Univerza *v Ljubljani*
Fakulteta *za elektrotehniko*

REAL-TIME OPERATING SYSTEMS:
LABORATORY EXERCISES

Založba
FE

JANEZ PUHAN

University of Ljubljana
Faculty of electrical engineering

Real-time operating systems: Laboratory exercises

Janez Puhan

Ljubljana, 2019

Kataložni zapis o publikaciji (CIP) pripravili v Narodni in univerzitetni knjižnici v Ljubljani

COBISS.SI-ID=299480576

ISBN 978-961-243-381-9 (pdf)

URL: http://fides.fe.uni-lj.si/~janezp/real-time_operating_systems_laboratory_exercises.pdf

Založnik: Založba FE, Ljubljana
Izdajatelj: Fakuleta za elektrotehniko, Ljubljana
Urednik: prof. dr. Sašo Tomažič
1. elektronska izdaja

Contents

Preface

The laboratory exercises described in this script are part of the Real-time operating systems course. The course is held in the third semester of the 2nd Cycle Postgraduate Study Programme in Electrical Engineering, study programme option Electronics, at the Faculty of electrical engineering of the University of Ljubljana, Slovenia. The students are introduced into RTOS[1] concepts through nine laboratory exercises. A µC[2] system with ARM[3] Cortex based processor core is used. The Arduino Due board with Olimex ARM-USB-OCD-H[5] JTAG[6] interface serves as a hardware platform. The open source FreeRTOS™ software is used as an operating system platform. The Eclipse IDE[7] for C/C++ Developers is used as a graphical interface to the GNU[8] tools (i.e., compiler, linker, debugger, etc.). The environment is installed on a PC[9] with installed Linux operating system. A solid knowledge of the C programming language is a required prerequisite.

[1] RTOS ... Real-Time Operating System
[2] µC ... Micro-Controller
[3] ARM ... Advanced RISC[4] Machines
[4] RISC ... Reduced Instruction Set Computer
[5] ARM-USB-OCD-H... ARM - USB[6] - On-Chip Debugger - High speed
[6] USB... Universal Serial Bus
[7] JTAG ... Joint Test Action Group
[8] IDE ... Integrated Development Environment
[9] GNU ... GNU's Not Unix
[10] PC ... Personal computer
FreeRTOS is a trademark of Real Time Engineers Ltd.

Exercise 1

Installing IDE

Prepare a working environment to program the Arduino Due board through the Olimex ARM-USB-OCD-H interface on a Linux pre-installed PC[1]. Use Eclipse IDE for C/C++ Developers as a graphical interface, GCC[2] as ARM cross-compiler, and OpenOCD for communication with the ARM-USB-OCD-H interface. Find the required software on the Internet. Create and cross-compile a template project with an empty `main()` function. Use ASF[3] code for µC initialization from reset to the start of the `main()` function. Upload the cross-compiled project to the Arduino Due board.

Explanation

Installing the environment

The Eclipse is a platform consisting of several components used to develop applications in various programming languages. Since our code will be in the C programming language the component Eclipse IDE for C/C++ Developers needs to be installed. The Eclipse IDE for C/C++ Developers tarball can be downloaded from the Eclipse website [1]. Open a terminal window and extract the files in tarball. To open the terminal window, go to the *Applications* in the menu bar, select *Accessories*, and run the *Terminal* program. A terminal window with the user home directory as the current working directory is opened. To extract the tarball (i.e., the `eclipse-cpp-kepler-SR2-linux-gtk-x86_64.tar.gz` file), use `tar` command.

```
user@host:~$
            tar xvfz eclipse-cpp-kepler-SR2-linux-gtk-x86_64.tar.gz
```

The Eclipse requires a Java VM[4] to run. Installing the Java SE[5] Development Kit solves that. The JDK[6] tarball can be downloaded from the Oracle website [2]. Change into the `eclipse` directory created at the previous tarball extraction and extract the JDK tarball there.

```
user@host:~$ cd eclipse
user@host:~/eclipse$ tar xvfz jdk-8u45-linux-x64.tar.gz
```

[1] The Wheezy release of the Debian Linux distribution
[2] GCC ... GNU Compiler Collection
[3] ASF ... Atmel Software Framework
[4] VM ... Virtual Machine
[5] SE ... Standard Edition
[6] JDK ... Java Development Kit

To ensure that the Eclipse will run on the installed JVM[7], it has to be specified in `eclipse.ini` file. The file can be edited with an arbitrary text editor (i.e., `vi`, `nano`, `gedit`, etc.). Add the following lines before the VM arguments line (i.e., before the `-vmargs` line) in `eclipse.ini`.

```
-vm
/home/user/eclipse/jdk1.8.0_45/bin/java
```

The Eclipse will serve as a graphical interface to the GNU tools (i.e., compiler, linker, debugger, etc.). The GNU tools (i.e., GCC) for ARM embedded processors will be used. GCC [3] is a collection of compilers supporting various programming languages and targeting various platforms (i.e., µCs or µPs[8]). In our case, the GCC ARM cross-compiler is required. The source code will be cross-compiled on a PC building an executable for an ARM Cortex processor. The GCC for ARM embedded processors tarball can be downloaded from the GCC ARM Embedded project on the Launchpad website [4]. Extract the tarball (i.e., the `gcc-arm-none-eabi-4_9-2015q1-20150306-linux.tar.bz2` file) into the `eclipse` directory.

```
user@host:~/eclipse$
       tar xvjf gcc-arm-none-eabi-4_9-2015q1-20150306-linux.tar.bz2
```

To communicate with the Olimex ARM-USB-OCD-H interface [5], the OCD software is required. The OCD (i.e., OpenOCD) provides programming and debugging of the target embedded system (i.e., µC on the Arduino Due board). To do so, a debug interface (i.e., the Olimex ARM-USB-OCD-H) is needed to produce the required electric signals (i.e., JTAG). In our case, the Eclipse will communicate with the ARM-USB-OCD-H interface. Thus the GNU ARM Eclipse OpenOCD distribution of the OpenOCD project [6] will be installed. The tarball can be downloaded from the GNU ARM Eclipse Plug-ins project on the Sourceforge website [7]. Extract the tarball (i.e., the `gnuarmeclipse-openocd-debian64-0.8.0-201503201909.tgz` file) into the `eclipse` directory.

```
user@host:~/eclipse$
     tar xvfz gnuarmeclipse-openocd-debian64-0.8.0-201503201909.tgz
```

The OpenOCD software needs to be properly configured to use the selected debug interface (i.e., the Olimex ARM-USB-OCD-H) talking to the selected target embedded system (i.e., the Atmel AT91SAM3X8E µC [8] on the Arduino Due board [9]). Create the following `openocd.cfg` configuration file in `openocd/0.8.0-201503201909/scripts` subdirectory of the `eclipse` directory.

```
source [find interface/ftdi/olimex-arm-usb-ocd-h.cfg]
source [find target/at91sam3ax_8x.cfg]
$_TARGETNAME configure -event gdb-attach {
    echo "Halting target"
    halt
}
```

The Olimex ARM-USB-OCD-H interface and the AT91SAM3X8E Arduino Due µC are specified in `openocd.cfg`. Also halting of the target processor is performed

[7] JVM ... Java Virtual Machine
[8] µP ... MicroProcessor

on the GDB[9] attach event[10]. Use `chmod` command to set `openocd.cfg` permissions to read/write for the owner and read for everyone else.

```
user@host:~/eclipse$
            chmod 644 openocd/0.8.0-201503201909/scripts/openocd.cfg
```

OpenOCD software needs the `lib32ncurses5` package to be installed. Also the `libcanberra-gtk-module` is required by Eclipse. To install both packages, `root` user permissions are required.

```
root@host:~# apt-get update
root@host:~# apt-get install lib32ncurses5
root@host:~# apt-get install libcanberra-gtk-module
```

The Olimex ARM-USB-OCD-H interface is identified by the *udev* daemon when plugged in. The *udev* identifies a new device and creates its name according to the rules in `/etc/udev/rules.d` directory. The `99-openocd.rules` file contains rules for various interfaces (including the ARM-USB-OCD-H) the OpenOCD can work with. It has to be copied into the `/etc/udev/rules.d` directory. The rules has to be reloaded to take effect. The `root` user permissions are required.

```
root@host:~# cp /home/user/eclipse/openocd/0.8.0-201503201909/con
                              trib/99-openocd.rules /etc/udev/rules.d
root@host:~# udevadm control --reload-rules
```

To use the Olimex ARM-USB-OCD-H interface, the user has to be a member of the `plugdev` group.

```
root@host:~# usermod -a -G plugdev user
```

It is time to run the freshly installed Eclipse for the first time.

```
user@host:~/eclipse$ ./eclipse
```

To make the Eclipse environment work with the Olimex ARM-USB-OCD-H OpenOCD interface and AT91SAM3X8E µC, the Eclipse extensions for GNU tools for ARM embedded processors have to be installed. These extensions are provided by the GNU ARM plug-ins. Since debugging sessions are powered by the GDB, the C/C++ GDB Hardware Debugging plug-in is a prerequisite. It is a part of the CDT[11] plug-ins. The CDT zip file (i.e., the `cdt-master-8.3.0.zip` file) can be downloaded from the Eclipse website [1]. To install the C/C++ GDB Hardware Debugging plug-in into the Eclipse, select the *Install New Software...* menu item from the *Help* menu in menu bar. The *Install* dialog box opens. Press the *Add...* button to add a new software repository. In *Add Repository* dialog box shown in Fig. 1.1 specify the CDT repository, i.e., *Name:* CDT, *Location:* absolute path to the `cdt-master-8.3.0.zip` file.

After the repository is specified, the *Install* dialog box regains the focus. Select the C/C++ GDB Hardware Debugging plug-in from the CDT Optional Features list as shown in Fig. 1.2. Press the *Next* > button and follow the installation procedure.

[9] GDB ... GNU debugger
[10] Occurs when the GDB connects to the target (i.e., at the beginning of the debug session).
[11] CDT ... C/C++ Development Tooling

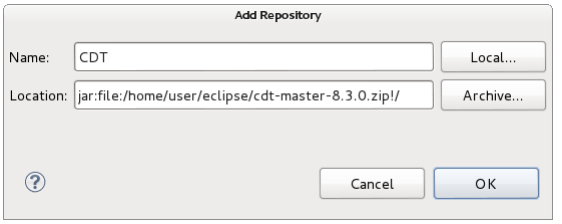


Figure 1.1: The *Add Repository* dialog box

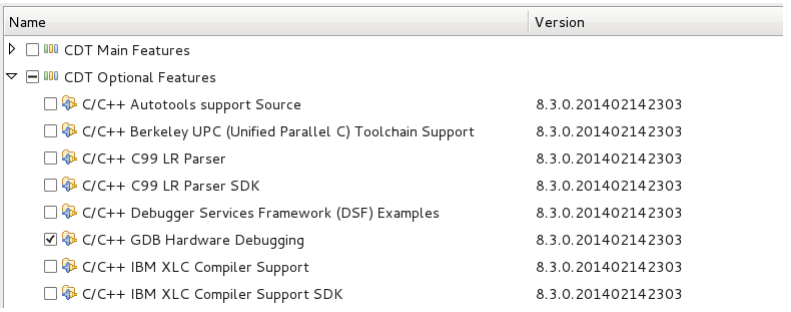


Figure 1.2: Select C/C++ GDB Hardware Debugging plug-in in the *Install* dialog box

Install the GNU ARM plug-ins in the same manner. The zip file (i.e., the `ilg.gnuarmeclipse.repository-2.8.1-201504061754.zip` file) can be downloaded from the GNU ARM Eclipse Plug-ins project on the Sourceforge website [7]. This time specify the repository as *Name:* GNU ARM Eclipse Plug-ins, and *Location:* absolute path to the `ilg.gnuarmeclipse.repository-2.8.1-20150406175 4.zip` file. In the *Install* dialog box select the entire package of the GNU ARM C/C++ Cross Development Tools plug-ins.

Finally, a path to the OpenOCD binary directory has to be configured in the Eclipse environment. To do so, select the *Preferences* menu item from the *Window* menu in menu bar. The *Preferences* dialog box opens. Select the *String Substitution* item from *Run/Debug* as shown in Fig. 1.3. Select the *openocd_path* variable and press the *Edit...* button. In the *Edit Variable: openocd_path* dialog box specify the absolute path to the OpenOCD binary directory, i.e., absolute path to the `openocd/0.8.0-201503201909/bin` directory.

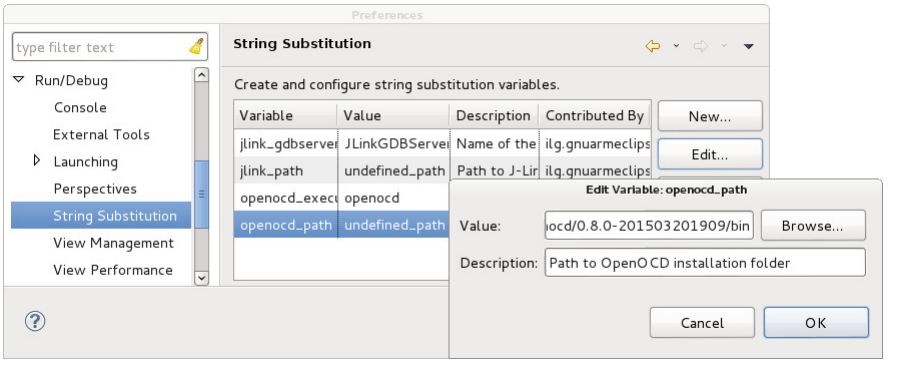


Figure 1.3: Specifying absolute path to the OpenOCD binary directory

At this point, a working environment consisting of the Eclipse with the required

plug-ins, the GNU tools and the OpenOCD software is installed as shown in Fig. 1.4.

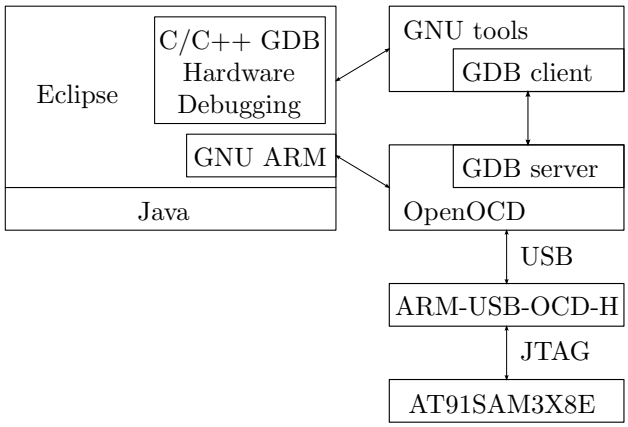


Figure 1.4: Working environment

A few handy settings of the Eclipse environment follow to ease the usage of the created working environment. The settings are optional.

Window | *Preferences* → *Preferences* dialog box → *General* | *Editors* | *Text Editors* → enable *Show print margin* and *Show line numbers* → press the *Apply* button

Window | *Preferences* → *Preferences* dialog box → *General* | *Workspace* → disable *Build automatically* and enable *Save automatically before build* → press the *Apply* button

Window | *Preferences* → *Preferences* dialog box → *C/C++* | *Build* | *Console* → enable *Bring console to top when building (if present)* and *Wrap lines on the console*, set *Limit console output (number of lines)* to 5000 → press the *Apply* button

Window | *Preferences* → *Preferences* dialog box → *C/C++* | *Code Analysis* → disable all problems → press the *Apply* button

Window | *Preferences* → *Preferences* dialog box → *C/C++* | *Code Style* | *Formatter* → set *Active profile* to *GNU [built-in]* → press the *Apply* button

Window | *Preferences* → *Preferences* dialog box → *C/C++* | *Editor* → in the *Documentation tool comments* section, set *Workspace default* to *Doxygen* → press the *Apply* button

Window | *Preferences* → *Preferences* dialog box → *C/C++* | *Editor* | *Folding* → in the *Initially fold this region types* section, disable *Header Comments* → press the *Apply* button

Window | *Preferences* → *Preferences* dialog box → *C/C++* | *Indexer* → in the *Build configuration for the indexer* section, select *Use active build configuration* → press the *Apply* button

Window | *Preferences* → *Preferences* dialog box → *Run/Debug* | *Launching* → in the *Launch Operation* section, enable *Always launch the previously launched application* → press the *Apply* button

Selecting an appropriate µC boot mode

The Atmel AT91SAM3X8E µC on the Arduino Due board has three non-volatile memory blocks that can retain their contents when not powered. Those are ROM[12] (16kB) starting at the 0x00000000 address, the first Flash memory bank (256kB) starting at 0x00080000, and the second Flash memory bank (256kB) starting at 0x000c0000. The reset vector[13] of the µC can reside in any of them. The location of the reset vector is selected by the GPNVM[14] bits (see Tab. 1.1) [8].

GPNVM bit	if bit value = 0	if bit value = 1
0 (security bit)	Flash access enabled	Flash access disabled
1 (boot mode selection)	reset vector in ROM	reset vector in Flash
2 (flash selection)*	reset vector in Flash0	reset vector in Flash1

*used only when GPNVM bit1 = 1

Table 1.1: GPNVM bits

Any kind of outside access to Flash is disabled when the GPNVM bit0 is set. Therefore, the code in the Flash is protected and cannot be read by the third party. The protected code can only be deleted by tying the *Erase* pin to high voltage level for at least 220ms (i.e., pressing the *ERASE* button on the Arduino Due board for 220ms [9]). The GPNVM bits are also erased by this procedure. Thus, the access to fresh empty Flash is enabled. Of course, the GPNVM bit0 must not be set during the code development.

The GPNVM bit1 selects the location of the reset vector. When ROM is selected, the SAM-BA[15] program hard-coded there is started. It programs[16] the on-chip Flash memory via the UART[17] or USB. On the other hand, when the Flash is selected, the reset vector is read from the first or the second Flash bank regarding the GPNVM bit2. Since the GDB will be used for uploading the code into the on-chip Flash, the GPNVM bit1 must be set. SAM-BA will not be used.

The code can be compiled for either the first, or the second Flash bank. The bank is selected in the linker script provided by the ASF. Since the ASF uses the first Flash bank (i.e., Flash0), the GPNVM bit2 must not be set.

The default value of the GPNVM bits is zero (i.e., when the *ERASE* button is pressed). To get the desired values (i.e., GPNVM bits = 0b010), the GPNVM bits have to be set with the OpenOCD. Plug in the Olimex USB-ARM-OCD-H debug interface with the Arduino Due board connected over the JTAG. Open two terminal windows (*Applications* → *Accessories* → *Terminal*). In the first terminal start the OpenOCD debugger.

```
user@host:~/eclipse/openocd/0.8.0-201503201909/bin$ ./openocd
```

[12] ROM ... Read-Only Memory
[13] Reset vector is loaded into the program counter register at power-up. It defines the µC starting address.
[14] GPNVM ... General Purpose Non-Volatile Memory
[15] SAM-BA ... Smart ARM MCU[18] - Boot Assistant
[16] SAM-BA starts the FFPI[19] to program the on-chip Flash.
[17] UART ... Universal Asynchronous Receiver/Transmitter
[18] MCU ... µC Unit
[19] FFPI ... Fast Flash Programming Interface

```
GNU ARM Eclipse 64-bit Open On-Chip Debugger 0.8.0-00063-gbda7f5c
                                                    (2015-01-01-00:00)
Licensed under GNU GPL v2
For bug reports, read
http://openocd.sourceforge.net/doc/doxygen/bugs.html
Info : only one transport option; autoselect 'jtag'
adapter speed: 500 kHz
adapter_nsrst_delay: 100
jtag_ntrst_delay: 100
cortex_m reset_config sysresetreq
adapter speed: 500 kHz
Info : clock speed 500 kHz
Info : JTAG tap: sam3.cpu tap/device found: 0x4ba00477
                                (mfg: 0x23b, part: 0xba00, ver: 0x4)
Info : sam3.cpu: hardware has 6 breakpoints, 4 watchpoints
```

Connect to the OpenOCD debugger via telnet in the second terminal. Use localhost 4444 port. The GPNVM bits can be set and viewed with the `at91sam3` OpenOCD command [6].

```
user@host:~$ telnet localhost 4444
Trying ::1...
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Open On-Chip Debugger
> reset init
JTAG tap: sam3.cpu tap/device found: 0x4ba00477
                                (mfg: 0x23b, part: 0xba00, ver: 0x4)
target state: halted
target halted due to debug-request, current mode: Thread
xPSR: 0x01000000 pc: 0x0010004c msp: 0x20001000
> at91sam3 gpnvm clr 0
> at91sam3 gpnvm set 1
> at91sam3 gpnvm clr 2
> at91sam3 gpnvm
sam3-gpnvm0: 0
sam3-gpnvm1: 1
sam3-gpnvm2: 0
> exit
Connection closed by foreign host.
```

Creating a template project

To create an empty template project in the Eclipse environment, select the *Project...* submenu item from the *File* | *New* menu. In the *New Project* dialog box shown in Fig. 1.5 select the *C/C++* | *C Project*. In the next, *C Project* dialog box shown in Fig. 1.6 set the *Project name* and select *Makefile project* | *Empty Project* for the *Project type*, and *Cross ARM GCC* for the *Toolchains*.

