3. First Program

There is a long tradition in programming tutorials to present some variant of the "Hello World" example as the first program.

This makes sense in desktop programming, where there is a huge hardware and software infrastructure already prepared that we can take for granted and where a single line of code seems to be a "basic" programming statement.

This does not work in the embedded world.

Before we will come to the point where we will be able to display a "Hello World" message, we will be already pretty advanced in the use of Ada on ARM Cortex-M. For now, let's start with something simple - as simple as an empty program that does absolutely nothing. We will use it to exercise all development tools.

Our program will be composed of two files: the specification (similar to a header file in C or C++) and a body. The specification (in file `program.ads`) will be very short:

```
package Program is

   procedure Run;
   pragma Export (C, Run, "run");

end Program;
```

Ada uses packages for organizing program elements and here we have package `Program` that will act as a logical container for everything else. That "everything else" is just a single procedure called `Run` - this procedure has no parameters and does not produce any value and is exactly what we need for a starting point of our program.

Note that you can use different names for both the package and the startup procedure, there is no obligation to use any specific name. You could, for example, use `Main` as a name for the startup procedure.

Note also that there is a pragma that defines an external name for the `Run` procedure (the external name will be `run`, but you can use anything else) - the external name is something that other tools will be able to see and refer to. That is, there will be a single procedure in the program, but it will be visible under two different names depending on the point of view: in Ada the procedure will be visible as `Run` and external tools will see the same procedure as `run`.

Another source file is needed with the body of the procedure `Run` - this file, named `program.adb`, contains:

```
package body Program is

   procedure Run is
   begin
      loop
         null;
      end loop;
   end Run;

end Program;
```

The procedure `Run` is the only entity in the package `Program` and is itself very simple: it contains an empty loop (the only statement in the loop is a null statement), which prevents the procedure from returning. We do not want this procedure to return anywhere, as it is a startup procedure, the root of the whole call graph in our program.

Just to help you with some analogies, the C programmer might write a similar procedure like this:

```
void run(void)
{
    for (;;)
    {
    }
}
```

I hope that the meaning of Ada keywords above is now well understood.

Note that it is not strictly true that procedure `Run` does nothing at all: actually, it is very busy wasting energy by endlessly spinning in the loop, millions of times per second. This is not the optimal solution, especially for battery-powered devices, but we will keep it like that until later chapters.

It is not necessary to know assembly language, but we will occasionally have a look at the assembly output for our program. You can ask the compiler to generate the assembly output with this command:

```
$ gcc -S -mcpu=cortex-m0 -mthumb -mfloat-abi=soft program.adb
```

or

```
$ gcc -S -mcpu=cortex-m3 -mthumb program.adb
```

(Remember that with cross-compilation tools, your compiler will have a different name, for example it will be `arm-eabi-gcc` with GNAT on Windows.)

The first variant above is appropriate for devices with the Cortex-M0 chip, which does not support floating-point instructions, the second variant will be used for Cortex-M3 or better.

The `-S` parameter above tells the compiler to prepare the assembly output instead of an object file. The `-mcpu` and `-mthumb` options are needed to limit the set of instructions to those understood by the target microcontroller - in the first variant above, the Cortex-M0 chip was selected, which does not support floating-point instructions, and the second variant will be used for Cortex-M3 or better. When working with a specific chip, it makes sense to use values that correspond with the target device - thanks to this, the compiler will be free to use more elaborate instruction sets, leading to executables that are both smaller and faster.

If the above command was executed without errors, a new file (`program.s`) should be created in the same directory. Its content might be similar to this:

```
        .cpu cortex-m0
        .eabi_attribute 27, 3
        .fpu vfp
        .eabi_attribute 20, 1
        .eabi_attribute 21, 1
        .eabi_attribute 23, 3
        .eabi_attribute 24, 1
        .eabi_attribute 25, 1
        .eabi_attribute 26, 2
        .eabi_attribute 30, 6
        .code   16
        .file   "program.adb"
        .global program_E
        .data
        .type   program_E, %object
        .size   program_E, 1
program_E:
        .space  1
        .text
        .align  2
        .global run
        .code   16
        .thumb_func
        .type   run, %function
run:
        push    {r7, lr}
        add     r7, sp, #0
.L4:
        mov     r8, r8
.L2:
        mov     r8, r8
        b       .L4
        .size   run, .-run
        .ident  "GCC: (Debian 4.6.3-8+rpi1) 4.6.3"
        .section        .note.GNU-stack,"",%progbits
```

It is not important to understand everything in this file, but there is one thing that is worth noting: the `run` symbol that marks the beginning of our `Run` function. The `run` name is exactly what we have used as external procedure name and this was propagated to the assembly output.

