13] UM1061 Description of STM32F2xx Standard Peripheral Library (*Description of STM32F2xx Standard Peripheral Library.pdf*)

are open-source and user can always see what is going on „behind the scenes“. When necessary, it is always possible to mix HAL- and register-level programming style.

Let us take a closer look on *gpio_init()* function. First we shall configure the orange LED GPIO pin control. After the reset, all GPIO pins are in inactive state (open drain) and GPIO ports are not clocked. We have to (a) provide clock for the chosen GPIO port, and (b) configure the chosen pin settings. This is achieved by the following code:

```
void gpio_init()
{
GPIO_InitTypeDef GPIO_InitStruct;

   RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOD, ENABLE);
   GPIO_InitStruct.GPIO_Pin   = GPIO_Pin_13;
   GPIO_InitStruct.GPIO_Mode  = GPIO_Mode_OUT;
   GPIO_InitStruct.GPIO_Speed = GPIO_Speed_50MHz;
   GPIO_InitStruct.GPIO_OType = GPIO_OType_PP;
   GPIO_InitStruct.GPIO_PuPd  = GPIO_PuPd_NOPULL;
   GPIO_Init(GPIOD, &GPIO_InitStruct);
}
```

Let us see what each part of code above means. As one can notice, we do not access processor registers directly, neither we write any "magic" hex number constants to these registers, and we use the high level HAL approach instead. First let us see the definition of *GPIO_InitTypeDef* structure (in header file *stm32f4xx_gpio.h*), which is rather self-explanatory:

```
typedef struct
{
  uint32_t GPIO_Pin;              /*!< Specifies the GPIO pins to be configured.
                                       This parameter can be any value of @ref GPIO_pins_define */
  GPIOMode_TypeDef GPIO_Mode;     /*!< Specifies the operating mode for the selected pins.
                                       This parameter can be a value of @ref GPIOMode_TypeDef */
  GPIOSpeed_TypeDef GPIO_Speed;   /*!< Specifies the speed for the selected pins.
                                       This parameter can be a value of @ref GPIOSpeed_TypeDef */
  GPIOOType_TypeDef GPIO_OType;   /*!< Specifies the operating output type for the selected pins.
                                       This parameter can be a value of @ref GPIOOType_TypeDef */
  GPIOPuPd_TypeDef GPIO_PuPd;     /*!< Specifies the operating Pull-up/Pull down for the selected pins.
                                       This parameter can be a value of @ref GPIOPuPd_TypeDef */
} GPIO_InitTypeDef;
```

The *GPIO_InitTypeDef* structure references some typedefs:

```
typedef enum
{
  GPIO_Mode_IN   = 0x00, /*!< GPIO Input Mode */
  GPIO_Mode_OUT  = 0x01, /*!< GPIO Output Mode */
  GPIO_Mode_AF   = 0x02, /*!< GPIO Alternate function Mode */
  GPIO_Mode_AN   = 0x03  /*!< GPIO Analog Mode */
} GPIOMode_TypeDef;


typedef enum
{
  GPIO_Speed_2MHz   = 0x00, /*!< Low speed */
  GPIO_Speed_25MHz  = 0x01, /*!< Medium speed */
  GPIO_Speed_50MHz  = 0x02, /*!< Fast speed */
  GPIO_Speed_100MHz = 0x03  /*!< High speed on 30 pF (80 MHz Output max speed on 15 pF) */
} GPIOSpeed_TypeDef;


typedef enum
{
  GPIO_OType_PP = 0x00,
  GPIO_OType_OD = 0x01
} GPIOOType_TypeDef;


typedef enum
{
  GPIO_PuPd_NOPULL = 0x00,
  GPIO_PuPd_UP     = 0x01,
  GPIO_PuPd_DOWN   = 0x02
} GPIOPuPd_TypeDef;
```

Pins that are used in a structure are defined via pin masks:

```
#define GPIO_Pin_0                 ((uint16_t)0x0001)  /* Pin 0 selected */
#define GPIO_Pin_1                 ((uint16_t)0x0002)  /* Pin 1 selected */
#define GPIO_Pin_2                 ((uint16_t)0x0004)  /* Pin 2 selected */
#define GPIO_Pin_3                 ((uint16_t)0x0008)  /* Pin 3 selected */
#define GPIO_Pin_4                 ((uint16_t)0x0010)  /* Pin 4 selected */
#define GPIO_Pin_5                 ((uint16_t)0x0020)  /* Pin 5 selected */
#define GPIO_Pin_6                 ((uint16_t)0x0040)  /* Pin 6 selected */
#define GPIO_Pin_7                 ((uint16_t)0x0080)  /* Pin 7 selected */
#define GPIO_Pin_8                 ((uint16_t)0x0100)  /* Pin 8 selected */
#define GPIO_Pin_9                 ((uint16_t)0x0200)  /* Pin 9 selected */
#define GPIO_Pin_10                ((uint16_t)0x0400)  /* Pin 10 selected */
#define GPIO_Pin_11                ((uint16_t)0x0800)  /* Pin 11 selected */
#define GPIO_Pin_12                ((uint16_t)0x1000)  /* Pin 12 selected */
#define GPIO_Pin_13                ((uint16_t)0x2000)  /* Pin 13 selected */
#define GPIO_Pin_14                ((uint16_t)0x4000)  /* Pin 14 selected */
#define GPIO_Pin_15                ((uint16_t)0x8000)  /* Pin 15 selected */

#define GPIO_Pin_All               ((uint16_t)0xFFFF)  /* All pins selected */
```

Before considering how to use these typedefs and defines, let us see the function call *RCC_AHB1PeriphClockCmd*:

```
RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOD, ENABLE);
```

This function is contained in module *stm32f4xx_rcc.c*. If we open this source file, we shall see the function definition:

```
/**
  * @brief  Enables or disables the AHB1 peripheral clock.
  * @note   After reset, the peripheral clock (used for registers read/write access)
  *         is disabled and the application software has to enable this clock before
  *         using it.
  * @param  RCC_AHBPeriph: specifies the AHB1 peripheral to gates its clock.
  *          This parameter can be any combination of the following values:
  *            @arg RCC_AHB1Periph_GPIOA:       GPIOA clock
  *            @arg RCC_AHB1Periph_GPIOB:       GPIOB clock
  *            @arg RCC_AHB1Periph_GPIOC:       GPIOC clock
  *            @arg RCC_AHB1Periph_GPIOD:       GPIOD clock
  *            @arg RCC_AHB1Periph_GPIOE:       GPIOE clock
  *            @arg RCC_AHB1Periph_GPIOF:       GPIOF clock
  *            @arg RCC_AHB1Periph_GPIOG:       GPIOG clock
  *            @arg RCC_AHB1Periph_GPIOG:       GPIOG clock
  *            @arg RCC_AHB1Periph_GPIOI:       GPIOI clock
  *            @arg RCC_AHB1Periph_CRC:         CRC clock
  *            @arg RCC_AHB1Periph_BKPSRAM:     BKPSRAM interface clock
  *            @arg RCC_AHB1Periph_CCMDATARAMEN CCM data RAM interface clock
  *            @arg RCC_AHB1Periph_DMA1:        DMA1 clock
  *            @arg RCC_AHB1Periph_DMA2:        DMA2 clock
  *            @arg RCC_AHB1Periph_ETH_MAC:     Ethernet MAC clock
  *            @arg RCC_AHB1Periph_ETH_MAC_Tx:  Ethernet Transmission clock
  *            @arg RCC_AHB1Periph_ETH_MAC_Rx:  Ethernet Reception clock
  *            @arg RCC_AHB1Periph_ETH_MAC_PTP: Ethernet PTP clock
  *            @arg RCC_AHB1Periph_OTG_HS:      USB OTG HS clock
  *            @arg RCC_AHB1Periph_OTG_HS_ULPI: USB OTG HS ULPI clock
  * @param  NewState: new state of the specified peripheral clock.
  *          This parameter can be: ENABLE or DISABLE.
  * @retval None
  */

void RCC_AHB1PeriphClockCmd(uint32_t RCC_AHB1Periph, FunctionalState NewState)
```

This function serves to enable or disable clock for some peripheral connected to AHB1 bus interface. Since the orange LED is connected to the Port D, which is connected to the AHB1 bus, we need to use this function to provide peripheral clock. Please note that there are also other peripheral clock enable functions, depending on bus interface:

```
void RCC_AHB2PeriphClockCmd(uint32_t RCC_AHB2Periph, FunctionalState NewState)
void RCC_AHB3PeriphClockCmd(uint32_t RCC_AHB3Periph, FunctionalState NewState)
void RCC_AHB3PeriphClockCmd(uint32_t RCC_AHB3Periph, FunctionalState NewState)
void RCC_APB1PeriphClockCmd(uint32_t RCC_APB1Periph, FunctionalState NewState) etc.
```

These functions have exactly the same meaning but they affect peripherals connected to their respective bus interfaces. For example, Port D cannot be enabled clock with function *RCC_AHB2PeriphClockCmd* because it is not connected to AHB2 interface. For each peripheral consult the microcontroller documentation and/or comments in *stm32f4xx_rcc.c* to determine to which bus interface every peripheral is connected.

In function call

```
RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOD, ENABLE);
```

the first argument is defined in *stm32f4xx_rcc.h* as a symbolic constant:

```
/** @defgroup RCC_AHB1_Peripherals
  * @{
  */
#define RCC_AHB1Periph_GPIOA             ((uint32_t)0x00000001)
#define RCC_AHB1Periph_GPIOB             ((uint32_t)0x00000002)
#define RCC_AHB1Periph_GPIOC             ((uint32_t)0x00000004)
#define RCC_AHB1Periph_GPIOD             ((uint32_t)0x00000008)
#define RCC_AHB1Periph_GPIOE             ((uint32_t)0x00000010)
#define RCC_AHB1Periph_GPIOF             ((uint32_t)0x00000020)
#define RCC_AHB1Periph_GPIOG             ((uint32_t)0x00000040)
#define RCC_AHB1Periph_GPIOH             ((uint32_t)0x00000080)
#define RCC_AHB1Periph_GPIOI             ((uint32_t)0x00000100)
#define RCC_AHB1Periph_CRC               ((uint32_t)0x00001000)
#define RCC_AHB1Periph_FLITF             ((uint32_t)0x00008000)
#define RCC_AHB1Periph_SRAM1             ((uint32_t)0x00010000)
#define RCC_AHB1Periph_SRAM2             ((uint32_t)0x00020000)
#define RCC_AHB1Periph_BKPSRAM           ((uint32_t)0x00040000)
etc.
```

Almost all function arguments in HAL rely either on custom typedef structures or symbolic constants defines like in this case. Typdefs and constants are defined in header files (along with API function interfaces), while function implementations are in C module files (along with detailed comments how to use these functions).

Hint: Although the document "*UM1061 Description of STM32F2xx Standard Peripheral Library*" provides detailed definition of the whole HAL API, it is easier to locate functions and explanations using project template and *Functions* tab.

