ARM-ASM-Tutorial

Aus der Mikrocontroller.net Artikelsammlung, mit Beiträgen verschiedener Autoren (siehe Versionsgeschichte)

The ARM processor architecture is widely used in all kinds of industrial applications and also a significant number of hobby and maker projects. This tutorial aims to teach the fundamentals of programming ARM processors in assembly language.

Tutorial by Niklas Gürtler. Thread in Forum for feedback and questions.

Inhaltsverzeichnis

Introduction

Why assembly?

Today, there is actually little reason to use assembly language for entire projects, because high-quality optimizing compilers for high-level languages (especially C and C++) are readily available as free open source software and because the ARM architecture is specifically optimized for high-level languages. However, knowledge in assembly is still useful for debugging certain problems, writing low-level software such as bootloaders and operating system kernels, and reverse engineering software for which no source code is available. Occasionally it is necessary to manually optimize some performance-critical code section. Sometimes claims are made that ARM processors can't be programmed in assembly. Therefore, this tutorial will show that this is very well possible by showing how to write entire (small) applications entirely in the ARM assembly language!

As most of the resources and tools for ARM focus on C programming and because of the complexity of the ARM ecosystem, the largest difficulty in getting started with ARM assembly is not the language

itself, but rather using the tools correctly and finding relevant documentation. Therefore, this tutorial will focus on the development environment and how the written assembly code is transformed into the final program. With a good understanding of the environment, all the ARM instructions can be learned simply by reading the architecture documentation.

Because of the complex ecosystem around ARM, a general introduction of the ARM processor market is necessary.

About ARM

Arm Holdings is the company behind the ARM architecture. Arm does not manufacture any processors themselves, but designs the “blueprints” for processor cores, which are then licensed by various semiconductor companies such as ST, TI, NXP and many others, who combine the processor with various support hardware (most notably flash and RAM memories) and peripheral modules to produce a final complete processor IC. Some of these peripheral modules are even licensed from other companies – for example, the USB controller modules by Synopsys are found in many different processors from various manufacturers.

Because of this licensing model, ARM processor cores are found in a very large variety of products for which software can be developed using a single set of tools (especially compiler, assembler and debugger). This makes knowledge about the ARM architecture, particularly the ARM assembly language, useful for a large range of applications.

Since the ARM processor cores always require additional hardware modules to function, both the ARM-made processor core and the manufacturer-specific periphery modules have to be considered when developing software for ARM systems. For example, the instruction set is defined by ARM and software tools (compiler, assembler) need to be configured for the correct instruction set version, while the clock configuration is manufacturer-specific and needs to be addressed by initialization code specifically made for one processor.

Architecture and processor variants

A processor’s architecture defines the interface between hardware and software. Its most important part is the instruction set, but it also defines e.g. hardware behavior under exceptional circumstances (e.g. memory access errors, division by zero, etc.). Processor architectures evolve, so they have multiple versions and variants. They also define optional functionality that may or may not be present in a processor (e.g. a floating-point unit). For ARM, the architectures are documented exhaustively in the “ARM Architecture Reference Manuals”.

While the architecture is an abstract concept, a processor core is a concrete definition of a processor (e.g. as a silicon layout or HDL) that implements a certain architecture. Code that only uses knowledge of the architecture (e.g. an algorithm that does not access any periphery) will run on any processor implementing this architecture. Arm, as mentioned, designs processor cores for their own

architectures, but some companies develop custom processors that conform to an ARM architecture, for example Apple and Qualcomm.

ARM architectures are numbered, starting with ARMv1 up until the most recent ARMv8. ARMv6 is the oldest architecture still in significant use, while ARMv7 is the most widespread one. Suffixes are appended to the version to denote variants of the architecture; e.g. ARMv7-M is for small embedded systems while ARMv7-A for more powerful processors. ARMv7E-M adds digital signal processing capabilities including saturating and SIMD operations.

Older ARM processors are named ARM1, ARM2 …, while after ARM11 the name “Cortex” was introduced. The Cortex-M family, including e.g. Cortex-M3 and Cortex-M4 (implementing ARMv7-M and ARMv7E-M architecture, respectively) is designed for microcontrollers, where power consumption, memory size, chip size and latency are important. The Cortex-A family, including e.g. Cortex-A8 and Cortex-A17 (both implementing ARMv7-A architecture) is intended for powerful processors (called “application processors”) for e.g. multimedia and communication products, particularly smartphones and tablets. These processors have much more processing power, typically feature high-bandwidth interfaces to the external world, and are designed to be used with high-level operating systems, most notably Linux (and Android).

An overview of ARM processors and their implemented architecture version can be found on Wikipedia. This tutorial will focus on the Cortex-M microcontrollers, as these are much easier to program without an operating system and because assembly language is less relevant on Cortex-A processors. However, the large range of ARM-based devices necessitates flexibility in the architecture specification and software tools, which sometimes complicates their use.

There is actually not a single, but three instruction sets for ARM processors:

- The “A32” instruction set for 32bit ARM architectures, also simply called “ARM” instruction set, favors speed over program memory consumption. All instructions are 4 bytes in size.
- The “A64” instruction set is for the new 64bit ARM processors
- The “T32” instruction set for 32bit ARM architectures, also known as “Thumb”, favors program memory consumption over speed. Most instructions are 2 bytes in size, and some are 4 bytes.

The 64bit Cortex-A application processors support all three instruction sets, while the 32bit ones only A32 and T32. The Cortex-M microcontrollers only support T32. Therefore, this tutorial will only talk about “thumb2”, the second version of the “T32” instruction set.

Prerequisites

First, suitable hardware and software need to be selected for demonstrating the usage of assembly language. For this tutorial, the choice of the specific microcontroller is of no great significance. However, to ensure that the example codes are easily transferable to your setup, it is recommended to use the same components.

Microcontroller selection

For the microcontroller, an STM32F103C8 or STM32F103RB by STMicroelectronics will be used. Both controllers are identical except for the flash size (64 KiB vs 128 KiB) and number of pins (48 vs 64). These controllers belong to ST's "mainstream" entry-level- family and are quite popular among hobbyist developers with many existing online resources. Several development boards with these controllers are available, for example: Nucleo-F103, "Blue Pill" (search for "stm32f103c8t6" on AliExpress, Ebay or Amazon), Olimexino-STM32, STM32-P103, STM32-H103, STM3210E-EVAL.

Processor type & documentation

First, the microcontroller manufacturer's documentation is used to find out what kind of ARM processor core and architecture is used for the chosen chip. This information is used to find all the relevant documentation.

- The first source of information is the STM32F103RB/C8 datasheet. According to the headline, this is a **medium-density** device. This term is ST-specific and denotes a product family with certain features. The very first paragraph states that this microcontroller uses a **Cortex-M3** processor core with 72 MHz. This document also contains the electrical characteristics and pinouts.
- The next important document is the STM32F103 reference manual that contains detailed descriptions of the periphery. Particularly, detailed information about periphery registers and bits can be found here.
- The ARM developer website provides information about the Cortex-M3 processor core, particularly the ARM Cortex-M3 Processor Technical Reference Manual (Manual can be downloaded as PFD. According to chapter 1.5.3, this processor implements the **ARMv7-M architecture**.
- The architecture is documented in the ARMv7M Architecture Reference Manual. Particularly, it contains the complete documentation of the instruction set.
- The ARM and Thumb-2 Instruction Set Quick Reference Card contains all ARM and Thumb-2 instructions, but is a bit cluttered because of its brevity.

For any serious STM32 development, you should be familiar with all these documents.

Debug adapter

There are many different ways of getting your program to run on an STM32 controller. A debug adapter is not only capable of writing software to the controller's flash, but can also analyze the program's behavior while it is running. This allows you to run the program one instruction at a time, analyze program flow and memory contents and find the cause of crashes. While it is not strictly necessary to use such a debugger, it can save a lot of time during development. Since entry-level models are available cheaply, not using one doesn't even save money. Debuggers connect to a host PC via USB (some via Ethernet) and to the microcontroller ("target") via JTAG or SWD. While these

two interfaces are closely related and perform the same function, SWD uses fewer pins (2 instead of 4, excluding reset and ground). Most STM32 controllers support JTAG, and all support SWD.

Documenting all possible way of flashing and debugging STM32 controllers is beyond the scope of this tutorial; a lot of information is already available online on that topic. Therefore, this tutorial will assume that the ST-Link debug adapter by STMicroelectronics is used, which is cheap and popular among hobbyists. Some of the aforementioned boards even include an ST-Link adapter, which can also be used “stand-alone” to flash an externally connected microcontroller. The examples should work with other adapters as well; please consult the appropriate documentation on how to use them.

Development Software

On the software part, several tools are needed for developing microcontroller firmware. Using a complete Integrated Development Environment (IDE) saves time and simplifies repetitive steps but hides some important steps that are necessary to gain a basic understanding of the process. Therefore, this tutorial will show the usage of the basic command line tools to demonstrate the underlying principles. Of course, for productive development, using an IDE is a sensible choice. The tools presented will work on Windows, Linux and Mac OS X (untested).

First, a text editor for writing assembly code is needed. Any good editor such as Notepad++, gedit or Kate is sufficient. When using Windows, the ST-Link Utility can be useful, but is not strictly required.

Next, an assembler toolchain is needed to translate the written assembly code into machine code. For this, the GNU Arm Embedded Toolchain is used. This is a collection of open source tools for writing software in Assembly, C and C++ for Cortex-M microcontrollers. Even though the package is maintained by ARM, the software is created by a community of open-source developers. For this tutorial, only the contained applications “binutils” (includes assembler & linker) and “GDB” (debugger) are really needed, but if you later decide to work with C or C++ code, the contained compilers will come in handy. Apart from that, this package is also shipped as part of several IDEs such as SW4STM32, Atollic TrueSTUDIO, emIDE, Embedded Studio and even Arduino – so if you (later) wish to work with one of these, your assembly code will be compatible with it.

Another component is required to talk with the debug adapter. For the ST-Link, this is done by OpenOCD, which communicates with the adapter via USB. Other adapters such as the J-Link ship with their own software.

Lastly, a calculator that supports binary and hexadecimal modes can be very helpful. Both the default Gnome calculator and the Windows calculator (calc.exe) are suitable.

Setup

Follow the instructions in the next chapters to set up your development environment.

Hardware

The only thing that needs to be done hardware-wise is connecting the debugger with your microcontroller. If you are using a development board with an integrated debugger (such as the Nucleo-F103), this is achieved by setting the jumpers accordingly (see the board's documentation – for e.g. the Nucleo-F103, both "CN2" jumpers need to be connected). When using an external debugger, connect the "GND", "JTMS/SWDIO" and "JTCK/SWCLK" pins of debugger and microcontroller. Connect the debugger's "nRESET" (or "nTRST" if it only has that) pin to the microcontroller's "NRST" input.

If your board has jumpers or solder bridges for the "BOOT0" pin, make sure that the pin is low. Applying power to the microcontroller board is typically done via USB.

Software

Linux

Some linux distributions ship with packages for the ARM toolchain. Unfortunately, these are often outdated and also configured slightly differently than the aforementioned package maintained by ARM. Therefore, to be consistent with the examples, it is strongly recommended to use the package by ARM.

Download the Linux binary tarball from the downloads page and extract it to some directory whose path does not contain any spaces. The extracted directory contains a subdirectory called "bin". Copy the full path to that directory (e.g. "/home/user/gcc-arm-none-eabi-8-2019-q3-update/bin").

Add this path to the "PATH" environment variable. On Ubuntu/Debian systems, this can be done via:

```
echo 'export PATH="${PATH}:/home/user/gcc-arm-none-eabi-8-2019-q3-update/bin"' | sudo tee /etc/profile.d/gnu-arm-embedded.sh
```

OpenOCD can be installed via the package manager, e.g. (Ubuntu/Debian):

```
sudo apt-get install openocd
```

After that, log out and back in (or just reboot). In a terminal, type `arm-none-eabi-as -version`. The output should look similar to this:

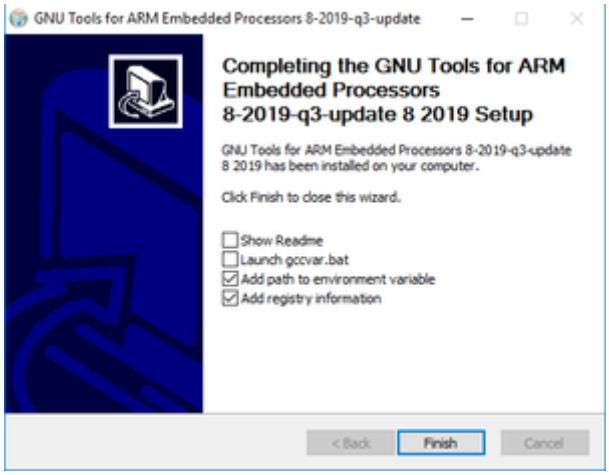
```
$ arm-none-eabi-as -version
GNU assembler (GNU Tools for Arm Embedded Processors 8-2019-q3-update) 2.32.0.20190703
Copyright (C) 2019 Free Software Foundation, Inc.
This program is free software; you may redistribute it under the terms of
the GNU General Public License version 3 or later.
This program has absolutely no warranty.
This assembler was configured for a target of `arm-none-eabi'.
```

Similarly, for `openocd -v`:

```
$ openocd -v
Open On-Chip Debugger 0.10.0
Licensed under GNU GPL v2
For bug reports, read
    http://openocd.org/doc/doxygen/bugs.html
```

If an error message appears, the installation isn't correct.

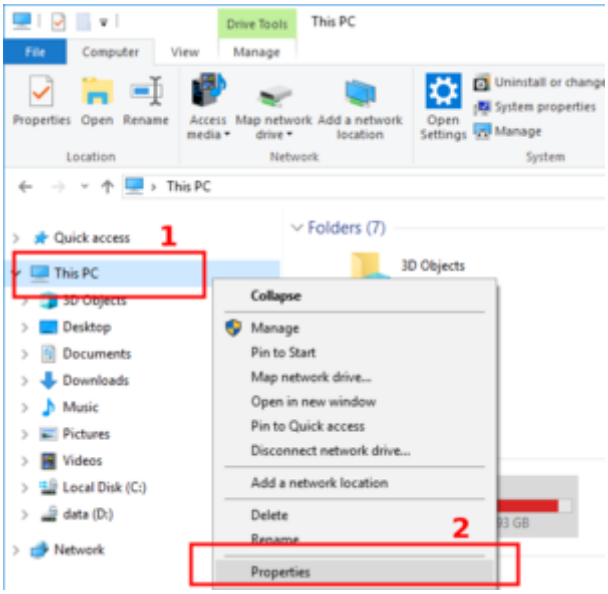
Windows



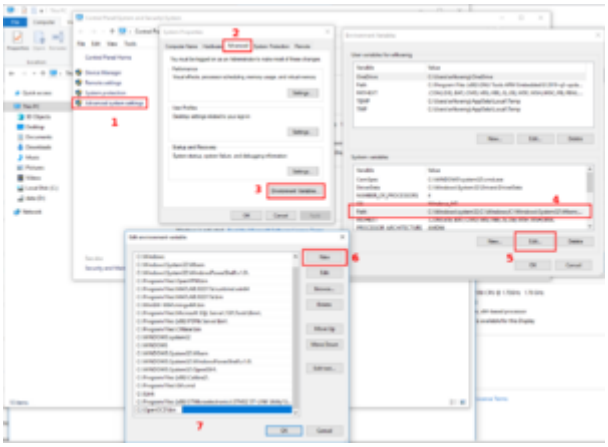
Options for installing GCC

Download the Windows installer from the downloads page and run it. Enable the options "Add path to environment variable" and "Add registry information", and disable "Show Readme" and "Launch gccvar.bat".

A Windows package for OpenOCD can be obtained from the gnu-mcu-eclipse downloads page. Download the appropriate file, e.g. " gnu-mcu-eclipse-openocd-0.10.0-12-20190422-2015-win64.zip". The archive contains a path like "GNU MCU Eclipse/OpenOCD/0.10.0-12-20190422-2015". Extract the contents of the inner directory (i.e. the subdirectories "bin", "doc", "scripts"…) into some directory whose path does not contain any spaces, e.g. "C:\OpenOCD". You should now have a directory "C:\OpenOCD\bin" or similar. Copy its full path.



Opening PC properties


Setting environment variable

Set the “Path” environment variable to include this path: Right-Click on “This PC”, then “Properties” → “Advanced System Settings”→ “Environment Variables”. In the lower list (labeled “System variables”), select “Path”. Click “Edit” → “New”, paste the path, and click “OK” multiple times.

Open a *new* command window (Windows Key + R, type “cmd” + Return). Type `arm-none-eabi-as -version`. The output should look similar to this:

```
C:\>arm-none-eabi-as -version
GNU assembler (GNU Tools for Arm Embedded Processors 8-2019-q3-update)
2.32.0.20190703
Copyright (C) 2019 Free Software Foundation, Inc.
This program is free software; you may redistribute it under the terms of
the GNU General Public License version 3 or later.
This program has absolutely no warranty.
This assembler was configured for a target of `arm-none-eabi'.
```

Similarly, for `openocd -v`:

```
C:\>openocd -v
GNU MCU Eclipse OpenOCD, 64-bitOpen On-Chip Debugger 0.10.0+dev-00593-
g23ad80df4 (2019-04-22-20:25)
Licensed under GNU GPL v2
For bug reports, read
        http://openocd.org/doc/doxygen/bugs.html
```

If an error message appears, the installation isn't correct.

Writing assembly applications

The full source code of the examples in the following chapters contain be found on GitHub. The name of the corresponding directory is given after each example code below.

First rudimentary program

After the software setup, you can begin setting up a first project. Create an empty directory for that, e.g. "prog1".

Inside the project directory, create your first assembly file "prog1.S" (".S" being the file name extension for assembly files in GNU context) with the following content:

```
.syntax unified
.cpu cortex-m3
.thumb

.word 0x20000400
.word 0x080000ed
.space 0xe4

nop     @ Do Nothing
b .     @ Endless loop
```

Example name: "EmptyProgram"

When this file is sent to the assembler, it will translate the instructions into binary machine code, with 2 or 4 bytes per instruction. These bytes are concatenated to form a program image, which is later written into the controller's flash memory. Therefore, assembly code more or less directly describes flash memory contents.

The lines starting with a dot "." are assembler directives that control the assembler's operation. Only some of those directives emit bytes that will end up in flash memory. The @ symbol starts a comment.

The first line lets the assembler use the new "unified" instruction syntax ("UAL" - Unified Assembler Language) instead of the old ARM syntax. The second line declares the used processor Cortex-M3, which the assembler needs to know in order to recognize the instructions available on that processor.

The third line instructs the assembler to use the Thumb (T32) instruction set. We can't start putting instructions in flash memory right away, as the processor expects a certain data structure to reside at the very beginning of the memory. This is what the ".word" and ".space" instructions create. These will be explained later.

The first "real" instruction is "nop", which will be the first instruction executed after the processor starts. "nop" is short for "No OPeration" - it causes the processor to do nothing and continue with the next instruction. This next instruction is "b .". "b" is short for "branch" and instructs the processor to jump to a certain "target" location, i.e. execute the instruction at that target next. In assembly language, the dot "." represents the current location in program memory. Therefore, "b ." instructs the processor to jump to this very instruction, i.e. execute it again and again in an endless loop. Such an endless loop is frequently found at the end of microcontroller programs, as it prevents the processor from executing random data that is located in flash memory after the program.

To translate this assembly code, open a terminal (linux) / command window (Windows). Enter the project directory by typing `cd <Path to Project Directory>`. Call the assembler like this:

```
arm-none-eabi-as -g prog1.S -o prog1.o
```

This instructs the assembler to translate the source file "prog1.S" into an object file "prog1.o". This is an intermediary file that contains binary machine code, but is not a complete program yet. The "-g"-Option tells the assembler to include debug information, which does not influence the program itself, but makes debugging easier. To turn this object file into a final program, call the linker like this:

```
arm-none-eabi-ld prog1.o -o prog1.elf -Ttext=0x8000000
```

This creates a file "prog1.elf" that contains the whole generated program. The "-Ttext" option instructs the linker to assume 0x8000000 as the start address of the flash memory. The linker might output a warning like this:

```
arm-none-eabi-ld: warning: cannot find entry symbol _start; defaulting to
0000000008000000
```

This is not relevant for executing the program without an operating system and can be ignored.

Flashing the program

To download the compiled application to the microcontroller that has been attached via ST-Link, use OpenOCD like so:

```
openocd -f interface/stlink-v2.cfg -f target/stm32f1x.cfg -c "program
prog1.elf verify reset exit"
```

Unfortunately, the application does not do anything that can be observed from the outside, except perhaps increase the current consumption.

Starting the debugger

To check whether the program is actually running, start a debugging session to closely observe the processor's behavior. First, run OpenOCD such that it acts as a GDB server:

```
openocd -f interface/stlink-v2.cfg -f target/stm32f1x.cfg
```

Then, open a new terminal/command window and start a GDB session:

```
arm-none-eabi-gdb prog1.elf
```

GDB provides its own interactive text-based user interface. First, type this command to let GDB connect to the already running OpenOCD instance:

```
target remote :3333
```

Then, stop the currently running program:

```
monitor reset halt
```

If this fails, hold your board's reset button just before executing the command and repeat until it succeeds. GDB can also download code to flash memory by simply typing:

```
load
```

Which will overwrite the previously flashed program (which, in this case, is identical anyways). After loading the program, reset the controller again:

```
monitor reset halt
```

Now, examine the contents of the CPU registers:

```
info reg
```

The output should look something like

```
r0             0x0                0
r1             0x0                0
r2             0x0                0
r3             0x0                0
r4             0x0                0
r5             0x0                0
r6             0x0                0
r7             0x0                0
r8             0x0                0
r9             0x0                0
r10            0x0                0
r11            0x0                0
r12            0x0                0
sp             0x0                0x0
```

```
lr             0x0             0
pc             0x8000000       0x8000000 <_stack+133693440>
xPSR           0x1000000       16777216
msp            0x20000400      0x20000400
psp            0x27e3fa34      0x27e3fa34
primask        0x0             0
basepri        0x0             0
faultmask      0x0             0
control        0x0             0
```

At this point, the processor is ready to start executing your program. The processor is halted just before the first instruction, which is “nop”. You can let the processor execute one single instruction (i.e. the “nop”) by typing

```
stepi
```

If you type `info reg` again, you will see that PC is now “0x80000ee”, i.e. the processor is about to execute the next instruction, “b .”. When you do

```
stepi
```

again (repeatedly), nothing more will happen – the controller is stuck in the mentioned endless loop, exactly as intended. You can instruct the processor to run the program continuously, without stopping after each instruction by typing

```
continue
```

You can interrupt the running program by pressing “Ctrl+C”. Run the commands

```
kill
quit
```

to exit GDB. You can terminate OpenOCD by pressing “Ctrl+C” in its terminal.

Using processor registers

The example program hasn’t done anything useful, but any “real” program will need to process some data. On ARM, any data processing is done via the processor registers. The 32bit ARM platforms have 16 processor registers, each of which is 32bit in size. The last three of those (r13-r15) have a special meaning and can only be used with certain restrictions. The first thirteen (r0-r12) can be used freely by the application code for data processing.

All calculations (e.g. addition, multiplication, logical and/or) need to be performed on those processor registers. To process data from memory, it first has to be loaded into a register, then processed, and stored back into memory. This is typical for RISC platforms and is known as a “load-store-architecture”.

As the starting point for any calculation, some specific values need to be put into the registers. The easiest way to do that is:

```
ldr r0, =123456789
```

The number 123456789 will be encoded as part of the program, and the instruction lets the processor copy it into the register “r0”. Any number and any register in the range r0-r13 can be used instead.

The instruction “mov” can be used to copy the contents from one register to another:

```
mov r1, r0
```

This copies r0 to r1. Unlike some other processor architectures, “mov” can not be used to access memory, but only the processor registers.

In ARM, 32bit numbers are called "words" and are most frequently used. 16bit numbers are known as half-words, and 8bit numbers as bytes, as usual.

