SATURDAY MAY 17TH, 2014

Programming ARM Cortex (STM32) under GNU/Linux

This article is a complete intruduction to programming ARM Cortex microcontrolers under GNU/Linux. I will describe how to set up the environment to be able to code, compile, and flash applications into your STM32 MCU. There is no need to install non-free, proprietary, user subordinating software.

What do we need?

We need only few obvious things:

- STM32 microcontroller
- programmer and debugger
- GNU/Linux operating system

In this article, I will use STM32 Nucleo board with STM32F401RE microcontroller. I really recommend this board for all beginners – it's powerfull and cheap. One of the best advantage of STM32 Nucleo board is that it does not require any separate probe as it integrates the ST-LINK/V2-1 debugger/programmer. Integrated ST-LINK can be also used to program/debug other MCUs.

If you own any other board/MCU (like STM32 Discovery) it shouldn't be a problem, all steps will be exactly the same except of selecting library version.

GNU/Linux software requirements

I will use the following software:

- **arm-none-eabi-gcc** – The GNU Compiler Collection – cross compiler for ARM EABI (bare-metal) target
- **arm-none-eabi-gdb** – The GNU Debugger for the ARM EABI (bare-metal) target
- **arm-none-eabi-binutils** – A set of programs to assemble and manipulate binary and object files for the ARM EABI (bare-metal) target
- **openocd** – Debugging, in-system programming and boundary-scan testing for

SPIS TREŚCI

embedded target devices
- **vim** – The text editor of my choice

All of above packages should be available in your GNU/Linux distribution via default repositories. Note that they could have different names, for example in Debian GNU/Linux and Ubuntu the **arm-none-eabi-gcc** is **gcc-arm-none-eabi**.

Tough choice

Before we start, you need to choose between two extremely different possibilities. The first possibility is to use high-level libraries provided by STMicroelectronics (containing the hardware abstraction layer (HAL) for the STM32 peripherals). The second possibility is to learn how to interact with hardware and write your own functions to control MCU internals. As usual, both ways have their own advantages and disadvantages. High-level libraries are developed to be universal which mean you can use the same functions to control different devices, in this case it is obvious that some part of such libraries need to perform a lot extra operations (it will consume more memory and CPU time). As it comes to STM32 HAL library – redundant and overprotective code could be (in most cases) optimized at compile time. High-level libraries are also developed to be an easy to use proxy to complicated internals. What would such an approach mean in practice? Well, this mean that such a library hides as much internals as it is possible – in this case user can focus on what to do instead of how to do it.

I strongly recommend to start without high-level libraries and spend some time reading about MCU internals (datasheet, programming manual, reference manual), this will imply better understanding of the processes taking place inside the processor.

How much is a programmer worth if she/he does not understand the programmed device?

In this part of article I will describe how to start with no external libraries. Second part of article (coming soon) will describe how to compile and use STM32 HAL Driver.

CMSIS – Cortex Microcontroller Software Interface Standard

The ARM® Cortex® Microcontroller Software Interface Standard (CMSIS) is a vendor-independent hardware abstraction layer for the Cortex-M processor series and specifies debugger interfaces. The CMSIS consists of the following components:

- CMSIS-CORE
- CMSIS-Driver
- CMSIS-DSP
- CMSIS-RTOS API
- CMSIS-Pack
- CMSIS-SVD
- CMSIS-DAP

You can read about CMSIS here.

For the purpose of this article, we will use only first component – CMSIS-CORE

CMSIS-CORE gives the user access to the processor core and the device peripherals. It defines:

- Hardware Abstraction Layer (HAL) for Cortex-M processor registers with standardized definitions for the SysTick, NVIC, System Control Block registers, MPU registers, FPU registers, and core access functions.
- System exception names to interface to system exceptions without having

CATEGORIES

- Git (1)
- Other (1)
- Politics (1)
- Programowanie (2)
- Operating systems (6)
  - GNU/Linux (5)
- Theraphosidae (1)
  - Poecilotheria (1)