ENABLE is defined as an enumerated constant in *stm32f4xx.h*:

```
typedef enum {DISABLE = 0, ENABLE = !DISABLE} FunctionalState;
```

After enabling the clock to Port D, it is necessary to initialize GPIO descriptor structure:

```
GPIO_InitStruct.GPIO_Pin      = GPIO_Pin_13;
GPIO_InitStruct.GPIO_Mode     = GPIO_Mode_OUT;
GPIO_InitStruct.GPIO_Speed    = GPIO_Speed_50MHz;
GPIO_InitStruct.GPIO_OType    = GPIO_OType_PP;
GPIO_InitStruct.GPIO_PuPd     = GPIO_PuPd_NOPULL;
```

In this case we want to initialize pin PD13 as output, push-pull type (instead of open-drain), with no internal pull-up or pull-down resistor. We set fast I/O speed (50 MHz setting). Slower speeds are desirable in cases when speed is not needed to mitigate electromagnetic intereference (EMI) issues due to fast rise time of output signal.

After the settings in GPIO descriptor structure are configured, we apply them to the chosen GPIO port (Port D in this case) and pin (pin 13) using the call to *GPIO_Init* API function:

```
GPIO_Init(GPIOD, &GPIO_InitStruct);
```

The prototype definition of functio *GPIO_Init* can be found in *stm32f4xx_gpio.h*:

```
void GPIO_Init(GPIO_TypeDef* GPIOx, GPIO_InitTypeDef* GPIO_InitStruct);
```

The first argument is of type *GPIO_TypeDef*. Data type *GPIO_TypeDef* can be found in header *stm32f4xx.h*:

```
typedef struct
{
  __IO uint32_t MODER;    /*!< GPIO port mode register,               Address offset: 0x00      */
  __IO uint32_t OTYPER;   /*!< GPIO port output type register,        Address offset: 0x04      */
  __IO uint32_t OSPEEDR;  /*!< GPIO port output speed register,       Address offset: 0x08      */
  __IO uint32_t PUPDR;    /*!< GPIO port pull-up/pull-down register,  Address offset: 0x0C      */
  __IO uint32_t IDR;      /*!< GPIO port input data register,         Address offset: 0x10      */
  __IO uint32_t ODR;      /*!< GPIO port output data register,        Address offset: 0x14      */
  __IO uint16_t BSRRL;    /*!< GPIO port bit set/reset low register,  Address offset: 0x18      */
  __IO uint16_t BSRRH;    /*!< GPIO port bit set/reset high register, Address offset: 0x1A      */
  __IO uint32_t LCKR;     /*!< GPIO port configuration lock register, Address offset: 0x1C      */
  __IO uint32_t AFR[2];   /*!< GPIO alternate function registers,     Address offset: 0x20-0x24 */
} GPIO_TypeDef;
```

The C structure *GPIO_InitTypeDef* contains logical fields which are other typedefs, enums or defines, and has not direct connection with microcontroller hardware registers. The meaning of *GPIO_TypeDef* structure is slightly different as it maps *directly* to hardware addresses of registers controlling GPIO port. The *GPIO_TypeDef* structure contains an "addressing template" with appropriate offsets and enables access to hardware registers conveniently through the C struct fields. Since e.g. all port control register banks have the same structure in memory address space, it is sufficient to provide base address of each port control register bank to map appropriate port with symbolic identifier. For example, the port D is accessed through the *GPIO_TypeDef*-typed pointer on predefined address:

```
#define GPIOD                 ((GPIO_TypeDef *) GPIOD_BASE)
```

In code, the symbol *GPIOD* refers to the typed pointer pointing to the address *GPIOD_BASE*, which is defined as:

```
/*!< Peripheral base address in the alias region */
#define PERIPH_BASE           ((uint32_t)0x40000000)
#define AHB1PERIPH_BASE       (PERIPH_BASE + 0x00020000)
#define GPIOD_BASE            (AHB1PERIPH_BASE + 0x0C00)
```

HAL API provides convenient mechanism of abstraction and relieves the programmer a burden of calculating the addresses of each individual register by hand. The same principles are used in all other parts of the HAL library.

4.4. Step 4: Avoiding hard-coded constants via #define statements

If we take a look on a code

```
void gpio_init()
{
GPIO_InitTypeDef GPIO_InitStruct;

   RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOD, ENABLE);
   GPIO_InitStruct.GPIO_Pin   = GPIO_Pin_13;
   GPIO_InitStruct.GPIO_Mode  = GPIO_Mode_OUT;
   GPIO_InitStruct.GPIO_Speed = GPIO_Speed_50MHz;
   GPIO_InitStruct.GPIO_OType = GPIO_OType_PP;
   GPIO_InitStruct.GPIO_PuPd  = GPIO_PuPd_NOPULL;
   GPIO_Init(GPIOD, &GPIO_InitStruct);
}
```

we can see that it would be rather tedious and error-prone to change e.g. LED location from PD13 to PA10. The problem is even more emphasized when the number of GPIO pins rises. Therefore, it is a good programming practice to avoid any hard-coded constants in your application code and move all hardware-specific references to #define statements. Let us consider the following code:

```
#include <stm32f4xx_gpio.h>                 // gpio control
#include <stm32f4xx_rcc.h>                  // reset anc clocking

#define LED3_ORANGE_RCC_GPIOx               RCC_AHB1Periph_GPIOD
#define LED3_ORANGE_GPIOx                   GPIOD
#define LED3_ORANGE_PinNumber               GPIO_Pin_13

void gpio_init()
{
GPIO_InitTypeDef GPIO_InitStruct;

   RCC_AHB1PeriphClockCmd(LED3_ORANGE_RCC_GPIOx, ENABLE);
   GPIO_InitStruct.GPIO_Pin   = LED3_ORANGE_PinNumber;
   GPIO_InitStruct.GPIO_Mode  = GPIO_Mode_OUT;
   GPIO_InitStruct.GPIO_Speed = GPIO_Speed_50MHz;
   GPIO_InitStruct.GPIO_OType = GPIO_OType_PP;
   GPIO_InitStruct.GPIO_PuPd  = GPIO_PuPd_NOPULL;
   GPIO_Init(LED3_ORANGE_GPIOx, &GPIO_InitStruct);
}

int main(void)
{
       gpio_init();
       while(1);
}
```

The code is almost the same like in the previous version but nothing in application code is now hard-coded. There is a single place where you change your port pin definitions. For example, if you want to move orange LED to some external LED connected on PA10, this can be achieved without affecting the application code at all, only #define statements:

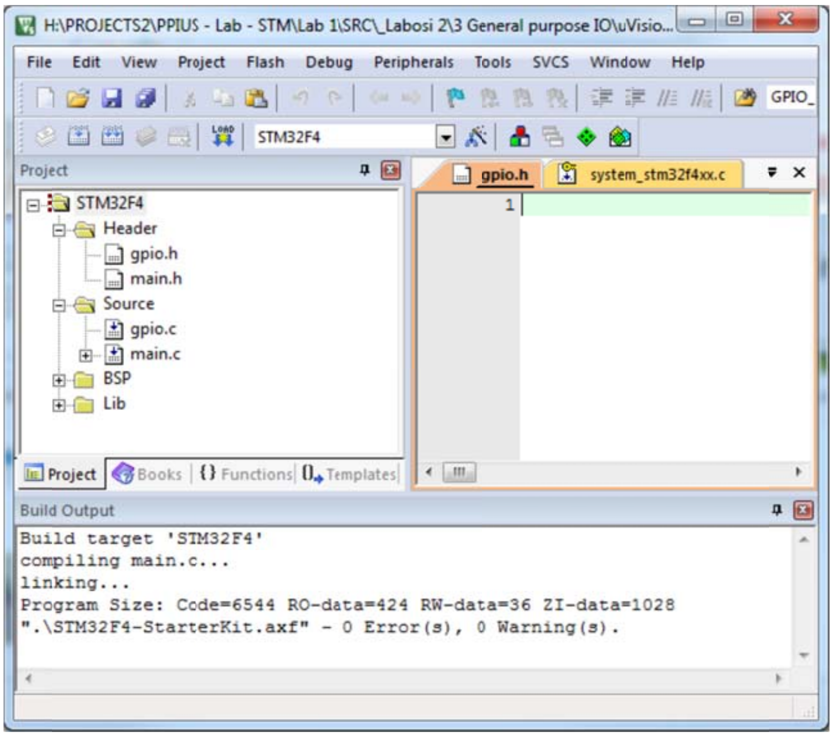
```
#define LED3_ORANGE_RCC_GPIOx               RCC_AHB1Periph_GPIOA
#define LED3_ORANGE_GPIOx                   GPIOA
#define LED3_ORANGE_PinNumber               GPIO_Pin_10
```

The application code is unaffected by this change and program is now easy to modify and maintain.

4.5. Step 5: Appropriate use of header files and module decomposition to build larger programs

It is a good programming practice to keep different functional parts of programs separated. Each module should have well defined functionality and consistent interface for easy interfacing with other modules. We shall demonstrate how to decouple GPIO initialization from the main program module and how to import this functionality into main module.

First we need to add to project two source files *main.c* (main application) and *gpio.c* (GPIO module). In order to consistently reference C modules among each other, the accompanying header files *main.h* and *gpio.h* have to be created and added to the project. It is good practice to keep C sources and headers under different groups, for example:



```
Build target 'STM32F4'
compiling main.c...
linking...
Program Size: Code=6544 RO-data=424 RW-data=36 ZI-data=1028
".\STM32F4-StarterKit.axf" - 0 Error(s), 0 Warning(s).
```