Accessing periphery

To write microcontroller programs that interact with the outside world, access to the controller’s periphery modules is required. Interaction with periphery happens mainly through periphery registers (also known as “special function registers”, SFR). Despite their name, they work quite differently from processor registers. Instead of numbers, they have addresses (in the range of 0x40000000-0x50000000) that are not contiguous (i.e. there are gaps), they cannot be directly used for data processing but need to be explicitly read and written before and after any calculations. Not all of them are 32bit; many have only 16bit, and some of those bits may not exist and can’t be accessed. The microcontroller manufacturer’s documentation uses names for these registers, but the assembler doesn’t know these. Therefore, the assembly code needs to use the numerical addresses.

The easiest way to get the microcontroller to do something that produces some visible result is to send a signal via an output pin to turn on an LED. Using a pin to send/receive arbitrary software-defined signals is called “GPIO” (General Purpose Input/Output). First, choose a pin – for example, PA8 (this one is available on all package variants). Connect an LED to this pin and to GND (“active high”). Use a series resistor to limit the current to max. 15mA (the absolute maximum being 25mA), e.g. 100Ω for a 3,3V supply and a standard LED. For higher loads (e.g. high-power LEDs or a relay) use an appropriate transistor.

As with most microcontrollers, the pins are grouped into so-called “ports”, each of which has up to 16 pins. The ports are named by letters of the alphabet, i.e. “GPIOA”, “GPIOB”, “GPIOC” etc. The number of ports and pins varies among the individual microcontroller types. The 16 pins of one port can be read or written in one single step.

Clock Configuration

Many ARM controllers feature a certain trap: Most periphery modules are disabled by default to save power. The software has to explicitly enable the needed modules. On STM32 controllers, this is done via the “RCC” (Reset and Clock Control) module. Particularly, this module allows the software to disable/enable the clock signal for each periphery module. Because MOSFET-based circuits (virtually all modern ICs) only draw power if a clock signal is applied, turning off the clock of unused modules can reduce the power usage considerably.

This is documented in the aforementioned reference manual in chapter 7. The subchapter 7.3.7 describes the periphery register “RCC_APB2ENR” which allows you to configure the clock signal for some peripheral modules. This register has 32 bits, of which 14 are “reserved”, i.e. can’t be used and should only be written with zeroes. Each of the available 18 bits enables one specific periphery module if set to “1” or disables it if set to “0”. According to the manual, the reset value of this register is 0, so all periphery modules are disabled by default. In order to turn on the GPIOA module to which the desired pin PA8 belongs, the bit “IOPAEN” needs to be set to “1”. This is bit number two in the register. Since registers can only be accessed to as a whole (individual bits can’t be addressed), a 32bit-value where bit two is “1” and all others are kept as “0” needs to be written. This value is 0x00000004.

To write to the register, its address needs to be given in the code. The addresses of the periphery registers are grouped by the periphery modules they belong to - each periphery module (e.g. RCC, GPIOA, GPIOB, USB, …) has its own base address. The addresses of the individual registers are specified as an offset that needs to be added to this base address to obtain the full absolute address of the register. Chapter 7.3.7 specifies the offset address of RCC_APB2ENR as “0x18”. Chapter 3.3 specifies the base addresses of all periphery modules – RCC is given as “0x40021000”. So, the absolute address of RCC_APB2ENR is “0x40021000+ 0x18=0x40021018”.

In short: To enable GPIOA, the value **0x00000004** needs to be written to address **0x40021018**.

According to the “load-store” principle, ARM processors can’t do this in a single step. Both the value to be written and the address need to reside in processor registers in order to perform the write access. So, what needs to done is:

- Load the value 0x00000004 into a register
- Load the value 0x40021018 into another register
- Store the value from the first register into the memory location specified by the second register.

This last step is performed by the “STR” instruction as follows:

```
.syntax unified
.cpu cortex-m3
.thumb

.word 0x20000400
.word 0x080000ed
.space 0xe4
```

```
ldr r0, =0x00000004
ldr r1, =0x40021018
str r0, [r1]            @ Set IOPAEN bit in RCC_APB2ENR to 1 to enable
GPIOA
b .
```

The square brackets are required but just serve as a reminder to the programmer that the contents of "r1" is used as an address. After the "str" instruction, the GPIOA periphery is enabled, but doesn't do anything yet.

GPIO Configuration

By default, all GPIO pins are configured as "input", even if there is no software to process the input data. Since inputs are "high-impedance", i.e. only a very small current can flow into/out of the pin, the risk of (accidental) short-circuits and damage to the microcontroller is minimized. However, this current is too small to light up an LED, so you have to configure the pin PA8 as "output". The STM32 support multiple output modes, of which the right one for the LED is "General Purpose Output Push-Pull, 2 MHz".

Access and configuration of GPIO pins is achieved via the registers of the GPIO periphery. The STM32 have multiple identical instances of GPIO modules, which are named GPIOA, GPIOB, … Each of those instances has a distinct base address, which are again described in chapter 3.3 of the reference manual (e.g. "0x40010800" for GPIOA, "0x40010C00" for GPIOB etc.). The registers of the GPIO module are described in chapter 9.2, and there is one instance of each register per GPIO module. To access a specific register of a specific GPIO module, the base address of that module needs to be added to the offset address of the register. For example, "GPIOA_IDR" has address "0x40010800+0x08=0x40010808", while "GPIOB_ODR" has address "0x40010C00+0x0C= 0x40010C0C".

Configuration of the individual GPIO pins happens through the "GPIOx_CRL" and "GPIOx_CRH" registers ("x" is a placeholder for the concrete GPIO module) – see chapters 9.2.1 and 9.2.2. Both registers are structured identically, where each pin uses 4 bits, so each of the two registers handles 8 pins in 8x4=32 bits. Pins 0-7 are configured by "GPIOx_CRL" and pins 8-15 by "GPIOx_CRH". Pin 0 is configured by bits 0-3 of "GPIOx_CRL", pin 1 by bits 4-7 of "GPIOx_CRL", pin 8 by bits 0-3 of "GPIOx_CRH" and so on.

The 4 bits per pin are split into two 2-bit fields: "MODE" occupies bits 0-1, and "CNF" bits 2-3. "MODE" selects from input and output modes (with different speeds). In output mode, "CNF" determines whether the output value is configured from software ("General Purpose" mode) or driven by some other periphery module ("Alternate function" mode), and whether two transistors ("Push-pull") or one ("open-drain") are used to drive the output. In input mode, "CNF" selects from analog mode (for ADC), floating input and input with pull-up/down resistors (depending on the value in the "GPIOx_ODR" register).

Therefore, to configure pin PA8 into “General Purpose Output Push-Pull, 2 MHz” mode, bits 0-3 of “GPIOA_CRH” need to be set to value “2”. The default value of “4” configures the pin as “input”. To keep the other pins at their “input” configuration, the value “0x44444442” needs to be written to register “GPIOA_CRH”, which has address “0x40010804”:

```
ldr r0, =0x44444442
ldr r1, =0x40010804
str r0, [r1]            @ Set CNF8:MODE8 in GPIOA_CRH to 2
```

Writing GPIO pins

The GPIO pin still outputs the default value, which is 0 for “low”. To turn on the LED, the output has to be set to “1” for “high”. This is achieved via the GPIOA_ODR register, which has 16bits, one for each pin (see chapter 9.2.4). To enable the LED, set bit 8 to one:

```
.syntax unified
.cpu cortex-m3
.thumb

.word 0x20000400
.word 0x080000ed
.space 0xe4

ldr r0, =0x00000004
ldr r1, =0x40021018
str r0, [r1]            @ Set IOPAEN bit in RCC_APB2ENR to 1 to enable
GPIOA

ldr r0, =0x44444442
ldr r1, =0x40010804
str r0, [r1]            @ Set CNF8:MODE8 in GPIOA_CRH to 2

ldr r0, =0x100
ldr r1, =0x4001080C
str r0, [r1]            @ Set ODR8 in GPIOA_ODR to 1 to set PA8 high

b .
```

Example name: “SetPin”

This program enables the GPIOA periphery clock, configures PA8 as output, and sets it to high. If you run it on your microcontroller, you should see the LED turn on – the first program to have a visible effect!

Data processing

ARM supports many instructions for mathematical operations. For example, addition can be performed as:

```
ldr r0, =222
ldr r1, =111
add r2, r0, r1
```

This will first load the value 222 into register r0, load 111 into r1, and finally add r0 and r1 and store the result (i.e. 333) in r2. The operand for the result is (almost) always put on the left, while the input operand(s) follow on the right.

You can also overwrite an input register with the result:

```
add r0, r0, r1
```

This will write the result to r0, overwriting the previous value. This is commonly shortened to

```
add r0, r1
```

The output operand can be omitted, and the first input (here: r0) will be overwritten. This applies to most data processing instructions. Other frequently used data processing instructions that are used in a similar fashion are:

- **sub** for subtraction
- **mul** for multiplication
- **and** for bitwise and
- **orr** for bitwise or
- **eor** for bitwise exclusive or (“xor”)
- **lsl** for logical left shift
- **lsr** for logical right shift

Most of these instructions can not only take registers as input, but also immediate arguments. Such an argument is encoded directly into the instruction without needing to put it into a register first. Immediate arguments need to be prefixed by a hash sign #, and can be decimal, hexadecimal or binary. For example,

```
add r0, r0, #23
```

adds 23 to the register r0 and stores the result in r0. This can again be shortened to

```
add r0, #23
```

Such immediate arguments can not be arbitrarily large, because they need to fit inside the instruction, which is 16 or 32 bit in size and also needs some room for the instruction and register numbers as well. So, if you want to add a large number, you have to use “ldr” first as shown to load it into a register.

Try out the above examples and use GDB to examine their behavior. Use GDB’s “info reg” command to display the register contents. Don't forget to execute both the “arm-none-eabi-as” and “arm-none-eabi-ld” commands to translate the program.

Reading periphery registers

The last example works, but has a flaw: Even though only a few bits per register need to be modified, the code overwrites all the bits in the register at once. The bits that should not be modified are just overwritten with their respective default value. If some of those bits had been changed before – for example to enable some other periphery module – these changes would be lost. Keeping track of the state of the register throughout the program is hardly practical. Since ARM does not permit modifying individual bits, the solution is to read the whole register, modify the bits as needed, and write the result back. This is called a “read-modify-write” cycle.

Reading registers is done via the “ldr” instruction. As with “str”, the address needs to be written into a processor register beforehand, and the instruction stores the read data into a processor register as well. Starting the with the “RCC_APB2ENR” register, you can read it via:

```
ldr r1, =0x40021018
ldr r0, [r1]
```

Even though the two “ldr” instruction look similar, they work differently – the first one loads a fixed value into a register (r1), while the second loads data from the periphery register into r1.

The loaded value should then be modified by setting bit two to “1”. This can be done with the “orr” instruction:

```
orr r0, r0, #4
```

After that, we can store r0 as before.

With the GPIOA_CRH register, it’s slightly more complicated: The bits 0, 2 and 3 need to be cleared, while bit 1 needs to be set to 1. The other bits (4-31) need to keep their value. To clear the bits, use the “and” instruction after loading the current periphery register value:

```
ldr r1, =0x40010804
ldr r0, [r1]
and r0, #0xfffffff0
orr r0, #2
str r0, [r1]            @ Set CNF8:MODE8 in GPIOA_CRH to 2
```

For the “GPIOx_ODR” registers, such tricks are not needed, as there is a special “GPIOx_BSRR” register which simplifies writing individual bits: This register can not be read, and writing zeroes to any bit has no effect on the GPIO state. However, if a 1 is written to any of the bits 0-15, the corresponding GPIO pin is set to high (i.e. the corresponding bit in ODR set to 1). If any of the bits 16-31 is written to 1, the corresponding pin is set to low. So, the pin can be set to 1 like this:

```
ldr r1, =0x40010810
ldr r0, =0x100
str r0, [r1]            @ Set BS8 in GPIOA_BSRR to 1 to set PA8 high
```

So, the modified program is:

```
.syntax unified
.cpu cortex-m3
.thumb

.word 0x20000400
.word 0x080000ed
.space 0xe4

ldr r1, =0x40021018
ldr r0, [r1]
orr r0, r0, #4
str r0, [r1]            @ Set IOPAEN bit in RCC_APB2ENR to 1 to enable
GPIOA

ldr r1, =0x40010804
ldr r0, [r1]
and r0, #0xfffffff0
orr r0, #2
str r0, [r1]            @ Set CNF8:MODE8 in GPIOA_CRH to 2

ldr r1, =0x40010810
ldr r0, =0x100
str r0, [r1]            @ Set BS8 in GPIOA_BSRR to 1 to set PA8 high

b .
```

Example name: “SetPin2”

Jump instructions

For a traditional “hello world” experience, the LED should not only light up, but blink, i.e. turn on and off repeatedly. Setting pin PA8 to low level can be achieved by writing a 1 to bit 24 in the “GPIO_BSRR” register:

```
ldr r1, =0x40010810
ldr r0, =0x1000000
str r0, [r1]
```

By pasting the this behind the instructions for turning on the LED, it will be turned on and off again. To get the LED to blink, those two blocks need to be repeated endlessly, i.e. at the end of the code there needs to be an instruction for jumping back to the beginning.

A simple endless loop was already explained: The “b .” instruction, which just executes itself repeatedly. To have it jump somewhere else, the dot needs to be substituted for the desired target address, for example:

```
.syntax unified
.cpu cortex-m3
.thumb

.word 0x20000400
.word 0x080000ed
.space 0xe4

ldr r1, =0x40021018
ldr r0, [r1]
orr r0, r0, #4
str r0, [r1]            @ Set IOPAEN bit in RCC_APB2ENR to 1 to enable
GPIOA

ldr r1, =0x40010804
ldr r0, [r1]
and r0, #0xfffffff0
orr r0, #2
str r0, [r1]            @ Set CNF8:MODE8 in GPIOA_CRH to 2

ldr r1, =0x40010810
ldr r0, =0x100
str r0, [r1]            @ Set BS8 in GPIOA_BSRR to 1 to set PA8 high

ldr r1, =0x40010810
ldr r0, =0x1000000
str r0, [r1]            @ Set BR8 in GPIOA_BSRR to 1 to set PA8 low

b 0x8000104
```

Example name: “Blink”

The address specified is an absolute address, which is the address of the “ldr” instruction at the beginning of the block for setting the pin to high. Actually, the branch instruction “b” is not capable of jumping directly to such an absolute address - again, because a 32 bit wide address can't be encoded in a 16/32 bit wide instruction. Instead, the assembler calculates the distance of the jump target and the location of the “b” instruction, and stores it into the instruction. When jumping backwards, this distance is negative.

When executing program code, the processor always stores the address of the currently executed instruction plus four in the r15 register, which is therefore also known as PC, the program counter. When encountering a “b” instruction, the processor adds the contained distance value to the PC value to calculate the absolute address of the jump target before jumping there.

This means that “b” performs a relative jump, and even if the whole machine code section were moved somewhere else in memory, the code would still work. However, the assembly language syntax does not really represent this, as the assembler expects absolute addresses which it then transforms into relative ones.

Specifying the target address directly as shown is very impractical, as it has to be calculated manually, and if the section of code is moved or modified, the address needs to be changed. To rectify this, the assembler supports labels: You can assign a name to a certain code location, and use this name to refer to the code location instead of specifying the address as a number. A label is defined by writing its name followed by a colon:

```
BlinkLoop:

ldr r1, =0x40010810
ldr r0, =0x100
str r0, [r1]            @ Set BS8 in GPIOA_BSRR to 1 to set PA8 high

ldr r1, =0x40010810
ldr r0, =0x1000000
str r0, [r1]            @ Set BR8 in GPIOA_BSRR to 1 to set PA8 low

b BlinkLoop
```

Example name: “Blink2”

This is purely a feature of the assembler – the generated machine code will be identical to the previous example. In “b BlinkLoop”, the assembler substitutes the label for the address it represents to calculate the relative jump distance. The assembler actually provides no direct way of directly specifying the relative offset that will be encoded in the instruction, but it can be done like this:

```
b (.+4+42*2)
```

The resulting instruction will contain “42” as the jump offset. As suggested by the syntax, the processor multiples this number by 2 (since instructions can only reside at even memory addresses, it would waste one bit of memory to specify the number directly) and adds to it the address of the “b” instruction plus 4. The assembly syntax is designed to represent the end result of the operation, so the assembler reverses the peculiar pre-calculations of the processor. If you want to do this calculation yourself, you have to again undo the assembler’s own calculation with the expression shown above. There is usually no reason to do that, though.

Counting Loops

The above example for a blinking LED does not really work yet – the LED blinks so fast the human eye can’t see it. The LED will just appear slightly dim. To achieve a proper blinking frequency, the code needs to be slowed down. The easiest way for that is to have the processor execute a large number of “dummy” instructions between setting the pin high and low. Simply placing many “nop” instructions isn’t possible though, as there is simply not enough program memory to store all of them. The solution is a loop that executes the same instructions a specific number of times (as opposed to the endless loops from the examples above). To do that, the processor has to count the number of loop iterations. It is actually easier to count *down* than up, so start by loading the desired number of iterations into a register and begin the loop by subtracting “1”:

```
ldr r2, =1000000
subs r2, #1
```

Now, the processor should make a decision: If the register has reached zero, terminate the loop; else, continue by again subtracting “1”. The ARM math instructions can automatically perform some tests on the result to check whether it is positive/negative or zero and whether an overflow occurred. To enable those checks, append an “s” to the instruction name – hence, “subs” instead of “sub”. The result of these checks is automatically stored in the “Application Program Status Register” (APSR) – the contained bits N, Z, C, V indicate whether the result was negative, zero, set the carry bit or caused an overflow. This register is usually not accessed directly. Instead, use the conditional variant of the “b” instruction, where two letters are appended to indicate the desired condition. The jump is only performed if the condition is met; otherwise, the instruction does nothing. The available condition codes are described in the chapter “Condition Codes” of this tutorial. The conditions are formulated in terms of the mentioned bits of the APSR. For example, the “bne” instruction only performs a jump if the zero (Z) flag is *not* set, i.e. when the result of the last math instruction (with an “s” appended) was *not* zero. The “beq” instruction is the opposite of that – it only performs a jump if the result *was* zero.

So, to perform the jump back to the beginning of the loop, add a label before the “subs” instruction, and put a “bne” instruction after the “subs” that jumps to this label if the counter has not reached zero yet:

```
ldr r2, =1000000
delay1:
subs r2, #1
bne delay1          @ Iterate delay loop
```

The actual loop consists only of the two instructions “subs” and “bne”. By placing two of those loops (with two different labels!) in between the blocks that turn the pins on and off, the blink frequency is lowered sufficiently such that it becomes visible:

```
.syntax unified
.cpu cortex-m3
.thumb

.word 0x20000400
.word 0x080000ed
.space 0xe4

ldr r1, =0x40021018
ldr r0, [r1]
orr r0, r0, #4
str r0, [r1]            @ Set IOPAEN bit in RCC_APB2ENR to 1 to enable
GPIOA

ldr r1, =0x40010804
ldr r0, [r1]
```

```
and r0, #0xfffffff0
orr r0, #2
str r0, [r1]            @ Set CNF8:MODE8 in GPIOA_CRH to 2

BlinkLoop:
ldr r1, =0x40010810
ldr r0, =0x100
str r0, [r1]            @ Set BS8 in GPIOA_BSRR to 1 to set PA8 high

ldr r2, =1000000
delay1:
subs r2, #1
bne delay1          @ Iterate delay loop

ldr r1, =0x40010810
ldr r0, =0x1000000
str r0, [r1]            @ Set BR8 in GPIOA_BSRR to 1 to set PA8 low

ldr r2, =1000000
delay2:
subs r2, #1
bne delay2          @ Iterate delay loop

b BlinkLoop
```

Example name: “BlinkDelay”

You might notice that the registers r0-r2 are loaded with the same values over and over again. To make the code both shorter and faster, take advantage of the available processor registers, and load the values that don’t change *before* the loop. Then, just use them inside the loop:

```
.syntax unified
.cpu cortex-m3
.thumb

.word 0x20000400
.word 0x080000ed
.space 0xe4

ldr r1, =0x40021018
ldr r0, [r1]
orr r0, r0, #4
str r0, [r1]            @ Set IOPAEN bit in RCC_APB2ENR to 1 to enable
GPIOA

ldr r1, =0x40010804
ldr r0, [r1]
and r0, #0xfffffff0
orr r0, #2
str r0, [r1]            @ Set CNF8:MODE8 in GPIOA_CRH to 2
```

```
ldr r0, =0x40010810     @ Load address of GPIOA_BSRR
ldr r1, =0x100          @ Register value to set pin to high
ldr r2, =0x1000000      @ Register value to set pin to low
ldr r3, =1000000        @ Iterations for delay loop

BlinkLoop:
str r1, [r0]            @ Set BS8 in GPIOA_BSRR to 1 to set PA8 high

mov r4, r3
delay1:
subs r4, #1
bne delay1          @ Iterate delay loop

str r2, [r0]            @ Set BR8 in GPIOA_BSRR to 1 to set PA8 low

mov r4, r3
delay2:
subs r4, #1
bne delay2          @ Iterate delay loop

b BlinkLoop
```

Example name: “BlinkDelay2”

Using RAM

Until now, all data in the example codes was stored in periphery or processor registers. In all but the most simple programs, larger amounts of data have to be processed for which the thirteen general-purpose processor registers aren’t enough. For this, the microcontroller features a block of SRAM that stores 20 KiB of data. Accessing data in RAM works similar to accessing periphery registers – load the address in a processor register and use “ldr” and “str” to read and write the data. After reset, the RAM contains just random ones and zeroes, so before the first read access, some value has to be stored.

As the programmer decides what data to place where, they have to keep track which address in memory contains what piece of data. You can use the assembler to help keeping track by declaring what kind of memory blocks you need and giving them names. To do this, you must first tell the assembler that the next directives refer to data instead of instructions with the “.data” directive. Then, use the “.space” directive for each block of memory you need. To assign names to the blocks, place a label definition (using a colon) right *before* that. After the definitions, put a “.text” directive to make sure the instructions after that will properly go to program memory (flash):

```
.data
var1:
    .space 4        @ Reserve 4 bytes for memory block “var1”
```

```
var2:
    .space 1        @ Reserve 1 byte for memory block “var2”

.text
@ Instructions go here...
```

Here, a data block of 4 bytes is reserved and named “var1”. Another block of 1 byte is named “var2”. Note that just inserting these lines will not modify the assembler output – these are just instructions to the assembler itself. To access these memory blocks, you can use “var1” and “var2” just like literal addresses. Load them into registers and use these with “ldr” and “str” like this:

```
.syntax unified
.cpu cortex-m3
.thumb

.word 0x20000400
.word 0x080000ed
.space 0xe4

.data
var1:
    .space 4        @ Reserve 4 bytes for memory block “var1”
var2:
    .space 1        @ Reserve 1 byte for memory block “var2”

.text


ldr r0, =var1           @ Get address of var1
ldr r1, =0x12345678
str r1, [r0]            @ Store 0x12345678 into memory block “var1”


ldr r1, [r0]            @ Read memory block “var1”
and r1, #0xFF       @ Set bits 8..31 to zero
ldr r0, =var2           @ Get address of var2
strb r1, [r0]           @ Store a single byte into var2

b .
```

Example name: “RAMVariables”

Note the use of “strb” - it works similar to “str”, but only stores a single byte. Since the processor register r1 is of course 32bit in size, only the lower 8 bits are stored, and the rest is ignored.

There is still something missing – nowhere in the code is there any address of the RAM. To tell the linker where the RAM is located, pass the option `-Tdata=0x20000000` to the `arm-none-eabi-ld` call to tell the linker that this is the address of the first byte of RAM. This program can't be flashed directly with OpenOCD, as OpenOCD doesn't recognize the RAM as such; GDB has to be used as

explained above. When a linker script is used as described in the next chapters (using the NOLOAD attribute), OpenOCD can again be used directly.

If you run this program via GDB, you can use the commands `x/1xw &var1` and `x/1xb &var2` to read the data stored in memory. After this quick introduction a more abstract overview is indicated.

Memory Management

If there is one thing that sets higher and lower level programming languages apart, it's probably memory management. Assembly programmers have to think about memory, addresses, layout of program and data structures all the time. Assembler and linker provide some help which needs to be used effectively. Therefore, this chapter will explain some more fundamentals of the ARM architecture and how the toolchain works.