RECENT POSTS

- (Polski) Uwolnij swoje Arduino – programowanie Arduino bez bibliotek od Arduino
- (Polski) Programowanie mikrokontrolerów – materiały
- Programming ARM Cortex (STM32) under GNU/Linux
- (Polski) Praktyczne przykłady wykorzystania środowiska tekstowego
- (Polski) Tajemnice wejścia/wyjścia – jak zrozumieć deskryptory plików, strumienie i potoki
- (Polski) Nasza POsłanka przeciwko nam?
- (Polski) Poczuj potęgę wolności z GNU/Linux
- (Polski) Wprowadzenie do systemu kontroli wersji Git

RECENT COMMENTS

- mirecta on Programming ARM Cortex (STM32) under GNU/Linux
- Ganesh Babu on Programming ARM Cortex (STM32) under GNU/Linux
- Radu on Programming ARM

compatibility issues.

- Methods to organize header files that makes it easy to learn new Cortex-M microcontroller products and improve software portability. This includes naming conventions for device-specific interrupts.
- Methods for system initialization to be used by each MCU vendor. For example, the standardized SystemInit() function is essential for configuring the clock system of the device.
- Intrinsic functions used to generate CPU instructions that are not supported by standard C functions.
- A variable to determine the system clock frequency which simplifies the setup the SysTick timer.

CMSIS-CORE files:

- <device>.h
  - system_<device>.h
  - core_<cpu>.h
- startup_<device>.s

<device> is replaced with the specific device name or device family name; i.e. **stm32f401xe**, <cpu> is replaced with MCU's Core shortcut; i.e. **cm0** (Cortex M0), **cm4** (Cortex M4).

We need to understand the role of this files:

- *<device>.h* – contains device specific informations: interrupt numbers (IRQn) for all exceptions and interrupts of the device, definitions for the Peripheral Access to all device peripherals (**all data structures and the address mapping for device-specific peripherals**). It also provide additional helper functions for peripherals that are useful for programming of these peripherals.
- *core_<cpu>.h* – defines the core peripherals and provides helper functions that access the core registers (SysTick, NVIC, ITM, DWT etc.).
- *startup_<device>.s* – startup code and system configuration code (reset handler which is executed after CPU reset, exception vectors of the Cortex-M Processor, interrupt vectors that are device specific).

STM32Cube

> *STMCube is an STMicroelectronics original initiative to ease developers life by reducing development efforts, time and cost. (...)*

As you can see, STMicroelectronics introduces STMCube as an initiative to ease developers life. They are sharing packages containing libraries, documentation and examples. Packages are delivered per series (such as STM32CubeF4 for STM32F4 series). In this article I will describe STM32CubeF4 package.

Getting STM32CubeF4

First of all, we need to download STM32CubeF4 package. You can get it from STMicroelectronics official site: here.

STM32CubeF4 content

After unpacking STM32CubeF4 package, we should have the following directory structure:

```
$ tree -L 2
.
├── Documentation
│   └── STM32CubeF4GettingStarted.pdf
├── Drivers
│   ├── BSP
│   ├── CMSIS
│   └── STM32F4xx_HAL_Driver
```

Cortex (STM32) under GNU/Linux
- yytyt on (Polski) Wprowadzenie do systemu kontroli wersji Git
- Patryk Jaworski on Programming ARM Cortex (STM32) under GNU/Linux
- norbert on Programming ARM Cortex (STM32) under GNU/Linux
- mi on (Polski) Nasza POsłanka przeciwko nam?
- ibiss on (Polski) Praktyczne przykłady wykorzystania środowiska tekstowego