An empty makefile project is created. A path to the GNU tools directory (i.e., `/home/user/eclipse/gcc-arm-none-eabi-4_9-2015q1/bin`) has to be set.

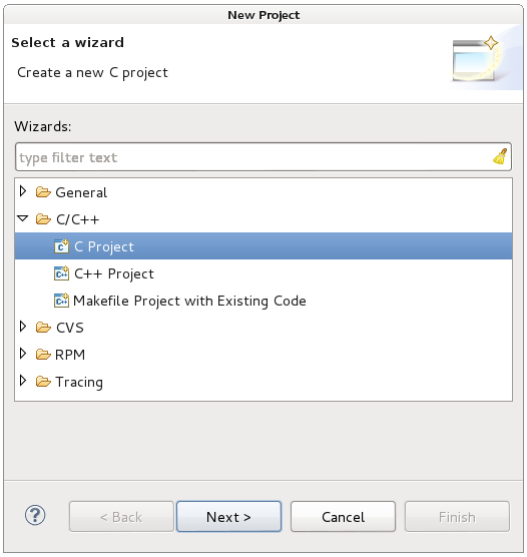


Figure 1.5: The *New Project* dialog box

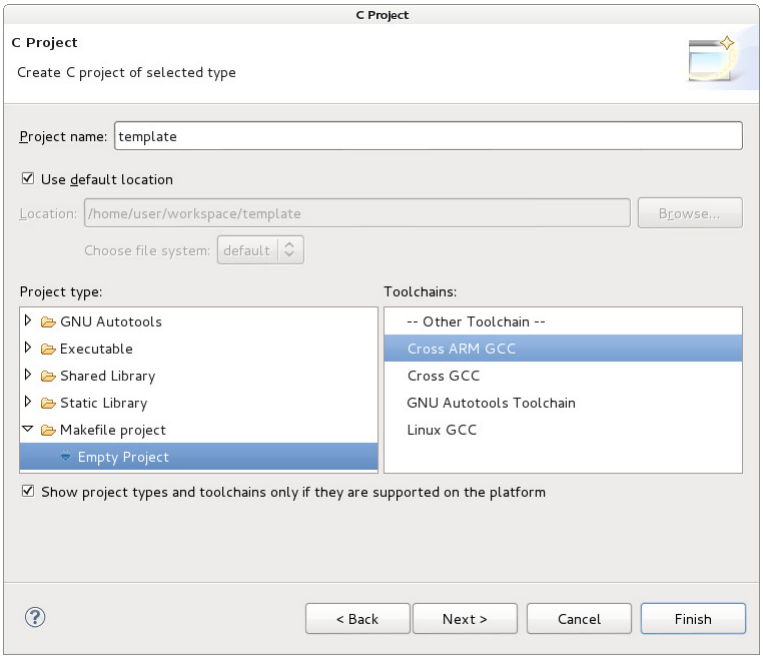


Figure 1.6: The *C Project* dialog box

Highlight the project in the *Project Explorer* view of the *C/C++* perspective[20]. Select the *Properties* menu item from the *Project* menu. In the *Properties for <projectname>*[21] dialog box set the following:

C/C++ Build | *Settings* → *Toolchains* tab → press the *Apply* button[22]

C/C++ Build | *Environment* → press the *Add...* button → *New variable* dialog box → set variable *Name* to `PATH` and *Value* to `/home/user/eclipse/gcc-arm-none-eabi-4_9-2015q1/bin` → press the *OK* button → back in the *Properties for <projectname>* dialog box press the *Apply* button (see Fig. 1.7)

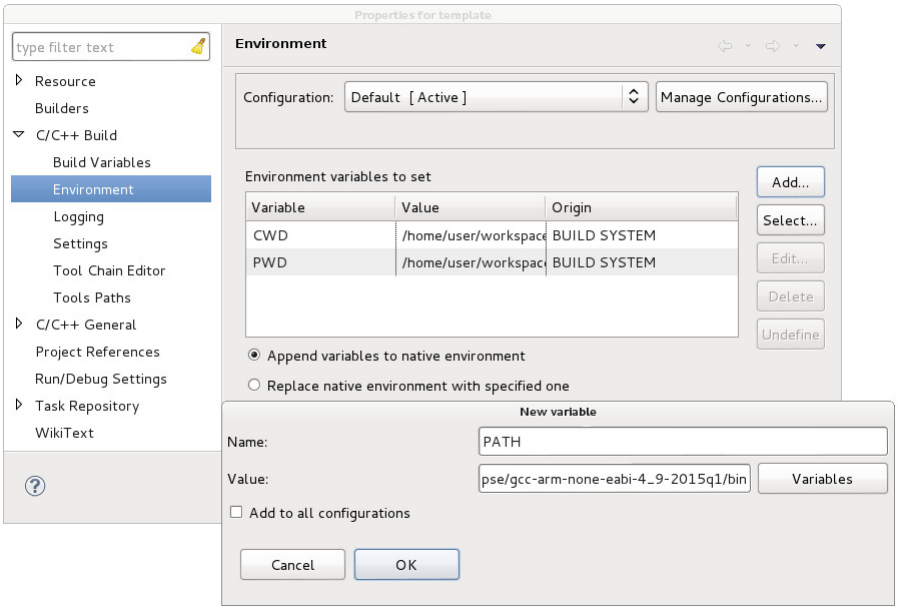


Figure 1.7: Path to the GNU tools directory

Atmel provides the ASF software library for its µCs. It contains source code for µC initialization, APIs[23] to peripheral units, etc. For Cortex based processors, the CMSIS[24] provided by ARM [10] is included. The ASF software library can be downloaded from the Atmel website [11]. It comes as a standalone archive file (i.e., as a `asf-standalone-archive-3.21.0.6.zip` file). Makefiles and linker scripts are added, so the code can be compiled and linked using the GCC [3] and GNU Make utility [12]. The AT91SAM3X8E µC source code files accompanied by makefiles and linker script (all extracted from the ASF library) can be downloaded from [13]. Note that only the files needed in this laboratory exercises are included.

There is a branched directory structure with a lot of various files in [13], which can be a bit confusing. Thus, the three key files are pointed out here:

- The `sam/utils/cmsis/sam3x/source/templates/gcc/startup_sam3x.c` file contains the exception table. The second entry in the table is the reset

[20] For explanation of the Eclipse environment views and perspectives, consult the documentation pages on the Eclipse website [1].
[21] The name `template` is used as *<projectname>* in figures.
[22] The toolchain settings have to be applied although no changes are made. Otherwise the build command (i.e., `make`) is not set. This is a bug.
[23] API ... Application Programming Interface
[24] CMSIS ... Cortex Microcontroller Software Interface Standard

vector loaded into the program counter register at power-up. The reset vector refers to the `Reset_Handler()` function also defined in this file. Thus, the µC starts in `Reset_Handler()` which after some basic initialization[25] calls the `main()` function. The `main()` function is considered as the beginning of the program in the C programming language.

- The `config.mk` contains the build settings used by the GNU Make utility. The compiler and linker flags, linker script filename, output (`elf`) filename, list of C and assembly source files, include paths, library paths, etc., are defined here.

- The `sam/utils/linker_scripts/sam3x/sam3x8/gcc/flash.ld` file is the linker script. Among others, the address of the selected Flash memory bank is defined here (see page 6).

Extract and copy the files from [13] into the project directory, e.g., `/home/user/workspace/<projectname>`.

To run and debug the freshly created template project, a debug configuration has to be defined. Highlight the project in the *Project Explorer* view of the *C/C++* perspective. Select the *Debug Configurations...* menu item from the *Run* menu. In the *Debug Configurations* dialog box select the *GDB OpenOCD Debugging* item and click the *New* button. Select the newly created *<projectname> Default* debug configuration under the *GDB OpenOCD Debugging* item. Define the configuration settings in tabs on the right side of the *Debug Configurations* dialog box as shown in Fig. 1.8.

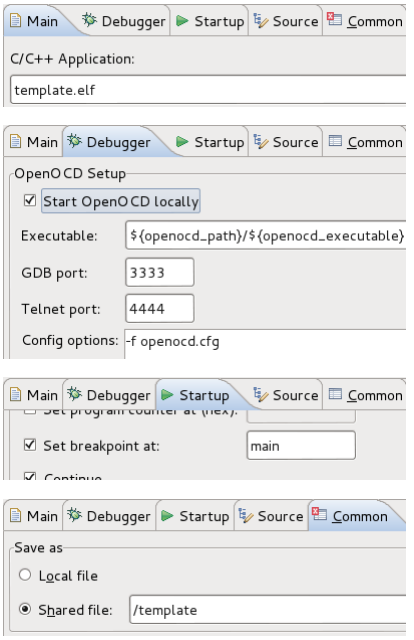


Figure 1.8: Debug configuration settings

[25] Copy the `relocate` segments to RAM[26], clear the `bss` segment, set the exception table address (e.g., to 0x0008000 = the first Flash memory bank), all according to the linker script, and initialize the `libc` standard C library. Note that the first entry in the exception table is the initial stack top address loaded into the `r13` SP[27] register at power-up.

[26] RAM ... Random Access Memory

[27] SP ... Stack Pointer

In the *Main* tab, define the *C/C++ Application* executable file as it is defined in the `config.mk` file. The `TARGET_FLASH = ....elf` line defines the name of the `elf` file.
In the *Debugger* tab, define the OpenOCD *Config options*. The options reside in the `openocd.cfg` file (see page 2) which has to be passed as an argument to the OpenOCD executable.
In the *Startup* tab, the debug starting-point can be defined with *Set breakpoint at* option. After the upload, the program on the target µC board starts with execution. It stops at the first breakpoint set to the `main()` function by default. By changing the *Set breakpoint at* option, the initial breakpoint can be placed elsewhere (e.g., to the `Reset_Handler()` function right "after" the reset vector and even before the basic initializations).
In the *Common* tab, the directory containing the `*.launch` file, where setting are saved, is specified.

There is an inconsistency in the standard `cdefs.h`[28] and the ASF `compiler.h`[29] header file. Both define the `__always_inline`[30] macro. Therefore one of the definitions is redundant. Since the definitions are not exactly identic, the ASF definifion is used and the definition in the `cdefs.h` header file is commented out. With this minor hack, the project can be compiled by selecting the *Build Project* menu item from the *Project* menu.

To upload the compiled `elf` file to the target µC board (i.e., the Arduino Due board) and start a debug session, select the *Debug Configurations...* menu item from the *Run* menu. Select the project under the *GDB OpenOCD Debugging* item and press the *Debug* button.

Enabling serial communication over the USB

When the Programming USB port on the Arduino Due board is connected to a Linux PC, it is identified as a new serial device (e.g., `/dev/ttyACM0`). The user will be able to access such a device if it is a member of the `dialout` group. The super user can add the user into the group with the following command:

```
root@host:~# usermod -a -G dialout user
```

The change takes effect at the next login.

An arbitrary serial terminal program is also required for serial communication. The PuTTY serial console can be used. It can be installed to a Linux PC with the commands:

```
root@host:~# apt-get update
root@host:~# apt-get install putty
```

carried out as the super user. To start PuTTY, select the *PuTTY SSH Client* submenu item from the *Applications* | *Internet* menu. Note that the serial terminal settings must match the UART configuration (see Appendix A). An example of the PuTTY serial settings is shown in Fig. 1.9.

[28] Located in the `include/sys` subdirectory, e.g., `/home/user/eclipse/gcc-arm-none-eabi-4_9-2015q1/arm-none-eabi/include/sys/cdefs.h`.
[29] Located in the `sam/utils` subdirectory, e.g., `/home/user/workspace/<projectname>/sam/utils/compiler.h`.
[30] In line 359 of `cdefs.h` and in line 162 of `compiler.h`.

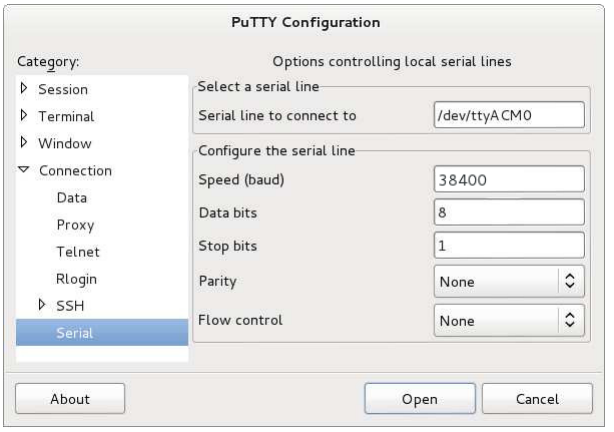


Figure 1.9: An example of the PuTTY serial settings

Exercise 2

Tasks and scheduling algorithms in FreeRTOS™

Create four FreeRTOS [14] tasks. Assign the same above idle priority to two tasks, and the idle priority to the other two. No task should ever end, each should run in an infinite loop. Assign a LED[1] and a button key on the external board to each task. In every iteration of the infinite loop, a task should turn its LED on and all the other LEDs off to indicate which task is running. The Idle task should turn all the LEDs off. Also, in every iteration of the infinite loop, a task should check all the button keys. If a button key belonging to a particular task is pressed, the task should be suspended, if released, the task should be resumed. Thus, a task is suspended while its button key is pressed, and vice versa. Observe various scheduling algorithms available in FreeRTOS. Can a running task explicitly request a context switch? How a delay can be effectively implemented?

Explanation

Create a new empty project in the Eclipse working environment as explained in Exercise 1. Connect the Arduino Due board to the host PC over the Olimex ARM-USB-OCD-H interface, and to the external board button keys and LEDs, as shown in Fig. 2.1.

The default `main()` function of the freshly created empty project can be found in the `src/main.c` file. The first function called is `prvSetupHardware()` where the hardware initialization is performed. Inside the `prvSetupHardware()` function, the functions `sysclk_reinit()`, `NVIC_SetPriorityGrouping()` and `board_init()` perform the basic initialization of the AT91SAM3X8E µC and the Arduino Due board. To be able to read the keys and drive the LEDs, the pins from `PC21` to `PC26`, `PC28` and `PC29` have to be configured as GPIO[2] pins (see Fig. 2.1). The keys require pull-up resistors on their input pins, while the debouncing filters are not necessary. See Appendix A for GPIO pin configuration and usage.

FreeRTOS configuration

The FreeRTOS properties can be set by options in the `src/FreeRTOSConfig.h` configuration file. The relevant options will be introduced in parallel with the explanation of the exercises in this script.

[1] LED ... Light Emitting Diode
[2] GPIO ... General Purpose I/O[3]
[3] I/O ... Input Output

Tasks in FreeRTOS

A task is a standalone program implemented as a function taking one void pointer argument, i.e., a function of type `void` *func*`(void *)`. The FreeRTOS can 'simultaneously' run more than one task by using the time slicing technique. A task can be in one of four states (see Fig. 2.2):

- ready; task is ready to run whenever scheduled,
- running; there is always exactly one task in running state,
- suspended; task stays suspended until resumed, and
- blocked; task is waiting for an event.

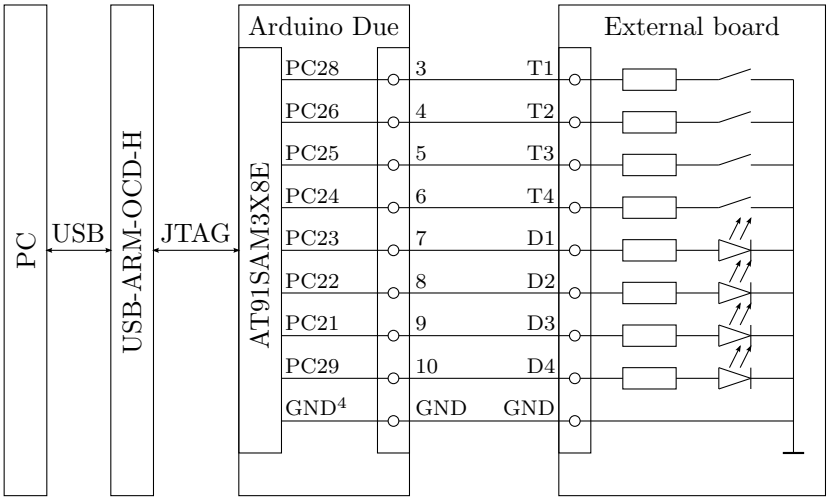


Figure 2.1: PC / Olimex ARM-USB-OCD-H / Arduino Due / External board connection for Exercises 2 in 3

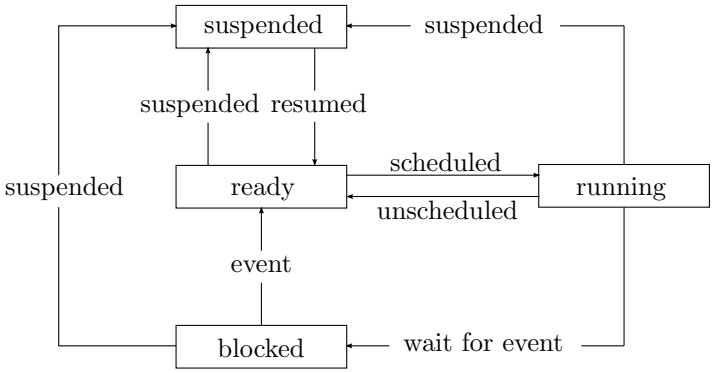


Figure 2.2: Task state machine

[4]GND ... Ground

Task creation and FreeRTOS starting

A new task is created by the `xTaskCreate()` function[5] [15] [16]. The declaration of the function is:

```
BaseType_t xTaskCreate( TaskFunction_t pvTaskCode,
                        char *pcName,
                        unsigned short usStackDepth,
                        void *pvParameters,
                        UBaseType_t uxPriority,
                        TaskHandle_t *pxCreatedTask );[6]
```

The function returns the `pdPASS` value on success. The function will fail if there is not enough heap memory available to allocate the task's stack and TCB[7]. A newly created task is placed into the ready state. The function arguments are:

`pvTaskCode`	...	pointer to task function
`pcName`	...	task name[8]
`usStackDepth`	...	task's stack size in words[9]
`pvParameters`	...	pointer passed to the task as an argument
`uxPriority`	...	task priority (zero is the lowest priority)
`pxCreatedTask`	...	pointer to where a handle to the created task is returned; if `NULL`, the handle is not returned

For instance, the function call

```
xTaskCreate(tsk, "Task", 150, (void *)2, 1, &xHnd);
```

creates a task named `"Task"` coded in the `tsk()` function of type `void tsk(void *)`. It has 600 bytes of stack. The pointer passed to the task function, e.g., `arg`, has value `0x00000002`. The task priority is one, just above the lowest. Pointer to a handle to the newly created task is written into the `xHnd` variable defined as `TaskHandle_t xHnd;`. Note that `xHnd` is actually a void pointer, and `&xHnd` is an address where this void pointer resides.