Address space

In the examples so far, addresses were used for periphery register accesses and jump instructions without really explaining what they mean, so it's time to catch up with that. To access periphery registers and memory locations in any memory type (RAM, Flash, EEPROM…), an address is required, which identifies the desired location. On most platforms, addresses are simply unsigned integers. The set of all possible addresses that can be accessed in a uniform way is called an "address space". Some platforms such as AVR have multiple address spaces (for Flash, EEPROM, and RAM+periphery) where each memory needs to be accessed in a distinct way and the programmer needs to know which address space an address belongs to – e.g. all three memory types have a memory location with address 123.

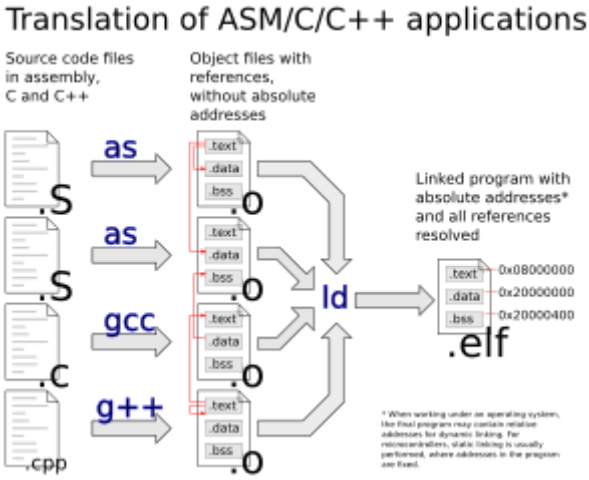
However, the ARM architecture uses only a single large address space where addresses are 32bit unsigned integers in the range of 0-4294967295. Each address refers to one byte of 8 bits. The address space is divided into several smaller ranges, each of which refers to a specific type of memory. For the STM32F103, this is documented in the datasheet in chapter 4. All addresses in all memory types are accessed in the same way – directly via the "ldr" and "str" instructions, or by executing code from a certain location, which can be achieved by jumping to the respective address with the "b" instruction. This also makes it possible to execute from RAM – simply perform a jump to an address that refers to some code located in RAM. Note that there are large gaps between the individual ranges in address space; attempting to access those usually leads to a crash.

While the addresses of periphery are fixed and defined by the manufacturer, the layout of program code and data in memory can be set by the programmer rather freely. Up until now, the example programs defined the flash memory contents in a linear fashion by listing the instructions on the order they should appear in flash memory. However, when translating multiple assembly source files into one program, the order in which the contents from those files appears in the final program isn't defined a priori. Also, even though in the last example the memory blocks for RAM were defined

before the code, the code actually comes first in address space. What makes all this work is the Linker.

The Linker

Usually the last step in translating source code into a usable program, the linker is an often overlooked, sometimes misunderstood but important and useful tool, if applied correctly. Many introductions into programming forego explaining its workings in detail, but as any trade, embedded development requires mastery of the tools! A good understanding of the linker can save time solving strange errors and allow you to implement some less common use cases, such as using multiple RAM blocks present in some microcontrollers, executing code from RAM or defining complex memory layouts as sometimes required by RTOSes.



Translation of native applications using assembler, compiler and linker

You have already used a linker – the command `arm-none-eabi-ld` calls the GNU linker that is shipped with the GNU toolchain. Until now, only one assembly source files was translated for each program. To translate a larger program that consists of three assembly files “file1.S”, “file2.S” and “file3.s”, the assembler would be called three times to produce three object code files “file1.o”, “file2.o” and “file3.o”. The linker would then be called to combine all three into a single output file.

When translating any of these assembly files, the assembler does not know of the existence of the other files. Therefore, it can’t know whether the contents of any other file will end up in flash memory before the currently processed file, and also can’t know the final location in flash memory of the machine code it is emitting and placing in the object file (ending .o). This means that the object file does not contain any absolute addresses (except for those of periphery registers, as these were specified explicitly). For example, when loading the address of the RAM data blocks (“ldr r0, =var1”) the assembler doesn’t know the address, only the linker does. Therefore, the assembler puts a placeholder in the object file that will be overwritten by the linker. A jump (“b” instruction) to a label defined in another assembly file works similarly; the assembler uses a placeholder for the address. For the jump instructions we used inside the same file (e.g. “b BlinkLoop”), a placeholder is not

necessary, as the assembler can calculate the distance of the label and the instruction and generate the relative jump itself. However, if the target resides within a different section (see below), this isn't possible, and a placeholder becomes necessary. As the contents of object files has no fixed address and can be moved around by the linker, these files are called relocatable.

On Unix Systems (including Linux), the Executable and Linkable Format (ELF) is used for both object files and executable program files. This format is also used by ARM, and the GNU ARM toolchain. Because it was originally intended to be used with operating systems, some of its concepts don't perfectly map the embedded use case. The object (.o) files created by the assembler and linker, and also the final program (usually no ending, but in embedded contexts and also in above example commands, .elf is used) are all in ELF format. The specification of ELF for ARM can be found here, and the generic specification for ELF on which the ARM ELF variant is based can be found here.

ELF files are structured into sections. Each section may contain code, data, debug information (used by GDB) and other things. In an object file, the sections have no fixed address. In the final program file, they have one. Sections also have various attributes that indicate whether its contents is executable code or data, is read-only and whether memory should be allocated for it. The linker combines and reorders the sections from the object files ("input sections") and places them into sections in the final program file ("output sections") while assigning them absolute addresses.

Another important aspect are symbols. A symbol defines a name for an address. The address of a symbol may be defined as an absolute number (e.g. 0x08000130) or as an offset relative to the beginning of a section (e.g. "start address of section .text plus 0x130"). Labels defined in assembly source code define symbols in the resulting object file. For example, the "var1" label defined in the last example results in a symbol "var1" in the "prog1.o" file whose address is set to be equal to the beginning of ".data". The symbol "var" is defined similarly, but with an offset of 4. After the linking process, the "prog1.elf" file contains a ".data" section with absolute address 0x20000000, and so the "var1" and "var2" symbols get absolute addresses as well.

As mentioned, the assembler puts placeholders in the object files when it doesn't know the address of something. In ELF files, there placeholders are called "relocation entries" and they reference symbols by name. When the linker sees such a relocation entry in one of its input files, it searches for a symbol in the input files with a matching name and fills in its address. If no symbol with that name was found, it emits this dreaded error:

```
(.text+0x132): undefined reference to `Foo'
```

Google finds almost a million results for that message, but knowing how the linker operates makes it easy to understand and solve – since the symbol was not found in any object file, make sure it is spelled correctly and that the object file that contains it is actually fed to the linker.

Linker Scripts

A linker script is a text file written in a linker-specific language that controls how the linker maps input sections to output sections. The example project hasn't explicitly specified one yet, which lets the linker use a built-in default one. This has worked so far, but results in a slightly mixed up program file (unsuitable symbols) and has some other disadvantages. Therefore, it's time to do things properly and write a linker script. Linker scripts aren't usually created on a per-project basis, but usually provided by the microcontroller manufacturer to fit a certain controller's memory layout. To learn how they work, a quick introduction into writing one will follow. The full documentation can be found here.

It's customary to name the linker script after the controller they are intended for, so create a text file "stm32f103rb.ld" or "stm32f103c8.ld" with the following contents:

```
MEMORY {
    FLASH   : ORIGIN = 0x8000000,   LENGTH = 128K
    SRAM    : ORIGIN = 0x20000000,  LENGTH = 20K
}

SECTIONS {
    .text : {
        *(.text)
    } >FLASH

    .data (NOLOAD) : {
        *(.data)
    } >SRAM
}
```

Example name: "LinkerScriptSimple"

This is this minimum viable linker script for a microcontroller. If you are using a STM32F103C8, replace the 128K by 64K. The lines inside the "MEMORY" block define the available memory regions on your microcontroller by specifying their start address and size within the address space. The names "FLASH" and "SRAM" can be chosen arbitrarily, as they have no special meaning. This memory definition has no meaning outside of the linker script, as it is just an internal helper for writing the script; it can even be left out and replaced by some manual address calculations.

The interesting part happens inside the "SECTIONS" command. Each sub-entry defines an output section that will end up in the final program file. These can be named arbitrarily, but the names ".text" and ".data" for executable code and data storage respectively are usually used. The asterisk expressions "*(.text)" and "(*.data)" tell the linker to put the contents of the input sections ".text" and ".data" at that place in the output section. In this case, the names for the input sections and output sections are identical. The input section names ".data", ".text" (and some more) are used by the assembler and C and C++ compilers by default, so even though they can be changed, it's best to keep them. You can however name the output sections arbitrarily, for example:

```
SECTIONS {
    .FlashText : {
        *(.text)
```

```
    } >FLASH

    .RamData (NOLOAD) : {
        *(.data)
    } >SRAM
}
```

The commands “>FLASH” and “>SRAM” tell the linker to calculate the address of the output sections according to the respective memory declaration above: The first output section with a “>FLASH” command will end up at address 0x8000000, the next with “>FLASH” right after that section and so on. The “>SRAM” works the same way with the start address “0x20000000”. The “NOLOAD” attribute does not change the linker’s behavior, but marks the corresponding output section as “not-loadable”, such that OpenOCD and GDB will not attempt to write it into RAM – the program has to take care of initializing any RAM data anyways when running stand-alone.

To specify the filename of the linker script, use the “-T” option:

```
arm-none-eabi-ld prog1.o -o prog1.elf -T stm32f103rb.ld
```

The `-Tdata` and `-Ttext` aren’t needed anymore, as the addresses are now defined in the linker script.

Since the linker script defines the sizes of the memory regions, the linker can now warn you when your program consumes too much memory (either flash or RAM):

```
arm-none-eabi-ld: prog1.elf section `.text' will not fit in region `FLASH'
arm-none-eabi-ld: region `FLASH' overflowed by 69244 bytes
```

Reserving memory blocks

Using the processor’s stack will be explained later, but you can already use the linker script to assign a memory block for it. It’s best to allocate memory for the stack at the *beginning* of SRAM, so put this before the “*(.data)” command:

```
. = . + 0x400;
```

Inside a linker script, the dot “.” refers to the current address in the output file; therefore, this command increments the address by 0x400, leaving an “empty” block of that size. The “.data” input section will be located after that, at address 0x20000400.

Defining symbols in linker scripts

As mentioned before, the controller requires a certain data structure called the “vector table” to reside at the very beginning of flash memory. It is defined in the assembler source file:

```
.word 0x20000400
.word 0x080000ed
```

```
.space 0xe4
```

The ".word" directive tells the assembler to output the given 32bit-number. Just like processor instructions, these numbers are put into the current section (.text by default, .data if specified) and therefore end up in flash memory. The first 32bit-number, which occupies the first 4 bytes in flash memory, is the initial value of the stack pointer which will be explained later. This number should be equal to the address of the first byte *after* the memory block that was reserved for the stack. The reserved block starts at address 0x20000000 and has size 0x400, so the correct number is 0x20000400. However, if the size of the reserved block was modified in the linker script, the above assembly line needs to be adjusted as well. To avoid any inconsistencies, and to be able to manage everything related to the memory-layout centrally in the linker script, it is desirable to replace the number in the assembly source file with a symbol expression. To do this, define a symbol in the linker script:

```
.data (NOLOAD) : {
    . = . + 0x400;
    _StackEnd = .;
    *(.data)
} >SRAM
```

Example name: "LinkerScriptSymbols"

This will define a symbol "_StackEnd" to have the value of ".", which is the current address, which at this point is 0x20000400. In the assembly source file, you can now replace the number with the symbol:

```
.word _StackEnd
```

The assembler will put a placeholder in the object file, which the linker will overwrite with the value of 0x20000400. This modification will not change the output file, but avoids putting absolute addresses in source files. The name "_StackEnd" was chosen arbitrarily; since names that start with an underscore and a capital letter may not be used in C and C++ programs, there is no possibility of conflict if any C/C++ source is added later. Typically, all symbols that are part of the runtime environment and should be "invisible" to C/C++ code are named this way. The same rule applies to names starting with two underscores.

The second entry of the vector table is the address of the very first instruction to be executed after reset. Currently the address is hard-coded as the first address after the vector table. If you wanted to insert some other code before this first instruction, this number would have to be changed. This is obviously impractical, and therefore the number should be replaced by a label as well. Since the code executed at reset is commonly known as the "reset handler", define it like that:

```
.syntax unified
.cpu cortex-m3
.thumb
```

```
.word _StackEnd
.word Reset_Handler
.space 0xe4

.type Reset_Handler, %function
Reset_Handler:

@ Put code here
```

The ".type" directive tells the assembler that the label refers to executable code. The exact meaning of this will be covered later. Leave the ".space" directive alone for now.

Absolute section placement

The vector table needs to be at the beginning of flash memory, and the examples have relied on the assembler putting the first things from the source file into flash memory first. This stops working if you use multiple source files. You can use the linker script to make sure the vector table is always at the beginning of flash memory. To do that, you first have to separate the vector table from the rest of the code so that the linker can handle it specially. This is done by placing the vector table in its own section:

```
.syntax unified
.cpu cortex-m3
.thumb

.section .VectorTable, "a"
.word _StackEnd
.word Reset_Handler
.space 0xe4

.text
.type Reset_Handler, %function
Reset_Handler:
```

Example name: "LinkerScriptAbsolutePlacement"

The ".section" directive instructs the assembler to put the following data into the custom section ".VectorTable". The "a" flag marks this section as allocable, which is required to have the linker allocate memory for it. To place the vector table at the beginning of flash memory, define a new output section in the linker script:

```
MEMORY {
    FLASH   : ORIGIN = 0x8000000,   LENGTH = 128K
    SRAM    : ORIGIN = 0x20000000,  LENGTH =  20K
}

SECTIONS {
    .VectorTable : {
```

```
        *(.VectorTable)
    } >FLASH

    .text : {
        *(.text)
    } >FLASH

    .data (NOLOAD) : {
        . = . + 0x400;
        _StackEnd = .;
        *(.data)
    } >SRAM
}
```

This puts the .VectorTable input section into the equally-named output section. It is also possible to put it into .text alongside the code:

```
MEMORY {
    FLASH   : ORIGIN = 0x8000000,   LENGTH = 128K
    SRAM    : ORIGIN = 0x20000000,  LENGTH =  20K
}

SECTIONS {
    .text : {
        *(.VectorTable)
        *(.text)
    } >FLASH

    .data (NOLOAD) : {
        . = . + 0x400;
        _StackEnd = .;
        *(.data)
    } >SRAM
}
```

Even though both variants produce the same flash image, the first one is slightly nicer to work with in GDB. The modified LED-blinker application now looks like:

```
.syntax unified
.cpu cortex-m3
.thumb

.section .VectorTable, "a"
.word _StackEnd
.word Reset_Handler
.space 0xe4

.text
.type Reset_Handler, %function
Reset_Handler:
```

```
ldr r1, =0x40021018
ldr r0, [r1]
orr r0, r0, #4
str r0, [r1]            @ Set IOPAEN bit in RCC_APB2ENR to 1 to enable
GPIOA

ldr r1, =0x40010804
ldr r0, [r1]
and r0, #0xfffffff0
orr r0, #2
str r0, [r1]            @ Set CNF8:MODE8 in GPIOA_CRH to 2


ldr r0, =0x40010810     @ Load address of GPIOA_BSRR
ldr r1, =0x100          @ Register value to set pin to high
ldr r2, =0x1000000      @ Register value to set pin to low
ldr r3, =1000000        @ Iterations for delay loop

BlinkLoop:
str r1, [r0]            @ Set BS8 in GPIOA_BSRR to 1 to set PA8 high

mov r4, r3
delay1:
subs r4, #1
bne delay1          @ Iterate delay loop

str r2, [r0]            @ Set BR8 in GPIOA_BSRR to 1 to set PA8 low

mov r4, r3
delay2:
subs r4, #1
bne delay2          @ Iterate delay loop

b BlinkLoop
```

Program Structure

Because the vector table is usually the same for all projects, it is handy to move it into a separate file, for example called “vectortable.S”:

```
.syntax unified
.cpu cortex-m3
.thumb

.section .VectorTable, "a"
.word _StackEnd
.word Reset_Handler
.space 0xe4
```

Assemble and link this source code with two assembler commands:

```
arm-none-eabi-as -g prog1.S -o prog1.o
arm-none-eabi-as -g vectortable.S -o vectortable.o
arm-none-eabi-ld prog1.o vectortable.o -o prog1.elf -T stm32f103rb.ld
```

This will result in the dreaded “undefined reference” error. To alleviate this, use the “.global” directive in the main source file “prog1.S”:

```
.syntax unified
.cpu cortex-m3
.thumb

.type Reset_Handler, %function
.global Reset_Handler
Reset_Handler:
@ Code here ...
```

This will tell the assembler to make the symbol “Reset_Handler” visible globally, such that it can be used from other files. By default, the assembler creates a *local* symbol for each label, which can’t be used from other source files (same as *static* in C). The symbol is still there in the final program file, though - it can be used for debugging purposes.

More assembly techniques

After having set up the project for using the linker properly, some more aspects of assembly programming will be introduced.

Instruction set state

As mentioned before, ARM application processors support both the T32 and A32/A64 “ARM” instruction sets, and are capable of dynamically switching between them. This can be used to encode time-critical program parts in the faster A32/64 instruction set, and less critical parts in the T32 “thumb” instruction set to save memory. Actually, reducing program size may improve performance too, because the cache memories may become more effective.

Even though the Cortex-M microcontrollers based on the ARMv7-M architecture do not support the A32/A64 instruction sets, some of the switching-logic is still there, requiring the program code to work accordingly. The switch between the instruction sets happens when jumping with the “bx” “Branch and Exchange” and “blx” “Branch with Link and Exchange” instructions. Since all instructions are of size 2 or 4, and code may only be stored at even addresses, the lowest bit of the address of any instruction is always zero. When performing a jump with “bx” or “blx”, the lowest bit of the target address is used to indicate the instruction set of the jump target: If the bit is “1”, the processor expects the code to be T32, else A32.

Another specialty of the “bx” and “blx” instructions is that they take the jump target address from a register instead as encoding it in the instruction directly. This called an indirect jump. An example of such a jump is:

```
ldr r0, =SomeLabel
bx r0
```

Such indirect jumps are necessary if the difference of the jump target address and the jump instruction is too large to be encoded in the instruction itself for a relative jump. Also, sometimes you want to jump to an address that has been passed from another part of the program, which e.g. happens in C/C++ code when using function pointers or virtual functions.

In these cases, you need to make sure that the lowest bit of the address passed to “bx/blx” via a register has the lowest bit set, to indicate that the target code is T32. Otherwise, the code will crash. This can be achieved by telling the assembler that the target label refers to code (and not data) via the already mentioned “.type” directive:

```
.type SomeLabel, %function
SomeLabel:
@ Some code...
```

That way, when you refer to the label to load its address into a register, the lowest bit will be set. Actually, using “.type” for all code labels is a good idea, even though it does not matter if you only refer to a label via the “b” instruction (including the conditional variant) which does not encode the lowest bit and does not attempt to perform an instruction set switch.

As was already shown, there is another case where the lowest bit matters: when specifying the address of the reset handler (and later, exception handler functions) in the vector table, the bit must be set, so the “.type” directive is necessary here too:

```
.type Reset_Handler, %function
```

If you were writing code for a Cortex-A processor, you would use “.arm” instead of “.thumb” to have your code (or performance critical parts of it) encoded as A32. The “.type” directive would be used as well, and the assembler would clear the lowest bit in the address to ensure the code is executed as A32. For example:

```
.cpu cortex-a8
.syntax unified

@ Small but slower code here
.thumb

.type Block1, %function
Block1:
ldr r0, =Block2
bx r0
```

```
@ Larger but faster code here
.arm

.type Block2, %function
Block2:
@ ...
```

The directive ".code 32" has the same meaning as ".arm", and ".code 16" the same as ".thumb" (although the name is slightly misleading, as T32 instructions can be 32 bit as well). There is also ".type Label, %object" to declare some label refers to data in flash or RAM; this is optional, but helps in working with analysis tools (see below).

Constants

The previous examples contain a lot of numbers (esp. addresses), the meaning of which is not obvious to the reader - so called "magic numbers". As code is typically read many times more than written/modified, readability is important, even for assembly code. Therefore, it is common practice to define constants that assign names to numbers such as addresses, and use names instead of the number directly.

The assembler actually does not provide any dedicated mechanism for defining constants. Instead, symbols as introduced before are used. You can define a symbol in any of the following ways:

```
RCC_APB2ENR = 0x40021018
.set GPIOA_CRH, 0x40010804
.equ GPIOA_ODR, 0x4001080C
```

and then use it in place of the number:

```
ldr r1, =RCC_APB2ENR
```

Replacing (almost) all numbers in the source code for the LED blinker by constants yields a source code like this:

```
.syntax unified
.cpu cortex-m3
.thumb

RCC_APB2ENR = 0x40021018
RCC_APB2ENR_IOPAEN = 4
GPIOA_CRH = 0x40010804

GPIOA_BSRR = 0x40010810
GPIOx_BSRR_BS8 = 0x100
GPIOx_BSRR_BR8 = 0x1000000

GPIOx_CRx_GP_PP_10MHz = 1
GPIOx_CRx_GP_PP_2MHz = 2
```

```
GPIOx_CRx_GP_PP_50MHz = 3

GPIOx_CRx_GP_OD_10MHz = 1|4
GPIOx_CRx_GP_OD_2MHz = 2|4
GPIOx_CRx_GP_OD_50MHz = 3|4

GPIOx_CRx_AF_PP_10MHz = 1|8
GPIOx_CRx_AF_PP_2MHz = 2|8
GPIOx_CRx_AF_PP_50MHz = 3|8

GPIOx_CRx_AF_OD_10MHz = 1|4|8
GPIOx_CRx_AF_OD_2MHz = 2|4|8
GPIOx_CRx_AF_OD_50MHz = 3|4|8

GPIOx_CRx_IN_ANLG = 0
GPIOx_CRx_IN_FLOAT = 4
GPIOx_CRx_IN_PULL = 8


DelayLoopIterations = 1000000

.text
.type Reset_Handler, %function
.global Reset_Handler
Reset_Handler:

ldr r1, =RCC_APB2ENR
ldr r0, [r1]
orr r0, r0, #RCC_APB2ENR_IOPAEN
str r0, [r1]            @ Set IOPAEN bit in RCC_APB2ENR to 1 to enable
GPIOA

ldr r1, =GPIOA_CRH
ldr r0, [r1]
and r0, #0xfffffff0
orr r0, #GPIOx_CRx_GP_PP_2MHz
str r0, [r1]            @ Set CNF8:MODE8 in GPIOA_CRH to 2


ldr r0, =GPIOA_BSRR     @ Load address of GPIOA_BSRR
ldr r1, =GPIOx_BSRR_BS8         @ Register value to set pin to high
ldr r2, =GPIOx_BSRR_BR8     @ Register value to set pin to low
ldr r3, =DelayLoopIterations        @ Iterations for delay loop

BlinkLoop:
str r1, [r0]            @ Set BS8 in GPIOA_BSRR to 1 to set PA8 high

mov r4, r3
delay1:
subs r4, #1
```

```
bne delay1          @ Iterate delay loop

str r2, [r0]            @ Set BR8 in GPIOA_BSRR to 1 to set PA8 low

mov r4, r3
delay2:
subs r4, #1
bne delay2          @ Iterate delay loop

b BlinkLoop
```

Example name: “BlinkConstants”

This is much more readable than before. In fact, you could even leave out the comments, as the code becomes more self-documenting. The addresses of periphery registers are defined individually, but the bits for the GPIO registers are the same for each GPIO module, so the names include an “x” to denote that they apply to all GPIO modules.

The “CRL”/“CRH” registers get a special treatment. Since the individual bits have little direct meaning, it would be pointless to name them. Instead, 15 symbols are defined to denote the 15 possible modes of operation per pin (combinations of input/output, open-drain vs. push-pull, analog vs. digital, floating vs. pull-resistors, and output driver slew rate). Each of the 15 symbols has a 4 bit value that needs to be written into the appropriate 4 bits of the register. To configure e.g. PA10 as General Purpose Open-Drain with 10 MHz slew rate:

```
ldr r1, =GPIOA_CRH
ldr r0, [r1]
and r0, #0xfffff0ff
orr r0, #(GPIOx_CRx_GP_OD_10MHz<<8)
str r0, [r1]
```

C-like arithmetic operators can be used in constant expressions, like + - * / and bitwise operators like | (or), & (and), << (left shift) and >> (right shift). Note that these calculations are always done by the assembler. In the example, or | is used to combine bit values.