Code in header files and C modules is given below:

```
/* gpio.h */
#ifndef GPIO_H
#define GPIO_H

#include <stm32f4xx.h>                                     // common stuff
#include <stm32f4xx_gpio.h>                                // gpio control
#include <stm32f4xx_rcc.h>                                 // reset anc clocking

#define LED3_ORANGE_RCC_GPIOx                     RCC_AHB1Periph_GPIOD
#define LED4_GREEN_RCC_GPIOx                      RCC_AHB1Periph_GPIOD
#define LED5_RED_RCC_GPIOx                        RCC_AHB1Periph_GPIOD
#define LED6_BLUE_RCC_GPIOx                       RCC_AHB1Periph_GPIOD

#define LED3_ORANGE_GPIOx                         GPIOD
#define LED4_GREEN_GPIOx                          GPIOD
#define LED5_RED_GPIOx                            GPIOD
#define LED6_BLUE_GPIOx                           GPIOD

#define LED3_ORANGE_PinNumber                     GPIO_Pin_13
#define LED4_GREEN_PinNumber                      GPIO_Pin_12
#define LED5_RED_PinNumber                        GPIO_Pin_14
#define LED6_BLUE_PinNumber                       GPIO_Pin_15

void gpio_init(void);

#endif
```

```
/* main.h */
#ifndef MAIN_H
#define MAIN_H

#include <stm32f4xx.h>                                      // common stuff
#include <stm32f4xx_gpio.h>                                 // gpio control
#include <stm32f4xx_rcc.h>                                  // reset anc clocking
#include <gpio.h>

#endif
```

```
/* gpio.c */
#include <gpio.h>

void gpio_init()
{
GPIO_InitTypeDef         GPIO_InitStruct;

        RCC_AHB1PeriphClockCmd(LED3_ORANGE_RCC_GPIOx, ENABLE);
        GPIO_InitStruct.GPIO_Pin          = LED3_ORANGE_PinNumber;
        GPIO_InitStruct.GPIO_Mode         = GPIO_Mode_OUT;
        GPIO_InitStruct.GPIO_Speed        = GPIO_Speed_50MHz;
        GPIO_InitStruct.GPIO_OType        = GPIO_OType_PP;
        GPIO_InitStruct.GPIO_PuPd         = GPIO_PuPd_NOPULL;
        GPIO_Init(LED3_ORANGE_GPIOx, &GPIO_InitStruct);

        RCC_AHB1PeriphClockCmd(LED4_GREEN_RCC_GPIOx, ENABLE);
        GPIO_InitStruct.GPIO_Pin          = LED4_GREEN_PinNumber;
        GPIO_Init(LED4_GREEN_GPIOx, &GPIO_InitStruct);

        RCC_AHB1PeriphClockCmd(LED5_RED_RCC_GPIOx, ENABLE);
        GPIO_InitStruct.GPIO_Pin          = LED5_RED_PinNumber;
        GPIO_Init(LED5_RED_GPIOx, &GPIO_InitStruct);

        RCC_AHB1PeriphClockCmd(LED6_BLUE_RCC_GPIOx, ENABLE);
        GPIO_InitStruct.GPIO_Pin          = LED6_BLUE_PinNumber;
        GPIO_Init(LED6_BLUE_GPIOx, &GPIO_InitStruct);


}
```

```
/* main.c */
#include <main.h>
int main(void)
{
        gpio_init();
        while(1);
}
```

The header file *gpio.h* contains all definitions and function prototype declarations that we want to use in external modules. The structure

```
#ifndef GPIO_H
#define GPIO_H
...
#endif GPIO_H
```

is called *include guards*, which prevents recursive inclusion of the same header file within a single translation unit (C file). By convention, at the beginning of the file it is checked whether the symbol (*<uppercase(filename)>_H*) has been already encountered in the current translation unit, thus preventing the collision with already parsed identifiers. Lack of include guards would require very careful use of include files, and make it even impossible to resolve complex include schemes in larger libraries and programs in many cases. Next, we include all header files needed for translation unit *gpio.c*:

```
#include <stm32f4xx.h>          // common stuff
#include <stm32f4xx_gpio.h>     // gpio control
#include <stm32f4xx_rcc.h>      // reset anc clocking
```

When *gpio.c* includes *gpio.h*, it will automatically include these referenced header files. The block of #define statements:

```
#define LED3_ORANGE_RCC_GPIOx     RCC_AHB1Periph_GPIOD
#define LED4_GREEN_RCC_GPIOx      RCC_AHB1Periph_GPIOD
#define LED5_RED_RCC_GPIOx        RCC_AHB1Periph_GPIOD
#define LED6_BLUE_RCC_GPIOx       RCC_AHB1Periph_GPIOD

#define LED3_ORANGE_GPIOx         GPIOD
#define LED4_GREEN_GPIOx          GPIOD
#define LED5_RED_GPIOx            GPIOD
#define LED6_BLUE_GPIOx           GPIOD

#define LED3_ORANGE_PinNumber     GPIO_Pin_13
#define LED4_GREEN_PinNumber      GPIO_Pin_12
#define LED5_RED_PinNumber        GPIO_Pin_14
#define LED6_BLUE_PinNumber       GPIO_Pin_15
```

extends the definition of the on-board LEDs, to include all four LEDs present on STM32F4DISCOVERY development board. Finally, the line

```
void gpio_init();
```

is not only the forward declaration of function *gpio_init()* in *gpio.c*, it also serves to any module that includes *gpio.h* to know declaration of *gpio_init()* function. The actual function code is not necessary for compilation (only function declaration from header), and actual machine code for the function will be resolved by linker either from the source code (if C file is present in a project) or precompiled static library (depending on project configuration).

The module file *gpio.c* needs only to include *gpio.h*:

```
#include <gpio.h>
```

to automatically reference all needed STM32F4 HAL functions, peripheral definitions (provided by #define statements) and forward function declarations (this is useful on a module-level only if some module functions uses them before function body implementations in C source file).

The include file *main.h* has include guards and the following include statements:

```
#include <stm32f4xx.h>                                  // common stuff
#include <stm32f4xx_gpio.h>                             // gpio control
#include <stm32f4xx_rcc.h>                              // reset anc clocking
#include <gpio.h>
```

The first three includes are reduntant in this case because they are already present in *gpio.h* includes. However, if we rely on *gpio.h* inclusion, use HAL functions, and at some later point we decide not to use *gpio.h* anymore, then we need to bring back two first lines to reference back HAL library. The shown approach is more robust and independant on whether we remove some header include later or not. Thanks to include guards, it does not matter that we include the same two header files in *gpio.h*, because include guards are present in all header files of STM32F4 HAL library.

The structure of the *main.c* program is now much more clear and oriented to application logic:

```
#include <main.h>
int main(void)
{
  gpio_init();
  while(1);
}
```

It is good practice to decouple peripheral access from program logic and contain all code accessing the peripherals in separate C modules to keep application logic clean of platform-specific code, making it less error-prone and easier to maintain. This approach of building applications by decoupling hardware access from program logic will be maintained in the rest of lab exercises.

4.6. Step 6: Adding Blinky functionlity

At this point our program initialized GPIO to control four on-board LEDs. In the next step we shall make the program to periodically turn on and off all four LED diodes. In *gpio.h* add the following constants:

```
#define LED3_ORANGE_ID            1
#define LED4_GREEN_ID             2
#define LED5_RED_ID               3
#define LED6_BLUE_ID              4
```

These constants are user-defined identifiers of each on-board LED. Each LED will be turned on and off by referencing the constant to driver function, instead of hard-coding individual pins to what the LEDs are actually wired. Add the following function to *gpio.h*:

```
void gpio_led_state(uint8_t LED_ID, uint8_t state);
```

In module *gpio.c* add the function implementation code:

```
void gpio_led_state(uint8_t LED_ID, uint8_t state)
{
BitAction bitValue;

        bitValue = (state == 0) ? Bit_SET : Bit_RESET;
        switch(LED_ID)
        {
                case LED3_ORANGE_ID:
                        GPIO_WriteBit(LED3_ORANGE_GPIOx, LED3_ORANGE_PinNumber, bitValue);
                        break;
                case LED4_GREEN_ID:
                        GPIO_WriteBit(LED4_GREEN_GPIOx, LED4_GREEN_PinNumber, bitValue);
                        break;
                case LED5_RED_ID:
                        GPIO_WriteBit(LED5_RED_GPIOx, LED5_RED_PinNumber, bitValue);
                        break;
                case LED6_BLUE_ID:
                        GPIO_WriteBit(LED6_BLUE_GPIOx, LED6_BLUE_PinNumber, bitValue);
                        break;
        }

}
```

This function will turn ON and OFF the chosen LED, referenced by LED ID define. It uses HAL function:

```
void GPIO_WriteBit(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin, BitAction BitVal)
```

where *BitVal* specifies the value to be written to the selected bit (Bit_RESET clears the port pin, and Bit_SET sets the port pin). Since that LED is turned on by driving the port pin low, the *state* parameter '0' in *gpio_led_state* function call must translate to high GPIO level, while '1' translates to active low GPIO level. Note that any changes in pin assigment to LEDs can be changed in #defines in *gpio.h* without affecting a single line of code in *gpio.c* implementation file.

Note that for integer values an explicit signed and size type identifier *uint8_t* is used. It is good practice to clearly define for each integer value whether it is signed or unsigned, and what size is it. Definitions of integer types are contained in *stdint.h*, which is already included in *stm32f4xx.h*.

Commonly used integer types defined in *stdint.h* are:

```
/* exact-width signed integer types */
typedef   signed          char int8_t;
typedef   signed short     int int16_t;
typedef   signed           int int32_t;
typedef   signed       __int64 int64_t;

/* exact-width unsigned integer types */
typedef unsigned          char uint8_t;
typedef unsigned short     int uint16_t;
typedef unsigned           int uint32_t;
typedef unsigned       __int64 uint64_t;
```

Main module now looks like this:

```
/* main.c */
#include <main.h>
int main(void)
{
int i;

        gpio_init();
        while(1)
        {
                for(i=0;i<1000000;i++);
                gpio_led_state(LED3_ORANGE_ID, 1);        // turn on
                gpio_led_state(LED4_GREEN_ID, 1);         // turn on
                gpio_led_state(LED5_RED_ID, 0);           // turn off
                gpio_led_state(LED6_BLUE_ID, 0);          // turn off
                for(i=0;i<1000000;i++);
                gpio_led_state(LED3_ORANGE_ID, 0);        // turn off
                gpio_led_state(LED4_GREEN_ID, 0);         // turn off
                gpio_led_state(LED5_RED_ID, 1);           // turn on
                gpio_led_state(LED6_BLUE_ID, 1);          // turn on
        }

}
```

The code will alternatively turn on and off pairs of on-board LEDs. Application code in *main.c* is completely decoupled from any platform-specific hardware calls. Any changes in hardware wiring to LEDs are configured in #define statements in *gpio.h* **at a single point**, without affecting both function implementation code in *gpio.c* and application logic in *main.c*. Pauses for blinking LEDs are implemented by simple for loops that provide no strict timings but this shortcoming will be corrected soon in the next chapter.

4.7. Lab outcomes:

Students must individually accomplish the following outcomes:

- working Keil MDK-ARM uVision Blinky project, following previously elaborated guidelines; the project must be buildable on any computer with installed Keil MDK uVision IDE (uV5), and must not rely on installed HAL library support on local Keil installation (i.e. with HAL library must be embedded in project, as explained),
- the project must be ready to be transferred to STM32F4DISCOVERY development board connected to a demonstration computer and ready for practical demonstration,
- students must understand all steps in building the solution and must be able to demonstrate individual steps on demand.

Copying other students' solutions without understanding how they work will be properly sanctioned!