The FreeRTOS scheduler is started by `vTaskStartScheduler()` function [15] [16] declared as:

```
void vTaskStartScheduler( void );
```

If the scheduler is started successfully, the function will never return. The FreeRTOS scheduler starts to 'simultaneously' execute the created tasks following the scheduling algorithm. The function also creates an additional Idle task with the lowest priority. Therefore, in case when no application created task is ready to be scheduled into the running state, the Idle task will be available. Note that exactly one task must be in the running state at all times. Starting FreeRTOS scheduler will fail if there is not enough heap memory available for allocating the Idle task.

---

[5] The `configSUPPORT_DYNAMIC_ALLOCATION` option must not be set to zero in the `src/FreeRTOSConfig.h` configuration file to make the `xTaskCreate()` function available.

[6] `BaseType_t` is the most suitable integer type for the architecture, i.e., a 32-bit integer type for the AT91SAM3X8E µC.

`TaskFunction_t` type is a pointer to a function of type: `void` func(`void *`arg).

`UBaseType_t` is the most suitable unsigned integer type for the architecture, i.e., an unsigned 32-bit integer type for the AT91SAM3X8E µC.

`TaskHandle_t` is a void pointer type.

[7] TCB ... Task Control Block, i.e., task data used by FreeRTOS

[8] Maximum number of caharacters (ending `NULL` character included) in task name is set by the `configMAX_TASK_NAME_LEN` option in the `src/configFreRTOSConfig.h` configuration file.

[9] For AT91SAM3X8E, one word is four bytes.

In that case, the `vTaskStartScheduler()` function returns. The FreeRTOS is thus started with a simple `vTaskStartScheduler()` call:

```
vTaskStartScheduler();
```

A pseudo code that creates the required tasks and starts the FreeRTOS scheduler is as follows:

```
create two tasks with priority one
create two tasks with priority zero (idle priority)
start scheduler
```

To use the `xTaskCreate()` and `vTaskStartScheduler()` functions, and the accompanying data types, the `FreeRTOS.h` and `task.h` header files have to be included.

```
#include <FreeRTOS.h>
#include <task.h>
```

Suspending and resuming a task

Before discussing the actual implementation of the task functions, a lesson, how a task can be suspended and resumed, is needed. A task can be suspended by the `vTaskSuspend()` function[10] call [15] [16]. The declaration of the function is:

```
void vTaskSuspend( TaskHandle_t pxTaskToSuspend );
```

The function suspends the specified task, i.e., places the task into the suspended state. The argument of the function is:
`pxTaskToSuspend` ... task handle
The following `vTaskSuspend()` function call, for example, suspends a task specified by the `xHnd` handle:

```
vTaskSuspend(xHnd);
```

Passing `NULL` argument is equivalent to passing a handle of the calling task. The task suspends itself.

To resume a suspended task, the `vTaskResume()` function[10] can be used [15] [16]. The declaration of the function is:

```
void vTaskResume( TaskHandle_t pxTaskToResume );
```

The function call transfers the specified task from suspended into ready state (see Fig. 2.2). The function call will have no effect if the task to be resumed is not in the suspended state. The argument of the function is:
`pxTaskToResume` ... task handle
The following `vTaskResume()` function call, for instance, resumes previously suspended task specified by the `xHnd` handle:

```
vTaskResume(xHnd);
```

[10] The `INCLUDE_vTaskSuspend` option must be set to one in the `src/FreeRTOSConfig.h` configuration file to make the `vTaskSuspend()` and `vTaskResume()` functions available.

Task implementation

The four tasks in this exercise are actually four instances of the same algorithm. In every iteration of an endless loop, the LED corresponding to the task is turned on, and the tasks belonging to currently pressed keys are suspended. Other LEDs are turned off, and other tasks are resumed. The pseudo code of the algorithm is as follows:

```
while forever
        turn LED belonging to this task on, others off
        get key positions
        suspend tasks belonging to pressed keys, resume others
```

The Idle task

When the FreeRTOS is started, an additional Idle task with the lowest, i.e., zero or idle, priority is created. The stack depth of the Idle task is set by the `configMINIMAL_STACK_SIZE` option in the `src/FreeRTOSConfig.h` configuration file. The Idle task basically just waits in an endless loop. Its only real occupation is releasing system resources, e.g., heap memory, after an application task is deleted. Otherwise, the Idle task is equivalent to any other application created task with idle priority. The Idle task is invisible to the application by default.

If the `configUSE_IDLE_HOOK` will be set to one in the `src/FreeRTOSConfig.h` configuration file, the `vApplicationIdleHook()` callback function is called in every iteration of the Idle task endless loop [15] [16]. The function has to be defined in the application code as:

```
void vApplicationIdleHook(void) {
    ...
}
```

The Idle task calls the `vApplicationIdleHook()` function regularly. The function have to return within a short period of time, or else the Idle task cannot perform the releasing promptly. To avoid an error when no task is available to enter the running state, the idle task must never be blocked, suspended or deleted.

The Idle task functionality required in this exercise is implemented in the `vApplicationIdleHook()` function. The pseudo code of the function is as follows:

```
turn all LEDs off
get key positions
suspend tasks belonging to pressed keys, resume others
```

Scheduling algorithms

With all the code ready, different scheduling algorithms can be tested.

Cooperative scheduling is selected when the `configUSE_PREEMPTION` option is set to zero in the `src/FreeRTOSConfig.h` configuration file. A context switch occurs when the running task ends, is blocked, is suspended, or explicitly requests a switch. The running task cannot be preempted by a higher priority task. The next task to enter running state is a ready task with the highest priority. If there is more than one candidate, a task, being in ready state longest, will be selected. A context switch can be explicitly requested by calling the `taskYIELD()` function [15] [16] declared as:

```
void taskYIELD( void );
```

Thus, the running task places itself into ready state by the following call:

```
taskYIELD();
```

The Idle task requests a context switch in every iteration of its endless loop. With cooperative scheduling, the request cannot be canceled by setting the `configIDLE_SHOULD_YIELD` option to zero. Therefore, the Idle task should not be able to block any application task.

Prioritized preemptive scheduling without time slicing is selected when the `configUSE_PREEMPTION` option is set to one and the `configUSE_TIME_SLICING` option is set to zero in the `src/FreeRTOSConfig.h` configuration file. A context switch occurs when a task with higher priority than running task becomes ready, or when the running task ends, is blocked, is suspended, or explicitly requests a switch. The next task to enter running state is a ready task with the highest priority. If there is more than one candidate, a task, being in ready state longest, will be selected.

The Idle task requests a context switch in every iteration of its endless loop. The requests can be canceled by setting the `configIDLE_SHOULD_YIELD` option to zero. Therefore, the Idle task can block an idle priority application task.

Prioritized preemptive scheduling with time slicing is selected when the `configUSE_PREEMPTION` and `configUSE_TIME_SLICING` options are set to one in the `src/FreeRTOSConfig.h` configuration file. The time slice frequency is set by the `configTICK_RATE_HZ` option. The time slice length[11] is therefore `configTICK_RATE_HZ`$^{-1}$ seconds.

With this scheduling algorithm, a context switch will occur when a task with higher priority than running task becomes ready, or at the beginning of a new time slice if a task with running task's priority is ready, or when the running task ends, is blocked, is suspended, or explicitly requests a switch. The next task to enter running state is a ready task with the highest priority. If there is more than one candidate, a task, being in ready state longest, will be selected.

The Idle task requests a context switch in every iteration of its endless loop. The requests can be canceled by setting the `configIDLE_SHOULD_YIELD` option to zero. Regardless canceling, the Idle task cannot block any application task since context switch takes place at the beginning of a new time slice.

Delay

If an application task at some point needs to wait for a predefined amount of time, the most efficient way will be to place the task into the blocked state for that period. That can be achieved with the `vTaskDelay()` function[13] [15] [16] declared as:

[11] A share of time consumed by the operating system overhead increases with time slice shortening. Therefore, the time slice should not be too short. On the other end, the maximum time slice length is 1s, which cannot be achieved by any MCK[12] frequency since the FreeRTOS uses the 24-bit SysTick timer, e.g., $t_{\mathrm{SLICE_{MAX}}}\big|_{f_{\mathrm{MCK}}=84\mathrm{MHz}} = \frac{2^{24}-1}{f_{\mathrm{MCK}}} < 200\mathrm{ms}$.

[12] MCK ... Master Clock

[13] The `INCLUDE_vTaskDelay` option in the must be set to one in the `src/FreeRTOSConfig.h` configuration file to make the `vTaskDelay()` function available.

`void vTaskDelay( TickType_t xTicksToDelay );`[14]

The function places the calling task into a blocked state for the specified number of time slices, i.e., ticks. When the period expires, the task is placed into ready state. The delay interval has to be inconveniently specified in number of ticks. To convert milliseconds into a number of ticks, the `pdMS_TO_TICKS()` macro can be used [15] [16]. For instance, the following `vTaskDelay()` call makes a task wait for 2s:

```
vTaskDelay(pdMS_TO_TICKS(2000));
```

Note that the `vTaskDelay()` function must never be called from the `vApplication IdleHook()` callback function since the Idle task must not be placed into the blocked state.

Include a delay into one or more of the four tasks and observe the scheduling algorithms.

[14] `TickType_t` is an unsigned 32-bit integer by default. It will be set to an unsigned 16-bit integer if the `configUSE_16_BIT_TICKS` option is set to one.

Exercise 3

Implementing other scheduling algorithms

Write a code for four finite tasks with various BTs[1]. A task should run in a finite loop and end after a predefined number of iterations defining the BT. Use button keys on the external board as asynchronous task triggers. Each time a button key is pressed, a request for a single run of the corresponding finite task should be issued. Use LEDs on the external board to indicate which task is currently running. Implement the FCFS[2], SJF[3], SRTF[4] and RR[5] scheduling algorithms by dynamically adjusting the task priorities. Which scheduling algorithms are cooperative, and which are preemptive? Which can cause the convoy effect, and/or the CPU[6] starvation? Why the FreeRTOS sometimes skips a task, i.e., the task is not run, although a request was issued?

Explanation

Create a new empty project in the Eclipse working environment as explained in Exercise 1. Connect the Arduino Due board to the host PC over the Olimex ARM-USB-OCD-H interface, and to the external board button keys and LEDs, as shown in Fig. 2.1. Configure the pins from `PC21` to `PC26`, `PC28` and `PC29` as GPIO pins. The pull-up resistors are required on input pins, i.e., keys. To properly detect the key pressing, the debouncing filters are also needed. See Appendix A for GPIO pin configuration and usage. Start the FreeRTOS scheduler as explained in Exercise 2. As the FreeRTOS is started without any tasks created, the only task at the beginning is the Idle task.

Ending a finite task

A finite task is usually executed at an event, i.e., a key press. The task ends when its job is done, e.g., the event is handled. In the FreeRTOS, a task is a standalone function that should never return. Therefore, it cannot just end. The `vTaskDelete()` function[7] [15] [16] must be called instead. The declaration of the function is:

```
void vTaskDelete( TaskHandle_t pxTask );
```

[1] BT ... Burst Time, i.e., task's running state time
[2] FCFS ... First Come First Serve
[3] SJF ... Shortest Job First
[4] SRTF ... Shortest Remaining Time First
[5] RR ... Round Robin
[6] CPU ... Central Process Unit
[7] The `INCLUDE_vTaskDelete` option in the must be set to one in the `src/FreeRTOSConfig.h` configuration file to make the `vTaskDelete()` function available.

The function informs the FreeRTOS kernel to delete the specified task. In general, any task can delete any other task. The task related system resources are released later by the Idle task (see Exercise 2). The function argument is:
`pxTask` ... task handle
Passing `NULL` argument is equivalent to passing a handle of the calling task. The task deletes itself. In that case, the function never returns. To end a finite task, the task must call the `vTaskDelete()` function and delete itself before ending:

```
void func(void *arg) {
        ...
    vTaskDelete(NULL);
}
```

To use the `vTaskDelete()` function, the `FreeRTOS.h` and `task.h` header files have to be included.

```
#include <FreeRTOS.h>
#include <task.h>
```

FCFS scheduling algorithm

The FCFS scheduling algorithm executes tasks one after another in FIFO[8] order as their requests arrive. There is no priority, nor preemption. In FreeRTOS, the FCFS scheduling algorithm can be implemented by cooperative scheduling and all tasks having the same idle priority. The idle priority allows the Idle task to participate, thus being able to release system resources after task ending. Since there is no task preemption, the FCFS is a cooperative algorithm.

While a long BT task is being executed, new single run task requests pile up. The newly arrived tasks are made to wait the line. Every request, short BT tasks are no exception, is added at the end of the line. Such an accumulation of waiting tasks is called the convoy effect. The convoy effect can significantly increase the average waiting time[9] that leads to lower CPU utilization [17]. The FCFS scheduling algorithm can cause the convoy effect.

A task is starving of CPU time when continuously denied to enter the running state. The CPU starvation effect occurs when a scheduling algorithm denies one or more tasks to be scheduled for infinite amount of time. Assuming finite task BTs, the FCFS scheduling algorithm cannot cause the CPU starvation under any task arriving scheme.

The FreeRTOS was started without any tasks created. The only task at the beginning is the Idle task. It turns all the LEDs off, and creates a task corresponding to the newly pressed key. The created task is placed into the ready state, which is in fact a run request. The Idle task functionality can be coded in the `vApplicationIdleHook()` callback function (see Exercise 2). The pseudo code of the function is:

```
turn all LEDs off
for each of the four keys
        if the key is down and was up in the previous Idle task iteration
                create new key corresponding task
        save key position for the next Idle task iteration
```

[8] FIFO ... First In First Out
[9] An average time interval between task request arrival and entering the running state.

Current key positions are saved into a global array in each iteration of the Idle task. They are used to detect a key pressed event.

The tasks implement the same algorithm in a finite loop. The pseudo code of the finite task algorithm is as follows:

```
for predefined number of iterations
        turn LED belonging to this task on, others off
        for each of the four keys
                if the key is down and was up in the previous iteration
                        create new key corresponding task
                save key position for the next iteration
delete this task
```

A new task is created on every key pressed event. Each task requires some operating system resources, e.g., some heap space. If the line of waiting tasks gets long, there may not be enough space left to create another task. The `xTaskCreate()` function fails (see Exercise 2), and the run request gets skipped. Obviously, skipping is more common when the heap size is small. The heap size can be set by the `configTOTAL_HEAP_SIZE` option in the `src/FreeRTOSConfig.h` configuration file.

SJF scheduling algorithm

The SJF scheduling algorithm executes tasks one after another. There is no preemption. A task with shorter BT has higher priority, though. Therefore, a new task request is not just added at the end of the waiting line, but is inserted into the line according to its BT. In FreeRTOS, the SJF scheduling algorithm can be implemented by cooperative scheduling and creating tasks with above idle priorities according to their BTs. There are four finite tasks with predefined BTs in this exercise. The priority of the task with the longest BT should be set to one, the next to two, etc., and the task with the shortest BT should have priority four. The maximum number of available priorities is set by the `configMAX_PRIORITIES` option in the `src/FreeRTOSConfig.h` configuration file. For SJF scheduling of four tasks, the `configMAX_PRIORITIES` option has to be at least five.

Obviously, the SJF scheduling algorithm is a cooperative algorithm. Since the shorter BT tasks get scheduled first, the waiting line is shorter than in FCFS algorithm. Consequently, the convoy effect is smaller or none, skipping requests is rarer. On the other hand, the SJF scheduling algorithm can cause the CPU starvation. Constantly arriving short BT tasks can block a long BT task infinitely.

The SJF version of the pseudo code of the Idle task callback function is only slightly modified FCFS version. The tasks has to be created with priorities according to their BTs:

```
turn all LEDs off
for each of the four keys
        if the key is down and was up in the previous Idle task iteration
                create new key corresponding task with priority reflecting its BT
        save key position for the next Idle task iteration
```

The same goes for the pseudo code of the finite tasks:

`for` predefined number of iterations
    turn LED belonging to this task on, others off
    `for` each of the four keys
        `if` the key is down and was up in the previous iteration
            create new key corresponding task with priority reflecting its BT
        save key position for the next iteration
delete this task

Critical section of code

Tasks may want to access the same resource, e.g., global variable, peripheral device register etc., at the same time. Since preemption can occur at any time, the outcome depends on the sequence in which the individual tasks access the resource. The phenomenon is called a race condition. A race condition cannot occur in cooperative scheduling.

Example: Task A wants to increment, and task B to decrement the same global variable. At the end, the variable value should be the same. Task A reads the variable. Before succeeding to store the incremented value back into the memory, task B preempts task A. Since the variable has not changed yet, task B decrements the original value. After a while, task A is rescheduled. It continues with incrementing/storing the previously read value. The global variable unexpectedly ends incremented. If the preemption take place a bit later, the expected result will be obtained. Task A would manage to store the incremented value, and task B would decrement the variable back to its original value.

Critical section of code is a region where a race condition can arise. If preemption is disabled during the critical section, a race condition cannot occur. The critical section becomes an atom, i.e., a region of code that cannot be interrupted. To make a section of code an atom, the `taskENTER_CRITICAL()` and `taskEXIT_CRITICAL()` macros can be used [15] [16]. The enclosed code becomes an atom.

```
    ...
taskENTER_CRITICAL();
  atom code
taskEXIT_CRITICAL();
    ...
```

Atoms should be very short. The code inside an atom is guaranteed to stay in the running state. It must not request a context switch, go blocked, suspended, or end. The `taskENTER_CRITICAL()` and `taskEXIT_CRITICAL()` macros can be nested.

Task priority modification

Initial task priority defined at task creation (see Exercise 2) can be dynamically modified. The `vTaskPrioritySet()` function[10] [15] [16] can be used. Its declaration is:

```
void vTaskPrioritySet( TaskHandle_t pxTask,
                       UBaseType_t uxNewPriority );
```

[10]The `INCLUDE_vTaskPrioritySet` option must be set to one in the `src/FreeRTOSConfig.h` configuration file to make the `vTaskPrioritySet()` function available.

The function modifies the specified task's priority. The arguments of the function are:

`pxTask`	...	task handle
`uxNewPriority`	...	task priority

Passing NULL as a task handle argument is equivalent to passing a handle of the calling task. The task modifies its own priority.

The following `vTaskPrioritySet()` function call, for instance, sets priority of the calling task to three:

```
vTaskPrioritySet(NULL, 3);
```

SRTF scheduling algorithm

The SRTF scheduling algorithm always runs a task with the shortest remaining time to completion. A new arrived task will preempt the running task if its BT is shorter than the remaining time of the running task. In FreeRTOS, the SRTF scheduling algorithm can be implemented by prioritized preemptive scheduling without time slicing. The task priority has to be initially set according to its BT. During the execution, the remaining time to completion decreases, and the task priority has to be correspondingly raised.

The SRTF scheduling algorithm is a preemptive algorithm. There is no convoy effect. Skipping requests due to a lack of system resources should not be an issue. The SRTF algorithm can cause the CPU starvation. Constantly arriving short BT tasks can block a long BT task infinitely.