Since these constants are actually symbols, they can collide with assembler labels, so you must not define a symbol with the same name as any label.

A different kind of constants are register aliases. Using the “.req” directive, you can define a name for a processor register:

```
MyData .req r7
ldr MyData, =123
add MyData, 3
```

This can be useful for large assembly blocks where the meaning of register data is not obvious. It also allows you to re-assign registers without having to modify many lines of code.

The Stack

In computer science, a stack is a dynamic data structure where data can be added and removed flexibly. Like a stack of books, the last element that was put on top must be taken and removed first (LIFO-structure - Last In, First Out). Adding an element is usually called “push”, and reading & removing “pop”.

Many processor architectures including ARM feature circuitry to deal with such a structure efficiently. Like most others, ARM does not provide a dedicated memory area for this - it just facilitates using an area that the programmer reserved for this purpose as a stack. Therefore, a part of the SRAM needs to be reserved for the stack.

On ARM, the program stores processor registers on the stack, i.e. 32bit per element. The stack is commonly used when the contents of some register will be needed again later after it has been overwritten by some complex operation that needs many registers. These accesses always come in pairs:

- Some operation that writes to r0
- **Push** (save) r0 to the stack
- Some operation that overwrites r0
- **Pop** (restore) r0 from the stack
- Use the value in r0 which is the same as initially assigned

ARM’s instructions for accessing the stack are unsurprisingly called “push” and “pop”. They can save/restore any of the registers r0-r12 and r14, for example:

```
ldr r0, =1000000
@ Use r0 ...
push { r0 } @ Save value 1000000

@ … Some code that overwrites r0 …

pop { r0 }  @ Restore value 1000000
@ Continue using r0 ...
```

It is also possible to save/restore multiple registers in one go:

```
ldr r0, =1000000
ldr r1, =1234567
@ Use r0 and r1 ...
push { r0, r1 } @ Save values 1000000 and 1234567

@ … Some code that overwrites r0 and r1 …

pop { r0, r2 }  @ Restore 1000000 into r0 and 1234567 into r2
@ Continue using r0 and r2...
```

It does not matter to which register the data is read back - in the previous example, the value that was held in r1 is restored into r2. In larger applications, many store-restore pairs will be nested:

```
ldr r0, =1000000
@ Use r0 ...
push { r0 } @ Save value 1000000

@ Inner Code Block:

ldr r0, =123
@ Use r0 …

push { r0 } @ Save value 123

@ Inner-Inner Code Block that overwrites r0

pop { r0 }  @ Restore value 123
@ Continue using r0 ...

pop { r0 }  @ Restore value 1000000 into r0

@ Continue using r0 …
```

The “inner” push-pop pair works with value 123, and the “outer” push-pop pair works with value 1000000. Assuming that the stack was empty at the beginning, it will contain 1000000 after the first “push”, and both 1000000 and 123 after the second push. After the first “pop” it contains only 1000000 again, and is empty after the second “pop”.

At the beginning of a push-pop pair, the current contents of the stack is irrelevant - it may be empty or contain many elements. After the “pop”, the stack will be restored to its previous state. This makes it possible to (almost) arbitrarily nest push-pop-pairs - after any inner push-pop-pair has completed, the stack is in the same state as before entering the inner pair, so the “pop” part of the outer pair doesn’t even notice the stack was manipulated in between. This is why it is important to make sure that each “push” has a matching “pop”, and vice-versa.

As mentioned, an area of memory has to be reserved for the stack. Access to the stack memory is managed via the stack pointer (SP). The stack pointer resides in the processor register r13, and “sp” is an alias for that. As the name implies, the stack pointer contains a 32bit memory address - specifically, the address of the first byte in the stack that contains any saved data.

When storing a 32bit register value using “push”, the stack pointer is **first** decremented by 4 before the value is written at the newly calculated address. To restore a value, the address currently stored in the stack pointer is read from memory, after which the stack pointer is incremented by 4. This is called a “full-descending” stack (see the ARM Architecture Reference Manual, chapter B1.5.6). On ARMv7-A (Cortex-A), this behaviour can be changed, but on ARMv7-M, it is dictated by the exception handling logic, which will be explained later.

An implication of this is that if the stack is empty, the stack pointer contains the address of the first byte **after** the stack memory area. If the stack is completely full, it contains the address of the very first byte **inside** the stack memory area. This means that the stack grows **downward**. Since the stack is empty at program start, the stack pointer therefore needs to be initialized to the first address after the memory area. Before executing the first instruction, the processor loads the first 4 bytes from the flash into the stack pointer. This is why "_StackEnd" was defined and used to place the address of the first byte after the stack memory region into the first 4 bytes of flash.

The stack pointer must always be a multiple of 4 (see chapter B5.1.3 in the ARM Architecture Reference Manual). It is a common error (which is even present in the example projects by ST!) to initialize the stack pointer to the last address *inside* the stack memory area (e.g. 0x200003FF instead of 0x20000400), which is not divisible by four. This can cause the application to crash or "just" slow it down. Actually, the ARM ABI requires the stack pointer to be a multiple of 8 for public software interfaces, which is important for e.g. the "printf" C function. So, when calling any external code, make sure the stack pointer is a multiple of 8.

In the previous examples, the stack memory area was defined with a size of 0x400, i.e. 1KiB. Choosing an appropriate stack size is critical for an application; if it is too small, the application will crash, if it is too large, memory is wasted that could be used otherwise. Traditionally, the stack is configured to reside at the *end* of available memory, e.g. 0x20005000 for the STM32F103. As the linker starts allocating memory for data (using ".data" in assembly or global/static variables in C) at the beginning of the memory, the stack is as far away from that regular data as possible, minimizing the chance of a collision. However, if the stack grows continuously, the stack pointer might end up pointing into the regular data area (".data" or C globals) or heap memory (used by "malloc" in C). In that case, writing to the stack silently overwrites some of the regular data. This can result in all kinds of hard to find errors. Therefore, the example codes put the stack area at the *beginning* of RAM, and the regular data after that - if the stack grows too large, the stack pointer will reach values below 0x20000000, and any access will result in an immediate "clean" crash. It is probably easy to find the code location that allocates too much stack memory, and possibly increase the stack size. Using the Cortex-M3's memory protection unit (MPU) enables even more sophisticated strategies, but that is out of scope for this tutorial.

Function calls

Many programming languages feature a "function" concept. Also known as a "procedures" or "subprograms", functions are the most basic building blocks of larger applications, and applying them correctly is key for clean, reusable code. The assembler does not know about functions directly, so you have to build them yourself. A function is a block of code (i.e. a sequence of instructions) that you can jump to, does some work, and then jumps back to the place from which the first jump originated. This ability to jump back is the main difference from any other block of assembly code. To make this explicit, such a jump to a function is known as a "call" (as in "calling a function"). The location in code that starts the jump to the function is known as the "caller", and the called function as "callee". From the perspective of the caller, calling a function resembles a "user-defined"

instruction - it performs some operation after which the code of the caller continues as before. To make the jump back possible, the address of the *next* instruction after the one that started the function call needs to be saved, so that the function can jump back to that location (without calling the function directly again).

This is done via the Link Register (LR), which is the processor register r14. Function calls are performed with the “bl” instruction. This instruction performs a jump, much like the well-known “b”, but also saves the address of the next instruction in LR. When the function is finished, it returns to the caller by jumping to the address stored in LR. As already mentioned, jumping to a location from a register is called an indirect jump, which is performed by the “bx” instruction. So, to return from a function, use “bx lr”:

```
.text
.type Reset_Handler, %function
.global Reset_Handler
Reset_Handler:

bl EnableClockGPIOA     @ Call function to enable GPIOA's peripheral clock

@ Some more code ...
ldr r1, =GPIOA_CRH
ldr r0, [r1]
and r0, #0xfffffff0
orr r0, #GPIOx_CRx_GP_PP_2MHz
str r0, [r1]


.type EnableClockGPIOA, %function
EnableClockGPIOA:
    ldr r1, =RCC_APB2ENR
    ldr r0, [r1]
    orr r0, r0, #RCC_APB2ENR_IOPAEN
    str r0, [r1]    @ Set IOPAEN bit in RCC_APB2ENR to 1 to enable GPIOA
    bx lr   @ Return to caller
```

Here, the code to enable the clock for GPIOA was packaged into a function. To enable this clock, only a single line is now required - “bl EnableClockGPIOA”.

When calling a function, the “bl” instruction automatically makes sure to set the lowest bit in LR such that the subsequent “bx lr” will not crash because of an attempted instruction set switch, which is not possible on Cortex-M. If you need to call a function indirectly, use “blx” with a register, and remember to ensure that the lowest bit is set, typically via “.type YourFunction, %function”. Usually, all the code of an application resides within functions, with the possible exception of the Reset_Handler. The order in which functions are defined in the source files does not matter, as the linker will always automatically fill in the correct addresses. If you want to put functions in separate source files, remember to use “.global FunctionName” to make sure the symbol is visible to other files.

Using the stack for functions

In large applications it is common for functions to call other functions in a deeply nested fashion. However, a function implemented as shown can't do that - using "bl" would overwrite the LR, and so the return address of the outer function would be lost, and that function couldn't ever return. The solution is to use the stack: At the beginning of a function that calls other functions, use "push" to save the LR, and at the end use "pop" to restore it. For example, the blink program could be restructured like this:

```
.syntax unified
.cpu cortex-m3
.thumb

RCC_APB2ENR = 0x40021018
RCC_APB2ENR_IOPAEN = 4
GPIOA_CRH = 0x40010804

GPIOA_BSRR = 0x40010810
GPIOx_BSRR_BS8 = 0x100
GPIOx_BSRR_BR8 = 0x1000000

GPIOx_CRx_GP_PP_2MHz = 2

DelayLoopIterations = 1000000

.text
.type Reset_Handler, %function
.global Reset_Handler
Reset_Handler:
    bl EnableClockGPIOA
    bl ConfigurePA8
    ldr r5, =5                      @ Number of LED flashes.
    bl Blink
    b .

.type Blink, %function
Blink:
    push { lr }
    ldr r0, =GPIOA_BSRR             @ Load address of GPIOA_BSRR
    ldr r1, =GPIOx_BSRR_BS8         @ Register value to set pin to high
    ldr r2, =GPIOx_BSRR_BR8         @ Register value to set pin to low
    ldr r3, =DelayLoopIterations    @ Iterations for delay loop

    BlinkLoop:
        str r1, [r0]                @ Set BS8 in GPIOA_BSRR to 1 to set PA8
high

        bl Delay
```

```
        str r2, [r0]                    @ Set BR8 in GPIOA_BSRR to 1 to set PA8
low

        bl Delay

        subs r5, #1
        bne BlinkLoop

    pop { lr }
    bx lr

.type EnableClockGPIOA, %function
EnableClockGPIOA:
    ldr r1, =RCC_APB2ENR
    ldr r0, [r1]
    orr r0, r0, #RCC_APB2ENR_IOPAEN
    str r0, [r1]                        @ Set IOPAEN bit in RCC_APB2ENR to 1 to
enable GPIOA
    bx lr   @ Return to caller


.type ConfigurePA8, %function
ConfigurePA8:
    ldr r1, =GPIOA_CRH
    ldr r0, [r1]
    and r0, #0xfffffff0
    orr r0, #GPIOx_CRx_GP_PP_2MHz
    str r0, [r1]                        @ Set CNF8:MODE8 in GPIOA_CRH to 2
    bx lr

.type Delay, %function
Delay:
    mov r4, r3
    DelayLoop:
    subs r4, #1
    bne DelayLoop                       @ Iterate delay loop
    bx lr
```

Example name: “BlinkFunctions”

The Reset_Handler just became much prettier. There now are functions for enabling the GPIOA clock, configuring PA8 as output, and one that delays execution so that the LED blinking is visible. The “Blink” function performs the blinking, but only for 5 flashes, after which it returns (an endless blink-loop wouldn’t be good for demonstrating returns). As you see, LR is saved on the stack to allow “Blink” to call further functions.

The two lines

```
pop { lr }
bx lr
```

are actually longer than necessary. It is actually possible to directly load the return address from the stack into the program counter, PC:

```
pop { pc }
```

This way, the return address that was saved on the stack is directly used for the jump back. Just the same way, you can use “push” and “pop” to save and restore any other registers while your function is running.

Calling Convention

Actually building a large program as shown in the last example is a bad idea. The “Delay” function requires 1000000 to reside in r4. The “Blink” function relies on “Delay” not overwriting r0-r2, and r5, and requires the number of flashes to be given via r5. Such requirements can quickly grow into an intricate web of interdependencies, that make it impossible to write larger functions that call several sub-functions or restructure anything. Therefore, it is common to use a calling convention, which defines which registers a function may overwrite, which it should keep, how it should use the stack, and how to pass information back to the caller.

When building an entire application out of your own assembly code, you can invent your own calling convention. However, it is always a good idea to use existing standards: The AAPCS defines a calling convention for ARM. This convention is also followed by C and C++ compilers, so using it makes your code automatically compatible with those. The Cortex-M interrupt mechanism follows it too, which would make it awkward to adapt code that uses some other convention to Interrupts. The specification of the calling convention is quite complex, so here is a quick summary of the basics:

- Functions may only modify the registers r0-3 and r12. If more registers are needed, they have to be saved and restored using the stack. The APSR may be modified too.
- The LR is used as shown for the return address.
- When returning (via “bx lr”) the stack should be exactly in the same state as during the jump to the function (via “bl”).
- The registers r0-r3 may be used to pass additional information to a function, called parameters, and the function may overwrite them.
- The register r0 may be used to pass a result value back to the caller, which is called the return value.

This means that when you call a function, you must assume registers r0-r3 and r12 may be overwritten but the others keep their values. In other words, the registers r0-r3 and r12 are (if at all) saved *outside* the function (“caller-save”), and the registers r4-r11 are (if at all) saved *inside* the function (“callee-save”).

A function that does not call any other functions is called a “leaf-function” (as it is a leaf in the call tree). If such a function is simple, it might not require to touch the stack at all, as the return value is just saved in a register (LR) and it might only overwrite the registers r0-r3 and r12, which the caller

can make sure to contain no important data. This makes small functions efficient, as register accesses are faster than memory accesses, such as to the stack.

If all your functions follow the calling convention, you can call any function from anywhere and be sure about what it overwrites, even if it calls many other functions on its own. Restructuring the LED blinker could look like this:

```
.syntax unified
.cpu cortex-m3
.thumb

RCC_APB2ENR = 0x40021018
RCC_APB2ENR_IOPAEN = 4
GPIOA_CRH = 0x40010804

GPIOA_BSRR = 0x40010810
GPIOx_BSRR_BS8 = 0x100
GPIOx_BSRR_BR8 = 0x1000000

GPIOx_CRx_GP_PP_2MHz = 2

DelayLoopIterations = 1000000

.text
.type Reset_Handler, %function
.global Reset_Handler
Reset_Handler:
    bl EnableClockGPIOA
    bl ConfigurePA8
    ldr r0, =5
    bl Blink
    b .

.type Blink, %function
Blink:
    push { r4-r7, lr }
    ldr r4, =GPIOA_BSRR     @ Load address of GPIOA_BSRR
    ldr r5, =GPIOx_BSRR_BS8         @ Register value to set pin to high
    ldr r6, =GPIOx_BSRR_BR8     @ Register value to set pin to low
    mov r7, r0                      @ Number of LED flashes.

    BlinkLoop:
        str r5, [r4]            @ Set BS8 in GPIOA_BSRR to 1 to set PA8
high

        ldr r0, =DelayLoopIterations        @ Iterations for delay loop
        bl Delay

        str r6, [r4]            @ Set BR8 in GPIOA_BSRR to 1 to set PA8 low
```

```
        ldr r0, =DelayLoopIterations        @ Iterations for delay loop
        bl Delay

        subs r7, #1
        bne BlinkLoop

    pop { r4-r7, pc }

.type EnableClockGPIOA, %function
EnableClockGPIOA:
    ldr r1, =RCC_APB2ENR
    ldr r0, [r1]
    orr r0, r0, #RCC_APB2ENR_IOPAEN
    str r0, [r1]    @ Set IOPAEN bit in RCC_APB2ENR to 1 to enable GPIOA
    bx lr   @ Return to caller

.type ConfigurePA8, %function
ConfigurePA8:
    ldr r1, =GPIOA_CRH
    ldr r0, [r1]
    and r0, #0xfffffff0
    orr r0, #GPIOx_CRx_GP_PP_2MHz
    str r0, [r1]            @ Set CNF8:MODE8 in GPIOA_CRH to 2
    bx lr

@ Parameters: r0 = Number of iterations
.type Delay, %function
Delay:
    DelayLoop:
    subs r0, #1
    bne DelayLoop           @ Iterate delay loop
    bx lr
```

Example name: “BlinkFunctionCallingConvention”

The three small functions at the end only use registers r0 and r1, which they are free to overwrite. The “Delay” function expects the number of iterations as a parameter in r0, which it then modifies. Therefore, the “Blink” function fills r0 before every call to “Delay”. Alternatively, “Delay” could use a fixed iteration count, i.e. the “ldr” could be moved into “Delay”. As the “Blink” function must assume that “Delay” overwrites r0-r3 and r12, it keeps its own data in r4-r7, which are guaranteed to be retained according to the calling convention. Since “Blink”, in turn, must preserve these registers for the function that called it, it uses “push” and “pop” to save and restore them. Note the shortened syntax “r4-r7” in the instructions. The number of LED flashes is passed in r0 as a parameter; as this register will be overwritten, this number is moved to r7.

Alternatively, “Blink” could re-load the constants each time they are used in r1/r2, such that only one register (r4) needs to be saved as it is needed to count the number of flashes:

```
.type Blink, %function
Blink:
    push { r4, lr }

    mov r4, r0

    BlinkLoop:
     ldr r1, =GPIOA_BSRR     @ Load address of GPIOA_BSRR
     ldr r2, =GPIOx_BSRR_BS8         @ Register value to set pin to high
     str r2, [r1]            @ Set BS8 in GPIOA_BSRR to 1 to set PA8 high

     ldr r0, =DelayLoopIterations        @ Iterations for delay loop
     bl Delay

     ldr r1, =GPIOA_BSRR     @ Load address of GPIOA_BSRR
     ldr r2, =GPIOx_BSRR_BR8     @ Register value to set pin to low
     str r2, [r1]            @ Set BR8 in GPIOA_BSRR to 1 to set PA8 low

     ldr r0, =DelayLoopIterations        @ Iterations for delay loop
     bl Delay

     subs r4, #1
     bne BlinkLoop

    pop { r4, pc }
```

Example name: “BlinkFunctionCallingConvention2”

A third variant would not use any of the callee-save-registers (r4-r11) at all, and instead just save r0 before the function calls and restore it as needed

```
.type Blink, %function
Blink:
    push { lr }

    BlinkLoop:
        push { r0 }

        ldr r1, =GPIOA_BSRR                 @ Load address of GPIOA_BSRR
        ldr r2, =GPIOx_BSRR_BS8             @ Register value to set pin to
high
        str r2, [r1]                        @ Set BS8 in GPIOA_BSRR to 1 to
set PA8 high

        ldr r0, =DelayLoopIterations        @ Iterations for delay loop
        bl Delay

        ldr r1, =GPIOA_BSRR                 @ Load address of GPIOA_BSRR
        ldr r2, =GPIOx_BSRR_BR8             @ Register value to set pin to
low
```

```
        str r2, [r1]                    @ Set BR8 in GPIOA_BSRR to 1 to
set PA8 low

        ldr r0, =DelayLoopIterations    @ Iterations for delay loop
        bl Delay

        pop { r0 }
        subs r0, #1
        bne BlinkLoop

    pop { pc }
```

Example name: “BlinkFunctionCallingConvention3”

The frequent stack accesses would however make this slower. Be sure to always document the meaning (and units, if applicable) of parameters e.g. via comments.

Conditional Execution

As mentioned, the conditional variants of the “b” instruction (e.g. “bne”) can be used to execute certain blocks of code only if a certain condition is met. First, more ways to formulate conditions will be shown. Next, the ARM instruction “it” will be introduced, which makes executing small blocks of code conditionally more efficient.

Conditions

All conditions for conditional execution depend on the outcome of some mathematical operation. When instructions such as “adds”, “subs”, “ands” are used, they update the flags in the APSR register depending on the outcome, which are then read by the conditional variants of “b” to decide whether to actually perform the jump.

Often it is necessary to compare two numbers without actually doing a calculation. This can be done with the “cmp” instruction to which you can pass two registers or a register and a literal:

```
cmp r0, #42
cmp r0, r1
```

The “cmp” instruction is very similar to “subs” - it subtracts the second operand from the first, but doesn’t save the result anywhere, i.e. the registers keep their values. Just the flags in the APSR are updated according to the result, just as with “subs”. For example, if both operands were equal, the result of the subtraction is zero, and the zero flag will be set. So, to test whether two numbers are equal:

```
cmp r0, #42
beq TheAnswer

@ This is executed if r0 is not 42
```

```
TheAnswer:
@ This is executed if r0 is 42
```

The “bne” instruction is the opposite of “beq”.

The “tst” instruction works similarly to “cmp”, but instead of subtracting, perform a bitwise “and” operation - like the “ands” instruction, but without keeping the result. This way, you can test whether a bit in a register is set:

```
tst r0, #4
beq BitNotSet

@ This is executed if bit 2 in r0 is set

BitNotSet:
@ This is executed if bit 2 in r0 is not set
```

A more useful use case for “tst” is to pass the same register twice. Applying “and” to the same value twice yields the same result as the input, so “tst” in this case effectively checks the properties of the input (negative/positive, zero):

```
tst r0, r0
beq ValueZero

@ This is executed if r0 is not zero

ValueZero:
@ This is executed if r0 is zero
```

There is also the “teq” instruction which performs an exclusive or operation.

As mentioned, the suffixes “eq” and “ne” are called condition codes. ARM has 14 of those which define how the flags in the APSR form the condition. The details about how a subtraction (by “subs” or “cmp”) sets the flags in the APSR and how their interpretation by the different condition codes correlates to the mathematical result are somewhat complicated, involving the way the 2’s complement format works and relying on the fact that subtracting works by adding a negated number. Instead of diving into all the details, a table with a more high-level view and a practical interpretation of the condition should be more helpful:

Code	Meaning	Unsigned / Signed	Flags	Condition after “cmp/subs r0, r1”	Condition after “tst r0, r0”
EQ	Equal	U+S	Z==1	r0 = r1	r0 = 0
NE	Not equal	U+S	Z==0	r0 ≠ r1	r0 ≠ 0

MI	Negative	S	N==1	---	r0 < 0
PL	Positive or Zero	S	N==0	---	r0 ≥ 0
VS	Overflow	S	V==1	r0-r1 out of range[1]	---
VC	No Overflow	S	V==0	r0-r1 in range[1]	---
HS	Unsigned higher or same	U	C==1	r0 ≥ r1	---
LO	Unsigned lower	U	C==0	r0 < r1	---
HI	Unsigned higher	U	C==1 and Z==0	r0 > r1	---
LS	Unsigned lower or same	U	C==0 or Z==1	r0 ≤ r1	---
GE	Signed greater or equal	S	N==V	r0 ≥ r1	---
LT	Signed less than	S	N!=V	r0 < r1	---
GT	Signed greater than	S	Z==0 and N==V	r0 > r1	---
LE	Signed less or equal	S	Z==1 or N!=V	r0 ≤ r1	---

1: Range meaning the numbers from -(2^31) until (2^31-1), inclusive

To determine which condition code you need, first think about whether the number is unsigned (range 0 to 2^32-1) or is using two's complement to represent signed numbers (range -2^31 to 2^31-1). Ignore all rows in the table with the wrong format.

If you want to compare two numbers, use the “cmp” instruction, and search for the desired condition within the “cmp”-condition column of the table. If you want to test a single number's properties, use the “tst”-column. Use the condition code from the first column with the conditional “b” instruction (“bne”, “beq”, “bmi”, “bpl”, “bhs”, ...) right after the appropriate “cmp”/”tst” instruction.