5. Timers and interrupts

Assignment:

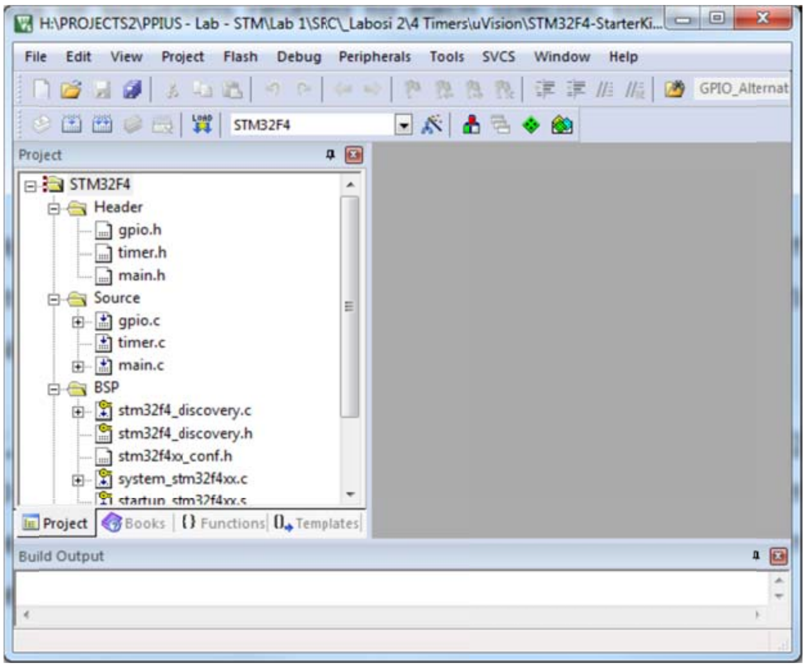
Make on-board LEDs to blink in programmable time intervals. Use timer peripheral to the measure time periods accurately and interrupts to trigger actions on timer overflow event.

Guidelines:

The previous chapter described in details how to structure a program and use HAL libraries. The following chapter(s) will describe only parts of solutions related to each specific topic, not repeating all the steps for building the solution as explained earlier.

5.1. Step 1: Create new project from template

Following the description in previous chapter(s), create new HAL-based project and add files *main.c*, *gpio.c*, and *timer.c*, with accompanying header files. The easiest way to accomplish this is to copy the project from the previous chapter in a separate folder and add *timer.h/c* files. The project should look like this:



Important: <u>Assignments in each chapter must be implemented in separate uVision projects!</u> This means that "*General-purpose input/output (GPIO) ports*" part of the lab exercise is one uVision project, while "*Timers and interrupts*" is another. You may use parts of the previous solutions for assignments in subsequent chapters.

5.2. Step 2: Configure timer

STM32F4 contains multiple types of timers with various advanced capabilities (see *RM0090 Reference manual for STM32F4 microcontroller family* for more details). In this exercise we shall use only simple timer feature, which generates an overflow event that can be either polled or trigger an interrupt. It is sufficient to use one of the general-purpose timers (TIM2-TIM5), which are described in chapter 18 of *RM0090 Reference manual*.

We add *timer.h* header file to our project:

```
/* timer.h */
#ifndef TIMER_H
#define TIMER_H

#include <stm32f4xx.h>        // common stuff
#include <stm32f4xx_gpio.h>   // gpio control
#include <stm32f4xx_rcc.h>    // reset anc clocking
#include <stm32f4xx_tim.h>    // timers

void timer2_init(void);
uint32_t timer2_get_millisec(void);
void timer2_wait_millisec(uint32_t ms);

#endif
```

This header file will provide an interface to our high-level timer API to be readily used in the main program module, decoupling hardware implementation details from the application logic. In timer module we need to additionally include *stm32f4xx_tim.h* header file, containing STM32F4 HAL functions related to timers. Our tiny timer wrapper API will expose the following functions to other modules:

- `void timer2_init(void)` - TIM2 timer initialization (along with helper variables); calling this function for the first time will initialize and start timer, and any subsequent call will restart the timer,
- `uint32_t timer2_get_millisec(void)` - number of milliseconds elapsed since the last call of *timer2_init()* function,
- `void timer2_wait_millisec(uint32_t ms)` - blocking call for waiting predefined number of milliseconds.

There are multiple ways to achieve the described functionality of the proposed API. In this example we shall do the following:

- initialize TIM2 timer to overflow every 1 ms (one millisecond software timer resolution) and generate an interrupt at each overflow,
- use module-level global variable to keep track of number of overflows; the variable will be incremented in TIM2 interrupt service routine (ISR) each millisecond; this will enable to keep track of time and provide an easy implementation of software timer for measuring arbitrary time intervals with millisecond resolution,
- enable the main module to read the elapsed time since the last timer restart in a non-blocking fashion,
- enable the main module to block execution for exact number of milliseconds.

Let us first implement the function for timer configuration. Add the following code to *timer.c* source file:

```
#include <timer.h>

uint32_t timer2_Ticks_Millisec;

void timer2_init(void)
{
TIM_TimeBaseInitTypeDef TIM_TimeBaseStructure; // init def struct for timer
NVIC_InitTypeDef NVIC_InitStructure; // init def struct for NVIC
uint16_t TimerPeriod; // 16-bit value because ARR register is 16-bit (although TIM2 is 32-bit!)
RCC_ClocksTypeDef RCC_Clocks; // for reading current clock setting - useful
uint32_t APB1_CLK; // APB1 clock - max. 42 MHz!

  RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM2, ENABLE); // enable TIM2 peripheral clock
  TIM_ITConfig(TIM2, TIM_IT_Update, DISABLE); // ensure that interrupt is disabled

 // Compute the value to be set in ARR register to generate signal frequency at 1.000 kHz (timebase 1 ms)
 // Warning - 168 MHz - this is default processor core frequency but it is NOT freq. at which peripherals
 // are clocked!
 // TimerPeriod = (uint16_t)((SystemCoreClock / 10000 ) - 1);
 // for 1 kHz division should be 1000 and ARR == 168000, but this cannot be represented with uint16;
 // therefore, it is necessery to divide by 10x more (10000) with prescaler x10;
 // to derive correct frequency, first check APB1 clock:
 RCC_GetClocksFreq(&RCC_Clocks); // fill query struct
 APB1_CLK = RCC_Clocks.PCLK1_Frequency;  // WARNING: APB1 is "slow" peripheral interface that with
 // SYSCLK = 168 MHz cannot be faster than 42 MHz (APB1 domain!) (pp. 213)
 // by default it is set to 13.44 MHz, and 13.44 MHz : 1000 = 13440.0, what would be acceptable for reload
 // without prescaler

 // Page 213:
 //      The timer clock frequencies are automatically set by hardware. There are two cases:
 //      1. If the APB prescaler is 1, the timer clock frequencies are set to the same frequency as
 //      that of the APB domain to which the timers are connected.
 //      2. Otherwise, they are set to twice (×2) the frequency of the APB domain to which the
 //      timers are connected.
 // => therefore, we need to multiplay APB1_CLK with 2:
 TimerPeriod = (uint16_t)(((APB1_CLK * 2)/ 1000 ) - 1);            // 1 kHz

 // Time Base configuration
 TIM_TimeBaseStructure.TIM_Prescaler = 0;
 TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up; // counts from 0 to autoreload, and then
   //back to 0
 TIM_TimeBaseStructure.TIM_Period = TimerPeriod;
 TIM_TimeBaseStructure.TIM_ClockDivision = 0;
 TIM_TimeBaseStructure.TIM_RepetitionCounter = 0;
 TIM_TimeBaseInit(TIM2, &TIM_TimeBaseStructure);

 // set TIM2 IRQ
 TIM_ITConfig(TIM2, TIM_IT_Update, ENABLE);

 // Enable the TIM2 global Interrupt
 NVIC_InitStructure.NVIC_IRQChannel = TIM2_IRQn;
 NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
 NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
 NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
 NVIC_Init(&NVIC_InitStructure);

 // setup global timekeeping variable (incremented each 1 ms)
 timer2_Ticks_Millisec = 0;

 // TIM2 counter enable
 TIM_Cmd(TIM2, ENABLE);
}
```

First we need to include *timer.h* header and add the global variable:

```
uint32_t timer2_Ticks_Millisec;
```

This variable is global within the *timer.c* module (i.e. it is visible to all functions contained in a *timer.c* compilation unit) but it will not be available to functions outside of *timer.c* module (because we did not put definition of *timer2_Ticks_Millisec* in *timer.h* header). This is intended behaviour because we do not want external module to directly access the variable that keeps track of time within *timer.c* module.

However, if we define global variable with the same name in some other module (e.g. in *main.c*), the program will not compile and linker will complain about multiple variable definitions with the same name.

There are situations when we want to let other modules to access the global variable contained in some compilation unit. If we wanted to export *timer2_Ticks_Millisec* variable and let some other module to use it, in *timer.h* header file we should declared it like this:

```
/* timer.h */
#ifndef TIMER_H
#define TIMER_H

...
extern uint32_t timer2_Ticks_Millisec;
...

#endif
```

All modules that use *timer.h* header will now see global variable *timer2_Ticks_Millisec* and will be able to access it from their function code. It is very important to use *extern* keyword in header file because it will prevent allocating storage for variable - the extern keyword serves only to declare the symbol name. Although it would be possible to manually add the line

```
extern uint32_t timer2_Ticks_Millisec;
```

in each module that wants to see *timer2_Ticks_Millisec* variable, this is not a good programming practice because all symbol name exports (whether they are variables or functions) should be kept in the accompanying header file. Another rule is that header files **should never allocate any storage**, what is achieved by enforcing *extern* keyword for each variable export declaration.

Next we consider the *timer2_init()* TIM2 initialization routine. Just like in GPIO example, we shall use two helper structs of custom type *TIM_TimeBaseInitTypeDef* and *NVIC_InitTypeDef* to initialize TIM2 timer and NVIC interrupt controller block for TIM2, respectively. The first step is to enable clock to TIM2 peripheral which is connected to APB1 bus interface:

```
RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM2, ENABLE); // enable TIM2 peripheral clock
```

Since we shall use the ISR, we must ensure that TIM2 interrupt is disabled before we configure interrupts:

```
TIM_ITConfig(TIM2, TIM_IT_Update, DISABLE); // ensure that interrupt is disabled
```

TIM2 will be used in counter mode (counting up) and configured to overflow every 1 ms. In order to provide the correct registers values, first we need to determine the speed at which TIM2 peripheral is clocked. System clock (SYSCLK) is set during the processor initialization stage in file *system_stm32f4xx.c*. The function *SetSysClock()* contains the code to set the system clock. By default, the processor core runs at the maximum frequency 168 MHz. However, this is not the frequency that clocks all parts of the microcontroller. The chapter 7 "*Reset and clock control for STM32Fxx*" in family user guide describes the clock system. It should be noted that:

"*Several prescalers are used to configure the AHB frequency, the high-speed APB (APB2) and the low-speed APB (APB1) domains. The maximum frequency of the AHB domain is 168 MHz. The maximum allowed frequency of the high-speed APB2 domain is 84 MHz. The maximum allowed frequency of the low-speed APB1 domain is 42 MHz" (TIM2 is connected to APB1!)*"

and

"*The timer clock frequencies are automatically set by hardware. There are two cases:*
1. If the APB prescaler is 1, the timer clock frequencies are set to the same frequency as that of the APB domain to which the timers are connected.
2. Otherwise, they are set to twice (×2) the frequency of the APB domain to which the timers are connected."