Let's try to compile the same program to create an object file:

```
$ gcc -c -mcpu=cortex-m0 -mthumb -mfloat-abi=soft program.adb
```

The `-c` option tells the compiler to only compile the given file, without attempting to link a full executable. A new file should

be created in the same directory, called `program.o`. It is a binary file.

We can check the list of all symbols used in this file with this command:

```
$ nm program.o
```

The output of this command might look like:

```
00000000 D program_E
00000000 T run
```

We need not worry about the `program_E` symbol (although you might want to check the assembly output above and see that it was there as well), but we will be happy to see `run` in this object file - this means that it contains our startup procedure.

Now, we can try to link our program into an executable file. The linking process will be controlled by a very short linker script, which I will explain in the next chapter - here, in order not to lose the flow (and assuming Arduino M0 as a target), let's invoke the linker with this command:

```
$ ld -T flash.ld -o program.elf program.o
```

As a result, the `program.elf` file should be created in the same directory. This file is an executable file, but we have no means to use it directly on the board. Still, it contains all the information that was accumulated so far - in particular, you can check its symbols:

```
$ nm program.elf
```

This time the output is a bit different:

```
0000410c D program_E
00004100 T run
```

Note that before (with the `program.o` file) the addresses on the left were all 0 and now the `run` symbol has address 4100 - the importance of this fact will be clarified in the next chapter, but for the time being keep in mind that these addresses are related to the memory layout of the target device and for each microcontroller can be different.

I have mentioned that there is no direct way to use this executable file on the board, so we need to convert it to the raw binary image that will contain exactly the bytes that we want to upload to the flash memory in the microcontroller. This command will do it:

```
$ objcopy -O binary program.elf program.bin
```

The last and final file, `program.bin`, contains the raw binary image that we want to upload to the board.

An alternative format that can be used for data transfer - and the one that we will use for Arduino M0 - is called Intel Hex and can be obtained with:

```
$ objcopy -O ihex program.elf program.hex
```

We will use extensions `.bin` and `.hex` to distinguish the final file formats.

The steps needed to actually upload such images to the flash memory are different for each board.

Try to become comfortable with all the steps presented above and in the following sections (and don't hesitate to experiment a bit with them), they will be regularly repeated with each new program.

Arduino M0

Connect the board to the USB port in your computer.

In order to reduce the effort needed to discover all the details, I have uploaded a very simple sketch from Arduino IDE to the board and carefully observed the commands that were printed in the output window. The last of these commands referred to the Avrdude tool with a dedicated configuration script - as already mentioned, it makes sense to copy them to some more comfortable place and after doing so, it should be possible to upload the executable file with this command:

```
$ ./avrdude -Cavrdude.conf -v -v -patmega2560 -cstk500v2
```

```
            -P/dev/cu.usbmodem0041 -b57600 -Uflash:w:program.hex:i
```

Note that `avrdude.cfg` is a configuration file from the Arduino IDE installation. The `program.hex` file is the image file created earlier.

The above command should be executed shortly after pressing the RESET button on the connected board - the bootloader that is already installed in the device waits for the first couple of seconds for the data transmission (the LED on the board blinks rapidly during that time) and these few seconds of time window is when the upload should be initiated from the host computer (note that during that short time window the device can be visible in the system under some temporary name like `/dev/cu.usbmodem0041` - use the Arduino IDE output to check that name on your particular host system), and if no upload is initiated during that time, the board will automatically transition to normal operation.

The result of executing the above command might look similar to this:

```
avrdude: AVR device initialized and ready to accept instructions

Reading | ################################################## | 100% 0.00s

avrdude: Device signature = 0x1e9801
avrdude: NOTE: "flash" memory has been specified, an erase cycle
         will be performed
         To disable this feature, specify the -D option.
avrdude: erasing chip
avrdude: reading input file "program.hex"
avrdude: writing flash (16748 bytes):

Writing | ################################################## | 100% 0.00s

avrdude: 16748 bytes of flash written
avrdude: verifying flash memory against program.hex:
avrdude: load data flash data from input file program.hex:
avrdude: input file program.hex contains 16748 bytes
avrdude: reading on-chip flash data:

Reading | ################################################## | 100% 0.00s

avrdude: verifying ...
avrdude: 16748 bytes of flash verified

avrdude: safemode: Fuses OK (H:00, E:00, L:00)

avrdude done.  Thank you.
```

The program is now in the flash memory and will start executing automatically or after each reset and power on.