Note that all the condition codes have a corresponding inverse code that has exactly the negated meaning. Most also have a swapped partner code, using which is equivalent to swapping the operands for cmp.

The IT instruction

Jumping is inefficient, so having many conditional jumps may slow down your program. The ARM architecture offers a way to make a few instructions conditional without requiring a jump via the “it” (if-then) instruction. It is used in place of a conditional jump after an instruction that set the flags (“cmp”, “tst”, “adds”…) and also needs a condition code. The next instruction right after the it will then only be executed when the condition is met, and skipped otherwise. You have to repeat the condition code and add it to that instruction; this is just to make the code clearer and avoid confusion.

```
ldr r4, =GPIOA_BSRR             @ Load address of GPIOA_BSRR
ldr r5, =GPIOx_BSRR_BS8         @ Register value to set pin to high

ldr r0, =1                      @ Load some date to compare
ldr r1, =2

cmp r0, r1                      @ Perform comparison

it hi                           @ Make the next instruction conditional
strhi r5, [r4]                  @ Set BS8 in GPIOA_BSRR to 1 to set PA8
high
```

This checks if r0 is higher than r1 (it isn’t), and only sets the pin PA8 to high if this condition is met. Up to 4 instructions can be made conditional like this; for each one, an additional “t” has to be appended to the “it” instruction:

```
cmp r0, r1                      @ Perform comparison

ittt hi                         @ Make the next instruction conditional
ldrhi r4, =GPIOA_BSRR               @ Load address of GPIOA_BSRR
ldrhi r5, =GPIOx_BSRR_BS8           @ Register value to set pin to high
strhi r5, [r4]                  @ Set BS8 in GPIOA_BSRR to 1 to set PA8
high
```

You can also add instructions that will be executed if the condition was *not* met (like an “else”-case in high-level-languages), by appending “e” instead of “t” to the “it” instruction. Since the “t” in “it” is fixed, the first instruction is always executed if the condition is met; only the next three instructions can be either a “then” case (“t”) or “else” case (“e”). You also have to provide the inverted condition code for the “else”-instructions:

```
ldr r4, =GPIOA_BSRR             @ Load address of GPIOA_BSRR
ldr r5, =GPIOx_BSRR_BS8         @ Register value to set pin to high
ldr r6, =GPIOx_BSRR_BR8         @ Register value to set pin to low

ldr r0, =1                      @ Load some date to compare
ldr r1, =2

cmp r0, r1                      @ Perform comparison
```

```
ite hi                          @ Make the next two instructions
conditional (if-then-else)
strhi r5, [r4]                  @ Set BS8 in GPIOA_BSRR to 1 to set PA8
high
strls r6, [r4]                  @ Set BR8 in GPIOA_BSRR to 1 to set PA8 low
```

There are several restrictions on which instructions may appear within an it-block. Most importantly, instructions that set the flags are forbidden here, as is the “b” instruction except for the last instruction in an “it” block. Directly jumping to one of the conditional instructions is forbidden too.

In T32 code, only the conditional “b” instruction is capable of encoding a condition code together with some operation, so the “it” instruction is provided to make any instruction conditional. On A32, most instructions include a condition code and can therefore be conditional, and the “it” instruction is actually ignored by the assembler here. You can and should still put “it” into code intended for A32, as this makes it compatible with T32. This is one of the reasons why A32 is more time-efficient, and T32 more space-efficient.

Conditional instructions sometimes make surprisingly compact programs. For example, the euclidean algorithm for calculating the greatest common divisor (gcd) of two numbers can be written in ARM assembly like this:

```
gcd:
    cmp r0, r1
    ite gt
    subgt r0, r0, r1
    suble r1, r1, r0
    bne gcd
```

While the C equivalent is actually longer:

```
int gcd(int a, int b) {
    while (a != b) {
        if (a > b)
            a = a - b;
        else
            b = b - a;
    }
    return a;
}
```

The usage of conditional instructions is also faster than using conditional jumps. Note that the final “bne” instruction is independent of the “if-then” block; it just directly uses the result of “cmp”.

8/16 bit arithmetic

So far, all numbers had 32 bit. However, especially for space reasons, smaller numbers are needed with 8 or 16 bit. Cortex-M3 doesn't provide any instructions for calculating 8 or 16 bit numbers

directly. Instead, after loading such a number from memory into a processor register, it has to be extended into 32bit to allow the 32bit instructions to work properly. When storing the result back, only the lower 8/16 bit are used. If 8/16bit overflow behavior is required (i.e. overflow at -128/127 for 8bit signed, 0/256 for 8bit unsigned, -32768/32767 for 16bit signed, 0/65536 for 16bit unsigned) for calculations, the numbers have to be truncated after each calculation. This actually makes it slightly less efficient to deal with smaller numbers.

A 16bit value (“halfword”) can be read from memory with the ldrh instruction:

```
ldr r0, =SomeAddress
ldrh r1, [r0]
```

“ldrh” loads 16bit from memory, writes them into the lower 16 bits of the destination register (here: r1), and sets the upper 16bits to zero. If the value is signed, it has to be sign-extended so that it can be used with 32bit-calculations:

```
ldr r0, =SomeAddress
ldrh r1, [r0]
sxth r1, r1
```

The “sxth” instruction copies the sign bit (i.e. bit 15) into the upper 16 bits (“sign-extension”); this makes sure that negative 16bit-numers keep their value when interpreted as 32 bits. The “ldrsh” instruction combines both “ldrh” and “sxth”. “ldrb”, “sxtb”, “ldrsb” are for loading and sign-extending 8bit-values and the combination of both, respectively.

To simulate 8/16bit overflow behaviour after a mathematical operation, use uxtb/uxth for unsigned 8/16 bit numbers, or sxtb/sxth for signed 8/16 bit numbers:

```
add r0, #1
uxth r0, r0
```

The “uxth”/”uxtb” instructions copy the lower 16/8 bits of a register into another one, setting the upper 16/24 bits to zero. This way, if r0 contained 65535 before, the result will be 0 instead of 65536 after using “uxth”.

This is a common trap when coding in C - when using e.g. the "uint16_t" type for local variables such as loop counters, this implicitly requests 16bit overflow behavior, requiring the truncating after each calculation, even though the overflow may actually never happen. This is why e.g. uint16_fast_t should be used for local variables, as this is 32 bit on ARM, which is faster.

Alignment

There are certain restrictions on the address when accessing data in memory using the “str”/”ldr” variants:

- The “ldrd”/”strd”/”ldm”/”stm” instructions, which can load/store multiple registers at once, always require the address to be a multiple of 4. If it isn’t, the program will crash.
- The “ldr”/”str” instructions require the address to be a multiple of 4, and “strh”/”ldrh” require it to be a multiple of 2. If it isn’t, the behaviour depends on the ARM version:
  - On ARMv6-M and before, the program will crash.
  - On ARMv7-M:
    - If the CCR.UNALIGN_TRP is set to zero (the default), the access will be slow
    - If the CCR.UNALIGN_TRP bit is set to one, the program will crash, emulating the ARMv6-M behaviour

For “strb”/”ldrb” there are no such requirements.

The number of which the address needs to be a multiple of is called the “alignment” (e.g. 2-byte-alignment, 4-byte-alignment, …). An access with an address that is a multiple of 2/4 as specified above is called an “aligned access”; others are called “unaligned access” (which are slow or cause a crash).

Even though slow accesses may be acceptable, it is still a good idea to make sure all accesses are always correctly aligned in case the code is ported to an ARM version or operating system that requires it. The addresses of periphery registers are already aligned correctly, so there is no need to worry. When placing data in RAM however, you should make sure that the addresses of the individual elements that are accessed via one of the “ldr” variants are aligned properly. For example, if a previous example code was modified like this:

```
.data
var2:
    .space 1        @ Reserve 1 byte for memory block “var2”
var1:
    .space 4        @ Reserve 4 bytes for memory block “var1”

.text
@ Instructions go here...
```

The address of “var1” will not be a multiple of 4, and an access via “ldr” would be unaligned. This could be improved by adding a space of 3 bytes in between:

```
.data
var2:
    .space 1        @ Reserve 1 byte for memory block “var2”
    .space 3
var1:
    .space 4        @ Reserve 4 bytes for memory block “var1”

.text
@ Instructions go here...
```

This would require you to keep in mind all the other things in memory that were declared before, which is impractical especially if multiple assembly files are used. Therefore, the assembler offers the ".align" directive:

```
.data
var2:
    .space 1        @ Reserve 1 byte for memory block "var2"
    .align 2
var1:
    .space 4        @ Reserve 4 bytes for memory block "var1"

.text
@ Instructions go here...
```

When using ".align X", the assembler makes sure that the next address will be a multiple of 2^X, so in this case, a multiple of 2^2=4. The assembler will therefore insert 0 to 2^X-1 bytes of space. The section containing the directive in the object code file will also be marked to require that alignment, such that the linker will automatically place it at the appropriate location in address space.

Offset addressing

The various "ldr"/"str" instructions can optionally perform some calculation on the address before executing the memory access. What is shown for "ldr" here works for "str" and the variants for halfwords and bytes equivalently. There are several variants for this. This first one adds a fixed offset that is encoded within the instruction itself to the address:

```
ldr r0, [r1, #8]
```

This adds 8 to r1 and uses the result as the address to access. The number can also be negative. This variant is useful for accessing members of a heterogeneous container organized like a C struct or the registers in a periphery module. For example, you can load the base address of a periphery module into a register, and then access the various registers using offset-addressing without having to load each address individually:

```
GPIOA=0x40010800
GPIOx_CRH = 0x04
GPIOx_BSRR = 0x10

GPIOx_BSRR_BS8 = 0x100
GPIOx_BSRR_BR8 = 0x1000000

GPIOx_CRx_GP_PP_2MHz = 2

.text
.type Reset_Handler, %function
.global Reset_Handler
Reset_Handler:
    bl EnableClockGPIOA
```

```
    ldr r1, =GPIOA

    ldr r0, [r1, #GPIOx_CRH]
    and r0, #0xfffffff0
    orr r0, #GPIOx_CRx_GP_PP_2MHz
    str r0, [r1, #GPIOx_CRH]        @ Set CNF8:MODE8 in GPIOA_CRH to 2

    ldr r0, =GPIOx_BSRR_BS8         @ Register value to set pin to high
    str r0, [r1, #GPIOx_BSRR]       @ Set BS8 in GPIOA_BSRR to 1 to set PA8
high

    b .
```

Example name: “OffsetAddressing”

This way, you can avoid repeated loads of similar addresses. This variant is also capable of writing the newly calculated address back into the address register by appending a “!”:

```
ldr r0, [r1, #8]!
```

This will add 8 to r1, write the result into r1, and also use it as an address from which to load 4 bytes and store them into r0. The variant

```
ldr r0, [r1], #8
```

works just the opposite - r1 is used as an address from which to load the data, and “r1+8” is written back to r1. The next variant adds two registers to obtain the memory address:

```
ldr r0, [r1, r2]
```

This loads the data from the address calculated by “r1+r2”. The second register (here: r2) can also be optionally shifted left by a fixed number of bits in the range 0-3:

```
ldr r0, [r1, r2, lsl #2]
```

This shifts r2 left by two bits (i.e. multiplies it by 4), adds it to r1, and uses that as the address (r2 itself is not modified).

Iterating arrays

The offset addressing mechanism is perfectly suited to iterating arrays. This could be used to make an array defining a sequence of LED flashes that is iterated by the LED blinker application. Such an array would contain the duration of each on-and off-cycle (as passed to the “Delay” function) and be placed in flash memory:

```
.text
.type Reset_Handler, %function
.global Reset_Handler
```

```
Reset_Handler:
    bl EnableClockGPIOA
    bl ConfigurePA8
    bl Blink
    b .

.type Blink, %function
Blink:
    push { r4-r8, lr }
    ldr r4, =GPIOA_BSRR     @ Load address of GPIOA_BSRR
    ldr r5, =GPIOx_BSRR_BS8         @ Register value to set pin to high
    ldr r6, =GPIOx_BSRR_BR8     @ Register value to set pin to low
    ldr r7, =BlinkTable             @ Move address of "BlinkTable" into r7
    ldr r8, =BlinkTableEnd          @ Move address of "BlinkTableEnd" into
r8


    BlinkLoop:
        str r5, [r4]            @ Set BS8 in GPIOA_BSRR to 1 to set PA8
high

        ldr r0, [r7], #4        @ Load delay iterations from table and
increment address
        bl Delay

        str r6, [r4]            @ Set BR8 in GPIOA_BSRR to 1 to set PA8 low

        ldr r0, [r7], #4        @ Load delay iterations from table and
increment address
        bl Delay

        cmp r7, r8
        blo BlinkLoop

    pop { r4-r8, pc }

.align 2
.type BlinkTable, %object
BlinkTable:
    .word   1000000, 1000000, 1000000, 1000000, 1000000, 1000000
    .word   2500000, 1000000, 2500000, 1000000, 2500000, 1000000
    .word   1000000, 1000000, 1000000, 1000000, 1000000, 1000000
BlinkTableEnd:
```

Example name: “BlinkPattern”

The “.word” directive is used to place a sequence of 32bit- numbers into flash memory. The label “BlinkTable” will refer the the start address of the array, and “BlinkTableEnd” to the first address *after* the array. These two addresses are loaded into registers before the loop. The “.align” directive is used to make sure the 32bit-words are stored at properly aligned addresses. Inside the loop, the “ldr”

instruction is used to load a 32bit-word from the array and pass it to the “Delay” function. The r7 register is advanced by 4 bytes to the next 32bit-word. This is done twice, for the on-and off-time. At the end of the loop, the address register is compared with the address of “BlinkTableEnd” - until that address has been reached, the loop will continue.

Another possibility is to keep the base address of the array in a register, and increment another register that contains the offset:

```
.type Blink, %function
Blink:
    push { r4-r9, lr }
    ldr r4, =GPIOA_BSRR     @ Load address of GPIOA_BSRR
    ldr r5, =GPIOx_BSRR_BS8 @ Register value to set pin to high
    ldr r6, =GPIOx_BSRR_BR8 @ Register value to set pin to low
    ldr r7, =BlinkTable         @ Move address of "BlinkTable" into r7
    ldr r8, =0
    ldr r9, =18

    BlinkLoop:
        str r5, [r4]            @ Set BS8 in GPIOA_BSRR to 1 to set PA8
high

        ldr r0, [r7, r8, lsl #2]        @ Load delay iterations from table
        add r8, #1
        bl Delay

        str r6, [r4]            @ Set BR8 in GPIOA_BSRR to 1 to set PA8 low

        ldr r0, [r7, r8, lsl #2]        @ Load delay iterations from table
        add r8, #1
        bl Delay

        cmp r8, r9
        blo BlinkLoop

    pop { r4-r9, pc }
```

Example name: “BlinkPattern2”

Here, r8 is incremented in steps of 1 to denote the index in the array. The “lsl” syntax for “ldr” is used to multiply r8 by 4 (since each word is 4 bytes in size) and add it to r7, which contains the array’s base address. At the end of the loop, r8 is compared with 18, which is the number of entries in the array. This variant is actually less efficient, as it needs to keep both the base address and the index in registers and also has to increment the index in each iteration.

Literal loads

Regardless of architecture, any processor obviously needs to work with addresses in its own address space a lot. ARM can do calculations with its 32bit addresses just fine, but there is a bottleneck: The instruction set itself. To work with any address, it needs to be initially loaded into a processor register, but ARM instructions are only 16 or 32bit in size - not enough space for an arbitrary 32bit number plus the instruction encoding. Allowing even larger instructions (e.g. 40 bit) would complicate matters, so ARM instead uses several tricks to deal with this problem, which will be discussed here.

The “ldr r0, =1234” syntax allows you to load any arbitrary 32bit numbers, but is not actually a machine code instruction, but is translated by the assembler into one. In this chapter, the actual instructions for loading immediate numbers are discussed.

The “mov”-instruction

The most basic way of loading an immediate number into a register is the “mov” instruction:

```
mov r0, #1234
```

This allows you to load any 16bit number (0 to 2^16-1) into a register. “mov” also includes some clever encodings that allow you to load certain commonly-used patterns:

- Any 32bit number that consists of one byte of arbitrary bits (i.e. 8 adjacent arbitrary bits) at any location, and zeros otherwise, e.g. 0x00000045, 0x00045000, 0x7f800000.
- Any 32bit number that consists of the same byte repeated 2 or 4 times in fixed places, as in 0x23002300, 0x00230023, 0x23232323
- The bit-wise negated result of any of these two patterns, e.g. 0xffffffba, 0xfffbafff, 0x807fffff or 0xdcffdcff. The assembler will actually use the “mvn” instruction for this, which works identically to “mov”, but negates the value.

By specifying a number that falls into one of these patterns, the assembler will automatically use the appropriate encoding. The first two ways of encoding numbers are not only available with “mov”, but also several other mathematical instructions that expect some immediate value: “add”, “and”, “bic”, “cmn”, “cmp”, “eor”, “mov”, “mvn”, “orn”, “orr”, “rsb”, “sbc”, “sub”, “teq”, “tst”. In the ARM Architecture Reference Manual, check the description of the instructions and look out for “ThumbExpandImm” to see whether it supports the first two patterns above.

You can also use the “mvn” instruction directly, e.g.:

```
mov r0, #0xf807ffff
mvn r0, #0x07f80000
```

both lines are identical and write the number 0xf807ffff into r0.

The “movt” instruction

While supporting many common patterns, this does not allow arbitrary 32 bit numbers. One way to load any 32bit number is to split the number into two 16bit halves, and use both “mov” and “movt” to combine these two half-words into one register:

```
mov r0, #0xabcd
movt r0, #0x1234
```

The “movt” instruction loads the given number into the upper 16 bits of the register, so this example loads 0x1234abcd into r0. The order is important, as “mov” overwrites the upper 16 bits with zeros, but “movt” keeps the lower 16 bits. If a single “mov” can't fit the desired number, the combination of “mov” and “movt” is the fastest way of loading any 32bit number. As two 32bit instructions are needed, this consumes 8 bytes of program memory. If you want to load the address of a symbol into a register, you need to tell the assembler to split it automatically. This can be achieved by prefixing the symbol with “:lower16:” or “:upper16:”, e.g.:

```
movw r0, #:lower16:GPIOA_BSRR
movt r0, #:upper16:GPIOA_BSRR
```

Note that “movw” needs to be specified in this case to explicitly tell the assembler to use the “mov” variant that accepts 16bit numbers (which it otherwise does automatically when a direct value is given).

PC-relative loads

The other way of loading arbitrary 32bit values into registers is to place the value directly in flash memory, and load it from there using “ldr”:

```
@ Some code …
mov r0, … address of Literal ...
ldr r1, [r0]
@ More code …
Literal:
    .word 0x12345678
```

However, there is a Chicken-And-Egg problem - the address of “Literal” is a 32bit number itself, so how to load it into r0? Luckily, there is a register that contains a number close to the one needed - the program counter (PC, r15) indicates the address of the instruction currently being executed. By reading it and adding a small offset that fits into the instruction itself, the address of “Literal” can be obtained, provided that “Literal” is located close enough. Consider this example of the EnableClockGPIOA function:

```
.align 2
.type EnableClockGPIOA, %function
EnableClockGPIOA:
    add r1, pc, #12
    ldr r1, [r1]
    ldr r0, [r1]
```

```
    orr r0, r0, #RCC_APB2ENR_IOPAEN
    str r0, [r1]    @ Set IOPAEN bit in RCC_APB2ENR to 1 to enable GPIOA
    bx lr   @ Return to caller

.align 2
    .word RCC_APB2ENR
```

The 32bit-value “RCC_APB2ENR” is stored in flash memory. The “add” instruction is used to add the offset 12 to the address of the instruction itself to obtain the address of said 32bit-value, which is then loaded via “ldr”. The offset 12 is actually not easy to calculate and even depends on the alignment of the “add” instruction itself (hence the “.align” to ensure a consistent example). The assembler is capable of doing the calculation on its own, for which the “adr” instruction is used:

```
.align 2
.type EnableClockGPIOA, %function
EnableClockGPIOA:
    adr r1, LiteralRCC_APB2ENR
    ldr r1, [r1]
    ldr r0, [r1]
    orr r0, r0, #RCC_APB2ENR_IOPAEN
    str r0, [r1]    @ Set IOPAEN bit in RCC_APB2ENR to 1 to enable GPIOA
    bx lr    @ Return to caller

.align 2
LiteralRCC_APB2ENR:
    .word RCC_APB2ENR
```

The label LiteralRCC_APB2ENR refers to the address of the 32bit-value in memory. “adr” is actually a variant of “add” that instructs the assembler to calculate the offset and place it into the instruction itself, which then lets the processor add it to PC and write the result to r1. This address is then used by “ldr”.

The “adr” instruction is useful when the address of some literal is explicitly needed; for example, in the blinker program, it can be used to obtain the addresses of the array:

```
adr r7, BlinkTable              @ Move address of "BlinkTable" into r7
adr r8, BlinkTableEnd           @ Move address of "BlinkTableEnd" into r8
```

However, for loading a single value, the address is actually not needed. In this case, “adr” and “ldr” can be combined:

```
.type EnableClockGPIOA, %function
EnableClockGPIOA:
    ldr r1, LiteralRCC_APB2ENR
    ldr r0, [r1]
    orr r0, r0, #RCC_APB2ENR_IOPAEN
    str r0, [r1]    @ Set IOPAEN bit in RCC_APB2ENR to 1 to enable GPIOA
    bx lr    @ Return to caller
```

```
.align 2
LiteralRCC_APB2ENR:
    .word RCC_APB2ENR
```

This special variant of “ldr” lets the assembler calculate to offset as with “adr”, adds it to “PC” at runtime and loads the data found at the address into r1. This is much easier than the first variant, as all calculations are done automatically. It is still somewhat cumbersome having to write three lines just to obtain a single 32bit value. Therefore, the assembler offers this already introduced syntax:

```
ldr r1, =RCC_APB2ENR
```

This is a special command for the assembler. If possible, the assembler will use the “mov” or “mvn” instruction to load the value. If the value won’t fit, it will be put into flash memory, and a “ldr” instruction as above will be used. In this case, the “ldr rX, =...” syntax is equivalent to the combination of specifying a label for the value, the “.word” directive and “ldr rX, <Label>”. Therefore, this syntax is usually the best way to load immediates.

The assembler places the literals at the end of the file. If the file is long, the offset will be too long for the “ldr” and “adr” instructions and the assembler will emit an error. You can instruct the assembler to place all literals that have been declared so far at a specific point using the “.ltorg” directive. It is recommended to place an “.ltorg” after each function (after the “bx lr”) - just make sure that execution will never reach there. If a single function is so long that an “.ltorg” at the end is too far away from “ldr”/”adr” at the beginning, you can place an “.ltorg” somewhere in the middle and jump over it with “b”.

In summary, the following rules can help make literal loads more efficient

- Avoid literal loads if possible; try to calculate needed values from other values that have already been loaded, possibly by using offset-addressing in “ldr”/”str”
- When accessing multiple registers of a single periphery module, load its base address once and use offset addressing to access the individual registers
- If you need a pointer to a location in flash memory, try using “adr”
- If speed is important, use “movw”+”movt” to load the value
- Else, use “ldr rX, =...” to have the assembler choose the optimal encoding
- Place “.ltorg” after each function

The “ldr … =” instruction can also be used to load any immediate 32bit value into the PC to cause a jump to that address, simply by specifying “pc” as the target register. If you perform an ordinary branch (via “b” or “bl”) to some function whose address is too far away from the current code location, the linker will insert a “wrapper” function that does exactly that to perform the “far” jump. That function is called a “veneer”.

The SysTick timer

An important aspect of many embedded systems is to control timing of technical processes. In the blinker example, the timing of the LED flashes was handled by having the processor execute dummy instructions to pass time. It is however virtually impossible to accurately predict the runtime of any piece of code on a complex processor such as ARM ones, and the runtime may vary among multiple runs and depending on the actual micro controller and its configuration. For a simple LED blinker this may be acceptable, but not for e.g. a closed loop controller for some mechanical actor. Therefore, almost all micro controllers and also application processors feature one or more hardware timers, which allow to measure time independently of the execution speed of the software. Timer features vary widely among different processors, but that basic idea is to increment or decrement some digital counter at each clock cycle and trigger some event when it reaches a certain value.