A section of code from getting to saving the current key position for the next iteration is critical and should be an atom. Otherwise, a key press requesting a short BT task will be handled twice if the remaining time of the running task is longer. The SRTF version of the pseudo code of the Idle task callback function is as follows:

```
turn all LEDs off
for each of the four keys
        enter critical section
        if the key is down and was up in the previous Idle task iteration
                create new key corresponding task with priority reflecting its BT
        save key position for the next Idle task iteration
        exit critical section
```

The tasks implement essentially the same algorithm in a finite loop. Since the remaining time constantly decreases, each task should be adequately raising its priority during the execution. The number of the remaining finite loop iterations can be used as a remaining time measure.

In this exercise, there are four tasks with predefined number of iterations. Suppose the number of iterations of the first task is A, of the second B, of the third C, and of the fourth D, and $A < B < C < D$. The initial priority of the first task is set to four, of the second to three, of the third to two, and of the fourth to one, thus reflecting the tasks' BTs. The priority is then raised as the number of remaining finite loop iterations decreases. The following pseudo code implements the described mechanism:

```
for predefined number of iterations, i.e., one of A, B, C, or D
    turn LED belonging to this task on, others off
    for each of the four keys
        enter critical section
        if the key is down and was up in the previous iteration
            create new key corresponding task with priority reflecting its BT
        save key position for the next iteration
        exit critical section
    if number of iterations left equals to any of 1, A + 1, B + 1, or C + 1
        increment priority
delete this task
```

Note that number of remaining iterations is decremented after the iteration is completed. The task priority is gradually raised. The final priority before task completion is five. Therefore, the `configMAX_PRIORITIES` option has to be at least six.

RR scheduling algorithm

The RR scheduling algorithm assigns one time slice per task in circular manner. Each task gets at most one time slice of CPU before the context switch takes place. There is no priority. A newly arrived task request is added at the end of the line of tasks. In FreeRTOS, the RR scheduling algorithm can be implemented by prioritized preemptive scheduling with time slicing and all tasks having the same idle priority. The idle priority allows the Idle task to participate, thus being able to release system resources after task ending.

The RR scheduling algorithm is a preemptive algorithm. The convoy effect and skipping requests due to a lack of system resources both increase with the time slice length. For an infinitely long time slice, the RR converts into the FCFS algorithm. The RR algorithm cannot cause the CPU starvation under any task arriving scheme.

Since a preemption can occur at any place in the code, a section of code from getting to saving the current key position for the next iteration is critical and should be an atom. To obtain the RR version of the pseudo code of the Idle task callback function, the FCFS version has to be equipped with critical section markers:

```
turn all LEDs off
for each of the four keys
    enter critical section
    if the key is down and was up in the previous Idle task iteration
        create new key corresponding task
    save key position for the next Idle task iteration
    exit critical section
```

The same goes for the pseudo code of the finite tasks:

```
for predefined number of iterations
        turn LED belonging to this task on, others off
        for each of the four keys
                enter critical section
                if the key is down and was up in the previous iteration
                        create new key corresponding task
                save key position for the next iteration
                exit critical section
delete this task
```

Exercise 4

Assembly language function

In the assembly language of the AT91SAM3X8E µC, write an external function which performs an addition of two arbitrary long unsigned integers. The function should receive four arguments. The first argument should be a pointer to the final sum, i.e., the address where the function writes the result. The next two arguments should be pointers to both summands. And the fourth argument should provide the length of the integers in 32-bit words. The function should return the final carry bit value. To test the function, write a program reading two 128-bit long unsigned integers in a hexadecimal form from the `stdin` stream, and writing their sum to the `stdout` stream. Use the UART peripheral device as the stdio[1], and an arbitrary serial terminal program as a console.

Explanation

Create a new empty project in the Eclipse working environment as explained in Exercise 1. Connect the Arduino Due board to the host PC over the Olimex ARM-USB-OCD-H interface as shown in Fig. 4.1. Configure the UART peripheral device and the stdio in serial mode as explained in Appendix A.

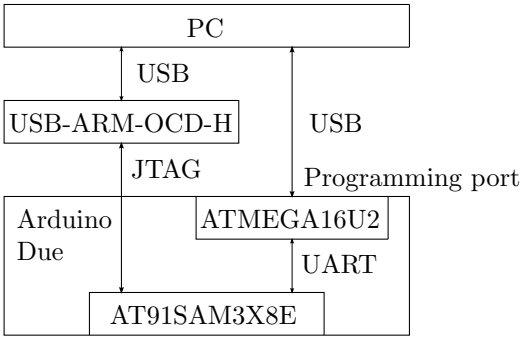


Figure 4.1: PC / Olimex ARM-USB-OCD-H / Arduino Due connection for Exercise 4

Function description

As it can be understood from the exercise text, the function has the following declaration:

[1]stdio ... Standard I/O

```
uint32_t func(uint32_t *sum,
              uint32_t *summand1,
              uint32_t *summand2,
              uint32_t length);[2]
```

It has to be declared as an `extern` function, since its definition will reside in a separate assembly code file.

The function usage from the C code is demonstrated with the following lines. The two 128-bit (= 4 × 32bit) summands `pulNum1` and `pulNum2` are added, the result is written into `pulSum`, and the final carry bit is returned.

```
uint32_t pulNum1[4], pulNum2[4], pulSum[4], ulC;
    ...
ulC = func(pulSum, pulNum1, pulNum2, 4);
```

Both summands and the sum are the arrays of four 32-bit unsigned integers. Each array represents a 128-bit number. The function adds the numbers from the `pulNum1` and `pulNum2` arrays and writes the result into the `pulSum` array as shown in Fig. 4.2. The length of the numbers is four times 32-bit, i.e., 128-bit.

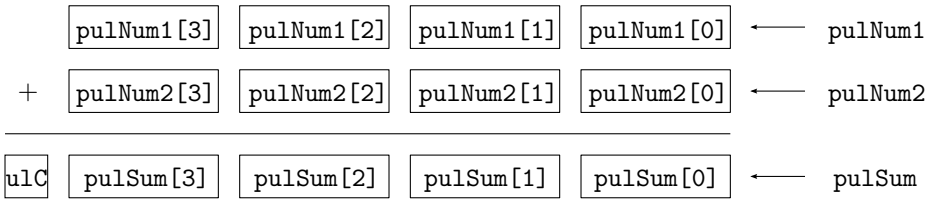


Figure 4.2: The addition of two 128-bit long unsigned integer numbers

Function implementation

The AT91SAM3X8E µC is based on the ARM Cortex-M3 processor [18]. The processor has 16 32-bit core registers labeled from `r0` to `r15`. There are 13 general-purpose registers `r0` to `r12`, an SP register `r13`, a link register `r14`, and a program counter `r15`.

When writing a subroutine, i.e., an external function, in the AT91SAM3X8E µC assembly language, the subroutine calling convention for the ARM architecture [19] has to be taken into account. By following the convention, the assembly subroutine can be transparently called from the C code as an external function. A short summary of the convention for the purpose of this exercise follows:

- The `r0` to `r3` registers are used to pass the arguments into the subroutine, and to pass the result value out. If there are more arguments, or the result is larger, the stack will be used. The `r0` to `r3` registers do not need to be restored before returning.

- The `r4` to `r11` registers can be freely used by the subroutine. They must be restored before returning. Therefore, the used registers are usually pushed into the stack on subroutine entry, and restored from it before returning.

- The `r12` register is a scratch register and can be used for any purpose.

[2] `uint32_t` is a 32-bit unsigned integer type.

- The `r13` register is the SP register and must not be used for any other purpose. The stack operates in full-descending mode, i.e., the SP register points to the last item on stack and the stack grows downwards to lower memory addresses.
- The `r14` register is the link register with the returning address. Note that the link register must be stored, e.g., to stack, when a subsubroutine is called.

According to the convention, the first argument, i.e., pointer to the sum, is passed in the `r0` register, the second and the third, i.e., pointers to both summands, in `r1` and `r2`, and the fourth, i.e., length, in `r3`. Stack is not used. Before returning, the result, i.e., the final carry bit value, has to be stored into the `r0` register to be passed back.

The summands are added by parts, 32-bits at a time. The carry bit from the previous iteration is added in each step. Obviously, the number of iterations equals to the `length` argument. The final carry bit is stored into the `r0` register for returning. The registers used in the subroutine must be saved, i.e., pushed to the stack, at the subroutine beginning, and restored, i.e., popped off the stack, at the end. The pseudo code of the subroutine is as follows:

```
        save working registers (push to the stack)
        C = 0
iter:   C, [r0] ← [r1] + [r2] + C
        increment pointers r0, r1 and r2
        decrement r3
        if r3 is not zero
                go to iter
        r0 ← C
        restore working registers (pop off the stack)
        return
```

As the exercise requires, the subroutine has to be written in assembly language in a separate assembly code file. In the empty project, the `src/sum.S` file is prepared for that purpose. Since the subroutine represents an external function to be used from the C code, it has to be visible outside the assembly file. Thus, the `.global` assembler directive [20] is required. The assembly file structure is:

```
        .thumb
        .syntax unified
        .global func
        .text
func:    ...
         ...
        .end
```

The ARM Cortex-M3 processors implements the ARMv7-M Thumb instruction set [21] which is quite extensive. To code the function from this exercise, only some basic variations of load and store, data-processing and branch instructions are required. A few selected instructions can be found in Tab. 4.1. The triangular brackets `<>` denote a required field, the square brackets `[]` denote address dereferencing, e.g., `[rn]` denotes a value stored at the memory location address in the `rn` register, and the curly brackets `{}` denote an optional field.

instruction	operation
`stmfd <rn>{!},<regs>`	regs → `[rn]`[3]
`ldmfd <rn>{!},<regs>`	`[rn]` → regs[3]
`mov{s} <rd>,#<const>`	`const` → `rd`[4]
`mov{s} <rd>,<rm>`	`rm` → `rd`[4]
`ldr <rd>,=<c32>`	`c32` → `rd`[5]
`ldr <rd>,[<rn>{,#+/-<imm>}]`	`[rn±imm]` → `rd`[6]
`ldr <rd>,[<rn>,#+/-<imm>]!`	`[rn±imm]` → `rd`, `rn±imm` → `rn`[6]
`ldr <rd>,[<rn>],#+/-<imm>`	`[rn]` → `rd`, `rn±imm` → `rn`[6]
`ldr <rd>,[<rn>,<rm>{,<shift>}]`	`[rn+`*shift*`(rm)]` → `rd`[7]
`str <rs>,[<rn>{,#+/-<imm>}]`	`rs` → `[rn±imm]`[8]
`str <rs>,[<rn>,#+/-<imm>]!`	`rs` → `[rn±imm]`, `rn±imm` → `rn`[8]
`str <rs>,[<rn>],#+/-<imm>`	`rs` → `[rn]`, `rn±imm` → `rn`[8]
`str <rs>,[<rn>,<rm>{,<shift>}]`	`rs` → `[rn+`*shift*`(rm)]`[9]
`cmp <rn>,#<const>`	`rn−const`[10]
`cmp <rn>,<rm>{,<shift>}`	`rn−`*shift*`(rm)`[11]
`cmn <rn>,#<const>`	`rn+const`[10]
`cmn <rn>,<rm>{,<shift>}`	`rn+`*shift*`(rm)`[11]
`add{s} {<rd>,}<rn>{,#<const>}`	`rn+const` → `rd`[12 18]
`add{s} {<rd>,}<rn>,<rm>{,<shift>}`	`rn+`*shift*`(rm)` → `rd`[13 18]
`adc{s} {<rd>,}<rn>{,#<const>}`	`rn+const+C` → `rd`[14 18]
`adc{s} {<rd>,}<rn>,<rm>{,<shift>}`	`rn+`*shift*`(rm)+C` → `rd`[15 18]
`sub{s} {<rd>,}<rn>{,#<const>}`	`rn−const` → `rd`[16 18]
`sub{s} {<rd>,}<rn>,<rm>{,<shift>}`	`rn−`*shift*`(rm)` → `rd`[17 18]
`b{cond}` <*label*>	*label* → `r15`[19]

`rd`, `rs`, `rm`, `rn` ... arbitrary core register, i.e., `r0` to `r15`[20]

`regs` ... list of comma separated registers surrounded by `{}`, e.g., `{r4,r5,r7}`

`!` ... write the final address back to `rn`

`s` ... update the flags in the PSR[21]

`const` ... 32-bit constant, entire range not available[22]

`c32` ... 32-bit constant

`imm` ... up to 12-bit offset added to or subtracted from the `rn` register[23] (default: `imm` = `0`)

`shift` ... shift rule (default: `shift` = `lsl #0`):

- `lsl #<n>` ... logical shift left `n` bits[24]
- `lsr #<n>` ... logical shift right `n` bits[24]
- `asr #<n>` ... arithmetic shift right `n` bits[24]
- `ror #<n>` ... rotate right `n` bits[24]
- `rrx` ... rotate right one bit with extend over the carry flag

`cond` ... condition under which the instruction is executed:

- `al` (always) → execute always (default),
- `eq` (equal) → execute if `Z` = 1,
- `ne` (not equal) → execute if `Z` = 0,
- `cs` (carry set) → execute if `C` = 1,
- `cc` (carry clear) → execute if `C` = 0, etc.

For complete list, see [21].

Table 4.1: Selected instructions from the ARMv7-M Thumb instruction set

Testing program in C

To test the assembly summing function, a few additional lines of C code are needed. Write an auxiliary program preparing the two summands, calling the function, and printing the result. Since the stdio is configured in serial mode, the stdio functions (e.g. `putchar()`, `getchar()`, `printf()`, etc.) can be used for reading form / writing to the UART device. An arbitrary serial terminal program running on the host PC can serve as a console as described in Exercise 1. The pseudo code of the auxiliary program is as follows:

`while` forever
    obtain two 128-bit summands
    call the assembly function to add them
    print the result

A hint: Read each summand character by character from the `stdin` standard stream using the `getchar()` function. Read eight characters at a time, i.e., a 32-bit number in a hexadecimal format. Use the `strtoul()` standard function to convert the eight character string into a 32-bit unsigned integer. After addition, use the `%08lx` format specifier of the `printf()` standard function to print a 32-bit

[3]Store (`stm`) / load (`ldm`) multiple registers, i.e., `regs`, to / from consecutive memory locations in full descending mode starting at the address in the `rn` register.

[4]Load the `const` value / `rm` register into the `rd` register.

[5]Load the `c32` value into the `rd` register. The same is achieved when the 32-bit constant is labeled:

```
label: .long <c32>
       ...
       ldr   <rd>, label
```

For the `.long` assembler directive, see [20].

[6]Obtain the value from the memory location address in the `rn` register increased or decreased by the `imm` offset value and load it into the `rd` register. The calculated address can be written back to the `rn` register.

[7]Calculate the memory location address by adding the `rn` register and the `rm` register shifted according to the `shift` rule. Load the value from the obtained address into the `rd` register.

[8]Store the `rs` register to the memory location address in the `rn` register increased or decreased by the `imm` offset value. The calculated address can be written back to the `rn` register.

[9]Calculate the memory location address by adding the `rn` register and the `rm` register shifted according to the `shift` rule. Store the `rs` register to the obtained address.

[10]Set the PSR flags according to the `rn`$\mp$`const` term.

[11]Shift the `rm` register according to the `shift` rule and set the PSR flags according to the `rn`$\mp$*shift*(`rm`) term.

[12]Add the `rn` register and the `const` value, and load the result into the `rd` register.

[13]Shift the `rm` register according to the `shift` rule, add the `rn` register, and load the result into the `rd` register.

[14]Add the `rn` register, the `const` value and the carry flag, and load the result into the `rd` register.

[15]Shift the `rm` register according to the `shift` rule, add the `rn` register and the carry flag, and load the result into the `rd` register.

[16]Subtract the `const` value from the `rn` register and load the result into the `rd` register.

[17]Shift the `rm` register according to the `shift` rule, subtract it from the `rn` register, and load the result into the `rd` register.

[18]If the `rd` register is omitted, the result will be loaded into the `rn` register.

[19]Branch to the target, i.e., *label*, address by setting the program counter register, i.e., `r15`.

[20]Available registers that can be specified at a particular field depend on the instruction encoding. In some cases the SP register, i.e., `r13`, or program counter register, i.e., `r15`, cannot be specified, or a particular register combination is not possible, etc. See [21] for details.

[21]PSR ... Program Status Register

[22]The entire 32-bit range is not available. Available constant values depend on the instruction encoding. See [21] for details.

[23]The offset value construction depends on the instruction encoding. See [21] for details.

[24]The range for `n` depends on the `shift` rule and the instruction encoding. Also all `shift` rules are not always available. See [21] for details.

unsigned integer in a hexadecimal format.

Exercise 5

MPU

Suppose that the PB27 pin, to which the Arduino Due on-board LED is connected, is configured as an output GPIO pin initialized to zero[1]. Observe the following section of the program code (include directives, hardware initialization code, etc., are omitted):

```
void prvRecursiveFunc(uint32_t ulDepth) {
    if(ulDepth) prvRecursiveFunc(ulDepth - 1);
}

int main(void) {
    static uint32_t ulVar;
        ...
    prvRecursiveFunc(DEPTH);
    if(ulVar != 0) pio_set(PIOB, PIO_PB27);
    for(;;);
}
```

The initial value of the ulVar static variable is zero and should never change. The on-board LED should stay off, the PB27 output pin should not be set.

Gradually increase the constant *DEPTH* and find out when the on-board LED is turned on. Explain the phenomena. Observe the SP register during the prvRecursiveFunc() function call. Configure the MPU[2] to catch the stack overflow events. Measure the approximate amount of stack, that a simple

```
printf("Hello World!");
```

function call needs.

Explanation

Create a new empty project in the Eclipse working environment as explained in Exercise 1. Connect the Arduino Due board to the host PC over the Olimex ARM-USB-OCD-H interface as shown in Fig. 4.1. Configure the UART peripheral device and the stdio in serial mode, and the PB27 pin as an output GPIO pin as explained in Appendix A.

Type the given code in, start with small *DEPTH*, e.g., $DEPTH = 10$. Compile, upload and run the program. The on-board LED stays off, as expected regarding

[1] See Appendix A.
[2] MPU ... Memory Protection Unit

the code. Increase the constant *DEPTH* and repeat the procedure. Observe that for *DEPTH* large enough[3], the LED turns on.

Program memory map

To understand why the condition `ulVar != 0` becomes true when the constant *DEPTH* is increased and consequently the on-board LED is turned on, the program memory map has to be examined. The program memory map shows the address space parts used to store the program code and data. It is defined by the `sam/utils/linker_scripts/sam3x/sam3x8/gcc/flash.ld` linker script. The ASF provided linker script (see Exercise 1) is slightly customized to accommodate the MPU port of the FreeRTOS. The program memory map for the AT91SAM3X8E µC defined by the linker script from [13] is shown in Fig. 5.1. With another linker script, a different memory map can be obtained.