To ensure what is the exact clock frequency at which APB1 peripherals are clocked, it is useful to check it by calling the function

```
RCC_GetClocksFreq(&RCC_Clocks);
APB1_CLK = RCC_Clocks.PCLK1_Frequency;
```

With default settings in *system_stm32f4xx.c* this frequency is 13.44 MHz. To calculate the constant that needs to be written in TIM2 autoreload register we need to use the formula:

```
TimerPeriod = (uint16_t)(((APB1_CLK * 2)/ 1000 ) - 1);             // 1 kHz
```

The result is stored as unsigned 16-bit integer because TIM2 ARR register (auto-reload register) is 16-bit wide (see datasheet). Finally, we need to configure struct of type *TIM_TimeBaseInitTypeDef* defined in *stm32f4xx_tim.h*:

```
typedef struct
{
  uint16_t TIM_Prescaler;         /*!< Specifies the prescaler value used to divide the TIM clock.
                                       This parameter can be a number between 0x0000 and 0xFFFF */

  uint16_t TIM_CounterMode;       /*!< Specifies the counter mode.
                                       This parameter can be a value of @ref TIM_Counter_Mode */

  uint32_t TIM_Period;            /*!< Specifies the period value to be loaded into the active
                                       Auto-Reload Register at the next update event.
                                       This parameter must be a number between 0x0000 and 0xFFFF.  */

  uint16_t TIM_ClockDivision;     /*!< Specifies the clock division.
                                      This parameter can be a value of @ref TIM_Clock_Division_CKD */

  uint8_t TIM_RepetitionCounter;  /*!< Specifies the repetition counter value. Each time the RCR
downcounter
                                       reaches zero, an update event is generated and counting restarts
                                       from the RCR value (N).
                                       This means in PWM mode that (N+1) corresponds to:
                                          - the number of PWM periods in edge-aligned mode
                                          - the number of half PWM period in center-aligned mode
                                       This parameter must be a number between 0x00 and 0xFF.
                                       @note This parameter is valid only for TIM1 and TIM8. */
} TIM_TimeBaseInitTypeDef;
```

We shall not use prescaler so we put

```
TIM_TimeBaseStructure.TIM_Prescaler = 0;
```

Counter modes are defined in the same header file:

```
#define TIM_CounterMode_Up                 ((uint16_t)0x0000)
#define TIM_CounterMode_Down               ((uint16_t)0x0010)
#define TIM_CounterMode_CenterAligned1     ((uint16_t)0x0020)
#define TIM_CounterMode_CenterAligned2     ((uint16_t)0x0040)
#define TIM_CounterMode_CenterAligned3     ((uint16_t)0x0060)
```

We shall use counter up mode:

```
TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up;
```

which is described on pp. 579 of family user manual:

"*In upcounting mode, the counter counts from 0 to the auto-reload value (content of the TIMx_ARR register), then restarts from 0 and generates a counter overflow event.*".

Next we need to set the value of ARR register:

```
TIM_TimeBaseStructure.TIM_Period = TimerPeriod;
TIM_TimeBaseStructure.TIM_ClockDivision = 0;
TIM_TimeBaseStructure.TIM_RepetitionCounter = 0;
```

and call TIM2 initialization function with reference to TIM2 resource (typed pointer to register bank struct) and content of TIM2 configuration helper struct:

```
TIM_TimeBaseInit(TIM2, &TIM_TimeBaseStructure);
```

Now TIM2 has defined time base and counting mode. Next, we shall configure TIM2 peripheral to generate interrupt request to NVIC on counter overflow:

```
TIM_ITConfig(TIM2, TIM_IT_Update, ENABLE);
```

TIM_ITConfig function enables or disables the specified TIM interrupts. The second parameter defines flags referring to events that can cause an IRQ (see the function source code comments in *stm32f4xx_tim.c*):

```
  * @param  TIM_IT: specifies the TIM interrupts sources to be enabled or disabled.
  *          This parameter can be any combination of the following values:
  *            @arg TIM_IT_Update: TIM update Interrupt source
  *            @arg TIM_IT_CC1: TIM Capture Compare 1 Interrupt source
  *            @arg TIM_IT_CC2: TIM Capture Compare 2 Interrupt source
  *            @arg TIM_IT_CC3: TIM Capture Compare 3 Interrupt source
  *            @arg TIM_IT_CC4: TIM Capture Compare 4 Interrupt source
  *            @arg TIM_IT_COM: TIM Commutation Interrupt source
  *            @arg TIM_IT_Trigger: TIM Trigger Interrupt source
  *            @arg TIM_IT_Break: TIM Break Interrupt source
  *
  * @note   For TIM6 and TIM7 only the parameter TIM_IT_Update can be used
  * @note   For TIM9 and TIM12 only one of the following parameters can be used: TIM_IT_Update,
  *          TIM_IT_CC1, TIM_IT_CC2 or TIM_IT_Trigger.
  * @note   For TIM10, TIM11, TIM13 and TIM14 only one of the following parameters can
  *          be used: TIM_IT_Update or TIM_IT_CC1
  * @note   TIM_IT_COM and TIM_IT_Break can be used only with TIM1 and TIM8
```

The *TIM_IT_Update* parameter will trigger interrupt request to NVIC following the counter overflow.

However, this will still not trigger interrupt service routine to run, only interrupt request to NVIC. To enable interrupt we need to configure NVIC by calling *NVIC_Init()* function on properly filled *NVIC_InitTypeDef* structure:

```
// Enable the TIM2 global Interrupt
NVIC_InitStructure.NVIC_IRQChannel = TIM2_IRQn;
NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
NVIC_Init(&NVIC_InitStructure);
```

NVIC_InitTypeDef structure is defined in *misc.h*:

```
typedef struct
{
  uint8_t NVIC_IRQChannel;      /*!< Specifies the IRQ channel to be enabled or disabled.
                                This parameter can be an enumerator of @ref IRQn_Type
                                enumeration (For the complete STM32 Devices IRQ Channels
                                list, please refer to stm32f4xx.h file) */

  uint8_t NVIC_IRQChannelPreemptionPriority;  /*!< Specifies the pre-emption priority for the IRQ channel
                                specified in NVIC_IRQChannel. This parameter can be a value
                                between 0 and 15 as described in the table @ref MISC_NVIC_Priority_Table
                                A lower priority value indicates a higher priority */

  uint8_t NVIC_IRQChannelSubPriority;         /*!< Specifies the subpriority level for the IRQ channel
                                  specified NVIC_IRQChannel. This parameter can be a value
                                  between 0 and 15 as described in the table @ref MISC_NVIC_Priority_Table
                                  A lower priority value indicates a higher priority */

  FunctionalState NVIC_IRQChannelCmd;         /*!< Specifies whether the IRQ channel defined in
                                  NVIC_IRQChannel will be enabled or disabled.
                                  This parameter can be set either to ENABLE or DISABLE */
} NVIC_InitTypeDef;
```

IRQ channels are defined in *stm32f4xx.h* in enum *IRQn_Type*:

```
typedef enum IRQn
{
/******  Cortex-M4 Processor Exceptions Numbers
****************************************************************/
  NonMaskableInt_IRQn         = -14,    /*!< 2 Non Maskable Interrupt*/
  MemoryManagement_IRQn       = -12,    /*!< 4 Cortex-M4 Memory Management Interrupt*/
  BusFault_IRQn               = -11,    /*!< 5 Cortex-M4 Bus Fault Interrupt  */
  UsageFault_IRQn             = -10,    /*!< 6 Cortex-M4 Usage Fault Interrupt */
  SVCall_IRQn                 = -5,     /*!< 11 Cortex-M4 SV Call Interrupt */
  DebugMonitor_IRQn           = -4,     /*!< 12 Cortex-M4 Debug Monitor Interrupt */
  PendSV_IRQn                 = -2,     /*!< 14 Cortex-M4 Pend SV Interrupt  */
  SysTick_IRQn                = -1,     /*!< 15 Cortex-M4 System Tick Interrupt */
/******  STM32 specific Interrupt Numbers ***/

...                                     /*!< other IRQ sources*/

  TIM2_IRQn                   = 28,     /*!< TIM2 global Interrupt */

...                                     /*!< other IRQ sources*/

} IRQn_Type;
```

Priority level is set to maximum by setting the preemption priority and subpriority to zero. Now the TIM2 interrupt request to NVIC will actually generate interrupt service routine (ISR) call upon TIM2 overflow. The final step is to run TIM2 peripheral:

```
TIM_Cmd(TIM2, ENABLE);
```

Before running the timer we shall reset global variable millisecond counter for later use:

```
timer2_Ticks_Millisec = 0;
```

How to write ISR routine and connect the code with interrupt vector table?

Interrupt vector table is defined in startup file *startup_stm32f4xx.s* (assembly code). Let us consider important parts of startup file:

```
; Amount of memory (in bytes) allocated for Stack
; Tailor this value to your application needs
; <h> Stack Configuration
;   <o> Stack Size (in Bytes) <0x0-0xFFFFFFFF:8>
; </h>

Stack_Size      EQU     0x00000400

                AREA    STACK, NOINIT, READWRITE, ALIGN=3
Stack_Mem       SPACE   Stack_Size
__initial_sp


; <h> Heap Configuration
;   <o>  Heap Size (in Bytes) <0x0-0xFFFFFFFF:8>
; </h>

Heap_Size       EQU     0x00000200

                AREA    HEAP, NOINIT, READWRITE, ALIGN=3
__heap_base
Heap_Mem        SPACE   Heap_Size
__heap_limit

                PRESERVE8
                THUMB


; Vector Table Mapped to Address 0 at Reset
                AREA    RESET, DATA, READONLY
                EXPORT  __Vectors
                EXPORT  __Vectors_End
                EXPORT  __Vectors_Size

__Vectors       DCD     __initial_sp               ; Top of Stack
                DCD     Reset_Handler              ; Reset Handler
                DCD     NMI_Handler                ; NMI Handler
                DCD     HardFault_Handler          ; Hard Fault Handler
                DCD     MemManage_Handler          ; MPU Fault Handler
                DCD     BusFault_Handler           ; Bus Fault Handler
                DCD     UsageFault_Handler         ; Usage Fault Handler
                DCD     0                          ; Reserved
                DCD     0                          ; Reserved
                DCD     0                          ; Reserved
                DCD     0                          ; Reserved
                DCD     SVC_Handler                ; SVCall Handler
                DCD     DebugMon_Handler           ; Debug Monitor Handler
                DCD     0                          ; Reserved
                DCD     PendSV_Handler             ; PendSV Handler
                DCD     SysTick_Handler            ; SysTick Handler

                ; External Interrupts
                DCD     WWDG_IRQHandler                   ; Window WatchDog
                DCD     PVD_IRQHandler                    ; PVD through EXTI Line detection
                DCD     TAMP_STAMP_IRQHandler             ; Tamper and TimeStamps through the EXTI line
                DCD     RTC_WKUP_IRQHandler               ; RTC Wakeup through the EXTI line
                DCD     FLASH_IRQHandler                  ; FLASH
                DCD     RCC_IRQHandler                    ; RCC
                DCD     EXTI0_IRQHandler                  ; EXTI Line0
                DCD     EXTI1_IRQHandler                  ; EXTI Line1
                DCD     EXTI2_IRQHandler                  ; EXTI Line2
                DCD     EXTI3_IRQHandler                  ; EXTI Line3
                DCD     EXTI4_IRQHandler                  ; EXTI Line4
                DCD     DMA1_Stream0_IRQHandler           ; DMA1 Stream 0
                DCD     DMA1_Stream1_IRQHandler           ; DMA1 Stream 1
                DCD     DMA1_Stream2_IRQHandler           ; DMA1 Stream 2
                DCD     DMA1_Stream3_IRQHandler           ; DMA1 Stream 3
                DCD     DMA1_Stream4_IRQHandler           ; DMA1 Stream 4
                DCD     DMA1_Stream5_IRQHandler           ; DMA1 Stream 5
                DCD     DMA1_Stream6_IRQHandler           ; DMA1 Stream 6
                DCD     ADC_IRQHandler                    ; ADC1, ADC2 and ADC3s
                DCD     CAN1_TX_IRQHandler                ; CAN1 TX
                DCD     CAN1_RX0_IRQHandler               ; CAN1 RX0
                DCD     CAN1_RX1_IRQHandler               ; CAN1 RX1
                DCD     CAN1_SCE_IRQHandler               ; CAN1 SCE
                DCD     EXTI9_5_IRQHandler                ; External Line[9:5]s
                DCD     TIM1_BRK_TIM9_IRQHandler          ; TIM1 Break and TIM9
                DCD     TIM1_UP_TIM10_IRQHandler          ; TIM1 Update and TIM10
                DCD     TIM1_TRG_COM_TIM11_IRQHandler     ; TIM1 Trigger and Commutation and TIM11
                DCD     TIM1_CC_IRQHandler                ; TIM1 Capture Compare
                DCD     TIM2_IRQHandler                   ; TIM2
                DCD     TIM3_IRQHandler                   ; TIM3
                DCD     TIM4_IRQHandler                   ; TIM4

                 ...
```