All ARMv7-M processors feature the so-called “SysTick”-Timer as part of the processor core itself. This is a rather simple 24bit-timer that counts from a configurable value back to zero, then resets to that value and triggers an event. This timer is frequently used as a time base for RTOS or other runtime libraries. The timer uses three periphery registers: “RVR” contains the value from which to count down. “CVR” contains the current value, and “CSR” contains some status and control bits. The timer can be used for the “Delay” function like this:

```
SCS = 0xe000e000
SCS_SYST_CSR = 0x10
SCS_SYST_RVR = 0x14
SCS_SYST_CVR = 0x18

@ Parameters: r0 = Number of iterations
.type Delay, %function
Delay:
    ldr r1, =SCS
    add r0, r0, r0, lsl #1

    str r0, [r1, #SCS_SYST_RVR]
    ldr r0, =0
    str r0, [r1, #SCS_SYST_CVR]

    ldr r0, =5
    str r0, [r1, #SCS_SYST_CSR]

DelayLoop:
    ldr r0, [r1, #SCS_SYST_CSR]
    tst r0, #0x10000
    beq DelayLoop

    ldr r0, =0
    str r0, [r1, #SCS_SYST_CSR]

    bx lr
```

The SysTick is part of the “System Control Space”, SCS. The SCS base address is defined as a symbol, and the relative addresses of the registers as well. The count value is stored in “RVR”, after which “CVR” has to be set to zero. The timer is started by writing “5” into the “CSR” register. The loop repeatedly reads the “CSR” register and continues until bit 16 is set. The “tst” instruction is used to perform an “and” operation with the register contents and an immediate value without keeping the result while just updating the flags. At the end, the “CSR” register is set to zero to disable the timer. The “add” instruction at the beginning is used to multiply the count value by 3: r0 is shifted left by one, i.e. multiplied by two, and then added to itself, as in r0*2^1+r0. This is a common trick to quickly multiply by constants. By including this multiplication, the duration is the same as with the previous “Delay” variant, which, on this microcontroller, uses about 3 cycles per loop iteration.

Managing timing this way (or any other kind of “Delay” function) is still not very accurate. The time needed to call the function, start the timer, return, and set the pins is added to the actual duration and may also vary each time. The timing errors accumulate over time - a clock implemented this way will quickly go wrong. The proper way to achieve accurate timing is to start the timer once, let it run continuously, and react to its events. The internal clock source used by the microcontroller is also quite inaccurate (up to 2.5% deviation), which can be improved upon by a quartz crystal (typical accuracy of e.g. 0.005%), which will be covered later. Reacting to events instead of calling a function that executes dummy code requires restructuring the program code, without using any kind of “Delay” function.

To do that, the timer is started once at program startup and kept running. After setting the LED pin, wait for the timer event, and repeat. In the last example, the values 3000000 and 7500000 are used for the timer register (3x1000000 and 3x2500000, respectively). Changing the timer value while it is running continuously is problematic, so one fixed value should be used; to achieve variable blinker duration, multiple timer events need to be counted. The greatest common denominator of the two numbers is 1500000, so to achieve the two different times, 2 and 5 timer events need to be registered, respectively. Since these numbers fit into a single byte, the table entries and corresponding access instructions are changed to byte. A function “StartSysTick” is implemented to start the timer once, and a function “WaitSysTick” to wait for a given number of timer events:

```
.syntax unified
.cpu cortex-m3
.thumb

RCC_APB2ENR = 0x40021018
RCC_APB2ENR_IOPAEN = 4
GPIOA_CRH = 0x40010804

GPIOA_BSRR = 0x40010810
GPIOx_BSRR_BS8 = 0x100
GPIOx_BSRR_BR8 = 0x1000000


GPIOx_CRx_GP_PP_2MHz = 2
```

```
SCS = 0xe000e000
SCS_SYST_CSR = 0x10
SCS_SYST_RVR = 0x14
SCS_SYST_CVR = 0x18
TimerValue=1500000

.text
.type Reset_Handler, %function
.global Reset_Handler
Reset_Handler:
    bl EnableClockGPIOA
    bl ConfigurePA8
    ldr r0, =TimerValue
    bl StartSysTick
    bl Blink
    b .

.type Blink, %function
Blink:
    push { r4-r8, lr }
    ldr r4, =GPIOA_BSRR             @ Load address of GPIOA_BSRR
    ldr r5, =GPIOx_BSRR_BS8         @ Register value to set pin to high
    ldr r6, =GPIOx_BSRR_BR8         @ Register value to set pin to low
    adr r7, BlinkTable              @ Move address of "BlinkTable" into r8
    adr r8, BlinkTableEnd           @ Move address of "BlinkTableEnd" into
r9


    BlinkLoop:
        str r5, [r4]                @ Set BS8 in GPIOA_BSRR to 1 to set PA8
high

        ldrb r0, [r7], #1           @ Load delay iterations from table and
increment address
        bl WaitSysTick

        str r6, [r4]                @ Set BR8 in GPIOA_BSRR to 1 to set PA8
low

        ldrb r0, [r7], #1           @ Load delay iterations from table and
increment address
        bl WaitSysTick

        cmp r7, r8
        blo BlinkLoop

    pop { r4-r8, pc }

.align 2
```

```
.type BlinkTable, %object
BlinkTable:
    .byte   2, 2, 2, 2, 2, 2
    .byte   5, 2, 5, 2, 5, 2
    .byte   2, 2, 2, 2, 2, 2
BlinkTableEnd:
.align 2

.type EnableClockGPIOA, %function
EnableClockGPIOA:
    ldr r1, =RCC_APB2ENR
    ldr r0, [r1]
    orr r0, r0, #RCC_APB2ENR_IOPAEN
    str r0, [r1]                    @ Set IOPAEN bit in RCC_APB2ENR to 1 to
enable GPIOA
    bx lr   @ Return to caller

.type ConfigurePA8, %function
ConfigurePA8:
    ldr r1, =GPIOA_CRH
    ldr r0, [r1]
    and r0, #0xfffffff0
    orr r0, #GPIOx_CRx_GP_PP_2MHz
    str r0, [r1]                    @ Set CNF8:MODE8 in GPIOA_CRH to 2
    bx lr
    .ltorg

@ r0 = Count-Down value for timer
.type StartSysTick, %function
StartSysTick:
    ldr r1, =SCS

    str r0, [r1, #SCS_SYST_RVR]
    ldr r0, =0
    str r0, [r1, #SCS_SYST_CVR]

    ldr r0, =5
    str r0, [r1, #SCS_SYST_CSR]

    bx lr

@ r0 = Number of timer events to wait for
.type WaitSysTick, %function
WaitSysTick:
    ldr r1,  =SCS

WaitSysTickLoop:
    ldr r2, [r1, #SCS_SYST_CSR]
    tst r2, #0x10000
    beq WaitSysTickLoop
```

```
    subs r0, #1
    bne WaitSysTickLoop

    bx lr
```

Example name: “BlinkSysTick”

This way, the blinker frequency will be as stable and accurate as possible with the given clock source.

Exceptions & Interrupts

Exceptions and interrupts play an important role in low-level development. They provide a facility for hardware to notify the software of events, such as received data blocks or a timer event. On ARM, interrupts are a sub-group of exceptions – there are some “system-level” exceptions mostly for dealing with processor errors and providing operating system support, while interrupts are “special” exceptions for events signaled by periphery modules. When writing “regular” microcontroller software, you will mostly work with interrupts.

Exceptions (and interrupts) interrupt normal program flow, and cause the processor to execute some other piece of code which is called the exception handler or Interrupt Service Routine (ISR) (even for the “system-level” exceptions that are not interrupts). After dealing with the indicated event, the ISR typically returns and normal program flow resumes. As exceptions can interrupt the program anytime, data (and periphery) may be in any kind of inconsistent state, so special care must be taken to avoid corrupting program state in an ISR. The ARMv7-M processor (including the Cortex-M3) provide sophisticated support for exceptions, with configurable priorities and nested exception calls. This chapter will only cover the basics for using exceptions.

On ARMv7-M, exception handlers are implemented as regular functions, for example:

```
.type SysTick_Handler, %function
.global SysTick_Handler
SysTick_Handler:
    @ Handle event ...
    bx lr
```

Like any other function, it has a label, returns with “bx lr”, and is also made globally visible to other source files using “.global”. The “.type … %function” is required here for the same reason as for the already-mentioned “Reset_Handler”. Exception handlers can be located anywhere in flash memory, among the other regular functions. To tell the processor where the exception handlers for the various exception types are located, the vector table needs to be adjusted. Until now, the vector table was defined as:

```
.syntax unified
.cpu cortex-m3
```

```
.thumb

.section .VectorTable, "a"
.word _StackEnd
.word Reset_Handler
.space 0xe4
```

Recall that the first 32bit-word in flash memory contains the initial stack pointer (defined via ".word _StackEnd") and the second word contains the address of the first instruction of the program (defined via ".word Reset_Handler"). Actually, resetting the controller is an exception too, and the code to be executed after reset (or start-up) is the handler for the reset exception (hence the name "Reset_Handler"). The next 228 bytes of flash memory contain 57 32bit-addresses of the handlers of the other exceptions, including interrupts. The ".space" directive just fills those with zeroes. To tell the processor the address of an exception handler, the appropriate entry in this table needs to be set to that address. In chapter 10.1.2, table 63 of the controller's reference manual, the format of the vector table, and which exception's address should go where, is defined. Only the interrupts up until position 42 actually exist on the STM32F103RB/C8, as defined in chapter 2.3.5 of the datasheet; everything from "TIM8_BRK" is only present on larger controllers. According to the table, the SysTick exception handler's address needs to be put at location 0x3C relative to the beginning of flash memory. Since the first 8 bytes are already occupied, 0x34 bytes of space are needed after those first 8 bytes.

```
.syntax unified
.cpu cortex-m3
.thumb

.section .VectorTable, "a"
.word _StackEnd
.word Reset_Handler
.space 0x34
.word SysTick_Handler
.space 0xac
```

With this modification, the SysTick_Handler function is now declared as the handler for the SysTick exception. By default, the SysTick timer does not trigger an exception. To do that, you have to set bit 2 in the SCS_SYST_CSR register. By placing the logic for the blinker into the timer's ISR, you get an interrupt-based blinker:

```
.syntax unified
.cpu cortex-m3
.thumb

RCC_APB2ENR = 0x40021018
RCC_APB2ENR_IOPAEN = 4
GPIOA_CRH = 0x40010804

GPIOA_BSRR = 0x40010810
```

```
GPIOx_BSRR_BS8 = 0x100
GPIOx_BSRR_BR8 = 0x1000000


GPIOx_CRx_GP_PP_2MHz = 2

SCS = 0xe000e000
SCS_SYST_CSR = 0x10
SCS_SYST_RVR = 0x14
SCS_SYST_CVR = 0x18
TimerValue=1500000

.data
Variables:
BlinkStep:
    .space 1
TimerEvents:
    .space 1

.text
.type Reset_Handler, %function
.global Reset_Handler
Reset_Handler:
    ldr r0, =Variables
    ldr r1, =0
    str r1, [r0, #(BlinkStep-Variables)]
    ldr r1, BlinkTable
    str r1, [r0, #(TimerEvents-Variables)]

    bl EnableClockGPIOA
    bl ConfigurePA8

    ldr r1, =GPIOx_BSRR_BS8
    ldr r0, =GPIOA_BSRR         @ Load address of GPIOA_BSRR
    str r1, [r0]


    ldr r0, =TimerValue
    bl StartSysTick
    SleepLoop:
        wfi
        b SleepLoop


.type SysTick_Handler, %function
.global SysTick_Handler
SysTick_Handler:
    ldr r0, =SCS
    ldr r0, [r0, #SCS_SYST_CSR]
    tst r0, #0x10000
```

```
    beq Return

    ldr r0, =Variables

    ldrb r1, [r0, #(BlinkStep-Variables)]

    cmp r1, #(BlinkTableEnd-BlinkTable)
    bhs Return

    ldrb r3, [r0, #(TimerEvents-Variables)]
    subs r3, #1

    itt ne
    strbne r3, [r0, #(TimerEvents-Variables)]
    bne Return

    add r1, #1
    cmp r1, #(BlinkTableEnd-BlinkTable)
    bhs SkipRestart

    ldr r2, =BlinkTable
    ldrb r3, [r2, r1]
    strb r3, [r0, #(TimerEvents-Variables)]

SkipRestart:
    strb r1, [r0, #(BlinkStep-Variables)]

    ands r1, #1
    ite eq
    ldreq r1, =GPIOx_BSRR_BS8
    ldrne r1, =GPIOx_BSRR_BR8

    ldr r0, =GPIOA_BSRR         @ Load address of GPIOA_BSRR
    str r1, [r0]

Return:
    bx lr

.align 2
BlinkTable:
    .byte   2, 2, 2, 2, 2, 2
    .byte   5, 2, 5, 2, 5, 2
    .byte   2, 2, 2, 2, 2
BlinkTableEnd:

.align 2

.type EnableClockGPIOA, %function
EnableClockGPIOA:
    ldr r1, =RCC_APB2ENR
```

```
    ldr r0, [r1]
    orr r0, r0, #RCC_APB2ENR_IOPAEN
    str r0, [r1]            @ Set IOPAEN bit in RCC_APB2ENR to 1 to enable
GPIOA
    bx lr   @ Return to caller

.type ConfigurePA8, %function
ConfigurePA8:
    ldr r1, =GPIOA_CRH
    ldr r0, [r1]
    and r0, #0xfffffff0
    orr r0, #GPIOx_CRx_GP_PP_2MHz
    str r0, [r1]            @ Set CNF8:MODE8 in GPIOA_CRH to 2
    bx lr
    .ltorg

@ r0 = Count-Down value for timer
.type StartSysTick, %function
StartSysTick:
    ldr r1, =SCS

    str r0, [r1, #SCS_SYST_RVR]
    ldr r0, =0
    str r0, [r1, #SCS_SYST_CVR]

    ldr r0, =7
    str r0, [r1, #SCS_SYST_CSR]

    bx lr
```

Example name: “BlinkSysTickInterrupt”

The regular program flow now consists only of initializing the periphery, timer, and the first step of the blinker (setting the pin high). After that, the processor should do nothing but wait for exceptions, which is achieved by a simple endless loop. The “wfi” instruction suspends the processor; when an exception occurs, the processor will wake up, execute the ISR, and return execution after the “wfi”. Therefore, “wfi” is usually put in an endless loop as shown. This technique can reduce the processor’s power consumption significantly, as it is only running when something needs to be done, as indicated via interrupts. The ISR first checks whether the interrupt flag in the timer register is set - this is necessary, since exceptions can sometimes occur “spuriously”, i.e. without an actual event causing it. The decision whether to set or reset the pin state is taken based on the lowest bit of the table index, such that the output alternates between 1 and 0.

The code inside the ISR needs to know which step in the blinking sequence is currently active, and how many timer events have already occurred inside the current step. Therefore, two 1-byte-variables are stored in RAM. To access them, offset addressing is used, where r0 contains the base address of the variables in memory, and the offsets inside “ldrb” and “strb” are set accordingly. The last number of the blink sequence table is omitted, since it is actually superfluous, because no action

is taken after the last delay has elapsed. Because the table size is now odd, an ".align" directive after it is required. Always putting ".align" after outputting data is a good idea anyways.

Since exceptions can occur at any point in regular program flow, the processor registers may contain some data that will be used after the exception handler returns. Therefore, if the exception handler writes anything into the registers, they need to be restored when returning from the exception. Upon exception entry, the Cortex-M3/4 processors automatically store the registers r0-r3, r12, r14 (LR) and APSR (including the flags) on the stack. The link register is filled with a special "dummy" value, and when the exception handler returns via "bx lr" using this value, the processor restores said registers to their previous state. This effectively means that you can implement exception handlers like any other function, i.e. freely overwrite r0-r3, r12 and the flags and push/pop r4-r11 and the LR if needed.

Macros

The assembler provides a few mechanisms to make assembly-language development easier. One of those are macros, which allow you to define snippets of assembly code that you can then insert easily whenever you need them. While looking similar to function invocations, the code inside the macro is actually copied each time the macro is used, so don't overuse them. Macros are started with ".macro" and end at the next ".endm" directive. For example, the following macro sets the LED pin to 0 or 1:

```
.macro SETLED value
    ldr r0, =GPIOA_BSRR
    ldr r1, =(((!\value) << 24) | (\value<<8))
    str r1, [r0]
.endm


SETLED 0
SETLED 1
```

The macro name is defined as "SETLED", and a single parameter with name "value" is given. By typing "\value", the value of the parameter is substituted in the macro body. Some bit-shifting is used to calculate the right bit pattern to write into BSRR to set or reset the pin accordingly.

Weak symbols

As explained before, labels defined in assembly files get translated into symbols in the object code files, which are resolved by the linker. Sometimes it is desirable to provide a "default" or "fallback" implementation of some function (or data block) which is only used when no other implementation is given. This can be achieved by marking the "fallback" variant with ".weak":

```
.type Function1, %function
.global Function1
.weak Function1
Function1:
```

```
    @ Default implementation …
…
bl Function1    @ Call the function
```

With this code alone, “Function1” will be used normally. If you put another function with the same name in a different assembly source file, that second variant will be used.

Symbol aliases

It is also possible to define aliases for symbols using “.thumb_set”, which sets the address accordingly. For example:

```
.type Function1, %function
.global Function1
Function1:
    @ Some Code

.thumb_set Function2, Function1
…
bl Function2    @ Call the function
```

When trying to call “Function2”, the linker will automatically fill in the address of “Function1”. This can also be combined with “.weak” to define a weak alias:

```
.type Function1, %function
.global Function1
Function1:
    @ Some Code

.weak Function2
.thumb_set Function2, Function1
…
bl Function2    @ Call the function
```

If you now define another “Function2” in a different assembly source file, that will be used. If not, “Function1” will be called, which is the target of the alias definition. This is useful if you want to define one default implementation for several different functions, for each of which you need one “.weak” and one “.thumb_set” directive.

Improved vector table

The techniques from the last three sections can be used to improve the definition of the vector table. The way it was defined before is not very flexible; to insert new entries, you have to calculate the new gap sizes and offsets. First, define a default handler ISR that is called by exceptions for which no other ISR is defined, and a macro that defines an alias for one exception with the default handler as the target, and finally a table of all exceptions by using the macro:

```
.syntax unified
.cpu cortex-m3
.thumb

.macro defisr name
    .global \name
    .weak \name
    .thumb_set \name, Default_Handler
    .word \name
.endm

.global VectorTable
.section .VectorTable, "a"
.type VectorTable, %object
VectorTable:
.word _StackEnd
defisr Reset_Handler
defisr NMI_Handler
defisr HardFault_Handler
defisr MemManage_Handler
defisr BusFault_Handler
defisr UsageFault_Handler
.word 0
.word 0
.word 0
.word 0
defisr SVC_Handler
defisr DebugMon_Handler
.word 0
defisr PendSV_Handler
defisr SysTick_Handler
defisr WWDG_IRQHandler
defisr PVD_IRQHandler
defisr TAMPER_IRQHandler
defisr RTC_IRQHandler
defisr FLASH_IRQHandler
defisr RCC_IRQHandler
defisr EXTI0_IRQHandler
defisr EXTI1_IRQHandler
defisr EXTI2_IRQHandler
defisr EXTI3_IRQHandler
defisr EXTI4_IRQHandler
defisr DMA1_Channel1_IRQHandler
defisr DMA1_Channel2_IRQHandler
defisr DMA1_Channel3_IRQHandler
defisr DMA1_Channel4_IRQHandler
defisr DMA1_Channel5_IRQHandler
defisr DMA1_Channel6_IRQHandler
defisr DMA1_Channel7_IRQHandler
defisr ADC1_2_IRQHandler
```

```
defisr USB_HP_CAN1_TX_IRQHandler
defisr USB_LP_CAN1_RX0_IRQHandler
defisr CAN1_RX1_IRQHandler
defisr CAN1_SCE_IRQHandler
defisr EXTI9_5_IRQHandler
defisr TIM1_BRK_IRQHandler
defisr TIM1_UP_IRQHandler
defisr TIM1_TRG_COM_IRQHandler
defisr TIM1_CC_IRQHandler
defisr TIM2_IRQHandler
defisr TIM3_IRQHandler
defisr TIM4_IRQHandler
defisr I2C1_EV_IRQHandler
defisr I2C1_ER_IRQHandler
defisr I2C2_EV_IRQHandler
defisr I2C2_ER_IRQHandler
defisr SPI1_IRQHandler
defisr SPI2_IRQHandler
defisr USART1_IRQHandler
defisr USART2_IRQHandler
defisr USART3_IRQHandler
defisr EXTI15_10_IRQHandler
defisr RTCAlarm_IRQHandler
defisr USBWakeUp_IRQHandler

.text

.type Default_Handler, %function
.global Default_Handler
Default_Handler:
    bkpt
    b.n Default_Handler
```

There are a few empty entries in the table that are not used by the processor. At the beginning, there is still the definition for the initial stack pointer and the “Reset_Handler”. If you replace your “vectortable.S” by this code, you get a “proper” vector table. The “SysTick_Handler” will continue to work as before, and if you need to define any other ISR, for example for USART1, just define a function by the exact name “USART1_IRQHandler”. The address of this function will automatically be put in the vector table. If an exception without a corresponding ISR occurs, the “Default_Handler” will be called, which uses the “bkpt” instruction to force a breakpoint via the attached debugger. This helps debugging missed exceptions while avoiding to define several individual dummy handler functions.

.include

Having to put the register and bit definitions (“RCC_APB2ENR”, “RCC_APB2ENR_IOPAEN”, ...) in each assembly source file is redundant and error-prone. Instead, you can put them into a separate file (e.g. called “stm32f103.inc”) and use the “.include” directive to reference it:

```
.syntax unified
.cpu cortex-m3
.thumb

.include "stm32f103.inc"

@ Normal code ...
```

The assembler will read the code from the included file and pretend it was written instead of the ".include" line. This can help improve code structure. While working on the project structure, you can also restructure the definitions for the GPIO registers to facilitate offset addressing:

```
GPIOA = 0x40010800

GPIOx_CRH = 0x4
GPIOx_BSRR = 0x10
GPIOx_BSRR_BS8 = 0x100
GPIOx_BSRR_BR8 = 0x1000000
```

The next example incorporates these changes in addressing the registers.