GITHUB

Regalis @ GitHub

- Rover5-STM32
  Rover 5 Robot Chassis with STM32F401RE (Nucleo)
- Jamii
- regalis-lcd-driver
  AVR driver for alphanumeric LCD modules (HD44780)
- regalis-signals
  Minimalistic implementation of signals and slots mechanism written in C++11 using some of new features including variadic templates, lambdas and smart pointers.
- mkinitramfs-squashfs
  SquashFS root support for mkinitcpio (remote and local image locations supported)
- jupiter-avr-robot
  Autonomous robot based on Atmel 8-bit microcontroller Atmega8
- regalis-newsletter
  Simple centralized newsletter system
- iiujasd
  II UJ ASD
- RegalisCMS
  Powerfull, hi-speed, modular Content Management System

```
├── _htmresc
│   ├── CMSIS_Logo_Final.jpg
│   ├── Eval_archi.bmp
│   ├── logo.bmp
│   ├── ReleaseNotes.html
│   ├── st_logo.png
│   └── STM32Cube_components.bmp
├── Middlewares
│   ├── ST
│   └── Third_Party
├── package.xml
├── Projects
│   ├── STM324x9I_EVAL
│   ├── STM324xG_EVAL
│   ├── STM32F401-Discovery
│   ├── STM32F429I-Discovery
│   ├── STM32F4-Discovery
│   └── STM32F4xx-Nucleo
├── Release_Notes.html
└── Utilities
    ├── CPU
    ├── Fonts
    ├── Log
    ├── Media
    └── PC_Software

22 directories, 9 files
```

We are mostly interested in **Drivers/** directory since it is the place where both CMSIS and STM32 HAL drivers are stored.

Let's find previously mentioned CMSIS files (<device>.h, core_<cpu>.h etc.). They are in the **Drivers/CMSIS/** directory:

```
$ tree Drivers/CMSIS/Include/
Drivers/CMSIS/Include/
├── arm_common_tables.h
├── arm_const_structs.h
├── arm_math.h
├── core_cm0.h
├── core_cm0plus.h
├── core_cm3.h
├── core_cm4.h
├── core_cm4_simd.h
├── core_cmFunc.h
├── core_cmInstr.h
├── core_sc000.h
└── core_sc300.h

0 directories, 12 files
```

Device specific files (<device>.h) are in the **Drivers/CMSIS/Device/ST/STM32F4xx/Include/** directory:

```
$ tree Drivers/CMSIS/Device/ST/STM32F4xx/Include/
Drivers/CMSIS/Device/ST/STM32F4xx/Include/
├── stm32f401xc.h
├── stm32f401xe.h
├── stm32f405xx.h
├── stm32f407xx.h
├── stm32f415xx.h
├── stm32f417xx.h
├── stm32f427xx.h
├── stm32f429xx.h
├── stm32f437xx.h
├── stm32f439xx.h
├── stm32f4xx.h
└── system_stm32f4xx.h

0 directories, 12 files
```

Note: this is basically all we need to create first project (without STM32 HAL library). Let's see where to find STM32 HAL Driver:

```
$ tree -F -L 1 Drivers/STM32F4xx_HAL_Driver/
Drivers/STM32F4xx_HAL_Driver/
├── Inc/
├── Release_Notes.html
└── Src/

2 directories, 1 file
```

STM32 HAL Driver defines a number of structures and functions to configure all of STM32 peripherals (like USART, SPI, GPIO, SDIO, DMA). HAL Driver is divided into multiple files:

```
$ tree -F -L 1 Drivers/STM32F4xx_HAL_Driver/Inc/
Drivers/STM32F4xx_HAL_Driver/Inc/
├── stm32f4xx_hal_adc_ex.h
├── stm32f4xx_hal_adc.h
├── stm32f4xx_hal_can.h
├── stm32f4xx_hal_conf_template.h
├── stm32f4xx_hal_cortex.h
├── stm32f4xx_hal_crc.h
├── stm32f4xx_hal_cryp_ex.h
├── stm32f4xx_hal_cryp.h
├── stm32f4xx_hal_dac_ex.h
├── stm32f4xx_hal_dac.h
├── stm32f4xx_hal_dcmi.h
├── stm32f4xx_hal_def.h
├── stm32f4xx_hal_dma2d.h
├── stm32f4xx_hal_dma_ex.h
├── stm32f4xx_hal_dma.h
(...) cut (...)
├── stm32f4xx_hal_spi.h
├── stm32f4xx_hal_sram.h
├── stm32f4xx_hal_tim_ex.h
├── stm32f4xx_hal_tim.h
├── stm32f4xx_hal_uart.h
├── stm32f4xx_hal_usart.h
├── stm32f4xx_hal_wwdg.h
├── stm32f4xx_ll_fmc.h
├── stm32f4xx_ll_fsmc.h
├── stm32f4xx_ll_sdmmc.h
└── stm32f4xx_ll_usb.h

0 directories, 57 files
```

I need to mention one special file named **stm32f4xx_hal_conf_template.h**. It is the only one file we need to copy into our project directory and name it **stm32f4xx_hal_conf.h**. But for now – let's forget about it.