The startup file contains some important initialization stuff that must be taken into account to properly write the application code. Very important setting is the stack size:

```
Stack_Size      EQU     0x00000400
```

The stack size is set to 1024 bytes by default. This size must provide enough space to hold nested function call stack frames, each containing function parameters, local variables etc. Although the default setting is sufficient for most cases, in some situations the stack size must be manually adjusted to accomodate need for deeply nested function calls, functions with large amount of local variables storage etc.

Next important thing is the *interrupt vector table* (IVT), that is placed at the memory address 0x00000000 after reset. Each DCD directive allocates (consecutive) 4 bytes of memory, starting with *__initial_sp* on address 0x00000000, *Reset_Handler* ISR function address at 0x00000004 etc. The DCD directives follow the structure of the interrupt vector table as defined by Cortex-M processor architecture. For our case the interesting entry is

```
                DCD     TIM2_IRQHandler                   ; TIM2
```

This positional IVT entry is given symbolic name *TIM2_IRQHandler*. If we define in any included compilation unit (i.e. C source file) a function with exactly the same name, the linker will insert the address of that function into interrupt vector table entry at this place. We can edit startup file and write another name for TIM2 ISR if we wish, but then we also need to call TIM2 ISR function that name in our code.

In *timer.c* we add TIM2 ISR function implementation:

```
void TIM2_IRQHandler(void)
{
	if (TIM_GetITStatus(TIM2, TIM_IT_Update) != RESET)
	{
		TIM_ClearITPendingBit(TIM2, TIM_IT_Update);
		timer2_Ticks_Millisec++;
	}
}
```

The linker will take care to insert the address of this function into appropriate IVT slot, which will cause IRQ to redirect code execution into this function. The ISR does the following (please see the HAL library source code and STM32F4 family user guide for further explanations):

- check whether the counter overflow event caused the interrupt to be triggered (*TIM_IT_Update* event - other events could have triggered TIM2 interrupt if it was set up in a different way, but they would be ignored),
- if interrupt was caused by auto-reload counter overflow, simply clear the interrupt pending bit (counter will start counting the next cycle automatically in auto-reload mode),
- increment global variable milliseconds counter.

The ISR basically does nothing but incrementing global variable *timer2_Ticks_Millisec*, enabling the application code to keep track on elapsed time since the last TIM2 initialization.

How can the application program read the value of *timer2_Ticks_Millisec*?

Although the application code could simply read the content of *timer2_Ticks_Millisec* global variable, it is not desirable because this variable could be changed at any time by interrupt (this is a *shared resource*). Each access to the shared resource must be done within critical section and protected against the non-atomic behaviour. It would be advisable not to read this variable directly but to use helper function which contains critical section. Therefore, we do not export *timer2_Ticks_Millisec* for outside modules and use helper function to read the value:

```
uint32_t timer2_get_millisec()
{
uint32_t value;

	NVIC_DisableIRQ(TIM2_IRQn);
	value = timer2_Ticks_Millisec;
	NVIC_EnableIRQ(TIM2_IRQn);
	return value;
}
```

Since the variable *timer2_Ticks_Millisec* is shared resource that could be changed by TIM2 interrupt, it should be protected by granting exclusive access to function *timer2_get_millisec()*. At the beginning of critical section, the TIM2 interrupt must be disabled. Then we read the content of *timer2_Ticks_Millisec* variable into local copy and enable TIM2 interrupt again. Since the variable *value* is locally visible, it is safe to enable interrupt after copying the global variable into local storage and return local copy to the outside caller function. The application code will use this function to read elapsed number of milliseconds at any moment in atomic way.

Another function that can be called from the user application code is *timer2_wait_millisec()*. This function will produce a blocking delay for supplied number of milliseconds, using the previously implemented function *timer2_get_millisec()*. It will first read the elapsed number of milliseconds and wait in a loop until the predefined delay elapses. The atomicity of access to the global variable is satisfied by calling a *timer2_get_millisec()* function that already provides critical section for reading the shared resource.

The *timer2_wait_millisec()* function is implemented like this:

```
void timer2_wait_millisec(uint32_t ms)
{
uint32_t t1, t2;

        t1 = timer2_get_millisec();
        while(1)
        {
                t2 = timer2_get_millisec();
                if ((t2 - t1) >= ms) break;
                if (t2 < t1) break;   // almost never occur, once in 49 days
        }
}
```

5.3. Step 3: Modify the main module to make LEDs blink with strict timing

Now that we have defined *timer.h* and *timer.c* files, we can easily modify the *main.c* module from the previous chapter to enable strictly timed LED blinking. The *main.h* looks like this:

```
/* main.h */
#ifndef MAIN_H
#define MAIN_H

#include <stm32f4xx.h>                                  // common stuff
#include <stm32f4xx_gpio.h>                             // gpio control
#include <stm32f4xx_rcc.h>                              // reset anc clocking
#include <gpio.h>
#include <timer.h>

#define DELAY_MS 1000

#endif
```

We include *timer.h* header to import TIM2 API. We also define a constant *DELAY_MS* which determines the LED blinking interval. To change this interval it is sufficient only to change the #define statement, without affecting the code in *main.c*. The *main.c* module now looks like this:

```
/* main.c */
#include <main.h>
int main(void)
{
        gpio_init();
        timer2_init();
        while(1)
        {
                timer2_wait_millisec(DELAY_MS);
                gpio_led_state(LED3_ORANGE_ID, 1);   // turn on
                gpio_led_state(LED4_GREEN_ID, 1);    // turn on
                gpio_led_state(LED5_RED_ID, 0);      // turn off
                gpio_led_state(LED6_BLUE_ID, 0);     // turn off
                timer2_wait_millisec(DELAY_MS);
                gpio_led_state(LED3_ORANGE_ID, 0);   // turn off
                gpio_led_state(LED4_GREEN_ID, 0);    // turn off
                gpio_led_state(LED5_RED_ID, 1);      // turn on
                gpio_led_state(LED6_BLUE_ID, 1);     // turn on
        }
}
```

The code is compact, easy to understand, decoupled from hardware implementation details, and without hard-coded constants. First we initialize GPIO by calling the function *gpio_init()*. Then we initialize TIM2 peripheral and NVIC by calling the function *timer2_init()*. The blocking delay is realized by calling the function *timer2_wait_millisec*(*DELAY_MS*), where delay can be changed in *main.h* header.

5.4. Lab outcomes:

Students must individually accomplish the following outcomes:

- working Keil MDK-ARM uVision project, following the guidelines for this part of lab exercise, with the same remarks like in the previous chapter regarding the project building and transfering to STM32F4DISCOVERY development board

6. Serial communication

Assignment:

Make a loopback serial interface that will receive characters from a computer terminal (e.g. HyperTerm) and echo back all received characters. Use USART1 (*Universal synchronous asynchronous receiver transmitter*) interface for communication with computer. As STM32F4DISCOVERY board does not have RS-232 interface implemented on board, and most of modern computers do not have RS-232 port, it is advisable to use either UART/USB bridge interface or UART/RS232 + RS232/USB interfaces to connect with computer. Set the communication parameters to 115200, 8, N, 1. The solution must use interrupt ISR for buffering the incoming characters, while sending the characters do not have to be buffered.

Guidelines:

6.1. Step 1: Create a new project from template

Make a new project based on template discussed in previous chapters. Add new empty files *main.c/h* and *usart.c/h*.

6.2. Step 2: Configure USART1 peripheral and ISR

In the header file *usart.h* we shall include header files, create necessary symbol definitions, and API function prototypes for exporting the serial communication API to other modules:

```
/* usart.h */
#ifndef USART_H
#define USART_H

#include <stm32f4xx.h>                          // common stuff
#include <stm32f4xx_rcc.h>                      // reset anc clocking
#include <stm32f4xx_gpio.h>                     // gpio control
#include <stm32f4xx_usart.h>                    // USART

#define BUFSIZE       16
#define BAUDRATE      115200

void USART1_Init(void);                         // init USART1 peripheral
void USART1_SendChar(char c);                   // blocking send character
int  USART1_Dequeue(char* c);                   // pop character from receive FIFO

#endif
```

Beside all previously explained header files, now we also include *stm32f4xx_usart.h* with all necessary HAL functions and definitions for control of on-chip USART peripherals. We define the size of first-in first-out (FIFO) buffer for received characters (*BUFSIZE = 16*), and baudrate setting (*BAUDRATE = 115200*), that can be easily changed by modifying #define values in this header file. For a minimum USART1 high-level API interface we define three functions:

- *USART1_Init()* - function that initializes USART1 peripheral and corrsponding GPIO pins,
- *USART1_SendChar()* - a high-level function that sends a single character through the USART1 interface; the function will block if USART1 is not ready to transmit and will wait until it is ready; the function guarantees that character will be sent, but does not provide any guarantees about the timing,

- *USART1_Dequeue()* - the function fetches a single character from a queue of received characters; in case that there are more than one character waiting to be processed in a queue, the one that first arrived will be fetched; function is non-blocking, meaning that it will not block on empty queue (it will return 0 if there are no characters to be processed); otherwise, the function will return 1, and fetched character will be returned through the *byref* parameter (supplied as a pointer to *char*).