Local Labels

Having to invent unique labels for all jump targets inside functions (e.g. for conditional code and loops) can be tedious. When using a disassembler (see below), each label will appear as its own functions. Therefore, the GNU assembler supports local labels. These are labels whose name consist only a number. Local names need not be unique; several labels called e.g. "1" may exist in one file. To perform a jump to a local label, use the number and append a "f" or "b" to indicate whether to jump forward or backward. Local labels can not be exported with the ".global" directive. The interrupt-based blinker can be modified like this using local labels:

```
.syntax unified
.cpu cortex-m3
.thumb

.include "stm32f103.inc"

TimerValue=1500000

.data
Variables:
BlinkStep:
    .space 1
TimerEvents:
    .space 1

.text
.type Reset_Handler, %function
```

```
.global Reset_Handler
Reset_Handler:
    ldr r0, =Variables
    ldr r1, =0
    str r1, [r0, #(BlinkStep-Variables)]
    ldr r1, BlinkTable
    str r1, [r0, #(TimerEvents-Variables)]

    bl EnableClockGPIOA
    bl ConfigurePA8

    ldr r1, =GPIOx_BSRR_BS8
    ldr r0, =GPIOA              @ Load address of GPIOA_BSRR
    str r1, [r0, #GPIOx_BSRR]


    ldr r0, =TimerValue
    bl StartSysTick
    1:
        wfi
        b 1b


.type SysTick_Handler, %function
.global SysTick_Handler
SysTick_Handler:
    ldr r0, =SCS
    ldr r0, [r0, #SCS_SYST_CSR]
    tst r0, #0x10000
    beq 2f

    ldr r0, =Variables

    ldrb r1, [r0, #(BlinkStep-Variables)]

    cmp r1, #(BlinkTableEnd-BlinkTable)
    bhs 2f

    ldrb r3, [r0, #(TimerEvents-Variables)]
    subs r3, #1

    itt ne
    strbne r3, [r0, #(TimerEvents-Variables)]
    bne 2f

    add r1, #1
    cmp r1, #(BlinkTableEnd-BlinkTable)
    bhs 1f

    ldr r2, =BlinkTable
```

```
    ldrb r3, [r2, r1]
    strb r3, [r0, #(TimerEvents-Variables)]

1:
    strb r1, [r0, #(BlinkStep-Variables)]

    ands r1, #1
    ite eq
    ldreq r1, =GPIOx_BSRR_BS8
    ldrne r1, =GPIOx_BSRR_BR8

    ldr r0, =GPIOA          @ Load address of GPIOA_BSRR
    str r1, [r0, #GPIOx_BSRR]

2:
    bx lr

.align 2
.type BlinkTable,%object
BlinkTable:
    .byte   2, 2, 2, 2, 2, 2
    .byte   5, 2, 5, 2, 5, 2
    .byte   2, 2, 2, 2, 2
BlinkTableEnd:

.align 2

.type EnableClockGPIOA, %function
EnableClockGPIOA:
    ldr r1, =RCC
    ldr r0, [r1, #RCC_APB2ENR]
    orr r0, r0, #(1 << RCC_APB2ENR_IOPAEN)
    str r0, [r1, #RCC_APB2ENR]          @ Set IOPAEN bit in RCC_APB2ENR to
1 to enable GPIOA
    bx lr   @ Return to caller

.type ConfigurePA8, %function
ConfigurePA8:
    ldr r1, =GPIOA
    ldr r0, [r1, #GPIOx_CRH]
    and r0, #0xfffffff0
    orr r0, #GPIOx_CRx_GP_PP_2MHz
    str r0, [r1, #GPIOx_CRH]            @ Set CNF8:MODE8 in GPIOA_CRH to 2
    bx lr
    .ltorg

@ r0 = Count-Down value for timer
.type StartSysTick, %function
StartSysTick:
    ldr r1, =SCS
```

```
    str r0, [r1, #SCS_SYST_RVR]
    ldr r0, =0
    str r0, [r1, #SCS_SYST_CVR]

    ldr r0, =7
    str r0, [r1, #SCS_SYST_CSR]

    bx lr
```

Example name: “BlinkLocalLabels”

Initializing RAM

The blinker program uses 2 byte-variables in memory, which have to be initialized to some value at startup. For large programs with many variables, this quickly becomes hard to maintain and also inefficient. Assembler and linker can help producing an “image” of how the RAM contents should look like after initializing, and place this image in flash memory alongside the normal program data. At startup, this image can be simply copied 1:1 into RAM in a loop. Most programs contain many variables that will be initialized with zero, so placing a (possibly large) block of zeroes in flash memory is wasteful; therefore, an additional loop is used to initialize all zero-variables to zero. Both techniques are also employed by C and C++ compilers, so implementing the initialization code is required there too. First, change the declaration of your variables by using “.byte”, “.hword” and “.word” and include the desired initialization value. Variables that should be initialized by zero get to be placed after a “.bss” directive to put them into the equally-named section. They don’t get an initialization value but just reserved space by using “.space”:

```
.data
TimerEvents:
    .byte 2

.bss
BlinkStep:
    .space 1
```

From the assembler’s point of view, the initialization data - in this case, just one byte of value “2” - will directly end up in RAM. However, this is not possible on microcontrollers, as the RAM always contains random data on startup and isn’t automatically initialized. To achieve that, change the linker script as follows:

```
MEMORY {
    FLASH   : ORIGIN = 0x8000000,   LENGTH = 128K
    SRAM    : ORIGIN = 0x20000000,  LENGTH =  20K
}

SECTIONS {
    .VectorTable : {
```

```
        *(.VectorTable)
    } >FLASH

    .text : {
        *(.text)
        . = ALIGN(4);
    } >FLASH

    .stack (NOLOAD) : {
        . = . + 0x400;
        _StackEnd = .;
    } >SRAM

    .data : {
        _DataStart = .;
        *(.data);
        . = ALIGN(4);
        _DataEnd = .;
    } >SRAM AT >FLASH

    _DataLoad = LOADADDR(.data);

    .bss (NOLOAD) : {
        _BssStart = .;
        *(.bss);
        . = ALIGN(4);
        _BssEnd = .;
    } >SRAM
}
```

Example name: “BlinkInitRAM”

The stack got put in its own section with the “NOLOAD” attribute, since it doesn’t need initializing. The data is now put in the “.data” section. The initial data for that section is put into flash memory via the “>SRAM AT >FLASH” construct. The addresses of symbols inside the “.data” section are still the addresses in RAM, so accesses to the symbols from assembly code still work. The symbol “_DataStart” is assigned the beginning of the initialized data in RAM, and “_DataEnd” the end. The “LOADADDR” function is used to get the beginning of the initialization data in flash, and assign it to “_DataLoad”. The “.bss” section contains all the variables that should be zero-initialized, and the symbols “_BssStart” and “_BssEnd” are set to its beginning and end address, respectively. It is marked with “NOLOAD” as well as we don't want to store (potentially many) zeroes in the linked program file, and we will explicitly initialize it (see below). As the beginning and size of the stack are already a multiple of 4, the beginning of “.data” is as well. The size of .data might not be a multiple of 4 however, so an “.=ALIGN(4)” command is inserted right before the definition of “_DataEnd”. This adds 0-3 dummy bytes by incrementing the location counter to make sure the address is a multiple of 4. The same thing is done right before “_BssEnd” and also at the end of the “.text” section, to make sure that “_BssEnd” and “_DataLoad” are multiples of 4 as well.

The only thing left is the actual initialization of the RAM. To do that, change the “Reset_Handler” as follows:

```
.text
.type Reset_Handler, %function
.global Reset_Handler
Reset_Handler:
    ldr r0, =_DataStart
    ldr r1, =_DataEnd
    ldr r2, =_DataLoad

    b 2f
1:  ldr r3, [r2], #4
    str r3, [r0], #4
2:  cmp r0, r1
    blo 1b

    ldr r0, =_BssStart
    ldr r1, =_BssEnd
    ldr r2, =0

    b 2f
1:  str r2, [r0], #4
2:  cmp r0, r1
    blo 1b

    bl EnableClockGPIOA
    bl ConfigurePA8

    ldr r1, =GPIOx_BSRR_BS8
    ldr r0, =GPIOA                  @ Load address of GPIOA_BSRR
    str r1, [r0, #GPIOx_BSRR]


    ldr r0, =TimerValue
    bl StartSysTick
    1:
        wfi
        b 1b
    .ltorg
```

The explicit initialization of the variables was removed. Instead, the addresses for “_DataStart”, “_DataEnd” and “_DataLoad” that were defined in the linker script are loaded. Then, a short loop repeatedly loads a word from flash (i.e. starting with “_DataLoad”) and stores it into RAM (starting at “_DataStart”). The address pointers are incremented by the “ldr”/”str” instructions after the access. The pointer for the RAM location is compared with the end of the RAM area (“_DataEnd”) to decide whether to jump back to the beginning of the loop. To start the loop, a jump directly to the comparison is performed; this avoids the need to do the comparison at the beginning and inside of

the loop. The second loop performs the zero-initialization of the area between "_BssStart" and "_BssEnd"; it works similarly, but does not need to load any data.

Unfortunately, the program as shown can't be translated - as the two variables now reside in two different sections (".data" and ".bss"), the offset addressing in the "SysTick_Handler" doesn't work anymore. Therefore, direct addressing has to be used:

```
.type SysTick_Handler, %function
.global SysTick_Handler
SysTick_Handler:
    ldr r0, =SCS
    ldr r0, [r0, #SCS_SYST_CSR]
    tst r0, #0x10000
    beq 2f

    ldr r0, =BlinkStep

    ldrb r1, [r0]

    cmp r1, #(BlinkTableEnd-BlinkTable)
    bhs 2f

    ldr r0, =TimerEvents
    ldrb r3, [r0]
    subs r3, #1

    itt ne
    strbne r3, [r0]
    bne 2f

    add r1, #1
    cmp r1, #(BlinkTableEnd-BlinkTable)
    bhs 1f

    ldr r2, =BlinkTable
    ldrb r3, [r2, r1]
    strb r3, [r0]

1:
    ldr r0, =BlinkStep
    strb r1, [r0]

    ands r1, #1
    ite eq
    ldreq r1, =GPIOx_BSRR_BS8
    ldrne r1, =GPIOx_BSRR_BR8

    ldr r0, =GPIOA              @ Load address of GPIOA_BSRR
    str r1, [r0, #GPIOx_BSRR]
```

```
2:
    bx lr
```

Peripheral interrupts

Interrupts, i.e. exceptions called by periphery modules, need a little extra code compared to the "core" exceptions including the SysTick. The Cortex-M's interrupt controller (the NVIC) contains several registers for configuring these interrupts. It is possible to configure the priority and manually trigger interrupts, but for most applications, the only necessary thing to do is enabling the desired interrupt. This is done via the registers "NVIC_ISER0" through "NVIC_ISER15", which are documented in the ARMv7M Architecture Reference Manual in chapter B3.4.4. Each of those registers contains 32 bits with which 32 of the interrupts can be enabled. The STM32F103RB/C8 has 43 interrupts, so only two of the possible 16 registers are present. The number of interrupts is given in chapter 2.3.5 of the controller's datasheet. So, to enable some interrupt x, the bit "x mod 32" in register NVIC_ISER*y* with y=x/32 has to be set. This register's address is 0xE000E100+y*4. Given an interrupt's number in r0, the following function does just that:

```
NVIC_ISER0 = 0xE000E100

@ r0 = IRQ Number
.type EnableIRQ, %function
EnableIRQ:
    ldr r1, =NVIC_ISER0

    movs r2, #1
    and r3, r0, #0x1F
    lsls r2, r2, r3

    lsrs r3, r0, #5
    lsls r3, r3, #2

    str r2, [r1, r3]

    bx lr
    .ltorg
```

Example name: "BlinkTIM1"

The "and" instruction calculates "x mod 32", and the following left-shift ("lsls") calculates the value where bit "x mod 32" is one, and all others are zero. To calculate the offset address "y*4", i.e. "(x/32)*4", the register is first shifted right by 5 bits and then shifted back left by 2 bits. This is the same as shifting 3 bits right and zeroing out the lower 2 bits; but two shift instructions actually consume less program memory space. Finally, the calculated value is written into the register by using offset addressing.

In addition to enabling the interrupt in the processor core's NVIC, it also has to be enabled in the periphery module. Many periphery modules support several different events, each of which has to be

enabled in the periphery's register individually. Depending on the controller, these can be mapped to one single processor interrupt (and hence, one single ISR) or multiple ones, and need to be configured in the NVIC appropriately.

This example uses the STM32's periphery timer TIM1 instead of the SysTick timer:

```
.syntax unified
.cpu cortex-m3
.thumb

.include "stm32f103.inc"

TimerValue=1500
TimerPrescaler=1000

.data
TimerEvents:
    .byte 2

.bss
BlinkStep:
    .space 1

.text
.type Reset_Handler, %function
.global Reset_Handler
Reset_Handler:
    ldr r0, =_DataStart
    ldr r1, =_DataEnd
    ldr r2, =_DataLoad

    b 2f
1:  ldr r3, [r2], #4
    str r3, [r0], #4
2:  cmp r0, r1
    blo 1b

    ldr r0, =_BssStart
    ldr r1, =_BssEnd
    ldr r2, =0

    b 2f
1:  str r2, [r0], #4
2:  cmp r0, r1
    blo 1b

    bl EnableClockGPIOA
    bl EnableClockTIM1
    bl ConfigurePA8
```

```
    ldr r1, =GPIOx_BSRR_BS8
    ldr r0, =GPIOA
    str r1, [r0, #GPIOx_BSRR]

    ldr r0, =TIM1_UP_IRQn
    bl EnableIRQ
    bl StartTIM1
    1:
        wfi
        b 1b
    .ltorg

.type TIM1_UP_IRQHandler, %function
.global TIM1_UP_IRQHandler
TIM1_UP_IRQHandler:
    ldr r0, =TIM1
    ldr r2, =(~(1 << TIMx_SR_UIF))

    ldr r1, [r0, #TIMx_SR]
    bics r1, r2
    beq 2f

    str r2, [r0, #TIMx_SR]

    ldr r0, =BlinkStep

    ldrb r1, [r0]

    cmp r1, #(BlinkTableEnd-BlinkTable)
    bhs 2f

    ldr r0, =TimerEvents
    ldrb r3, [r0]
    subs r3, #1

    itt ne
    strbne r3, [r0]
    bne 2f

    add r1, #1
    cmp r1, #(BlinkTableEnd-BlinkTable)
    bhs 1f

    ldr r2, =BlinkTable
    ldrb r3, [r2, r1]
    strb r3, [r0]

1:
    ldr r0, =BlinkStep
    strb r1, [r0]
```

```
    ands r1, #1
    ite eq
    ldreq r1, =GPIOx_BSRR_BS8
    ldrne r1, =GPIOx_BSRR_BR8

    ldr r0, =GPIOA
    str r1, [r0, #GPIOx_BSRR]

2:
    bx lr

.align 2
.type BlinkTable,%object
BlinkTable:
    .byte   2, 2, 2, 2, 2, 2
    .byte   5, 2, 5, 2, 5, 2
    .byte   2, 2, 2, 2, 2
BlinkTableEnd:

.align 2

.type EnableClockGPIOA, %function
EnableClockGPIOA:
    ldr r1, =RCC
    ldr r0, [r1, #RCC_APB2ENR]
    orr r0, r0, #(1 << RCC_APB2ENR_IOPAEN)
    str r0, [r1, #RCC_APB2ENR]              @ Set IOPAEN bit in RCC_APB2ENR
to 1 to enable GPIOA
    bx lr   @ Return to caller

.type EnableClockTIM1, %function
EnableClockTIM1:
    ldr r1, =RCC
    ldr r0, [r1, #RCC_APB2ENR]
    orr r0, r0, #(1 << RCC_APB2ENR_TIM1EN)
    str r0, [r1, #RCC_APB2ENR]              @ Set TIM1EN bit in RCC_APB2ENR
to 1 to enable TIM1
    bx lr   @ Return to caller
    .ltorg

.type ConfigurePA8, %function
ConfigurePA8:
    ldr r1, =GPIOA
    ldr r0, [r1, #GPIOx_CRH]
    and r0, #0xfffffff0
    orr r0, #GPIOx_CRx_GP_PP_2MHz
    str r0, [r1, #GPIOx_CRH]                @ Set CNF8:MODE8 in GPIOA_CRH
to 2
    bx lr
```

```
    .ltorg

@ r0 = Count-Down value for timer
.type StartTIM1, %function
StartTIM1:
    ldr r0, =TIM1
    ldr r1, =(1 << TIMx_CR1_URS)
    str r1, [r0, #TIMx_CR1]

    ldr r1, =TimerPrescaler
    str r1, [r0, #TIMx_PSC]

    ldr r1, =TimerValue
    str r1, [r0, #TIMx_ARR]

    ldr r1, =(1 << TIMx_DIER_UIE)
    str r1, [r0, #TIMx_DIER]

    ldr r1, =(1 << TIMx_EGR_UG)
    str r1, [r0, #TIMx_EGR]

    dsb

    ldr r1, =(1 << TIMx_CR1_CEN)
    str r1, [r0, #TIMx_CR1]

    bx lr
    .ltorg

@ r0 = IRQ Number
.type EnableIRQ, %function
EnableIRQ:
    ldr r1, =NVIC_ISER0

    movs r2, #1
    and r3, r0, #0x1F
    lsls r2, r2, r3

    lsrs r3, r0, #5
    lsls r3, r3, #2

    str r2, [r1, r3]

    bx lr
    .ltorg
```

The corresponding stm32f103.inc file with the added definitions for the timer registers is:

```
GPIOA = 0x40010800
```

```
GPIOx_CRH = 0x4
GPIOx_BSRR = 0x10
GPIOx_BSRR_BS8 = 0x100
GPIOx_BSRR_BR8 = 0x1000000


GPIOx_CRx_GP_PP_2MHz = 2

SCS = 0xe000e000
SCS_SYST_CSR = 0x10
SCS_SYST_RVR = 0x14
SCS_SYST_CVR = 0x18

RCC = 0x40021000
RCC_APB2ENR = 0x18
RCC_APB2ENR_IOPAEN = 2
RCC_APB2ENR_TIM1EN = 11

RCC_CR = 0x0
RCC_CR_PLLRDY = 25
RCC_CR_PLLON = 24
RCC_CR_HSERDY = 17
RCC_CR_HSEON = 16
RCC_CR_HSION = 0

RCC_CFGR = 0x04
RCC_CFGR_PLLMUL = 18
RCC_CFGR_USBPRE = 22
RCC_CFGR_PLLXTPRE = 17
RCC_CFGR_PLLSRC = 16
RCC_CFGR_PPRE2 = 11
RCC_CFGR_PPRE1 = 8
RCC_CFGR_HPRE = 4
RCC_CFGR_SWS = 2
RCC_CFGR_SW = 0

FLASH=0x40022000
FLASH_ACR=0
FLASH_ACR_PRFTBE = 4
FLASH_ACR_HLFCYA = 3
FLASH_ACR_LATENCY = 0

TIM1 = 0x40012C00

TIMx_CR1 = 0
TIMx_CR1_ARPE = 7
TIMx_CR1_URS = 2
TIMx_CR1_CEN = 0
```

```
TIMx_DIER = 0xC
TIMx_DIER_UIE = 0

TIMx_SR = 0x10
TIMx_SR_UIF = 0

TIMx_EGR = 0x14
TIMx_EGR_UG = 0

TIMx_PSC = 0x28
TIMx_ARR = 0x2C

TIM1_UP_IRQn = 25

NVIC_ISER0 = 0xE000E100
```

The source code enables the timer's clock in the RCC before configuring it. The timer supports both a freely configurable prescaler for dividing the clock and a freely configurable maximum value, both of which are set by the StartTIM1 function. The TIMx_DIER_UIE bit is set to enable the interrupt for the so-called "update event", which is triggered whenever the timer reaches the maximum value. A delicate sequence of register accesses is required to start the timer with the right configuration but without triggering the interrupt right away: To apply the modified settings immediately, the "TIMx_EGR_UG" bit is set to trigger an "artificial" update event. To prevent this from also triggering the interrupt, the "TIMx_CR1_URS" bit is set and cleared before and after, respectively. The timer is started by setting the "TIMx_CR1_CEN" bit at the end. Before that, a "dsb" instruction is inserted. This "Data Synchronization Barrier" waits until all write accesses before that have been completely processed - usually, the processors pipeline is working on several instructions at once. Because the timer configuration needs to be truly finished before starting the timer, this instruction is required. There are some other situations where the processor is too fast for the periphery and needs to be temporarily halted by a "dsb". If some periphery-accessing code works in step-by-step mode while debugging, but not when executing normally, a well-placed "dsb" might help.

The ISR "TIM1_UP_IRQHandler" is used for the timer. It checks the "TIMx_SR_UIF" bit to verify an update event has actually happened. In that case, the register is overwritten with the value 0xFFFFFFFE, i.e. all bits are written with "1" except the UIF bit. Writing ones has no effect on the bits in this register, and writing a zero clears the respective bit. Therefore, this write access clears the UIF bit but keeps the others. These interrupt flags must always be cleared as soon as possible in the ISR, or the periphery might trigger the interrupt again immediately. The rest of the ISR stays the same.

Analysis tools

When working on a low level directly with linker scripts and assembly code, it is frequently necessary to directly verify the translation output, as you can't rely on a compiler doing it right automatically, and flashing the program each time to see whether it works isn't the most efficient way. This was, in fact, important in creating the example codes for this tutorial. The "binutils" package, of which

assembler and linker are part of, offers a few tools that help with analyzing the assembler's and linker's output.

Disassembler

As the name implies, a disassembler is the opposite of an assembler - it turns binary machine code back into a (more or less) readable textual representation. If you feed an ELF file generated by the assembler or linker into the disassembler, it will read the header information to tell apart data (i.e. constants) and code, get names of symbols (and therefore, labels) and can even tell which instructions were generated from which assembly source file, if it was assembler with debug information (i.e. the "-g" flag was used). If you disassemble a binary flash image, the disassembler doesn't have all those information and will produce a much less readable output and attempt to decode data bytes as instructions.