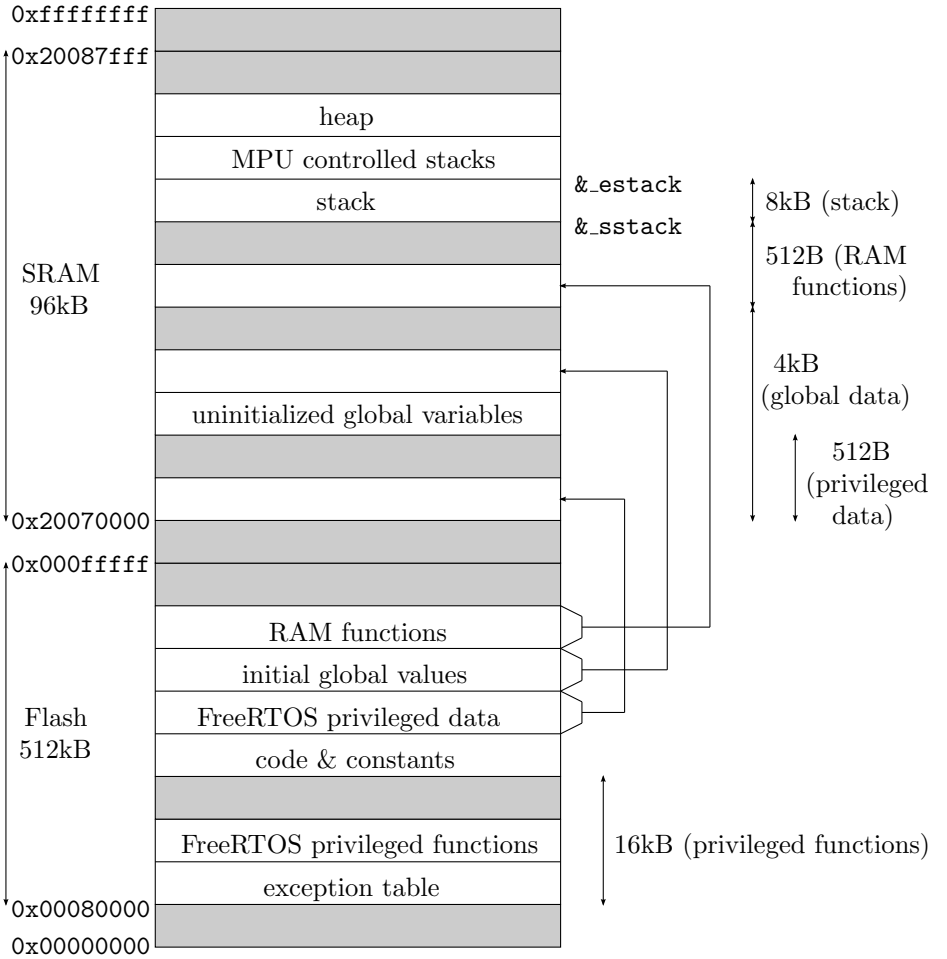


Figure 5.1: The program memory map defined by the linker script in [13]

The source code is compiled and linked into an output (`elf`) file that is uploaded into the on-chip Flash memory. At reset, the `Reset_Handler()` function performs the SRAM[4] initialization (see footnote[25] in Exercise 1). Initial values

[3]This is true for the project files downloaded from [13]. With another linker script and/or makefiles, different effects can be achieved.

[4]SRAM ... Static RAM

of the initialized global and static variables (FreeRTOS privileged data included), and RAM functions[5], are copied from Flash to RAM. The segment containing the uninitialized global and static variables (on Fig. 5.1 marked as 'uninitialized global variables') is cleared, thus setting the initial values of those variables to zero. The Flash and SRAM addresses required to preform the SRAM initialization are defined in the linker script. A symbol marking an individual address can be accessed from C code by declaring it as an extern unsigned 32-bit integer variable, e.g., symbols `_sstack` and `_estack` from Fig. 5.1 can be accessed when declared as:

```
extern uint32_t _sstack, _estack;
```

The extern variable defined in the linker script is placed at the specified address. The address can be obtained by the C language address operator `&`, e.g., `&_sstack` equals to `0x20071200` (= `0x20070000` + 4kB + 512B), and `&_estack` equals to `0x20073200` (= `&_sstack` + 8kB).

The `ulVar` is an uninitialized static variable. It resides in the 'uninitialized global variables' SRAM segment on Fig. 5.1. Its initial value is set to zero during the SRAM initialization.

Stack overflow

Stack is a memory space in SRAM organized as a LIFO[6] data structure (see Fig. 5.2). The AT91SAM3X8E µC uses a full-descending stack. That means that the SP register points to the last stacked item, and the stack grows towards lower addresses. The SP register is initialized to the stack bottom, i.e., `&_estack`, and grows downwards, i.e., towards `&_sstack`. The stack is used to store local variables. The µC state, i.e., return address, register values, etc., at function call is also pushed to stack.

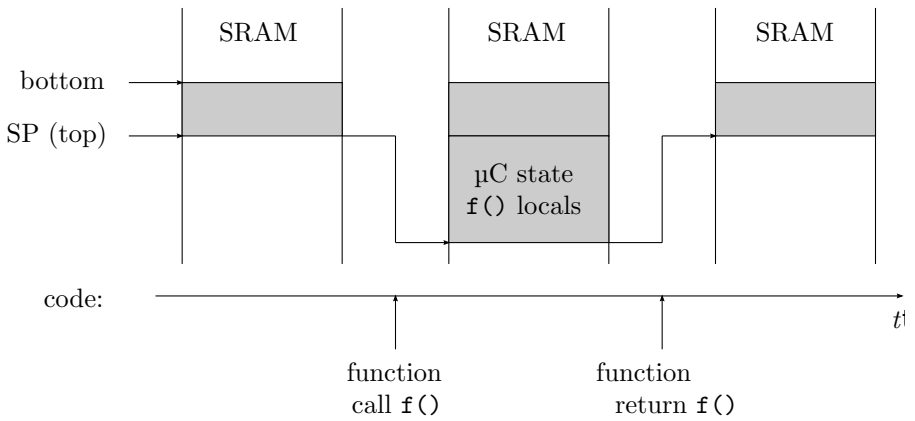


Figure 5.2: Stack

When the `prvRecursiveFunc()` function is called, the µC state is pushed onto the stack. There are no local variables, so no space is required on that account. The `prvRecursiveFunc()` function recursively calls itself. The µC state is pushed onto the stack again. The procedure is performed for *DEPTH* times. The stack grows to its final size achieved at the last recursive call, and then shrinks back

[5] A function marked as a RAM function is compiled to run from SRAM.
[6] LIFO ... Last In First Out

as the recursion unfolds. The stack growing and shrinking can be monitored by observing the SP register.

The program works fine as long as the stack top, i.e., the SP register, stays inside the stack segment boundaries[7], i.e., inside the [`&_sstack`, `&_estack`] address space interval, see Fig. 5.1). If the *DEPTH* constant is large enough, the stack will grow beyond the `&_sstack` boundary and sooner or later will flood the segment below, i.e., at a lower address. The phenomena is called stack overflow. If the flood reaches the 'uninitialized global variables' segment, the `ulVar` variable may become corrupted. The condition `ulVar != 0` does not hold any more, and the on-board LED is turned on. The error can be postponed to a larger *DEPTH* parameter by increasing the stack segment size[8].

Operation modes and SP registers

The AT91SAM3X8E µC is based on the ARM Cortex-M3 processor [18] having two operation modes: thread mode and handler mode. The µC is in handler mode when executing an interrupt or system exception handler, otherwise the µC is in thread mode.

There are two privilege levels[9] of code execution: privileged and unprivileged level. In handler mode, the code always runs at privileged level. In thread mode, the code can run at privileged or unprivileged level. The privilege level in thread mode is defined by the thread mode privilege level bit in the CONTROL register [8].

There are also two SP registers available, MSP[10] and PSP[11]. In handler mode, the MSP is always used. In thread mode, the MSP or PSP can be used. The SP register used in thread mode is defined by the active SP bit in the CONTROL register [8]. The SP register `r13` (see Exercise 4) is a copy of the currently used register, MSP or PSP.

CONTROL register

TMPL ... thread mode privilege level (0 ... privileged, 1 ... unprivileged)
ASP ... active SP (0 ... MSP, 1 ... PSP)

At the reset, the µC starts in thread mode at privileged level using MSP. The initial MSP register value, i.e., `&_estack`, is loaded from the first entry in the exception table defined in the `sam/utils/cmsis/sam3x/source/templates/gcc/startup_sam3x.c` file.

The `msr` instruction can be used to set the SP register value, PSP or MSP. Both registers can also be set from the C code with the `__set_PSP()` and `__set_MSP()` CMSIS function pair. The declarations are:

```
void __set_PSP(uint32_t topOfProcStack);
void __set_MSP(uint32_t topOfMainStack);
```

[7] The stack segment size, i.e., the difference `&_estack` − `&_sstack`, is defined by the `__stack_size__` symbol. Its default value is defined in the linker script and is set to `0x2000` (= 8kB).

[8] Different value for the `__stack_size__` linker symbol can be specified in the `config.mk` file by defining the linker flag [3]:
`LDFLAGS = -Wl,--defsym,__stack_size__=`*value*

[9] Do not confuse privilege levels with FreeRTOS privileged functions and data.

[10] MSP ... Main SP

[11] PSP ... Process SP

The following two C lines set PSP at three quarters, and MSP at the end of the stack segment. Note how the address at three quarters, i.e., `&_sstack` + $\frac{3}{4}\times$(`&_estack` − `&_sstack`), is calculated to avoid the overflow during the calculation.

```
__set_PSP(3 * ((uint32_t)&_estack / 4) + (uint32_t)&_sstack / 4);
__set_MSP((uint32_t)&_estack);
```

Two stacks are obtained (see Fig. 5.3). The stack that is currently used is determined by the operation mode and ASP bit in the CONTROL register. The main stack can overflow the process stack which can overflow the segment below.

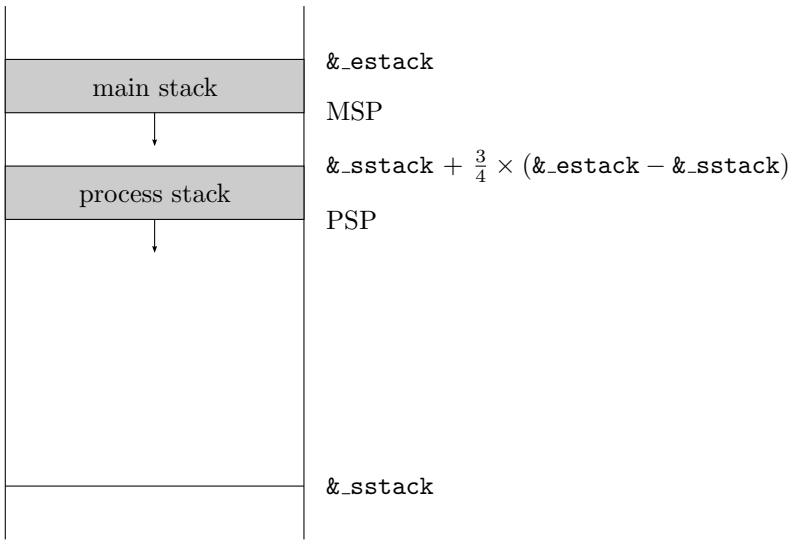


Figure 5.3: Stack

The `msr` instruction can also be used to set the CONTROL register. A CMSIS wrapper function `__set_CONTROL()` is available to set the register from C code. The declaration is:

```
void __set_CONTROL(uint32_t control);
```

CMSIS does not provide the CONTROL register bit masks. To run the thread code at unprivileged level using the PSP, the CONTROL register must be set by hand:

```
__set_CONTROL(0x00000003);
```

A privileged level is required to execute the `msr` instruction. The `msr` instruction must be followed by the instruction synchronization barrier explained in the next section. However, one should be careful when modifying the active SP value from a C function. The local variables may get lost, the return information surely is.

Synchronization barriers

The code instructions are not executed strictly one after another in a modern

processor. Speculative reads[12], out-of-order executions[13], buffered memory transfers[14], etc., take place due to the processor optimizations. The subsequent instruction may modify an already speculatively read memory location, or it can affect an ongoing memory transfer by reconfiguring the MPU, etc. A barrier between the previous and the next instruction is required in such cases. There are two kinds of barriers: data synchronization barrier, and instruction synchronization barrier.

The data synchronization barrier ensures that all memory transfers complete before the next instruction starts with execution. It is set by the `dsb` instruction. The `dsb` instruction can be accessed from the C code with the CMSIS `__DSB()` function. Its declaration is:

```
void __DSB(void);
```

The instruction synchronization barrier ensures that all instructions complete before the next instruction starts with execution. It is set by the `isb` instruction. The `isb` instruction can be accessed from the C code with the CMSIS `__ISB()` function. Its declaration is:

```
void __ISB(void);
```

As mentioned in the previous section, the `__ISB()` call is required after the `__set_PSP()`, `__set_MSP()`, or `__set_CONTROL()` function call.

MPU

The AT91SAM3X8E µC includes a MPU that is capable of trigging a system exception on a forbidden memory access [8, 18]. From the programmer's point of view, a system exception is equivalent to a regular interrupt. The regular interrupt request is handled by the NVIC[15] whereas the system exception request is usually issued by a system unit like MPU. A system exception request does not have a priority over the regular interrupt requests by default. The priority level is configurable for the most of the system exceptions.

MPU setup

The MPU defines up to eight configurable address space regions with configurable access rights, and a background region spreading over the entire 4GB address space. The regions are numbered from 0 to 7, the background region number is −1. When two or more regions overlap, e.g., any region always overlaps with the background region, the region with the highest number defines the access rights, e.g., the background region is always overruled. The MPU is disabled by default. The regions and their access rights become relevant when the MPU is enabled. If the background region access is granted, the access will be able only from the privileged level code.

The configuration of the MPU registers has to be done partially by hand since the CMSIS does not provide MPU manipulating functions. A brief summary of

[12]The processor reads the data or instruction, that may be needed in the future, beforehand.

[13]The processor does not wait idle during an execution of a costly several cycles long instruction. If the processor recognizes that the next instruction does not depend on the result of the previous one, it will proceed with executing the next instruction. Thus, the next instruction can complete before the previous one.

[14]Buffered memory transfer is a memory read or write that completes afterwards. In the mean time the subsequent instruction(s) is(are) already executing.

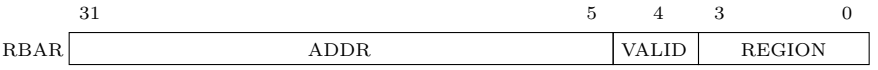
[15]NVIC ... Nested Vector Interrupt Controller

the MPU registers[16] follows.

MPU Control register

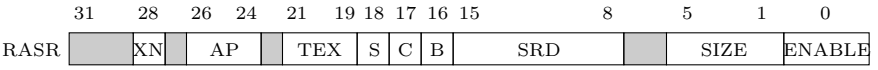
ENABLE	...	enable the MPU
HFNMIENA	...	enable MPU during hard fault[18] and NMI[19] handler
PRIVDEFENA	...	enable background region access, privileged level is required

MPU Region Base Address register



REGION	...	region number
VALID	...	region number valid (if zero, the settings refer to the current region)
ADDR	...	region base address (has to be aligned to the region size)

MPU Region Attribute and Size register



ENABLE	...	enable the region
SIZE	...	region size ($= 2^{\text{SIZE}+1}\text{B} \geq 32\text{B}$)
SRD	...	subregion disable bits[20]
B	...	bufferable bit[21]
C	...	cacheable bit[21]
S	...	shareable bit[21]
TEX	...	type extension[21]
AP	...	access permission

AP	privileged	unprivileged
`0b000`	no access	no access
`0b001`	RW[22]	no access
`0b010`	RW	RO
`0b011`	RW	RW
`0b101`	RO	no access
`0b11x`	RO	RO

XN	...	no execution (disable instruction fetches)

Stack fence

The stack overflow will be intercepted by the MPU if a no-access region is placed just before the stack boundary. If the code runs at unprivileged level and is using

[16] Only registers needed in this exercise are listed. For example, the RO[17] `TYPE` register is not listed. It contains an information about MPU regions and comes handy when writing a code portable across various versions of Cortex based µCs with different MPUs. For detailed explanation see [8].

[17] RO ... Read-Only

[18] A hard fault system exception occurs when an exception is requested inside the exception handler, or in case of a request that cannot be regularly managed.

[19] NMI ... Non-Maskable Interrupt

[20] Regions greater than 128B are divided into eight equal subregions.

[21] The flags define caching policy and shareability, i.e., the ability of the region to be shared by multiple processors. Since the T91SAM3X8E µC has a singe processor and no cache, the TEX, S, C and B flags should have the following values:

region in	TEX	S	C	B
Flash	`0b000`	0	1	0
SRAM	`0b000`	1	1	0
external SRAM*	`0b000`	1	1	1
pripherals	`0b000`	1	0	1

*not present on Arduino Due board

[22] RW ... Read-Write

the PSP with stack configuration from Fig. 5.3, the stack will grow towards the `&_sstack` boundary. A no-access region must be placed just before the stack floods the segment below (see Fig. 5.4). The fence, i.e., no-access region, occupies a part of the stack segment space, thus the maximum stack size is consequently slightly smaller. Since the code is running at unprivileged level, the no-access region is required for the unprivileged level only.

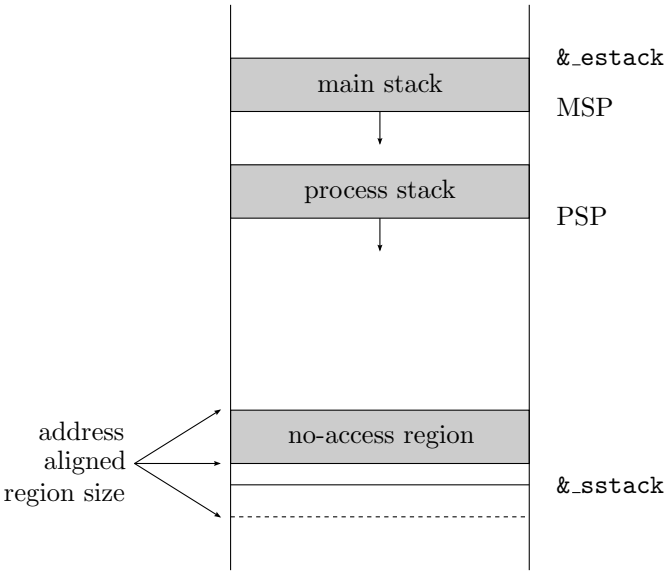


Figure 5.4: Stack fence

To configure a no-access region for the unprivileged level near the `&_sstack` address, the MPU registers must be properly set:

```
MPU->CTRL &= ~MPU_CTRL_ENABLE_Msk;[23]
MPU->RBAR = ((((uint32_t)&_sstack) & align_mask) + region_size) |
                                                MPU_RBAR_VALID_Msk | 1;[24]
MPU->RASR = 0x01060000 | size | MPU_RASR_ENABLE_Msk;[25]
```

First, the MPU is disabled for the configuration. The region base address and region number are set in the RBAR. The `&_sstack` address is striped of the least significant bits to get the first region size aligned address beneath the `&_sstack` (the dashed line in Fig. 5.4). E.g.: for the smallest 32B region size, the *align_mask* is `0xfffffffe`, thus using all the ADDR bits in the RBAR; for the 64B region size, the *align_mask* is `0xffffffc0`, etc. The region base address is one *region_size* above the obtained address. The region number is one. The access permissions and other region attributes are set in the RASR. The `0x01060000` constant defines a region in SRAM with no-access from the unprivileged code, i.e., XN = `0`, AP = `0b001`, TEX = `0b000`, S = `1`, C = `1`, B = `0`. The region size is $2^{\text{SIZE}+1}$B, e.g., *size* = `0x00000008` for 32B, *size* = `0x0000000a` for 64B, etc.

The background region access cannot be granted to the unprivileged code. With one no-access region configured and others disabled, the entire address space

[23] `MPU` is a CMSIS defined address where the MPU registers start. It is type casted into a pointer to a structure whose elements are 32-bit integers representing the MPU registers. Mask `MPU_CTRL_ENABLE_Msk` enables the MPU.

[24] `MPU_RBAR_VALID_Msk` is a region number valid mask.