At the beginning of *usart.c* module, we first add the internal global variables that will take care of received characters buffering and interface transmission ready state information:

```
/* usart.c */
#include <usart.h>

// RX FIFO buffer
char RX_BUFFER[BUFSIZE];
int  RX_BUFFER_HEAD, RX_BUFFER_TAIL;
// TX state flag
uint8_t TxReady;
```

The character array *RX_BUFFER* holds all unprocessed incoming characters. The buffer is organized as a FIFO queue using static C character array. Therefore, we need auxiliary variables pointing to *head* (*RX_BUFFER_HEAD*) and *tail* (*RX_BUFFER_TAIL*) to implement queue functionality with static array. This global *RX_BUFFER* queue is filled with characters by USART1 ISR as they arrive. USART1 ISR is executed on each received character, and its only job is to place newly received character in *RX_BUFFER* FIFO (ISR must be very short, and provide only minimum necessary work with I/O registers). The main program will dequeue characters from *RX_BUFFER* FIFO in a main control loop. Buffering of incoming characters enables less strict timings in the main control loop, which can be even more relaxed by increasing the FIFO depth (*BUFSIZE* constant).

The meaning of the *TxReady* flag will be explained as follows. At the beginning, the USART1 is not sending any characters, and it is ready to send a new one (*TxReady=1*). At some moment we decide to transmit a new character through the USART interface. Depending on the baudrate, transmission of a single character will take some time, and any new characters must be written in data buffer *after* we complete transmission of the previous one (for that reason, just before the character transmission, we set the flag *TxReady=0*, signaling to the main program that it must not try to send any new characters while *TxReady=0*). At the moment the serial interface complete with sending one character, the USART1 ISR will be triggered, signaling that the USART interface has just finished with transmission of the current character (this is the moment when USART interface is ready to receive a next character for tranmission, and the main program will be informed about this through the *TxReady* flag, that ISR will change to *TxReady=1*). Then the main program sends a new character, setting *TxReady=0* and waiting the ISR to reset it to *TxReady=1*, upon end of the character transmission.

This scenario holds for all microcontrollers with hardware USART FIFO length of one character. Some microcontrollers have hardware FIFO depth of more than one character, and some (like STM32F4 family) provide hardware FIFO in RAM by means of DMA. In this example will shall use only software buffering of received characters, and no support for buffering of transmitted characters (i.e. sending them one by one in a blocking manner as described, without corrupting characters by prematurely writes to USART data transmit registers).

Before going any futher, we must implement the function for USART1 initialization:

```
// init USART1
void USART1_Init(void)
{
GPIO_InitTypeDef         GPIO_InitStruct;
USART_InitTypeDef        USART_InitStruct;
NVIC_InitTypeDef         NVIC_InitStructure;

   // enable peripheral clocks (note: different bus interfaces for each peripheral!)
   RCC_APB2PeriphClockCmd(RCC_APB2Periph_USART1, ENABLE);
   RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOB, ENABLE);

   // map port B pins for alternate function
   GPIO_InitStruct.GPIO_Pin = GPIO_Pin_6 | GPIO_Pin_7; // Pins 6 (TX) and 7 (RX) will be used for USART1
   GPIO_InitStruct.GPIO_Mode = GPIO_Mode_AF;       // GPIO pins defined as alternate
                 function
   GPIO_InitStruct.GPIO_Speed = GPIO_Speed_50MHz;  // I/O pins speed (signal rise time)
   GPIO_InitStruct.GPIO_OType = GPIO_OType_PP;     // push-pull output
   GPIO_InitStruct.GPIO_PuPd = GPIO_PuPd_UP;       // activates pullup resistors
   GPIO_Init(GPIOB, &GPIO_InitStruct);             // set chosen pins

   // set alternate function to USART1 (from multiple possible alternate function choices)
   GPIO_PinAFConfig(GPIOB, GPIO_PinSource6, GPIO_AF_USART1);       // pins will automatically be assigned
        to TX/RX - refer to datasheet to see AF mappings
   GPIO_PinAFConfig(GPIOB, GPIO_PinSource7, GPIO_AF_USART1);

   // use USART_InitStruct to config USART1 peripheral
   USART_InitStruct.USART_BaudRate = BAUDRATE;     // set baudrate from define
   USART_InitStruct.USART_WordLength = USART_WordLength_8b;// 8 data bits
   USART_InitStruct.USART_StopBits = USART_StopBits_1;     // 1 stop bit
   USART_InitStruct.USART_Parity = USART_Parity_No;        // no parity check
   USART_InitStruct.USART_HardwareFlowControl = USART_HardwareFlowControl_None; // no HW control flow
   USART_InitStruct.USART_Mode = USART_Mode_Tx | USART_Mode_Rx; // enable both character transmit and
        receive
   USART_Init(USART1, &USART_InitStruct);          // set USART1 peripheral

   // set interrupt triggers for USART1 ISR (but do not enable USART1 interrupts yet)
   USART_ITConfig(USART1, USART_IT_TXE,   DISABLE);// should be disbled
   USART_ITConfig(USART1, USART_IT_TC,    ENABLE); // transmission completed event (for reseting TxReady
        flag)
   USART_ITConfig(USART1, USART_IT_RXNE,  ENABLE); // character received (to trigger buffering of new
        character)

   TxReady = 1;                                    // USART1 is ready to transmit
   RX_BUFFER_HEAD = 0; RX_BUFFER_TAIL = 0;         // clear rx buffer

   // prepare NVIC to receive USART1 IRQs
   NVIC_InitStructure.NVIC_IRQChannel = USART1_IRQn;                 // configure USART1 interrupts
   NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;// max. priority
   NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;                // max. priority
   NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;                   // enable USART1 interrupt in NVIC
   NVIC_Init(&NVIC_InitStructure);                                                      // set NVIC
        for USART1 IRQ

   // enables USART1 interrupt generation
   USART_Cmd(USART1, ENABLE);
}
```

GPIO_InitTypeDef struct will be used to define TX and RX USART1 pins on GPIO port, *NVIC_InitTypeDef* will be used to setup USART1 interrupt in NVIC, and *USART_InitTypeDef* will be used to configure USART1 parameters.

First we must provide clock to the USART1 peripheral (connected to APB2 bus), but also to the Port B, because TX and RX pins will be mapped to this port:

```
   // enable peripheral clocks (note: different bus interfaces for each peripheral!)
   RCC_APB2PeriphClockCmd(RCC_APB2Periph_USART1, ENABLE);
   RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOB, ENABLE);
```

Next we set up GPIO pins that will be mapped to USART1.TX (PB6), and USART1.RX (PB7):

```
// map port B pins for alternate function
GPIO_InitStruct.GPIO_Pin = GPIO_Pin_6 | GPIO_Pin_7; // Pins 6 (TX) and 7 (RX) will be used for USART1
GPIO_InitStruct.GPIO_Mode = GPIO_Mode_AF;       // GPIO pins defined as alternate
           function
GPIO_InitStruct.GPIO_Speed = GPIO_Speed_50MHz;  // I/O pins speed (signal rise time)
GPIO_InitStruct.GPIO_OType = GPIO_OType_PP;     // push-pull output
GPIO_InitStruct.GPIO_PuPd = GPIO_PuPd_UP;       // activates pullup resistors
GPIO_Init(GPIOB, &GPIO_InitStruct);             // set chosen pins
```

By default, PB6 and PB7 are plain GPIO pins without any corrspondance with USART1. How to know what GPIO pins we can be used as USART1 TX and RX pins?

This cannot be answered just by reading STM32F4 family guide, because there are too many different microcontrollers sub-types and IC packages, that differ in number of pins, exposed GPIO ports etc. Therefore, we need to open the datasheet of the microcontroller we use (e.g. STM32F407VG), and find the chapter „*Pinouts and pin description*“. In the table 7 „*STM32F40x pin and ball definitions*“ there are all pin definitions, along with alternate functions, for all package types. For microcontroller on STM32F4DISCOVERY board (STM32F407VG) we need to refer to LQFP100 package. In the column „*Alternate functions*“ we can see all possible alternate functions that can be assigned to some pin. For LQFP100 package and pins PB6 and PB7 we can see:

Pin number (LQFP100)	**Pin name (after reset)**	**Pin type**	**I/O structure**	**Alternate functions**
92	PB6	I/O	FT	I2C1_SCL/ TIM4_CH1 / CAN2_TX / DCMI_D5/USART1_TX/ EVENTOUT
93	PB7	I/O	FT	I2C1_SDA / FSMC_NL / DCMI_VSYNC / USART1_RX/ TIM4_CH2/ EVENTOUT

We see all alternate functions that can be assigned to PB6 and PB7, and we could also use other pins (e.g. PA9 for USART1_TX etc.). Because we decided to use PB6 and PB7, we must refer to these pins (*GPIO_InitStruct.GPIO_Pin = GPIO_Pin_6 | GPIO_Pin_7*), and change their mode to alternate function (*GPIO_InitStruct.GPIO_Mode = GPIO_Mode_AF*). Because there are few possible alternate functions for these pins, we must decide which one to use:

```
GPIO_PinAFConfig(GPIOB, GPIO_PinSource6, GPIO_AF_USART1);
GPIO_PinAFConfig(GPIOB, GPIO_PinSource7, GPIO_AF_USART1);
```

It is sufficient to tell that we use USART1 on PB6 and PB7, and correct functions will be automatically assigned to each pin (because it is uniquely defined by AF mapping which pin will be TX and RX, respectively).

After setting the GPIO pin mappings, we need to set up USART1 peripheral:

```
// use USART_InitStruct to config USART1 peripheral
USART_InitStruct.USART_BaudRate = BAUDRATE;     // set baudrate from define
USART_InitStruct.USART_WordLength = USART_WordLength_8b;// 8 data bits
USART_InitStruct.USART_StopBits = USART_StopBits_1;     // 1 stop bit
USART_InitStruct.USART_Parity = USART_Parity_No;        // no parity check
USART_InitStruct.USART_HardwareFlowControl = USART_HardwareFlowControl_None; // no HW control flow
USART_InitStruct.USART_Mode = USART_Mode_Tx | USART_Mode_Rx; // enable both character transmit and
        receive
USART_Init(USART1, &USART_InitStruct);          // set USART1 peripheral
```

We set the baudrate (provided by BAUDRATE define statement), data frame description (8 bits, 1 stop bit, no parity), and without using hardware flow control. We must also excplicitely define what capabilities we use (receive and transmit).