Minimal configuration – no external libraries

The idea:

- create project directory containing:
  - **main.c** – main program
  - **system.c** – implementation of CMSIS *system_stm32f4xx.h* (system initialization – clock source, flash memory configuration etc.)
- copy startup code into project directory
- copy linker script into project directory
- compile, link and write code to MCU's flash memory

Let's start with describing MCU's startup procedure. After reset (power on) MCU works with HSI (internal high-speed oscilator) as system clock source. In my case (STM32F401RE), HSI = 16MHz. Assuming that we boot from Main Flash memory, MCU starts code execution from the boot memory starting from **0×00000004**. This

is the place where we need to put an address of initialization function. This function is usually named **Reset_Handler** and must do the following job:

- set stack pointer (usually at the end of SRAM)
- copy .data section from flash to SRAM
- zero fill the .bss section (in SRAM)
- call CMSIS **SystemInit()** function
- call libc **__libc_init_array()** function
- call **main()**

STMicroelectronics provides startup code in file **startup_stm32f401xe.s** (assembler), we need to copy it from **STM32CubeF4Root/Drivers/CMSIS/Device/ST/STM32F4xx/Source/Templates/gcc/startup_stm32f401xe.s** or write own implementation.

Now, let's discuss the role of **SystemInit()** function:

- configure embedded linear voltage regulator
- configure clock source:
  - calibrate internal HSI
  - set HSI as PLL source
  - configure PLL
  - enable PLL
  - wait until PLL becomes stable
  - configure Flash memory:
    - enable instruction cache
    - enable prefetch buffer
    - set correct latency
  - set system clock source to PLL
  - configure HCLK
  - configure APB1 and APB2 prescallers

Note: I use HSI as an input clock for PLL. You can replace it with HSE if you are using external, more accurate clock source.

I already implemented all above steps for my board (Nucleo with STM32F401RE):

```
/*
*
* Copyright (C) Patryk Jaworski <regalis@regalis.com.pl>
*
* This program is free software: you can redistribute it and/or modify
* it under the terms of the GNU General Public License as published by
* the Free Software Foundation, either version 3 of the License, or
* (at your option) any later version.
*
* This program is distributed in the hope that it will be useful,
* but WITHOUT ANY WARRANTY; without even the implied warranty of
* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
* GNU General Public License for more details.
*
* You should have received a copy of the GNU General Public License
* along with this program. If not, see <http://www.gnu.org/licenses/>.
*
*/
#include <stm32f4xx.h>

/* Helpers for SystemInitError() */
#define SYSTEM_INIT_ERROR_FLASH 0x01
#define SYSTEM_INIT_ERROR_PLL 0x02
#define SYSTEM_INIT_ERROR_CLKSRC 0x04
#define SYSTEM_INIT_ERROR_HSI 0x08

void SystemInit() {
        /* Enable Power Control clock */
        RCC->APB1ENR |= RCC_APB1LPENR_PWRLPEN;
        /* Regulator voltage scaling output selection: Scale 2 */
        PWR->CR |= PWR_CR_VOS_1;

        /* Wait until HSI ready */
        while ((RCC->CR & RCC_CR_HSIRDY) == 0);

        /* Store calibration value */
        PWR->CR |= (uint32_t)(16 << 3);
```

```
        /* Disable main PLL */
        RCC->CR &= ~(RCC_CR_PLLON);
        /* Wait until PLL ready (disabled) */
        while ((RCC->CR & RCC_CR_PLLRDY) != 0);

        /*
         * Configure Main PLL
         * HSI as clock input
         * fvco = 336MHz
         * fpllout = 84MHz
         * fusb = 48MHz
         * PLLM = 16
         * PLLN = 336
         * PLLP = 4
         * PLLQ = 7
         */
        RCC->PLLCFGR = (uint32_t)((uint32_t)0x20000000 | (uint32_t)(16 << 0) | (uint32_t)(336
                                        RCC_PLLCFGR_PLLP_0 | (uint32_t)(7 << 24));

        /* PLL On */
        RCC->CR |= RCC_CR_PLLON;
        /* Wait until PLL is locked */
        while ((RCC->CR & RCC_CR_PLLRDY) == 0);

        /*
         * FLASH configuration block
         * enable instruction cache
         * enable prefetch
         * set latency to 2WS (3 CPU cycles)
         */
        FLASH->ACR |= FLASH_ACR_ICEN | FLASH_ACR_PRFTEN | FLASH_ACR_LATENCY_2WS;

        /* Check flash latency */
        if ((FLASH->ACR & FLASH_ACR_LATENCY) != FLASH_ACR_LATENCY_2WS) {
                SystemInitError(SYSTEM_INIT_ERROR_FLASH);
        }

        /* Set clock source to PLL */
        RCC->CFGR |= RCC_CFGR_SW_PLL;
        /* Check clock source */
        while ((RCC->CFGR & RCC_CFGR_SWS_PLL) != RCC_CFGR_SWS_PLL);

        /* Set HCLK (AHB1) prescaler (DIV1) */
        RCC->CFGR &= ~(RCC_CFGR_HPRE);

        /* Set APB1 Low speed prescaler (APB1) DIV2 */
        RCC->CFGR |= RCC_CFGR_PPRE1_DIV2;

        /* SET APB2 High speed srescaler (APB2) DIV1 */
        RCC->CFGR &= ~(RCC_CFGR_PPRE2);
}

void SystemInitError(uint8_t error_source) {
        while(1);
}
```