[25] Mask `MPU_RASR_ENABLE_Msk` enables the specified region.

becomes unaccessible from the unprivileged code. To enable the unprivileged access outside the no-access region, a full-access region over the entire address space is required. The no-access region will stay in power and will overrule the full-access one if the full-access region number is smaller. The following lines configure a full-access region, i.e., XN = 0, AP = `0b011`, TEX = `0b000`, S = `1`, C = `1`, B = `1`, over the entire address space, i.e., base address = `0x00000000`, size = 4GB. The region number is zero.

```
MPU->RBAR = MPU_RBAR_VALID_Msk;
MPU->RASR = 0x0307003e | MPU_RASR_ENABLE_Msk;
```

The unused regions have to be disabled. To disable the i^{th} region, the RBAR and RASR must be appropriately set:

```
MPU->RBAR = MPU_RBAR_VALID_Msk | i;
MPU->RASR = 0;
```

The MPU starts acting when enabled. Since the entire-space full-access region is configured, enabling the background region access, i.e., setting the PRIVDEFENA bit in the CTRL register, is irrelevant.

```
MPU->CTRL = MPU_CTRL_ENABLE_Msk;
```

The MPU configuration modification may affect an ongoing memory transfer. Or the new configuration may not be used immediately after the MPU setup because of a speculative read or out-of-order execution. To avoid the first issue, the data synchronization barrier, i.e., `__DSB()`, is required before the MPU setup. Setting the data and instruction synchronization barriers, i.e., `__DSB()` and `__ISB()`, after the MPU setup eliminates the latter concern.

Memory management fault exception

An illegal memory access detected by the MPU causes the memory management fault. When enabled, the `MemManage_Handler()` exception handler is carried out. The memory management fault exception is enabled by setting the `MEMFAULTENA` bit in the `SHCSR`[26]:

```
SCB->SHCSR |= SCB_SHCSR_MEMFAULTENA_Msk;
```
[27]

The memory management fault exception has a configurable priority level[29]. The default priority level obtained at µC reset can be modified by the CMSIS `NVIC_SetPriority()` function[30]. The following `NVIC_SetPriority()` function call sets the priority level of the memory management fault exception to the lowest:

---

[26] SHCSR ... System Handler Control and State Register

[27] `SCB` is a CMSIS defined address where the SCB[28] registers start. It is type casted into a pointer to a structure whose elements are 32-bit integers representing the SCB registers. Mask `SCB_SHCSR_MEMFAULTENA_Msk` enables the memory management fault exception.

[28] SCB ... System Control Block

[29] Do not confuse system exception and interrupt priority levels with FreeRTOS task priorities.

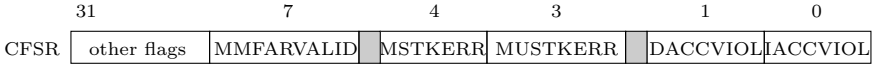
[30] Declared as `void NVIC_SetPriority(IRQn_Type IRQn, uint32_t priority);` where `IRQn_Type` is an enumerated type with interrupt/exception identifiers, `IRQn` is the identifier, and `priority` is a number from 0 (the highest) to 15 (the lowest priority).

```
NVIC_SetPriority(MemoryManagement_IRQn, 15);
```
[31]

Starting address of the `MemManage_Handler()` exception handler is defined in the exception table in the `sam/utils/cmsis/sam3x/source/templates/gcc/startup_sam3x.c` file. The handler is declared a as a weak symbol. A new `MemManage_Handler()` function definition in the application code overrides the weak declaration.

```
void MemManage_Handler(void) {
    ...
}
```

The reason for the memory management fault can be retrieved from the five bits in CFSR[32].



IACCVIOL	...	instruction access violation
DACCVIOL	...	data access violation
MUSTKERR	...	error on unstacking at exception return
MSTKERR	...	error on stacking at exception entry
MMFARVALID	...	the address in MMFAR[33] is valid

CFSR bit masks are not provided by the CMSIS. An individual bit has to be masked out by hand.

Suppose that the code runs in thread mode at unprivileged level using the PSP register. The memory management fault happens because the stack enters the no-access region. The cause of the fault is data access violation (DACCVIOL). The memory management fault starts the `MemManage_Handler()` exception handler. Before starting, the current µC state needs to be pushed onto the stack. The operating mode, privilege level and current SP register are changed when the exception handler is started, i.e. after storing the current µC state. In other words, the stacking at exception entry is performed in the original mode, privilege level and SP register (see Fig. 5.5). Since the stack is already in the no-access region, the stacking at exception entry fails. Another memory management fault happens, this time because of an error on stacking at exception entry (MSTKERR). The address in the MMFAR is the address accessed at the first memory management fault.

The two memory management faults cannot be separated when the access violation is caused by pushing data onto the stack, although the exception handler uses another stack, and no-access region does not apply at privileged level. A memory management fault because of stack access is always followed by a memory management fault because of an error on stacking at exception entry.

If for any reason a memory management fault occurs in the `MemManage_Handler()` exception handler, a hard fault will occur. Hard fault is a system exception with a fixed priority of -1, higher than any configurable priority (from 0 to 15). The `HardFault_Handler()` exception handler is started. It is again declared as a weak symbol and can be overridden in the application code.

[31] The enumerated value `MemoryManagement_IRQn` represents the memory management fault identifier.

[32] CFSR ... Configurable Fault Status Register, can be accessed from C code by `SCB->CFSR`.

[33] MMFAR ... Memory Management Fault Address Register contains the address whose access attempt caused the memory management fault. The register can be accessed from C code by `SCB->MMFAR`.

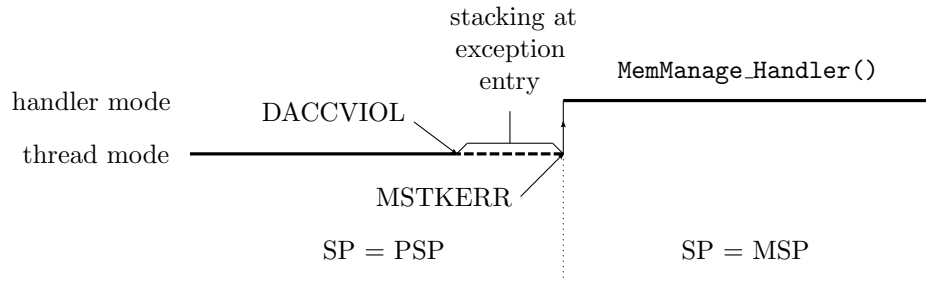


Figure 5.5: Stacking at exception entry

MPU configuration and memory management fault exception summary

The discussed stack fence is realized with proper configuration of the MPU regions. According to the explanation in the previous sections, the pseudo code for configuring the MPU is as follows:

enable memory fault exception
set memory fault exception priority
data barrier
disable MPU
configure MPU regions
enable MPU
set PSP and MSP
set unprivileged level and PSP as SP in thread mode
data barrier
instruction barrier

Note that by modifying the PSP and MSP registers, the current stack is lost. Therefore, the operation must not be performed in a function since the return address will be corrupted. Setting the PSP and MSP registers must be done from the `main()` function which never returns. Also, there must not be any local variables in the `main()` function since the PSP/MSP modification corrupts them too.

When a memory management fault occurs, the `MemManage_Handler()` exception handler is started. Since the µC state was not stored, the handler cannot properly return. Without an operating system, the memory management fault is irrecoverable. For the purpose of this exercise, the `MemManage_Handler()` code can print the cause of the fault and enter the endless loop. The pseudo code of the handler is as follows:

print memory management fault information regarding the CFSR bits
`while` forever
    do nothing

Measuring the amount of stack needed by a function call

To measure the amount of stack required by the `printf("Hello World!");` function call, configure a relatively wide no-access region, i.e., stack fence, to ensure intercepting the stack overflow. Set the current SP register and call the `printf()` function. The stack size, i.e., the difference between the initial SP register value and the edge of the stack fence, is large enough when the memory management fault does not occur.

To prevent stack overshooting, the no-access region has to be wide-enough. It has to be wider than the amount of stack required at function entry. If the no-access region is not wide enough, the SP register may be set beyond the fence to accommodate local variables. Yet the stack is not touched. If a local variable beyond the fence is used first, the undetected stack overflow will occur. The code can become unpredictable. Luckily, if the damage is not sincere, the stack overflow will be detected later when a variable inside the no-access region is used.

A hint: a `printf("Hello World!");` function call requires less than 2kB stack space.

Exercise 6

Stack management in FreeRTOS™

Write a FreeRTOS task that periodically prints the 'Hello World!' string to the `stdout` stream using the `printf()` function. Use the UART peripheral device as the stdio. Use FreeRTOS provided stack overflow detection mechanisms to measure the stack size required by the task. Verify the measured size using the FreeRTOS MPU port.

Explanation

Create a new empty project in the Eclipse working environment as explained in Exercise 1. Connect the Arduino Due board to the host PC over the Olimex ARM-USB-OCD-H interface as shown in Fig. 4.1. Configure the UART peripheral device and the stdio in serial mode as explained in Appendix A.

Stack high water mark

The stack size of the task is specified at task creation (see Exercise 2). The remaining stack space decreases when the stack grows, and increases when the stack shrinks. The minimum remaining stack space during the task execution is called the stack high water mark. It measures how close the task came to the stack size limit. A sizable stack high watermark indicates that the stack is too large, thus waisting the RAM, whereas a small stack high water mark warns about stack overflow.

The `uxTaskGetStackHighWaterMark()` function[1] [15] [16] can be used to obtain the stack high water mark. The declaration of the function is:

```
UBaseType_t uxTaskGetStackHighWaterMark( TaskHandle_t xTask );
```

The function returns the minimum remaining stack space, i.e., the stack high water mark[2], in number of words. The argument of the function is:

`xTask` ... task handle

Passing `NULL` argument is equivalent to passing a handle of the calling task. For instance, the following `uxTaskGetStackHighWaterMark()` function call returns the stack high watermark of the calling task:

```
UBaseType_t uxMark = uxTaskGetStackHighWaterMark(NULL);
```

[1] The `INCLUDE_uxTaskGetStackHighWaterMark` option must be set to one in the `src/FreeRTOSConfig.h` configuration file to make the `uxTaskGetStackHighWaterMark()` function available.

[2] A predefined byte is loaded into the entire stack space at task creation. The high water mark is a point in the stack space where the byte remained intact.

To use the `uxTaskGetStackHighWaterMark()` function, the `FreeRTOS.h` and `task.h` header files have to be included.

```
#include <FreeRTOS.h>
#include <task.h>
```

Periodic task

FreeRTOS measures the time in number of time slices, i.e., ticks. A serial number of the current tick can be obtained by the `xTaskGetTickCount()` function [15] [16]. The declaration of the function is:

```
TickType_t xTaskGetTickCount( void );
```

The function returns the total number of ticks since the FreeRTOS was started.

To wait for a specified point in time, the `vTaskDelayUntil()` function[3] [15] [16] can be used. The declaration of the function is:

```
void vTaskDelayUntil( TickType_t *pxPreviousWakeTime,
                      TickType_t xTimeIncrement );
```[4]

The function places the calling task into the blocked state until the specified time, i.e., `*pxPreviousWakeTime` + `xTimeIncrement`, is reached, and updates the initial time to the specified time, i.e., `*pxPreviousWakeTime` = `*pxPreviousWakeTime` + `xTimeIncrement`. If the specified time is in the past[5], the calling task will not be placed into the blocked state, only the initial time is updated. Therefore, if the task is denied the CPU time for some period of time, it will be able to compensate the missed cycles afterwards. The function arguments are:

`pxPreviousWakeTime`	...	pointer to initial time in the total number of ticks since FreeRTOS start
`xTimeIncrement`	...	time interval in number of ticks

A periodic task that executes at a predefined frequency rate can be implemented in the following way:

```
void periodic(void *arg) {
   TickType_t xLastWakeTime = xTaskGetTickCount();
   for(;;) {
         ...
      vTaskDelayUntil(&xLastWakeTime, pdMS_TO_TICKS(period_in_ms));
   }
}
```

To obtain some information about the stack space required by the `printf( "Hello World");` function call, printing stack high water mark is useful. The pseudo code of the periodic task is as follows:

[3]The `INCLUDE_vTaskDelayUntil` option must be set to one in the `src/FreeRTOSConfig.h` configuration file to make the `vTaskDelayUntil()` function available.

[4]Do not confuse the `vTaskDelayUntil()` function with the `vTaskDelay()` function from Exercise 2. The `vTaskDelayUntil()` function waits until an absolute time is reached, whereas the `vTaskDelay()` waits until an instant relative to when the function was called is reached.

[5]The tick count overflow error is avoided by verifying if the current tick count is inside the initial and specified time in a circular manner.

get current tick
`while` forever
    print 'Hello World!'
    obtain stack high water mark
    print the mark
    go to blocked state until specified time is reached

Do not use the `printf()` function to print the stack high water mark since it may require even more stack space than the `printf("Hello World");` call. Therefore, the stack high water mark value would not be the consequence of the 'Hello World!' call anymore. Use the `putchar()` function instead.

Create the periodic task and start the FreeRTOS scheduler as explained in Exercise 2.

FreeRTOS stack overflow detection mechanisms

The required task's stack size can be estimated by the stack high water mark value during the application development. However, FreeRTOS also provides two run time stack overflow detection mechanisms [15] [16]. The first one checks the SP value at every context switch. It assumes that if the stack overflow appears, it will happen at the context switch since the additional stack for saving the task's state is needed at that moment. Therefore, the first detection mechanism misses *temporary* stack overflows that happen during the task execution and are *over* before the context switch.

The second mechanism additionally implements the stack high water mark technique. A predefined byte is loaded into the entire stack space at task creation. Beside checking the SP, the mechanism also checks if the bytes near the stack boundary were overwritten. The checking procedure is again performed at every context switch.

The first mechanism is selected when the `configCHECK_FOR_STACK_OVERFLOW` option in the `src/FreeRTOSConfig.h` configuration file is set to one, and the second when to two. When a stack overflow is detected, the `vApplicationStackOverflowHook()` callback function will be called. The function has to be defined in the application code as:

```
void vApplicationStackOverflowHook( TaskHandle_t *pxTask,
                                    signed char *pcTaskName );
```

The received arguments are:
`pxTask` ... pointer to task that exceeded its stack
`pcTaskName` ... task name
When the `vApplicationStackOverflowHook()` callback function is called, the stack overflow already took place, thus making an unrecoverable damage. Recovering from the stack overflow is impossible. The `vApplicationStackOverflowHook()` function serves primarily for debugging purposes and ends in an endless loop[6].

[6] FreeRTOS kernel uses the system timer interrupt, i.e., the SysTick counter [8]. The priority level of the SysTick interrupt is set by the `configKERNEL_INTERRUPT_PRIORITY` option in the `src/FreeRTOSConfig.h` configuration file and should be the lowest possible. Interrupts that use the FreeRTOS API functions must have priority levels equal or below the `configMAX_SYSCALL_INTERRUPT_PRIORITY` option. The `configKERNEL_INTERRUPT_PRIORITY` level must never be configured above `configMAX_SYSCALL_INTERRUPT_PRIORITY`.

The `taskDISABLE_INTERRUPTS()` macro call disables the interrupts that have priority levels equal or below the `configMAX_SYSCALL_INTERRUPT_PRIORITY` option. In other words, if the rule above is applied, the macro will stop all the code directly related to the FreeRTOS. Therefore, the macro call is usually the first line of the `vApplicationStackOverflowHook()` function.

Do not confuse interrupt priority levels with FreeRTOS task priorities.

Since it cannot detect *temporary* overflows, the first detection mechanism cannot be used for measuring the `printf("Hello World");` stack requirements. The second mechanism, on the other hand, provides similar service as the stack high water mark measurement.

Stack overshoot

A stack overshoot appears when the required amount of stack at function entry is larger than the entire stack space. In that case, the SP register is immediately shifted outside the stack space. If the function returns before the context switch, and if, for any reason, only a memory near the SP is used, whereas the larger part of the required space is left untouched, the stack overflow will pass undetected. The SP is back inside the stack space and the part of the stack space monitored by the detection mechanism is intact. A stack overflow was caused because of stack overshooting. Under certain circumstances, the `printf("Hello World");` call can cause a stack overshoot.

MPU restricted task

To detect stack overshooting with greater certainty, the MPU should be used. Instead of `xTaskCreate()` (see Exercise 2), the `xTaskCreateRestricted()` function has to be used to create an MPU restricted task [16]. The function declaration is:

```
BaseType_t xTaskCreateRestricted(
                      TaskParameters_t *pxTaskDefinition,
                      TaskHandle_t *pxCreatedTask );
```

The function returns the `pdPASS` value on success. The function will fail if there is not enough heap memory available for allocating the task's stack and TCB. A newly created task is placed into the ready state. The function arguments are:

`pxTaskDefinition`	...	pointer to a structure with task definition
`pxCreatedTask`	...	pointer to where a handle to the created task is returned; if `NULL`, the handle is not returned

The `TaskParameters_t` structure is defined as:

```
typedef struct {
    TaskFunction_t pvTaskCode;
    char *pcName;
    uint16_t usStackDepth;
    void *pvParameters;
    UBaseType_t uxPriority;
    StackType_t *puxStackBuffer;
    MemoryRegion_t xRegions[ portNUM_CONFIGURABLE_REGIONS ];
} TaskParameters_t;[7]
```

The elements of the structure are[8]:

[7] `uint16_t` is a 16-bit unsigned integer type.
`StackType_t` is a 32-bit unsigned integer type.
`portNUM_CONFIGURABLE_REGIONS` is three for ARM Cortex-M3 based processors.

[8] Most of the elements are the same as the arguments of the `xTaskCreate()` function (see Exercise 2).

`pvTaskCode`	...	pointer to task function
`pcName`	...	task name
`usStackDepth`	...	task's stack size in words
`pvParameters`	...	pointer passed to the task as an argument
`uxPriority`	...	task priority[9]
`puxStackBuffer`	...	pointer to task's stack region
`xRegions`	...	array of additional MPU regions

The `puxStackBuffer` pointer defines starting address of the task's stack. Since stack of an MPU restricted task is MPU controlled, it has to be an individual MPU region. That means, that its size must be a power of two, and its starting address must be region size aligned (see Exercise 5). In C programming language, a stack space that meet the above requirements can be declared as an aligned global array placed into the 'MPU controlled stacks' SRAM segment (see Fig. 5.1):

```
StackType_t stack[size]
            __attribute__((aligned(sizeof(StackType_t) * size)))
            __attribute__((section(".mpu_stacks")));
```

The `xRegions` array defines task specific MPU regions[10] that are reconfigured at context switch. Each region is described by the `MemoryRegion_t` structure defined as:

```
typedef struct {
    void *pvBaseAddress;
    unsigned long ulLengthInBytes;
    unsigned long ulParameters;
} MemoryRegion_t;
```

The elements of the structure are:

`pvBaseAddress`	...	region starting address
`ulLengthInBytes`	...	region length in bytes
`ulParameters`	...	region access permission

For instance, an MPU restricted task with 512 bytes of stack and RW access to the global data in SRAM (see Fig. 5.1) is created with the following code:

```
extern uint32_t __SRAM_segment_start__[];
extern uint32_t _Globals_Region_Size[];
StackType_t pxStack[128] __attribute__((aligned(512)))
                         __attribute__((section(".mpu_stacks")));
```

[9] Tasks execute their code at an unprivileged level (see Exercise 5) by default. A task created by the `xTaskCreateRestricted()` function can be elevated to the privileged level of code execution by setting the `portPRIVILEGE_BIT` bit in the `uxPriority` element, e.g., `uxPriority = portPRIVILEGE_BIT|2` defines a task running at privileged level with priority two.

[10] By default, the MPU enables RO access to Flash memory, RW access to peripherals [8], and RW access to FreeRTOS privileged data in SRAM. The privilege level of execution is required to access the FreeRTOS privileged functions in Flash and privileged data in SRAM (see Fig. 5.1). The background region access is enabled. The default MPU settings do no change at context switch.