Arduino/Genuino Zero

The Arduino Zero and Arduino M0 boards come from different vendors (which are in disagreement with regard to the brand name) and even though they share a lot in terms of design, they use different loaders.

The Zero board can be programmed with the use of OpenOCD, which, after analyzing the Arduino IDE output and copying the involved files to some more accessible place, can be invoked with this command:

```
$ ./openocd -s scripts/ -f arduino_zero.cfg -c 'telnet_port disabled;
            program program.bin verify reset 0x00002000; exit'
```

Note that `arduino_zero.cfg` is a configuration file from the Arduino IDE installation. This file, together with a whole set of other scripts, is located in the `scripts` directory. The `program.bin` file is the binary executable created earlier and the 0x00002000 value above is an offset that will be explained in the next chapter.

The result of executing the above command might look similar to this:

```
Open On-Chip Debugger 0.9.0-gd4b7679 (2015-06-10-19:16)
Licensed under GNU GPL v2
For bug reports, read
        http://openocd.org/doc/doxygen/bugs.html
Info : only one transport option; autoselect 'swd'
adapter speed: 500 kHz
adapter_nsrst_delay: 100
cortex_m reset_config sysresetreq
Info : CMSIS-DAP: SWD  Supported
Info : CMSIS-DAP: Interface Initialised (SWD)
Info : CMSIS-DAP: FW Version = 02.01.0157
Info : SWCLK/TCK = 1 SWDIO/TMS = 1 TDI = 1 TDO = 1 nTRST = 0 nRESET = 1
Info : CMSIS-DAP: Interface ready
Info : clock speed 500 kHz
Info : SWD IDCODE 0x0bc11477
```

```
Info : at91samd21g18.cpu: hardware has 4 breakpoints, 2 watchpoints
target state: halted
target halted due to debug-request, current mode: Thread
xPSR: 0x81000000 pc: 0x00002208 msp: 0x20008000
** Programming Started **
auto erase enabled
Info : SAMD MCU: SAMD21G18A (256KB Flash, 32KB RAM)
Warn : Adding extra erase range, 00000000 to 0x00001fff
wrote 8192 bytes from file program.bin in 1.193115s (6.705 KiB/s)
** Programming Finished **
** Verify Started **
verified 269 bytes in 0.076565s (3.431 KiB/s)
** Verified OK **
** Resetting Target **
```

The program is now in the flash memory and will start executing after reset (which is ensured by the OpenOCD at the end) or after each power on.

Note that since the M0 and Zero boards share the same microcontroller and pin mapping, later examples will focus on only one of these boards.

Arduino Due

Connect the board to the USB port in your computer and try to figure out how it was detected, exactly in the same way as you would do with Arduino IDE.

The Arduino IDE uses some tricks with USB transmission speed to force the board to reset in the loader mode. We will not attempt to repeat these tricks and instead will do the upload manually.

First, press the ERASE button on the board (it is in the area below descriptions for pins 15 and 16); after that press RESET (in the corner) and execute the bossac command:

```
$ bossac -p tty.usbmodemfa141 -U false -e -w -v -b program.bin -R
```

You should see something like this:

```
Erase flash
Write 269 bytes to flash
[==============================] 100% (2/2 pages)
Verify 269 bytes of flash
[==============================] 100% (2/2 pages)
Verify successful
Set boot flash true
```

After that your program is in the flash memory of the board and from now on every time the board is reset (or powered on), the program will automatically start execution.

STM32 Nucleo-32

Connect the board to the USB port in your computer.

We will use the OpenOCD tool, which in its standard package already contains the configuration script for STM32F0x boards. The appropriate command is:

```
$ ./openocd -s scripts/ -f board/st_nucleo_f0.cfg
            -c 'program program.bin reset exit 0x08000000'
```

The `program.bin` file is the binary executable created earlier and the 0x08000000 value above is an offset that will be explained in the next chapter.