The disassembler from binutils is called "objdump". Invoking it on the blinker looks like this:

```
$ arm-none-eabi-objdump -d -s prog1.elf

prog1.elf:      file format elf32-littlearm

Contents of section .VectorTable:
 8000000 00040020 ed000008 ed010008 ed010008  ... ............
 8000010 ed010008 ed010008 ed010008 00000000  ................
 8000020 00000000 00000000 00000000 ed010008  ................
 8000030 ed010008 00000000 ed010008 49010008  ............I...
 8000040 ed010008 ed010008 ed010008 ed010008  ................
 8000050 ed010008 ed010008 ed010008 ed010008  ................
 8000060 ed010008 ed010008 ed010008 ed010008  ................
 8000070 ed010008 ed010008 ed010008 ed010008  ................
 8000080 ed010008 ed010008 ed010008 ed010008  ................
 8000090 ed010008 ed010008 ed010008 ed010008  ................
 80000a0 ed010008 ed010008 ed010008 ed010008  ................
 80000b0 ed010008 ed010008 ed010008 ed010008  ................
 80000c0 ed010008 ed010008 ed010008 ed010008  ................
 80000d0 ed010008 ed010008 ed010008 ed010008  ................
 80000e0 ed010008 ed010008 ed010008            ............
Contents of section .text:
 80000ec 0f481049 104a03e0 52f8043b 40f8043b  .H.I.J..R..;@..;
 80000fc 8842f9d3 0d480e49 4ff00002 01e040f8  .B...H.IO.....@.
 800010c 042b8842 fbd300f0 47f800f0 4bf84ff4  .+.B....G...K.O.
 800011c 80710848 01600848 00f058f8 30bffde7  .q.H.`.H..X.0...
 800012c 00040020 04040020 f0010008 04040020  ... ... .......
 800013c 08040020 10080140 60e31600 4ff0e020  ... ...@`...O..
 800014c 006910f4 803f1dd0 1a480178 b1f1110f  .i...?...H.x....
 800015c 18d21948 0378013b 1cbf0370 12e001f1  ...H.x.;...p....
 800016c 0101b1f1 110f02d2 144a535c 03701148  .........JS\.p.H
 800017c 017011f0 01010cbf 4ff48071 4ff08071  .p......O..qO..q
 800018c 0f480160 70470202 02020202 05020502  .H.`pG..........
```

```
 800019c 05020202 02020200 0a490868 40f00400  .........I.h@...
 80001ac 08607047 08490868 20f00f00 40f00200  .`pG.I.h ...@...
 80001bc 08607047 04040020 00040020 92010008  .`pG... ... ....
 80001cc 10080140 18100240 04080140 4ff0e021  ...@...@...@O..!
 80001dc 48614ff0 00008861 4ff00700 08617047  HaO....aO....apG
 80001ec 00befde7                               ....
Contents of section .data:
 20000400 02000000                             ....
Contents of section .ARM.attributes:
 0000 41200000 00616561 62690001 16000000  A ...aeabi......
 0010 05436f72 7465782d 4d330006 0a074d09  .Cortex-M3....M.
 0020 02                                   .
Contents of section .debug_line:
 0000 98000000 02001e00 00000201 fb0e0d00  ................
 0010 01010101 00000001 00000100 70726f67  ............prog
 0020 312e5300 00000000 000502ec 00000803  1.S.............
 0030 15012121 22212f2f 21222121 30212f21  ..!!"!//!"!!0!/!
 0040 222f302f 21232130 21036120 2f2f362f  "/0/!#!0!.a //6/
 0050 030c2e32 030a2e2f 212f2222 222f2221  ...2.../!/"""/"!
 0060 21222121 222f2f22 21212321 222f212f  !"!!"//"!!#!"/!/
 0070 30212303 0d9e2121 2f212421 212f2f21  0!#...!!/!$!!//!
 0080 03422035 030c2e03 0d2e0311 2e36030b  .B 5.........6..
 0090 2e30212f 222f2202 01000101 3b000000  .0!/"/".....;...
 00a0 02002400 00000201 fb0e0d00 01010101  ..$.............
 00b0 00000001 00000100 76656374 6f727461  ........vectorta
 00c0 626c652e 53000000 00000005 02ec0100  ble.S...........
 00d0 0803d000 01210201 000101          .....!.....
Contents of section .debug_info:
 0000 22000000 02000000 00000401 00000000  "...............
 0010 ec000008 ec010008 00000000 08000000  ................
 0020 12000000 01802200 00000200 14000000  ......".........
 0030 04019c00 0000ec01 0008f001 00082100  ..............!.
 0040 00000800 00001200 00000180           ............
Contents of section .debug_abbrev:
 0000 01110010 06110112 01030e1b 0e250e13  .............%..
 0010 05000000 01110010 06110112 01030e1b  ................
 0020 0e250e13 05000000                 .%......
Contents of section .debug_aranges:
 0000 1c000000 02000000 00000400 00000000  ................
 0010 ec000008 00010000 00000000 00000000  ................
 0020 1c000000 02002600 00000400 00000000  ......&.........
 0030 ec010008 04000000 00000000 00000000  ................
Contents of section .debug_str:
 0000 70726f67 312e5300 2f746d70 2f746573  prog1.S./tmp/tes
 0010 7400474e 55204153 20322e32 392e3531  t.GNU AS 2.29.51
 0020 00766563 746f7274 61626c65 2e5300 .vectortable.S.

Disassembly of section .text:

080000ec <Reset_Handler>:
```

```
 80000ec:    480f                ldr    r0, [pc, #60]    ; (800012c
<Reset_Handler+0x40>)
 80000ee:    4910                ldr    r1, [pc, #64]    ; (8000130
<Reset_Handler+0x44>)
 80000f0:    4a10                ldr    r2, [pc, #64]    ; (8000134
<Reset_Handler+0x48>)
 80000f2:    e003                b.n    80000fc <Reset_Handler+0x10>
 80000f4:    f852 3b04     ldr.w    r3, [r2], #4
 80000f8:    f840 3b04     str.w    r3, [r0], #4
 80000fc:    4288                cmp    r0, r1
 80000fe:    d3f9                bcc.n    80000f4 <Reset_Handler+0x8>
 8000100:    480d                ldr    r0, [pc, #52]    ; (8000138
<Reset_Handler+0x4c>)
 8000102:    490e                ldr    r1, [pc, #56]    ; (800013c
<Reset_Handler+0x50>)
 8000104:    f04f 0200     mov.w    r2, #0
 8000108:    e001                b.n    800010e <Reset_Handler+0x22>
 800010a:    f840 2b04     str.w    r2, [r0], #4
 800010e:    4288                cmp    r0, r1
 8000110:    d3fb                bcc.n    800010a <Reset_Handler+0x1e>
 8000112:    f000 f847     bl    80001a4 <EnableClockGPIOA>
 8000116:    f000 f84b     bl    80001b0 <ConfigurePA8>
 800011a:    f44f 7180     mov.w    r1, #256    ; 0x100
 800011e:    4808                ldr    r0, [pc, #32]    ; (8000140
<Reset_Handler+0x54>)
 8000120:    6001                str    r1, [r0, #0]
 8000122:    4808                ldr    r0, [pc, #32]    ; (8000144
<Reset_Handler+0x58>)
 8000124:    f000 f858     bl    80001d8 <StartSysTick>
 8000128:    bf30                wfi
 800012a:    e7fd                b.n    8000128 <Reset_Handler+0x3c>
 800012c:    20000400     .word    0x20000400
 8000130:    20000404     .word    0x20000404
 8000134:    080001f0     .word    0x080001f0
 8000138:    20000404     .word    0x20000404
 800013c:    20000408     .word    0x20000408
 8000140:    40010810     .word    0x40010810
 8000144:    0016e360     .word    0x0016e360

08000148 <SysTick_Handler>:
 8000148:    f04f 20e0     mov.w    r0, #3758153728    ; 0xe000e000
 800014c:    6900                ldr    r0, [r0, #16]
 800014e:    f410 3f80     tst.w    r0, #65536    ; 0x10000
 8000152:    d01d                beq.n    8000190 <SysTick_Handler+0x48>
 8000154:    481a                ldr    r0, [pc, #104]    ; (80001c0
<ConfigurePA8+0x10>)
 8000156:    7801                ldrb    r1, [r0, #0]
 8000158:    f1b1 0f11     cmp.w    r1, #17
 800015c:    d218                bcs.n    8000190 <SysTick_Handler+0x48>
 800015e:    4819                ldr    r0, [pc, #100]    ; (80001c4
```

```
<ConfigurePA8+0x14>)
 8000160:    7803                ldrb    r3, [r0, #0]
 8000162:    3b01                subs    r3, #1
 8000164:    bf1c                itt    ne
 8000166:    7003                strbne    r3, [r0, #0]
 8000168:    e012                bne.n    8000190 <SysTick_Handler+0x48>
 800016a:    f101 0101     add.w    r1, r1, #1
 800016e:    f1b1 0f11     cmp.w    r1, #17
 8000172:    d202                bcs.n    800017a <SysTick_Handler+0x32>
 8000174:    4a14                ldr    r2, [pc, #80]    ; (80001c8
<ConfigurePA8+0x18>)
 8000176:    5c53                ldrb    r3, [r2, r1]
 8000178:    7003                strb    r3, [r0, #0]
 800017a:    4811                ldr    r0, [pc, #68]    ; (80001c0
<ConfigurePA8+0x10>)
 800017c:    7001                strb    r1, [r0, #0]
 800017e:    f011 0101     ands.w    r1, r1, #1
 8000182:    bf0c                ite    eq
 8000184:    f44f 7180     moveq.w    r1, #256    ; 0x100
 8000188:    f04f 7180     movne.w    r1, #16777216    ; 0x1000000
 800018c:    480f                ldr    r0, [pc, #60]    ; (80001cc
<ConfigurePA8+0x1c>)
 800018e:    6001                str    r1, [r0, #0]
 8000190:    4770                bx    lr

08000192 <BlinkTable>:
 8000192:    0202 0202 0202 0205 0205 0205 0202 0202    ................
 80001a2:                                               .

080001a3 <BlinkTableEnd>:
    ...

080001a4 <EnableClockGPIOA>:
 80001a4:    490a                ldr    r1, [pc, #40]    ; (80001d0
<ConfigurePA8+0x20>)
 80001a6:    6808                ldr    r0, [r1, #0]
 80001a8:    f040 0004     orr.w    r0, r0, #4
 80001ac:    6008                str    r0, [r1, #0]
 80001ae:    4770                bx    lr

080001b0 <ConfigurePA8>:
 80001b0:    4908                ldr    r1, [pc, #32]    ; (80001d4
<ConfigurePA8+0x24>)
 80001b2:    6808                ldr    r0, [r1, #0]
 80001b4:    f020 000f     bic.w    r0, r0, #15
 80001b8:    f040 0002     orr.w    r0, r0, #2
 80001bc:    6008                str    r0, [r1, #0]
 80001be:    4770                bx    lr
 80001c0:    20000404     .word    0x20000404
 80001c4:    20000400     .word    0x20000400
```

```
 80001c8:    08000192     .word    0x08000192
 80001cc:    40010810     .word    0x40010810
 80001d0:    40021018     .word    0x40021018
 80001d4:    40010804     .word    0x40010804

080001d8 <StartSysTick>:
 80001d8:    f04f 21e0     mov.w    r1, #3758153728    ; 0xe000e000
 80001dc:    6148               str    r0, [r1, #20]
 80001de:    f04f 0000     mov.w    r0, #0
 80001e2:    6188               str    r0, [r1, #24]
 80001e4:    f04f 0007     mov.w    r0, #7
 80001e8:    6108               str    r0, [r1, #16]
 80001ea:    4770               bx    lr

080001ec <Default_Handler>:
 80001ec:    be00               bkpt    0x0000
 80001ee:    e7fd               b.n    80001ec <Default_Handler>
```

This is a lot of information. The “-d” flag tells objdump to disassemble code sections, and the “-s” flag lets it output data sections. At first, it prints the contents of “.VectorTable”. Each line is prefixed with the address of where this data is found in memory. Then, the 32bit data blocks from the vector table are output. The disassembler prints the bytes in the order they appear in memory, which, since the Cortex-M3 uses little endian, is reversed - for example, the printed “ed000008” actually refers to the address “0x080000ed”, which is the address of the “Reset_Handler” with the lowest bit set to one, as it is a thumb function. Most of the addresses in the vector table reflect the address of the default handler, 0x080001ec, except for the zero-entries and the SysTick_Handler. The contents of the “.text” section is the hexadecimal representation of the machine code, and hardly readable. The “.data” section contains a single “two” - this is the “02” put into “TimerEvents”. The contents of “.ARM.attributes:” and the various “.debug” sections is not very interesting, as it does not end up on the controller, and is only read by the various analysis tools to provide nicer output.

After that comes the actual disassembly. This is a list of all the instructions in the code section. The list is grouped by the symbols found in the input file. For C Code, each symbol usually matches one function, so each block in the disassembly represents one C function. In assembly code, if you put non-local labels into a function, that function will be split into multiple blocks by the disassembler, making it harder to read - the main reason for using local labels. Each instruction is translated into one line inside the blocks. The first column is the address where that instruction is found. The next column contains the hexadecimal representation of the 2 or 4 bytes that make up the machine code of that instruction, i.e. the actual content of flash memory. After that comes a textual representation of that instruction, as inferred by the disassembler. If the instruction contains some number, the disassembler sometimes outputs a semicolon followed by some interpretation of that number. If the instruction employs PC-relative addressing, that interpretation will be the absolute address. As many instructions have multiple spellings, there can be discrepancies between the original code and the disassembly. The disassembler will also output data, such as the “BlinkTable” and the literal pools, as

such. Using the ".type" directive is helpful in that case so that the disassembler does not attempt to interpret the data bytes as code.

objdump can also be used to disassembly raw binary files that can be obtained by reading back the flash memory of some controller. To do this, use this command line:

```
$ arm-none-eabi-objdump -b binary -m arm -D prog1.bin -Mforce-thumb --
adjust-vma=0x08000000
```

The address of the binary in flash memory is specified so that the printed instruction addresses are correct. However, as the disassembler can't tell data and code apart, the result will be of limited use. If you have to analyze a binary without having an ELF file or the source code, a more sophisticated disassembler such as IDA Pro is helpful. If you have the code and only need the disassembler to identify potential problems with the project (esp. the linker script), objdump is usually sufficient.

readelf

The "readelf" program is a powerful utility that can read and output various information from ELF files. The most useful option is the "-S" flag, which lets readelf print a summary of the sections in the respective file, e.g.:

```
$ arm-none-eabi-readelf -S prog1.elf
There are 15 section headers, starting at offset 0x11268:

Section Headers:
  [Nr] Name              Type            Addr    Off     Size   ES Flg Lk
Inf Al
  [ 0]                   NULL            00000000 000000 000000 00       0
0  0
  [ 1] .VectorTable      PROGBITS        08000000 010000 0000ec 00   A  0
0  1
  [ 2] .text             PROGBITS        080000ec 0100ec 000104 00  AX  0
0  4
  [ 3] .stack            NOBITS          20000000 020000 000400 00  WA  0
0  1
  [ 4] .data             PROGBITS        20000400 010400 000004 00  WA  0
0  1
  [ 5] .bss              NOBITS          20000404 010404 000004 00  WA  0
0  1
  [ 6] .ARM.attributes   ARM_ATTRIBUTES  00000000 010404 000021 00      0
0  1
  [ 7] .debug_line       PROGBITS        00000000 010425 0000db 00       0
0  1
  [ 8] .debug_info       PROGBITS        00000000 010500 00004c 00       0
0  1
  [ 9] .debug_abbrev     PROGBITS        00000000 01054c 000028 00       0
0  1
  [10] .debug_aranges    PROGBITS        00000000 010578 000040 00       0
```

```
0  8
  [11] .debug_str        PROGBITS        00000000 0105b8 00002f 01  MS  0
0  1
  [12] .symtab           SYMTAB          00000000 0105e8 0006a0 10       13
45  4
  [13] .strtab           STRTAB          00000000 010c88 000550 00       0
0  1
  [14] .shstrtab         STRTAB          00000000 0111d8 000090 00       0
0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  y (purecode), p (processor specific)
```

For each section, one line is output. The sections ".strtab", ".shstrtab", ".symtab" and "NULL" are an integral part of ELF and always present. The ".debug" sections are present if the source was assembled with the "-g" flag. The ".ARM.attributes" section defines for which ARM processor the contained code was translated. These sections don't end up on the microcontroller. The remaining sections were defined in the linker script: ".VectorTable" contains the addresses of the exception handlers, ".text" contains the program code and constant data for flash memory, ".stack" the stack in RAM, ".data" contains variables in RAM and ".bss" contains zero-initialized variables in RAM. For these sections, the column "Type" contains either "PROGBITS" or "NOBITS" that tells you whether the section in the ELF file actually contains some data - this is only the case for ".VectorTable", ".text" and ".data". The sections ".bss" and ".stack" only reserve memory that is written at runtime, but the ELF file doesn't contain data to be written in these sections. The column "Addr" defines where this section begins in the address space. The most useful column is "Size": If you sum up the sizes of the sections ".VectorTable", ".text" and ".data", you can obtain the used flash memory. By summing ob ".data", ".stack" and ".bss", you get the used amount of RAM. Note that ".data" is counted twice, as the initialization data is stored in flash.

nm

The "nm" utility prints the symbols defined in an ELF file, for example:

```
$ arm-none-eabi-nm prog1.elf
080001ec W ADC1_2_IRQHandler
20000404 b BlinkStep
08000192 t BlinkTable
080001a3 t BlinkTableEnd
20000408 B _BssEnd
20000404 B _BssStart
…
```

This can be helpful in analyzing errors in linker scripts where symbols might get assigned wrong addresses.

addr2line

The “addr2line” utility reads the debug information from an ELF file to determine which line in which source file produced the instruction found at a particular given address. For example:

```
$ arm-none-eabi-addr2line 0x080000f0 -e prog1.elf
/tmp/test/prog1.S:24
```

Here, line 24 of “prog1.S” contains the assembler command that produced the instruction that ends up at address 0x080000f0.

objcopy

The “objcopy” utility allows you to translate program files between different formats. It is useful to convert the ELF files to both the Intel Hex format and a simple binary representation. For example,

```
arm-none-eabi-objcopy -O ihex prog1.elf prog1.hex
```

produces a “.hex” file that contains an image of the flash contents in hexadecimal form. With

```
arm-none-eabi-objcopy -O binary prog1.elf prog1.bin
```

a binary file is created which contains an exact 1:1 image of the flash contents. Some flashing tools require these formats instead of ELF, and viewing the binary file with a hex editor can be interesting as well.

Interfacing C and C++ code

Since assembly is rarely used to implement entire complex projects, but mostly for few time-critical or especially low-level routines that are part of larger code bases written in a high-level-language, interfacing C and assembly code is an important topic, which will be covered here. While it is possible to write the main project structure in assembly and integrate some C modules, it is usually done the other way round. Most of the code shown is already ready to be included in C programs. Most of this topic works the same way for C++, apart from C++ exceptions (not to be confused with ARM processor exceptions) - but these are rarely used on embedded targets anyways.

If you compile C, C++ and assembly code into individual .o object files, you can link these together using “ld” as before. However, C and C++ code usually requires access to the respective standard library, and “ld” doesn’t link these by default - therefore it is necessary to substitute “ld” for a call to “gcc” or “g++” for C or C++, respectively. This will call “ld” internally and pass the required libraries.

Environment setup for C and C++

Many C projects use a reset handler and vector table implemented in assembly, although writing them in C is possible too. As required by the C standard, C programs start with the “main()” function, so the (assembly) reset handler should setup the environment such that it is ready for C, and then call

"main". The C code might then later call some assembly functions or inline assembly. When using C++ code, or some GCC extension for C code, it is required to call some additional functions before calling "main". This is used by C++ to call the constructors of global objects. The C and C++ compilers emit a table of function pointers to functions that should be called at startup. This table has to be put into flash memory by modifying the linker script as follows:

```
.text : {
    *(.text)
    . = ALIGN(4);

    _InitArrayStart = .;
    *(SORT(.preinit_array*))
    *(SORT(.init_array*))
    _InitArrayEnd = .;
} >FLASH
```

The table of function pointers is sorted to keep the order needed by the compiler. The symbols "_InitArrayStart" and "_InitArrayEnd" mark beginning and end of that table. A reset handler that performs the memory initialization as before and calls the table of initialization functions could look like this:

```
.syntax unified
.cpu cortex-m3
.thumb

.text
.type Reset_Handler, %function
.global Reset_Handler
Reset_Handler:
    ldr r0, =_DataStart
    ldr r1, =_DataEnd
    ldr r2, =_DataLoad

    b 2f
1:  ldr r3, [r2], #4
    str r3, [r0], #4
2:  cmp r0, r1
    blo 1b

    ldr r0, =_BssStart
    ldr r1, =_BssEnd
    ldr r2, =0

    b 2f
1:  str r2, [r0], #4
2:  cmp r0, r1
    blo 1b

    ldr r4, =_InitArrayStart
```

```
    ldr r5, =_InitArrayEnd

    b 2f
1:  ldr r0, [r4], #4
    blx r0
2:  cmp r4, r5
    blo 1b

    bl main
1:  bkpt
    b 1b
    .ltorg
```

Note that for iterating the table, registers r4 and r5 are used, since the called functions may not overwrite those. The “blx” instruction is needed to perform the indirect function call. When everything is set up, the main function is called. For embedded programs, the main function should never return (i.e. contain an endless loop). If it does, that’s an error, and to make it easier to find, an endless loop with a forced breakpoint is put right after the call to “main”.

Calling functions

To call assembly functions from C code and vice-versa, the assembly functions should observe the calling convention, as mentioned before. C functions can be called just like assembly functions from assembly code, by placing the parameters in register r0-r3 and on the stack, calling the function using “bl” and retrieving the return value from r0. To call an assembly function from C code, you need to declare it in C first just like a C function. For example, to call a function that takes 2 integer arguments and returns an integer:

```
int AssemblyFunction (int a, int b);
```

If you now define a function named “AssemblyFunction” in your assembly code and export it via “.global”, you can call it from C code just like any function.

Accessing global variables

Global variables defined in C can be accessed from assembly code just like variables defined in assembly code, by using the variable’s name. To access an assembly variable from C code, you need to declare it first by specifying the type. For example, to declare an integer variable:

```
extern int AssemblyVariable;
```

If you now define a variable named “AssemblyVariable” in your assembly code and export it via “.global”, you can access it from C code just like any variable. The “extern” is required to make sure the C code doesn’t attempt to declare another variable of the same name.

Clock configuration

By default, STM32 controllers use an internal RC-oscillator with 8 MHz as a clock source for the core and periphery. This oscillator is too inaccurate for implementing a clock or using serial interfaces such as UART, USB or CAN. To obtain a more accurate clock, an external quartz crystal is usually applied. Many STM32 boards feature an 8 MHz crystal. To use it, some initialization code is required that activates the microcontroller's built-in crystal-oscillator circuit and switches the clock input to that. The STM32 controllers also include a PLL which can multiply some input clock by a configurable factor before feeding it to the processor core and peripherals. This way, a precise and fast clock can be achieved - the STM32F103 supports up to 72 MHz core frequency. Unfortunately, flash memory is not capable of keeping up with such a high frequency. Therefore, when enabling a fast clock, the flash memory needs to be configured to use wait states depending on the frequency.

The following function configures the flash wait states, enables the crystal oscillator, configures the PLL to multiply the input clock by a factor of 9, and use that as the system clock. The prescaler for the internal bus APB1 is set to 2. Assuming an 8 MHz crystal, this achieves the maximum performance possible with this microcontroller - 72 MHz for the core and APB2 domain, 36 MHz for APB1. If a different crystal is used, the PLL factors have to be adjusted.

```
RCC = 0x40021000

RCC_CR = 0x0
RCC_CR_PLLRDY = 25
RCC_CR_PLLON = 24
RCC_CR_HSERDY = 17
RCC_CR_HSEON = 16
RCC_CR_HSION = 0

RCC_CFGR = 0x04
RCC_CFGR_PLLMUL = 18
RCC_CFGR_USBPRE = 22
RCC_CFGR_PLLXTPRE = 17
RCC_CFGR_PLLSRC = 16
RCC_CFGR_PPRE2 = 11
RCC_CFGR_PPRE1 = 8
RCC_CFGR_HPRE = 4
RCC_CFGR_SWS = 2
RCC_CFGR_SW = 0

FLASH=0x40022000
FLASH_ACR=0
FLASH_ACR_PRFTBE = 4
FLASH_ACR_HLFCYA = 3
FLASH_ACR_LATENCY = 0

.type ConfigureSysClock, %function
.global ConfigureSysClock
ConfigureSysClock:
    @ Turn on HSE
    ldr r0, =RCC
```

```
    ldr r1, =((1 << RCC_CR_HSION)|(1 << RCC_CR_HSEON))
    str r1, [r0, #RCC_CR]

    @ Configure (but not start yet) PLL
    @ Mul = 9, Prediv = 1, APB1 Prescaler = 2, APB2 Prescaler = 1, AHB
Prescaler = 1
    ldr r2, =(((9-2)<<RCC_CFGR_PLLMUL)|(1 << RCC_CFGR_USBPRE)|(1 <<
RCC_CFGR_PLLSRC)|(4 << RCC_CFGR_PPRE1))
    str r2, [r0, #RCC_CFGR]

    @ Pre-Calculate value for RCC_CR
    orr r1, #(1 << RCC_CR_PLLON)


    @ Wait for HSE ready
1:  ldr r3, [r0, #RCC_CR]
    ands r3, #(1 << RCC_CR_HSERDY)
    beq 1b

    @ Turn on PLL
    str r1, [r0, #RCC_CR]

    @ Pre-Calculate value for RCC_CFGR
    orr r2, #(2 << RCC_CFGR_SW)

    @ Wait for PLL ready
1:  ldr r3, [r0, #RCC_CR]
    ands r3, #(1 << RCC_CR_PLLRDY)
    beq 1b

    @ Set flash wait states to 2
    ldr r0, =FLASH
    ldr r3, =((1<<FLASH_ACR_PRFTBE)|(2<<FLASH_ACR_LATENCY))
    str r3, [r0, #FLASH_ACR]
    ldr r0, =RCC

    @ Switch system clock to PLL
    str r2, [r0, #RCC_CFGR]

    @ Pre-Calculate value for RCC_CR
    bic r1, #(1 << RCC_CR_HSION)


    @ Wait for switch to PLL
1:  ldr r3, [r0, #RCC_CFGR]
    and r3, #(3 << RCC_CFGR_SWS)
    cmp r3, #(2 << RCC_CFGR_SWS)
    bne 1b

    @ Turn off HSI to save power
```

```
    str r1, [r0, #RCC_CR]

    bx lr
    .ltorg
```

Many projects perform the clock configuration by the reset handler before calling the main function. If you want to follow that practice, place a “bl ConfigureSysClock” as the first instruction in the “Reset_Handler” - this way, all the setup will run with the higher clock frequency, making start-up faster. This and the completed startup code from the previous chapters is implemented in the “startup.S” file in the example repository. If you use it, put your code in the “main” function, where RAM and system clock will already be initialized. This is shown in the “BlinkStartup” example.

Project template & makefile

To quickly start your own project, a project template is supplied in the examples repository under the directory ProjectTemplate-STM32F103RB. Put your own application code in the program.S file. The startup.S and vectortable.S contain the reset handler with RAM initialization and the vector table with default handler, respectively. A linker script is included too.

The project also contains a makefile. This allows you to quickly translate your project without having to type the assembler and linker commands. Simply type

```
make
```

To translate the code and produce program.elf, program.bin and program.hex files. All “.S” files in the directory will be automatically translated. Writing makefiles is a complex topic on its own with a lot of information already available on the web, so no further explanations on that will be made here.

Kategorien:

- Seiten mit dem veralteten source-Tag
- Seiten mit Syntaxhervorhebungsfehlern
- ARM
- STM32
- Entwicklungstools
- Programmiersprachen