Besides the default MPU settings, a task has its own task specific MPU regions. To an ordinary task, RW access to the entire SRAM is granted. On the other hand, to an MPU restricted task, RW access only to its stack region is granted. To enable an MPU restricted task to access addresses outside the stack region, additional MPU regions may be specified in the `xRegions` array. The task specific MPU regions are reconfigured at context switch.

```
TaskParameters_t xDef = {tsk, "Task", 128, NULL, 1, pxStack,
    {{__SRAM_segment_start__,
      (uint32_t)_Globals_Region_Size,
      portMPU_REGION_READ_WRITE[11]},
     {0, 0, 0},
     {0, 0, 0}}};[12]
        ...
xTaskCreateRestricted(&xDef, NULL);
```

The task is coded in the `tsk()` function, its name is `"Task"`, has priority one, and runs at unprivileged level. The `NULL` pointer is passed as an argument and no task handle is returned. One additional MPU region enabling RW access to the global data space located at the start of the SRAM is specified for the task. For that purpose, the `__SRAM_segment_start__` and `_Globals_Region_Size` symbols defined in the `sam/utils/linker_scripts/sam3x/sam3x8/gcc/flash.ld` linker script are declared as extern sysmbols.

When an MPU restricted task attempts to access an illegal memory address, the MPU raises the memory management fault. The `MemManage_Handler()` exception handler is called. The handler is enabled and configured during the FreeRTOS start. Therefore, only a new definition of the `MemManage_Handler()` function is required in the application code (see Exercise 5).

Create the MPU restricted periodic task and start the FreeRTOS scheduler as explained in Exercise 2.

Suggested stack experiments

The exercise can be performed in many different ways. The course of action suggested here tries to highlight all the described techniques:

- Write a periodic 'Hello World!' task. Add stack high water mark printing. When creating the task, make sure the stack is large enough.
- Measure the required stack size by observing the stack high water mark while gradually decrease the size.
- Set the stack size slightly below the required size and test the first and the second FreeRTOS stack overflow detection mechanism.
- Set the stack size far below the required size and test the first and the second FreeRTOS stack overflow detection mechanism again.
- Create the task as an MPU restricted task. Test stack overflow error with various stack sizes.

[11] The `portMPU_REGION_ READ_WRITE` access permission mask defines RW access for privileged and unprivileged level.

[12] Note that the `main()` function stack is reused after the `vTaskStartScheduler()` function is called. Therefore, the `TaskParameters_t` structure, i.e., `xDef`, must be global. If the structure is local, it will become corrupted after the `vTaskStartScheduler()` call. This must not happen since the `xTaskCreateRestricted()` function does not make copies of the structure elements.

Exercise 7

Heap management in FreeRTOS™

Measure memory overhead per allocation and minimum allocation block size for FreeRTOS provided heap structure.

Write a FreeRTOS task that all the time allocates and releases memory blocks on the heap. The allocations should have random sizes and should be released in a random order. The number of currently allocated memory blocks should stay constant. The task should also measure the amount of time needed to perform an allocation, and the amount of time needed to perform a release. Use FreeRTOS provided best and first fit memory allocation algorithms. To monitor the heap status, write another FreeRTOS task that periodically reports the current and maximum allocation and release time, current number of free bytes on the heap, current fragmentation rate of the heap, and current and the smallest size of the free block at the top of the heap. Write a complementary code and modify the existing FreeRTOS code to obtain the fragmentation rate and the top of the heap free block size.

Explanation

Create a new empty project in the Eclipse working environment as explained in Exercise 1. Connect the Arduino Due board to the host PC over the Olimex ARM-USB-OCD-H interface as shown in Fig. 4.1. Configure the UART peripheral device and the stdio in serial mode, and the TC[1] device as a free running counter as explained in Appendix A.

Memory allocation algorithms

A heap is declared as a global array in the `src/main.c` source file [13]. The total size of the heap is set by the `configTOTAL_HEAP_SIZE` option in the `src/FreeRTOSConfig.h` configuration file. Because the heap's declaration aligns the array in order the MPU can be used (see Exercise 5), the `configTOTAL_HEAP_SIZE` option has to be a power of two. Without the alignment, any size is acceptable.

The FreeRTOS provides several memory allocation algorithms to reserve the memory on the heap. Two of them, the best fit algorithm, and the first fit algorithm are considered in this exercise.

In the best fit algorithm, the free memory blocks are ordered from the smallest to the largest. The best fit algorithm searches for the first, i.e., the smallest free block that can fit the required size. The block found is therefore equal or larger than the required size. If the block is larger than the required size, it will

[1] TC ... Timer Counter

be divided into two blocks thus leaving the surplus memory free. Two adjacent memory blocks are not merged when released. This makes the best fit algorithm more fragmentation prone. The best fit algorithm is shown in Fig. 7.1.

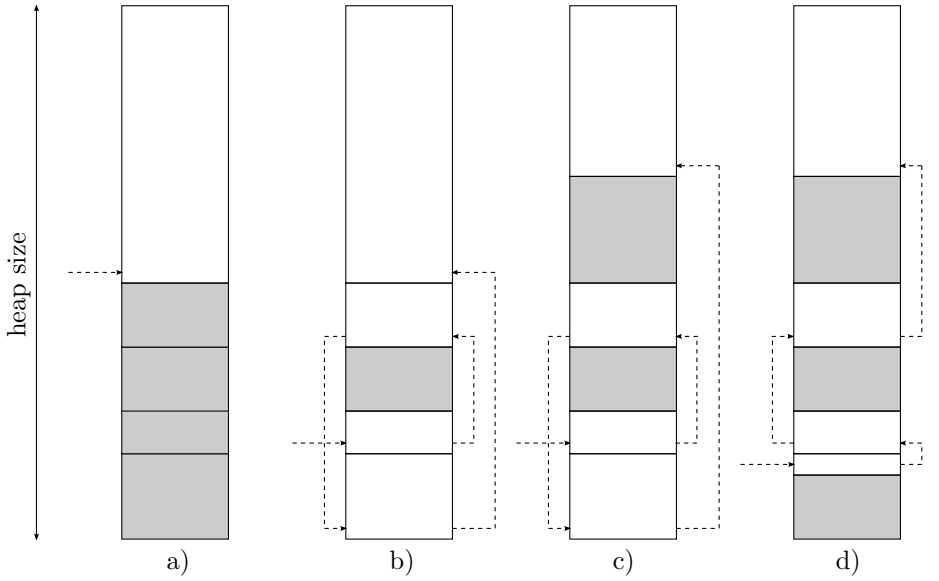


Figure 7.1: Best fit algorithm (dashed arrows denote free memory block order)
a) four memory blocks are allocated
b) the first, the second and the fourth block are released; adjacent blocks are not merged
c) a large block is allocated; although the first and the second block combined are large enough, a new block is formed
d) a block smaller than the first one is allocated; the first block is divided

In the first fit algorithm, the free memory blocks are address ordered. The first fit algorithm searches for the first memory block that can fit the required size. The block found is therefore equal or larger than the required size. If the block is larger than the required size, it will be divided into two blocks thus leaving the surplus memory free. Since the blocks are address ordered, the adjacent memory blocks are easily merged when released. This makes the first fit algorithm less fragmentation prone. The first fit algorithm is shown in Fig. 7.2.

The best fit and the first fit memory allocation algorithms are implemented in the `heap_2.c` and `heap_4.c` source files, respectively. Both reside in the `thirdparty/FreeRTOS/Source/portable/MemMang` subdirectory. The implementation file of the algorithm that is to be used has to be renamed into `heap.c`. The first fit algorithm is used by default.

Allocating and releasing a memory block on the heap

The well-known `malloc()` and `free()` standard C functions are not available by default. The memory allocation API is in many cases not provided on embedded systems. However, the FreeRTOS provides its own versions the `malloc()` and `free()` functions called `pvPortMalloc()` and `vPortFree()` [15] [16]. The declaration of the `pvPortMalloc()` function is:

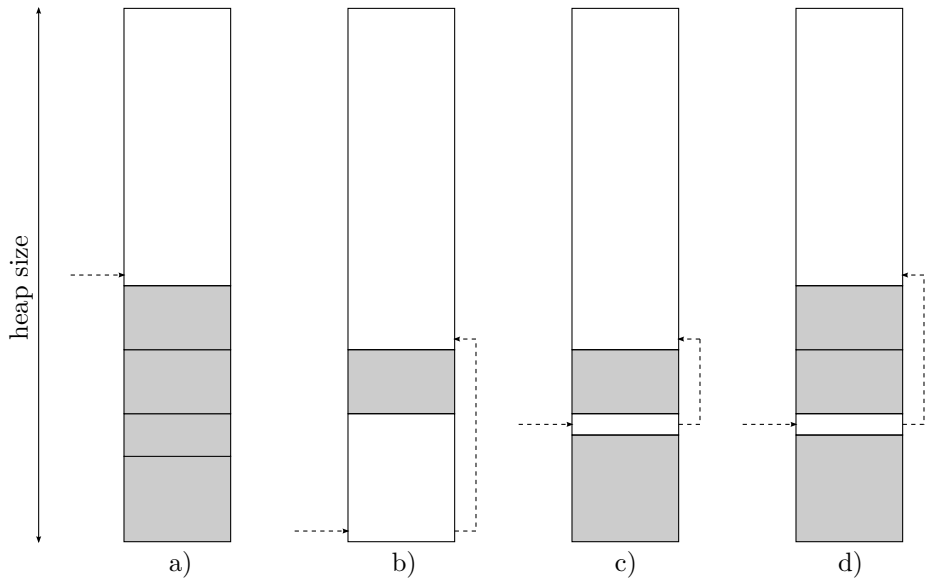


Figure 7.2: First fit algorithm (dashed arrows denote free memory block order)
a) four memory blocks are allocated
b) the first, the second and the fourth block are released; adjacent blocks are merged
c) a new block is allocated; the first block is large enough and is divided
d) another new block is allocated; since it does not fit into the remainder, a new block is formed

```
void *pvPortMalloc( size_t xSize );
```[2]

The function returns a pointer to the allocated memory block. If there is not enough heap memory available, the allocation will fail and `NULL` pointer will be returned. The argument of the function is:

`xSize` ... memory block size in bytes

The following `pvPortMalloc()` function call allocates 1kB memory block on heap:

```
void *pvBlock = pvPortMalloc(1024);
```

When a FreeRTOS provided memory allocation scheme is used, the `vApplicationMallocFailedHook()` callback function will be called on `pvPortMalloc()` failure if the `configUSE_MALLOC_FAILED_HOOK` option is set to one in the `src/FreeRTOSConfig.h` configuration file. The function has to be defined in the application code as:

```
void vApplicationMallocFailedHook(void);
```

An allocated memory block is released by the `vPortFree()` function. The declaration of the function is:

```
void vPortFree( void *pv );
```

The argument of the function is:

[2] `size_t` is defined in `stddef.h` as a long (32-bit) unsigned integer type.

`pv` ... pointer to memory block to be released

The following `vPortFree()` function call releases a memory block at `pvBlock` address:

```
vPortFree(pvBlock);
```

Free heap size

A current sum of free memory block sizes is obtained by the `xPortGetFreeHeapSize()` function [15]. Its declaration is:

```
size_t xPortGetFreeHeapSize( void );
```

For instance, the following `xPortGetFreeHeapSize()` function call returns a number of free bytes on the heap at the moment of call:

```
size_t xBytes = xPortGetFreeHeapSize();
```

The information about the free heap size is useful during the application development to estimate the application required heap size.

Allocation overhead and minimum block size

Each allocated memory block on the heap carries some information about itself. Therefore, it is slightly larger than the required size. The additional space is allocation overhead.

Requesting an allocation of an extremely small block can lead to heap fragmentation. Therefore, the allocation algorithm reserves a minimum size block on a below minimum size request.

To find out the allocation overhead and the minimum block size, the `xPortGetFreeHeapSize()` function in combination with testing allocation requests can be used. The pseudo code is:

initialize the heap[3]
obtain initial free heap size
allocate memory block
allocation overhead = initial free heap size − current free heap size −
− allocated memory block size
release memory block
allocate 1B
minimum block size = initial free heap size − current free heap size −
− allocation overhead

Allocation algorithm testing task

The testing task randomly allocates and releases memory on the heap. A set of random sized allocations is created at the beginning. In every iteration of an endless loop, a random allocation from the set is released, and a new random sized allocation is added to the set. A pseudo code of the task is:

[3]The heap initialization burns some bytes on the heap. It is preformed at the first `pvPortMalloc()` function call. Therefore, dummy `pvPortMalloc()` and `vPortFree()` calls do the trick.

```
for number of allocations
        allocate random size block
while forever
        enter critical section
        start time measurement
        release random allocation
        stop time measurement
        exit critical section
        calculate release time and save it into release global variable
        if the release time is the longest so far
                save it into release_max global variable
        enter critical section
        start time measurement
        allocate random size block
        stop time measurement
        exit critical section
        calculate allocation time and save it into allocate global variable
        if the allocation time is the longest so far
                save it into allocate_max global variable
```

The `rand()` standard C function returns a random integer value between zero and `RAND_MAX` macro[4]. It can be used for random number generation in the algorithm above.

The release and allocation time measurements must not be interrupted. Therefore, they are made atoms (see Exercise 3). See Appendix A for time measurement implementation.

Fragmentation rate and top of the heap free block

There is no universal definition of the heap fragmentation rate. For the purpose of this exercise, the definition

$$\text{fragmentation rate} = \frac{\text{free heap size} - \text{max free block size}}{\text{free heap size}}\ 100\%$$

is used to express the fragmentation rate in percentage.

The smallest size of the free block at the top of the heap during the application execution provides similar information about the heap size as the high water mark about the stack size (see Exercise 6). For the first fit allocation algorithm, the smallest top block size measures how close the heap came to an allocation failure[5]. A sizable smallest top block size indicates that the heap is too large, thus waisting the RAM, whereas a tiny smallest top block size warns about allocation failure.

Since the free blocks are not address ordered, the measurement is not necessarily valid when the best fit algorithm is used. Until the top block is also the largest, it will be the last in the free blocks chain and the measurement will be valid for the best fit algorithm too[5].

To get the fragmentation rate, and the current and the smallest size of the free block at the top of the heap, three additional memory management API functions are required. For the best fit allocation algorithm, the following code has to be inserted into the `thirdparty/FreeRTOS/Source/portable/MemMang/heap.c` source file[6]:

[4] The `RAND_MAX` constant equals to 2147483647.

[5] Supposing that the byte at the top of the heap is never allocated.

[6] Lines to be inserted are marked with *ins:*.

```
    static size_t xFreeBytesRemaining = configADJUSTED_HEAP_SIZE;
ins: static size_t xMinLastFreeBlockSize = configADJUSTED_HEAP_SIZE;
        ...
    void *pvPortMalloc( size_t xWantedSize ) {
            ...
        if( pxBlock != &xEnd ) {
ins:        size_t lastSize
                ...
            xFreeBytesRemaining -= pxBlock->xBlockSize;
ins:        lastSize = xPortGetLastFreeBlockSize();
ins:        if(lastSize < xMinLastFreeBlockSize)
ins:            xMinLastFreeBlockSize = lastSize;
        }
            ...
    }
        ...
ins: size_t xPortGetMaxFreeBlockSize(void) {
ins:    return xPortGetLastFreeBlockSize();
ins: }
ins: size_t xPortGetLastFreeBlockSize(void) {
ins:    BlockLink_t *pxBlock;
ins:    size_t lastSize = 0;
ins:    for(pxBlock = xStart.pxNextFreeBlock; pxBlock != &xEnd;
ins:        pxBlock = pxBlock->pxNextFreeBlock)
ins:            lastSize = pxBlock->xBlockSize;
ins:    return lastSize;
ins: }
ins: size_t xPortGetMinLastFreeBlockSize(void) {
ins:    return xMinLastFreeBlockSize;
ins: }
```

For the first fit algorithm, the code to be inserted is slightly different:

```
    static size_t xMinimumEverFreeBytesRemaining = 0U;
ins: static size_t xMinLastFreeBlockSize = 0U;
        ...
    void *pvPortMalloc( size_t xWantedSize ) {
            ...
        if( pxBlock != pxEnd ) {
ins:        size_t lastSize
                ...
            xFreeBytesRemaining -= pxBlock->xBlockSize;
ins:        lastSize = xPortGetLastFreeBlockSize();
ins:        if(lastSize < xMinLastFreeBlockSize)
ins:            xMinLastFreeBlockSize = lastSize;
                ...
        }
            ...
    }
        ...
ins: size_t xPortGetMaxFreeBlockSize(void) {
ins:    BlockLink_t *pxBlock;
ins:    size_t maxSize = 0;
ins:    for(pxBlock = xStart.pxNextFreeBlock; pxBlock != pxEnd;
```

```
ins:          pxBlock = pxBlock->pxNextFreeBlock)
ins:              if(pxBlock->xBlockSize > maxSize)
ins:                  maxSize = pxBlock->xBlockSize;
ins:      return maxSize;
ins: }
ins: size_t xPortGetLastFreeBlockSize(void) {
ins:      BlockLink_t *pxBlock;
ins:      size_t lastSize = 0;
ins:      for(pxBlock = xStart.pxNextFreeBlock; pxBlock != pxEnd;
ins:          pxBlock = pxBlock->pxNextFreeBlock)
ins:              lastSize = pxBlock->xBlockSize;
ins:      return lastSize;
ins: }
ins: size_t xPortGetMinLastFreeBlockSize(void) {
ins:      return xMinLastFreeBlockSize;
ins: }
          ...
      static void prvHeapInit( void ) {
              ...
          xFreeBytesRemaining = pxFirstFreeBlock->xBlockSize;
ins:      xMinLastFreeBlockSize = pxFirstFreeBlock->xBlockSize;
              ...
      }
```

The functions `xPortGetMaxFreeBlockSize()`, `xPortGetLastFreeBlockSize()`, and `xPortGetMinLastFreeBlockSize()` return the sizes of the currently largest free block[7], of the last free block[8], and of the smallest last free block[8] during the application execution, respectively.

The functions have to be declared throughout the various FreeRTOS source files. The following lines have to be inserted into the specified files:

in `thirdparty/FreeRTOS/Source/include/mpu_prototypes.h` :

```
size_t MPU_xPortGetMaxFreeBlockSize(void);
size_t MPU_xPortGetLastFreeBlockSize(void);
size_t MPU_xPortGetMinLastFreeBlockSize(void);
```

in `thirdparty/FreeRTOS/Source/include/mpu_wrappers.h` :

```
#define xPortGetMaxFreeBlockSize \
        MPU_xPortGetMaxFreeBlockSize
#define xPortGetLastFreeBlockSize \
        MPU_xPortGetLastFreeBlockSize
#define xPortGetMinLastFreeBlockSize \
        MPU_xPortGetMinLastFreeBlockSize
```

in `thirdparty/FreeRTOS/Source/include/portable.h` :

```
size_t xPortGetMaxFreeBlockSize(void) PRIVILEGED_FUNCTION;
size_t xPortGetLastFreeBlockSize(void) PRIVILEGED_FUNCTION;
size_t xPortGetMinLastFreeBlockSize(void) PRIVILEGED_FUNCTION;
```

[7]For the best fit algorithm, the `xPortGetMaxFreeBlockSize()` function is just a wrapper to the `xPortGetLastFreeBlockSize()` function since the last free block is also the largest.