The result of executing the above command might look similar to this:

```
Open On-Chip Debugger 0.9.0-gd4b7679 (2015-06-10-19:16)
Licensed under GNU GPL v2
For bug reports, read
        http://openocd.org/doc/doxygen/bugs.html
Info : The selected transport took over low-level target control.
        The results might differ compared to plain JTAG/SWD
adapter speed: 1000 kHz
adapter_nsrst_delay: 100
none separate
srst_only separate srst_nogate srst_open_drain connect_deassert_srst
Info : Unable to match requested speed 1000 kHz, using 950 kHz
Info : Unable to match requested speed 1000 kHz, using 950 kHz
```

```
Info : clock speed 950 kHz
Info : STLINK v2 JTAG v24 API v2 SWIM v11 VID 0x0483 PID 0x374B
Info : using stlink api v2
Info : Target voltage: 3.253649
Info : stm32f0x.cpu: hardware has 4 breakpoints, 2 watchpoints
target state: halted
target halted due to debug-request, current mode: Thread
xPSR: 0xc1000000 pc: 0x08000100 msp: 0x20001000
** Programming Started **
auto erase enabled
Info : device id = 0x10006444
Info : flash size = 32kbytes
target state: halted
target halted due to breakpoint, current mode: Thread
xPSR: 0x61000000 pc: 0x2000003a msp: 0x20001000
wrote 1024 bytes from file program.bin in 0.117594s (8.504 KiB/s)
** Programming Finished **
** Resetting Target **
shutdown command invoked
```

The program is now in the flash memory and will start executing after reset (which is ensured by the OpenOCD at the end) or after each power on.

STM32 Nucleo-144

Connect the board to the USB port in your computer.

We will use the OpenOCD tool, which from version 0.10 should contain the configuration script for STM32F7x boards, but since this tutorial was written before the official release of this version, some tips might be helpful if you try to use some earlier variant.

Check the `scripts/board/st_nucleo_f7.cfg` file and if it does not exist, create it as a copy of `st_nucleo_f4.cfg` in the same directory and replace the line:

```
source [find target/stm32f4x.cfg]
```

with:

```
source [find target/stm32f7x.cfg]
```

With this modification it should be possible to run the command:

```
$ ./openocd -s scripts/ -f board/st_nucleo_f7.cfg
            -c 'program program.bin reset exit 0x08000000'
```

The `program.bin` file is the binary executable created earlier and the 0x08000000 value above is an offset that will be explained in the next chapter.

The result of executing the above command might look similar to this:

```
Open On-Chip Debugger 0.10.0-dev-00247-g73b676c (2016-03-09-23:22)
Licensed under GNU GPL v2
For bug reports, read
        http://openocd.org/doc/doxygen/bugs.html
Info : The selected transport took over low-level target control.
        The results might differ compared to plain JTAG/SWD
adapter speed: 2000 kHz
adapter_nsrst_delay: 100
srst_only separate srst_nogate srst_open_drain connect_deassert_srst
srst_only separate srst_nogate srst_open_drain connect_deassert_srst
Info : Unable to match requested speed 2000 kHz, using 1800 kHz
Info : Unable to match requested speed 2000 kHz, using 1800 kHz
Info : clock speed 1800 kHz
Info : STLINK v2 JTAG v25 API v2 SWIM v13 VID 0x0483 PID 0x374B
Info : using stlink api v2
Info : Target voltage: 3.233618
Info : stm32f7x.cpu: hardware has 8 breakpoints, 4 watchpoints
stm32f7x.cpu: target state: halted
target halted due to debug-request, current mode: Thread
xPSR: 0x01000000 pc: 0x08000128 msp: 0x20010000
** Programming Started **
auto erase enabled
Info : device id = 0x10016449
Info : flash size = 1024kbytes
stm32f7x.cpu: target state: halted
target halted due to breakpoint, current mode: Thread
xPSR: 0x61000000 pc: 0x20000046 msp: 0x20010000
```

```
wrote 32768 bytes from file program.bin in 0.961748s (33.273 KiB/s)
** Programming Finished **
** Resetting Target **
shutdown command invoked
```

The program is now in the flash memory and will start executing after reset (which is ensured by the OpenOCD at the end) or after each power on.

Previous: Documentation and Tools, next: Linking and Booting.
See also Table of Contents.