It is time to write **main.c:**

```
/*
*
* Copyright (C) Patryk Jaworski <regalis@regalis.com.pl>
*
* This program is free software: you can redistribute it and/or modify
* it under the terms of the GNU General Public License as published by
* the Free Software Foundation, either version 3 of the License, or
* (at your option) any later version.
*
* This program is distributed in the hope that it will be useful,
* but WITHOUT ANY WARRANTY; without even the implied warranty of
* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
* GNU General Public License for more details.
*
* You should have received a copy of the GNU General Public License
* along with this program. If not, see <http://www.gnu.org/licenses/>.
*
*/
#include <stm32f4xx.h>

#define LED_PIN 5
#define LED_ON() GPIOA->BSRRL |= (1 << 5)
#define LED_OFF() GPIOA->BSRRH |= (1 << 5)

int main() {
        /* Enbale GPIOA clock */
        RCC->AHB1ENR |= RCC_AHB1ENR_GPIOAEN;
        /* Configure GPIOA pin 5 as output */
```

```
        GPIOA->MODER |= (1 << (LED_PIN << 1));
        /* Configure GPIOA pin 5 in max speed */
        GPIOA->OSPEEDR |= (3 << (LED_PIN << 1));

        /* Turn on the LED */
        LED_ON();

}
```

When we have all required files (system.c, main.c, startup_stm32f401xe.s), we can compile the project. I use the following command to compile **single file**:

```
$ arm-none-eabi-gcc -Wall -mcpu=cortex-m4 -mlittle-endian -mthumb -mthumb-interwork -ISTM32Cub
```

Options and arguments description:

- **-Wall** – enable all warnings
- **-mcpu=cortex-m4** – specify the target processor
- **-mlittle-endian** – compile code for little endian target
- **-mthumb** – generate core that executes in Thumb states
- **-mthumb-interwork** – generate code that supports calling between the ARM and Thumb instruction sets
- **-ISTM32CubeF4Root/Drivers/CMSIS/Include** – append directory to compiler list of directories which will be used to search for headers included with #include preprocessor directive. **Note:** replace *STM32CubeF4Root* with an absolute path to your STM32 Cube root directory
- **-DSTM32F401xE** – define target processor (used in device header files)
- **-Os** – optimize for size
- **-c** – do not run linker, just compile
- **system.c** – input file name
- **-o system.o** – output file name

You need to perform this operation for all your source files. After successfull compilation, you need to have **.o** files for all your **.c** and **.s** sources.