As we will be using interrupt driven I/O, we must define what events will trigger ISR:

```
USART_ITConfig(USART1, USART_IT_TXE,   DISABLE);
USART_ITConfig(USART1, USART_IT_TC,    ENABLE);
USART_ITConfig(USART1, USART_IT_RXNE,  ENABLE);
```

The *USART_IT_TC* („*Transmission Complete*“) flag will enable interrupt to be triggered at the end of character transmission (to signal to the main program that USART1 is ready to send the next character), and *USART_IT_RXNE* („*Received Data Ready to be Read*“) flag will enable interrupt to be triggered when a new character arrives. *USART_IT_TXE* flag („*Transmit Data Register Empty*“) should be disabled in this application because it does not have the same (desired) functionality as *USART_IT_TC* flag. The meaning of flags can be found in STM32F4 family guide (pp. 990, table 146. „*USART interrupt requests*“).

Next we define the initial states of global helper variables:

```
TxReady = 1;                                 // USART1 is ready to transmit
RX_BUFFER_HEAD = 0; RX_BUFFER_TAIL = 0;      // clear rx buffer
```

meaning that our USART1 interface is ready to send a new character (*TxReady = 1*), and receive characters FIFO is empty (*RX_BUFFER_HEAD = RX_BUFFER_TAIL = 0*).

At this point, NVIC has to be set to receive USART1 interrupts:

```
// prepare NVIC to receive USART1 IRQs
NVIC_InitStructure.NVIC_IRQChannel = USART1_IRQn;                  // configure USART1 interrupts
NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;// max. priority
NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;                 // max. priority
NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;                    // enable USART1 interrupt in NVIC
NVIC_Init(&NVIC_InitStructure);                                                        // set NVIC
       for USART1 IRQ
```

USART1 will start generating interrupts after we enable it:

```
// enables USART1 interrupt generation
USART_Cmd(USART1, ENABLE);
```

After we finish the initialization of USART1 peripheral, we need to implement the corresponding ISR:

```
void USART1_IRQHandler(void)
{
static char rx_char;
static char rx_head;

  // RX event
  if (USART_GetITStatus(USART1, USART_IT_RXNE) == SET)
  {
         USART_ClearITPendingBit(USART1, USART_IT_RXNE);
         rx_char = USART_ReceiveData(USART1);
         // check for buffer overrun:
         rx_head = RX_BUFFER_HEAD + 1;
         if (rx_head == BUFSIZE) rx_head = 0;
         if (rx_head != RX_BUFFER_TAIL)
         {
                 // adding new char will not cause buffer overrun:
                 RX_BUFFER[RX_BUFFER_HEAD] = rx_char;
                 RX_BUFFER_HEAD = rx_head;   // update head
         }
  }
  // TX event
  if (USART_GetITStatus(USART1, USART_IT_TC) == SET)
  {
          USART_ClearITPendingBit(USART1, USART_IT_TC);
          TxReady = 1;
  }
}
```

The ISR function name *USART1_IRQHandler* is already defined in the startup file interrupt vector table definition so it recommended to use exactly this name for ISR. When ISR fires, we need to determine the interrupt request cause:

```
  if (USART_GetITStatus(USART1, USART_IT_RXNE) == SET)
```

means that ISR was fired on received character, and

```
  if (USART_GetITStatus(USART1, USART_IT_TC) == SET)
```

means that ISR was fired upon the end of the last character transmission. When we receive a character, we need to do the following:

```
  if (USART_GetITStatus(USART1, USART_IT_RXNE) == SET)
  {
          USART_ClearITPendingBit(USART1, USART_IT_RXNE);
          rx_char = USART_ReceiveData(USART1);
          // check for buffer overrun:
          rx_head = RX_BUFFER_HEAD + 1;
          if (rx_head == BUFSIZE) rx_head = 0;
          if (rx_head != RX_BUFFER_TAIL)
          {
                  // adding new char will not cause buffer overrun:
                  RX_BUFFER[RX_BUFFER_HEAD] = rx_char;
                  RX_BUFFER_HEAD = rx_head;   // update head
          }
  }
```

First we need to clear interrupt pending bit. Then we use *USART_ReceiveData* HAL API function to read newly received character from USART1 interface (data register). We put this character in temporarily *rx_char* variable. What we want to do with a new character is to put it into global received characters queue, and finish ISR job as fast as possible, doing minimum interaction with peripheral registers, without doing any actual application-specific job. The rest of the code above will check whether the newly received character will cause the buffer overrun, and put it into the buffer only if there is a space left.

When sending a new character from the main program, we shall automatically set *TxReady = 0* (we shall see how a bit later), and prevent ourselves from sending a new character before the previous one has not been fully transmitted (i.e. until we detect that *TxReady = 1*). The flag is set back to *TxReady = 1* in ISR when character is transmitted:

```
  // TX event
  if (USART_GetITStatus(USART1, USART_IT_TC) == SET)
  {
          USART_ClearITPendingBit(USART1, USART_IT_TC);
          TxReady = 1;
  }
```

6.3. Step 3: Implement high-level API functions for sending and receiving characters

After we properly initialized USART1 peripheral and prepared logic in ISR regarding the global buffers and flags, we can finally implement high-level API functions for sending and receiving characters from the main program:

```
void USART1_SendChar(char c)
{
	while(!TxReady);
	USART_SendData(USART1, c);
	TxReady = 0;
}

int USART1_Dequeue(char* c)
{
int ret;

	ret = 0;
	*c = 0;
	NVIC_DisableIRQ(USART1_IRQn);
	if (RX_BUFFER_HEAD != RX_BUFFER_TAIL)
	{
		*c = RX_BUFFER[RX_BUFFER_TAIL];
		RX_BUFFER_TAIL++;
		if (RX_BUFFER_TAIL == BUFSIZE) RX_BUFFER_TAIL = 0;
		ret = 1;
	}
	NVIC_EnableIRQ(USART1_IRQn);
	return ret;
}
```

Let us consider the function *USART1_SendChar()*. This function will try to send a character *c* by first checking whether some other character is still being transmitted (state of the *TxReady* flag). If *TxReady=1*, the function will transmit the character (i.e. write it into the transmit buffer data register of USART1 interface) by means of *USART_SendData()* HAL function. As soon as a new character is written to USART1 data buffer, the program will set *TxReady=0*, and that flag will be set to *TxReady=1* by ISR upon end of character transmission. If we call again *USART1_SendChar()* immediatelly after this, the function will block in *while* loop on *TxReady=0* flag, until ISR unblocks it upon end of the previous character transmission. This is an example of unbuffered character sending with blocking, meaning that we send characters one by one (without buffering), in a polling manner (blocking on *TxReady* flag until the previous character is transmitted). Note that it would be easily possible to buffer outgoing character stream, and automatically send the remaining characters in buffer in ISR on USART_IT_TC event, but this is not done in this example for sake of simplicity.

The second function fetches single character (the oldest one) from the receive characters FIFO buffer. Since the receive buffer *RX_BUFFER* is a global variable that can be accessed both from the main function and USART1 interrupt service routine, care must be taken because this resource is shared between two lines of code execution. Therefore, any attempt in the main function to access or alter the values in *RX_BUFFER* or *RX_BUFFER_HEAD and RX_BUFFER_TAIL,* should be done in atomic way within the critical section where USART1 ISR must be temporarily disabled. The critical section is realized by simply enabling and disabling the USART1 ISR:

```
	NVIC_DisableIRQ(USART1_IRQn);
	...
	NVIC_EnableIRQ(USART1_IRQn);
```

Within a critical section, we first check whether the buffer is not empty, and in that case we fetch the oldest value to byref parameter (*char *c*), adjust the value of the buffer tail, and set the *ret* flag to 1, indicating that receive buffer was not empty when *USART1_Dequeue()* function was called:

```
        if (RX_BUFFER_HEAD != RX_BUFFER_TAIL)
        {
                *c = RX_BUFFER[RX_BUFFER_TAIL];
                RX_BUFFER_TAIL++;
                if (RX_BUFFER_TAIL == BUFSIZE) RX_BUFFER_TAIL = 0;
                ret = 1;
        }
```

If the receive buffer is empty, the function returns 0, and the content of byref parameter *char* c* should be ignored.

6.4. Step 4: Implement the loopback functionality in the main module

In *main.h* we import *usart.c* module functions through the *usart.h* header:

```
/* main.h */
#ifndef MAIN_H
#define MAIN_H

#include <stm32f4xx.h>                                      // common stuff
#include <stm32f4xx_gpio.h>                                 // gpio control
#include <stm32f4xx_rcc.h>                                  // reset anc clocking
#include <usart.h>
#endif
```

The *main.c* module source code implementing the loopback interface using high-level functions from *usart.c* is very simple:

```
/* main.c */
#include <main.h>
int main(void)
{
char c;

        USART1_Init();
        while(1)
        {
                if (USART1_Dequeue(&c) != 0)
                {
                        USART1_SendChar(c);
                }
        }
}
```

The *main.h* header contains all necessary common definitions and imports the *usart.c* module functionality. Calling the function *USART1_Init()* will initialize USART1 peripheral (communication settings), GPIO pins, interrupt service routine (IRQ triggers, ISR processing), and housekeeping variables internal to *usart.c* module (*TxReady* flag and *RX_BUFFER* queue). If we want to change anything in the current USART1 setup, we can do that in source code of *usart.h* and *usart.c*.

The main function contains an infinite loop that continously check for incoming characters. Characters are buffered to *RX_BUFFER* in USART1 ISR, and the main loop does not directly poll hardware registers, neither it must finish the current character processing before arrival of the new character. Timing restrictions are relaxed because ISR takes care of fast characters buffering into temporary memory storage. Since the *RX_BUFFER* is a memory resource shared between the main program and the USART1 ISR, reading and writing variables and data related to *RX_BUFFER* must be done in atomic way. Therefore, the *usart.h* header does not expose *RX_BUFFER* related variables outside of the module to prevent direct access in non-atomic way from external modules. The atomicity of *RX_BUFFER* access is achieved by implementing the critical section in *USART1_Dequeue()* function that wraps all internal operations on *RX_BUFFER*, and returns the result of operation and read character, if any.

If the character was successfully read, it is sent through UART1 interface right away. This is done by calling *USART1_SendChar()* function, which will automatically block until USART1 interface is ready for transmission (i.e. last character is completele transmitted), put the next character into outgoing buffer of USART1 interface, and mark *TxReady* value flag to 0, to prevent new character transmission until the current transfer is still in progress.

6.5. Lab outcomes:

Students must individually accomplish the following outcomes:

- working Keil MDK-ARM uVision project, following the guidelines for this part of lab exercise, with the same remarks as in the previous chapters regarding the project building and transfering to STM32F4DISCOVERY development board,
- program that demonstrates loopback functionality,
- extend the code example in a way that the program exits the loopback functionality when it receives '*x*' character from PC, and sends back the message *„now exiting loopback mode“*; write your own *printf()* function to send message (i.e. null-terminated string) through the USART1 interface, based on functions presented in this part of exercise.

Note: If you are using the UART/USB interface based on Silabs CP2102 shown in the picture below, you need to connect RXD signal to PB6 (TX), and TXD signal to PB7 (RX).