To link ***.o** files into single "executable", I use the following command:

```
$ arm-none-eabi-gcc -mcpu=cortex-m4 -mlittle-endian -mthumb -mthumb-interwork -DSTM32F401xE -T
```

Options and arguments (only new):

- **-TSTM32CubeF4Root/Projects/STM32F4xx-Nucleo/Templates/TrueSTUDIO/STM32F4xx-Nucleo/STM32F401CE_FLASH.ld** – use specific linker script, I use script provided in STM32 Cube package. As above, you need to replace *STM32CubeF4Root* with an absolute path to your STM32 Cube root directory
- **-Wl,--gc-sections** – enable garbage collection of unused input sections
- **system.o main.o startup_stm32f401xe.o** – input files
- **-o main.elf** – output file name

We need only one more step to upload code into our device – convert ELF binary into Intel Hex format:

```
$ arm-none-eabi-objcopy -Oihex main.elf main.hex
```

That is all. Now we can connect programmer/board and upload our code with **OpenOCD**. I use the following command to run openocd:

```
$ openocd -f /usr/share/openocd/scripts/board/st_nucleo_f401re.cfg
```

Note: script path may differ accross GNU/Linux disctributions, check content of openocd package in your distribution to find valid path.

After successfull connection, openocd will accept commands on localhost port 4444. We need to open new terminal and run:

```
$ telnet localhost 4444
```

Then, in openocd telnet session:

```
> reset halt
> flash write_image erase main.hex
> reset run
```

The best practice is to put all of above commands into single Makefile, I will describe how to do this in next part of this article (coming soon).

Happy hacking!

Share:

This entry was posted in GNU/Linux.
Bookmark the permalink.
5 Comments

← PREVIOUS **NEXT →**

5 THOUGHTS ON "PROGRAMMING ARM CORTEX (STM32) UNDER GNU/LINUX"

norbert says: Wednesday May 21st, 2014 at 05:48 PM

Hi,
your tutorial was quite helpful for me understanding how to compile source code using an arm-none-eabi-gcc toolchain for the nucleo F401RE board.
Could you provide a listing of the files in your final project folder too? (with the tree command for example?)
I struggle with the problem, that during startup the call to __libc_init_array seems to hang in an infinity loop … have you expericences something similar or have some advice?
Thx

Reply

Patryk Jaworski says: Wednesday May 21st, 2014 at 06:32 PM

Hi, my nucleo-templace directory looks like this:

```
$ tree nucleo-template
nucleo-template
├── delay.c (SysTick configuration to tick every ~1us)
├── delay.h
├── main.c (main program)
├── Makefile
├── system
│   ├── startup_stm32f401xe.s
│   └── STM32F401CE_FLASH.ld
├── system.c (clock source and flash memory configuration)
├── usart.c (uart configuration)
└── usart.h

1 directory, 9 files
```

Make sure you use the right command to link your object files. Following options are required to make it work:

```
-mcpu=cortex-m4   -mlittle-endian   -mthumb   -mthumb-interwork   -
```

```
DSTM32F401xE -TSTM32F401CE_FLASH.ld -Wl,--gc-sections
```

Reply

Radu says: Wednesday July 16th, 2014 at 07:46 PM

Thank you for your introduction into programming the STM32F4 under Linux. Looking forward to read your STM32F4Cube tutorial. Do you already have a template for a cube based project?

Reply

Ganesh Babu says: Sunday July 27th, 2014 at 07:06 AM

Hi, I would like to know If we can run liunx on STM32F405xx,,is it possible to run a linux Kernel on it…?If not why??Thanks for the valuable time…

Reply

mirecta says: Saturday November 1st, 2014 at 03:37 PM

can u explain:
why is need option -mthumb-interwork ?
in arm website is: all cortex-m cores is thumb capable only so they have not arm instruction set only thumb

Reply

LEAVE A REPLY

Your email address will not be published. Required fields are marked *

NAME *

EMAIL *

WEBSITE

COMMENT

Please type the characters of this captcha image in the input box

Post Comment