[8]If the byte at the top of the heap is never allocated, the last block in the chain of free blocks will be at the top of the heap.

in `thirdparty/FreeRTOS/Source/portable/Common/mpu_wrappers.c` :

```
size_t MPU_xPortGetMaxFreeBlockSize(void) {
    BaseType_t xRunningPrivileged = xPortRaisePrivilege();
    size_t xReturn = xPortGetMaxFreeBlockSize();
    vPortResetPrivilege(xRunningPrivileged);
    return xReturn;
}
size_t MPU_xPortGetLastFreeBlockSize(void) {
    BaseType_t xRunningPrivileged = xPortRaisePrivilege();
    size_t xReturn = xPortGetLastFreeBlockSize();
    vPortResetPrivilege(xRunningPrivileged);
    return xReturn;
}
size_t MPU_xPortGetMinLastFreeBlockSize(void) {
    BaseType_t xRunningPrivileged = xPortRaisePrivilege();
    size_t xReturn = xPortGetMinLastFreeBlockSize();
    vPortResetPrivilege(xRunningPrivileged);
    return xReturn;
}
```

Monitoring task

The monitoring task is a periodic task (see Exercise 6) continuously reporting the current status of the heap. The task wakes up on regular time intervals, obtain and print the requested heap parameters, and goes back into blocked state until the next time period. The priority of the monitoring task must be above the allocation algorithm testing task, so that the monitoring task can preempt the testing task. With heap information acquiring functions available, the monitoring task is straightforward. The pseudo code of the task is as follows:

get current tick
`while` forever
    print current and maximum allocation time
        (i.e., *allocate* and *allocate_max* global variables)
    print current and maximum release time
        (i.e., *release* and *release_max* global variables)
    obtain and print number of free bytes on the heap
    obtain size of the largest free block
    calculate and print fragmentation rate
    obtain and print size of the last free block (i.e., top of the heap block)
    obtain and print the smallest size of the last free block
    go to blocked state until specified time is reached

Create the allocation algorithm testing task, the monitoring task, and start the FreeRTOS scheduler as explained in Exercise 2.

Exercise 8

Deadlocks

Create an application able to simulate events like starting or stopping a specified task, and taking or giving a specified semaphore. Task starting and stopping events should be given in a predefined scenario defining start and stop time points of the tasks involved. A semaphore should be taken or given by pressing the corresponding button key on the external board. Use the external board LEDs to indicate which task is running. Report the semaphore events to the stdio. Produce different sequences of events leading to various deadlocks.

Explanation

Create a new empty project in the Eclipse working environment as explained in Exercise 1. Connect the Arduino Due board to the host PC over the Olimex ARM-USB-OCD-H interface, and to the external board button keys and LEDs, as shown in Fig. 2.1. Configure the UART peripheral device and the stdio in serial mode. Configure the key and LED pins as input and output GPIO pins. The pull-up resistors are required on input pins, i.e., keys. To properly detect the key pressing, the debouncing filters are also needed. See Appendix A for UART and GPIO pin configuration and usage.

Application organization

The application can be coded in different ways. The realization described here is of course not the only one possible. The proposed application organization is shown in Fig. 8.1. A master FreeRTOS task starts and stops other tasks following the predefined scenario. The master task also monitors the external board button keys and sends key events to a key queue. When the currently running task receives a key event, it will perform a corresponding semaphore action. A message about the action is sent to an UART queue. The tasks also alternatively drive the external board LEDs to indicate which one is currently running. When the master task receives the message from the UART queue, it will dispatch the message further to stdio to report a semaphore event to the user. If there are no tasks running, the Idle task will turn all the LEDs off.

Queue

Queue is a FIFO buffer providing a communication mechanism. A queue is created by the `xQueueCreate()` function[1] [15] [16]. The declaration of the function is:

[1] The `configSUPPORT_DYNAMIC_ALLOCATION` option must not be set to zero in the `src/FreeRTOSConfig.h` configuration file to make the `xQueueCreate()` and `xSemaphoreCreateBinary()` functions available.

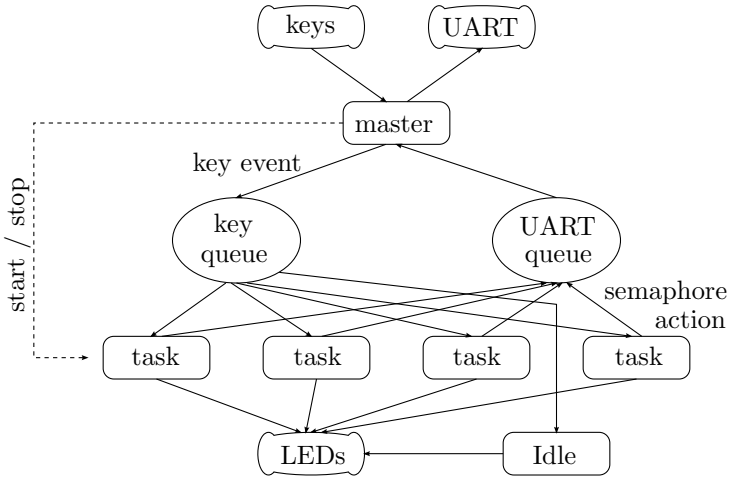


Figure 8.1: Application organization

```
QueueHandle_t xQueueCreate( UBaseType_t uxQueueLength,
                            UBaseType_t uxItemSize );
```[2]

The function returns a handle to the created queue on success. The function will return `NULL` if there is not enough heap memory available to allocate the queue buffer. The function arguments are:

`uxQueueLength` ... buffer space in number of items
`uxItemSize` ... one item size in bytes

For instance, the following `xQueueCreate()` function call creates a queue that is five unsigned 32-bit integers long:

```
QueueHandle_t xBuf = xQueueCreate(5, sizeof(uint32_t));
```

An item is written or sent to a queue by the `xQueueSend()` function [15] [16]. The declaration of the function is:

```
BaseType_t xQueueSend( QueueHandle_t xQueue,
                       void *pvItemToQueue,
                       TickType_t xTicksToWait );
```

The function returns the `pdPASS` value on success. The function will fail if the queue buffer is full and the item cannot be sent. The function arguments are:

`xQueue` ... queue handle
`pvItemToQueue` ... pointer to item to be sent
`xTicksToWait` ... maximum number of ticks to wait[3]

For instance, the following `xQueueSend()` function call sends an unsigned 32-bit integer to the `xBuf` queue, and returns immediately:

```
xQueueSend(xBuf, &ulNum, 0);
```

[2] `QueueHandle_t` is a void pointer type.

[3] If the queue is full, the item cannot be sent to it. The calling task is placed into blocked state and waits there for a queue space to become available for up to `xTicksToWait` ticks. If the `xTicksToWait` argument is set to `portMAX_DELAY`, the task is suspended for an indefinite time, until a queue space is available. In this case, the `INCLUDE_vTaskSuspend` option has to be set to one in the `src/FreeRTOSConfig.h` configuration file.

The `ulNum` variable is an unsigned 32-bit integer declared as `uint32_t ulNum;`.

An item is read or received from a queue by the `xQueueReceive()` function [15] [16]. The declaration of the function is:

```
BaseType_t xQueueReceive( QueueHandle_t xQueue,
                          void *pvBuffer,
                          TickType_t xTicksToWait );
```

The function returns the `pdPASS` value on success. The function will fail if the queue buffer is empty and the item cannot be received. The function arguments are:

`xQueue`	...	queue handle
`pvBuffer`	...	pointer to where received item is written
`xTicksToWait`	...	maximum number of ticks to wait[4]

For instance, the following `xQueueReceive()` function call receives an unsigned 32-bit integer from the `xBuf` queue, and returns immediately:

```
xQueueReceive(xBuf, &ulNum, 0);
```

The `ulNum` variable is an unsigned 32-bit integer declared as `uint32_t ulNum;`.

To use the `xQueueCreate()`, `xQueueSend()` and `xQueueReceive()` functions, the `FreeRTOS.h` and `queue.h` header files have to be included.

```
#include <FreeRTOS.h>
#include <queue.h>
```

Binary semaphore

Binary semaphore is a variable used for synchronization. A binary semaphore can be taken (locked), or given (unlocked). A task taking a semaphore, already taken by another task, must wait until the semaphore is given back, thus synchronizing with the other task. A binary semaphore is created by the `xSemaphoreCreateBinary()` function[1] [15] [16]. The declaration of the function is:

```
SemaphoreHandle_t xSemaphoreCreateBinary( void );
```[5]

The function returns a handle to the created binary semaphore on success. The semaphore is created in a taken (locked) state. The function will return `NULL` if there is not enough heap memory available to allocate the semaphore data structure. The following `xSemaphoreCreateBinary()` function call creates a binary semaphore:

```
SemaphoreHandle_t xSem = xSemaphoreCreateBinary();
```

A binary semaphore is given (unlocked) by the `xSemaphoreGive()` function [15] [16]. The declaration of the function is:

[4] If the queue is empty, the item cannot be received from it. The calling task is placed into blocked state and waits there for a queue item to arrive for up to `xTicksToWait` ticks. If the `xTicksToWait` argument is set to `portMAX_DELAY`, the task is suspended for an indefinite time, until a queue item arrives. In this case, the `INCLUDE_vTaskSuspend` option has to be set to one in the `src/FreeRTOSConfig.h` configuration file.

[5] `SemaphoreHandle_t` is a void pointer type.

```
BaseType_t xSemaphoreGive( SemaphoreHandle_t xSemaphore );
```

The function returns the `pdPASS` value on success. The function will fail if the semaphore is already given. The function argument is:
`xSemaphore` ... semaphore handle
For instance, the following `xSemaphoreGive()` function call gives the `xSem` semaphore:

```
xSemaphoreGive(xSem);
```

A binary semaphore is taken (locked) by the `xSemaphoreTake()` function [15] [16]. The declaration of the function is:

```
BaseType_t xSemaphoreTake( SemaphoreHandle_t xSemaphore,
                           TickType_t xTicksToWait );
```

The function returns the `pdPASS` value on success. The function will fail if the semaphore is already taken. The function arguments are:
`xSemaphore` ... semaphore handle
`xTicksToWait` ... maximum number of ticks to wait[6]
For instance, the following `xSemaphoreTake()` function call takes the `xSem` semaphore. In case the semaphore is already taken, the calling task is suspended for an indefinite time until the semaphore is given:

```
xSemaphoreTake(xSem, portMAX_DELAY);
```

To use the `xSemaphoreCreateBinary()`, `xSemaphoreGive()` and `xSemaphore Take()` functions, the `FreeRTOS.h` and `semphr.h` header files have to be included.

```
#include <FreeRTOS.h>
#include <semphr.h>
```

FreeRTOS software timer

The master task from Fig. 8.1 must have a higher priority than the other tasks. Therefore, it has to be a periodic task (see Exercise 6) regularly placing itself into blocked state in order to make the CPU available to others. A periodic task can be avoided if a FreeRTOS provided software timer is used. A software timer regularly calls a finite timer callback function, thus providing the same functionality as a periodic task.

Software timers are provided by a special Timer Service task. The task is created at the FreeRTOS scheduler start, likewise the Idle task. The Timer Service task priority and stack depth in words are set by the `configTIMER_TASK_PRIORITY` and `configTIMER_TASK_STACK_DEPTH` options in the `src/FreeRTOSConfig.h` configuration file. Commands to the software timers are passed to the Timer Service task by a special Timer Command queue. The length of the queue is set by the `configTIMER_QUEUE_LENGTH` option.

[6] If the semaphore is already taken (locked), the calling task cannot take it. The calling task is placed into blocked state and waits there for the semaphore to be given (unlocked) for up to `xTicksToWait` ticks. If the `xTicksToWait` argument is set to `portMAX_DELAY`, the task is suspended for an indefinite time, until the semaphore is given. In this case, the `INCLUDE_vTaskSuspend` option has to be set to one in the `src/FreeRTOSConfig.h` configuration file.

Before usage, a software timer needs to be created by the `xTimerCreate()` function[7] [15] [16]. The declaration of the function is:

```
TimerHandle_t xTimerCreate( char *pcTimerName,
                    TickType_t xTimerPeriod,
                    UBaseType_t uxAutoReload,
                    void *pvTimerID,
                    TimerCallbackFunction_t pxCallbackFunction );
```
[8]

The function returns a handle to a newly created timer. The timer is initially stopped. If there is not enough heap memory available to allocate the timer data structure, the function will fail and `NULL` will be returned. The arguments of the function are:

`pcTimerName`	...	timer name
`xTimerPeriod`	...	timer period in ticks
`uxAutoReload`	...	reload flag[9]
`pvTimerID`	...	pointer to timer identifier
`pxCallbackFunction`	...	pointer to timer callback function

For instance, the function call

```
TimerHandle_t xTimer =
  xTimerCreate("Master", pdMS_TO_TICKS(100), pdTRUE, NULL, master);
```

creates a software timer named `"Master"` with no identification. When started, the timer expires repeatedly on every 100ms. The callback function declared as `void master(TimerHandle_t);` is called by the Timer Service task on every expiration. The `xTimer` handle to a newly created software timer is returned.

A stopped timer is started by the `xTimerStart()` function[10] [15] [16]. The function sends a command to start the specified software timer to the Timer Service task using the Timer Command queue. The declaration of the function is:

```
BaseType_t xTimerStart( TimerHandle_t xTimer,
                        TickType_t xTicksToWait );
```

The function returns the `pdPASS` value on success. If the software timer was already started, it is reset. The function will fail if the Timer Command queue is full and the command cannot be sent. The function arguments are:

---

[7] The `configUSE_TIMERS` option must be set to one and the `configSUPPORT_DYNAMIC_ALLOCATION` option must not be set to zero in the `src/FreeRTOSConfig.h` configuration file to make the `xTimerCreate()` function available.

[8] `TimerHandle_t` is a void pointer type.
`TimerCallbackFunction_t` is a function pointer type. The function type is `void` *func*`(TimerHandle_t)`.

[9] The `uxAutoReload` flag specifies a one-shot or regular timer. When started, a one-shot timer (`uxAutoReload = pdFALSE`) expires only once after the `xTimerPeriod` ticks. When started, a regular timer (`uxAutoReload = pdTRUE`) expires repeatedly on every `xTimerPeriod` ticks.

[10] The `configUSE_TIMERS` option must be set to one in the `src/FreeRTOSConfig.h` configuration file to make the `xTimerStart()` function available.

`xTimer` ... software timer handle
`xTicksToWait` ... maximum number of ticks to wait[11]

For instance, the following `xTimerStart()` function call sends a command to start the `xTimer` software timer, and returns immediately:

```
xTimerStart(xTimer, 0);
```

To use the `xTimerCreate()` and `xTimerStart()` functions, the `FreeRTOS.h` and `timers.h` header files have to be included.

```
#include <FreeRTOS.h>
#include <timers.h>
```

Application initialization

Two semaphores can be driven with the four button keys available on the external board. The first semaphore is taken/given by pressing the key T1/T2, and the second semaphore by keys T3/T4. According to the Fig. 8.1, two queues and a master task are also needed. The pseudo code of the application initialization is:

create and give two semaphores
create key and UART queue
create and start software timer implementing master task
start scheduler

With application initialized, the FreeRTOS scheduler is started, and the master task is configured as a software timer callback function. The function itself is discussed in the next section. Note that the `main()` function stack is reused after the FreeRTOS scheduler is started. Therefore, the queue, semaphore and software timer handles must be global.

Master task

The master task is implemented in a software timer callback function. In each execution, the function has to: detect key pressed events and send them to the key queue, receive messages from the UART queue and send them to the stdio, and check a predefined scenario to start/stop the tasks that are due. To start a task, the `xTaskCreate()` function can be used, and `vTaskDelete()` to stop it. The pseudo code of the master task callback function is:

[11] If the Timer Command queue is full, the command cannot be sent to it. The calling task is placed into blocked state and waits there for a queue space to become available for up to `xTicksToWait` ticks. If the `xTicksToWait` argument is set to `portMAX_DELAY`, the task is suspended for an indefinite time, until a queue space is available. In this case, the `INCLUDE_vTaskSuspend` option has to be set to one in the `src/FreeRTOSConfig.h` configuration file. If the `xTimerStart()` function is called before the FreeRTOS scheduler is started, the `xTicksToWait` argument will be ignored.

```
for each of the four keys
        if the key is down and was up in the previous execution
                send key identifier to key queue
        save key position for the next execution
while reading from UART queue is successful
        send obtained item to stdio
get current tick
while current record in scenario is due
        perform the record, i.e., start/stop specified task
        go to next record, i.e., increase index pointing to current record
```

The algorithm above supposes that the records in the predefined scenario are time ordered. A global index pointing to the current record, i.e., the first unfulfilled record in the scenario, is used. The scenario data structure can be organized in different ways. For instance, a two dimensional array can be used, e.g.:

```
int32_t scenario[][3] =[12]
 {{ 5, 1,  2},  /* record 1: at  5s start task 1 with priority 2 */
  {10, 2,  3},  /* record 2: at 10s start task 2 with priority 3 */
  {15, 1, -1},  /* record 3: at 15s stop  task 1                 */
  {-1, 0,  0}}; /* end of scenario                               */
```

In the example scenario data structure above, each record contains three integers: time, task and priority. A negative priority signifies a task stop. A negative time marks the end of the record list.

Ordinary tasks

An ordinary task indicates that it is in the running state by turning the corresponding LED on the external board on. Since four LEDs are available, there can be up to four different ordinary tasks in the master's scenario. Ordinary tasks are started/stopped by the master task. The priority of an ordinary task must be set below the master task priority, i.e., below the Timer Service task priority. An ordinary task runs in an endless loop. Besides indicating its running state, an ordinary task checks the key queue in every iteration. If a key event is received, the corresponding semaphore action will be performed. A message about the progress of the performed semaphore action is send to the UART queue for the master task to dispatch it to the stdio. The pseudo code of an ordinary task is:

[12] `int32_t` is a 32-bit integer type.

```
while forever
    turn corresponding LED on and the others off
    if reading from key queue is successful
        if key T1 received
            send message 'trying to take semaphore one' to UART queue
            take semaphore one
            send message 'semaphore one taken' to UART queue
        if key T2 received
            enter critical section
            give semaphore one
            send message 'semaphore one given' to UART queue
            exit critical section
        if key T3 received
            send message 'trying to take semaphore two' to UART queue
            take semaphore two
            send message 'semaphore two taken' to UART queue
        if key T4 received
            enter critical section
            give semaphore two
            send message 'semaphore two given' to UART queue
            exit critical section
```

Semaphore take actions are indefinite. If the semaphore is already taken, the calling task will be suspended for an indefinite time until the semaphore is given.

A section of code, that gives a semaphore and sends a message about it, is critical. Otherwise, if a higher priority task is waiting for the semaphore, the task will be preempted immediately after giving the semaphore.

Idle task

The Idle task turns all the LEDs off indicating that no ordinary task is running. It also cannot give or take a semaphore. The received key events are ignored. The pseudo code of the Idle task hook function (see Exercise 2) is:

```
turn all LEDs off
read from key queue
```

Deadlock scenarios

A task, wanting to take an already taken semaphore, has to wait until the semaphore is given. A deadlock occurs when waiting never ends. Each task in a deadlock is waiting for another task in the deadlock to give the semaphore it is waiting for. Therefore, the tasks wait for each other indefinitely without a progress.

Recursive deadlock

If a task tries to take the same binary semaphore twice without giving it, a recursive deadlock will occur. The task locks itself. It is placed in a suspended state and waits indefinitely for itself to give the semaphore (see Fig. 8.2). This typically happens in recursive functions.

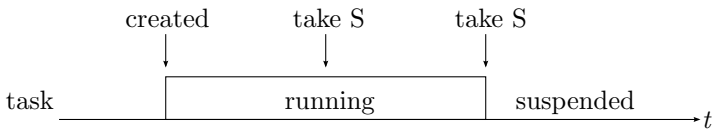


Figure 8.2: Recursive deadlock (S ... semaphore)

Termination deadlock

A task termination leaving a taken semaphore can lead to a termination deadlock. If another task tries to take the same semaphore, it will get suspended (see Fig. 8.3).

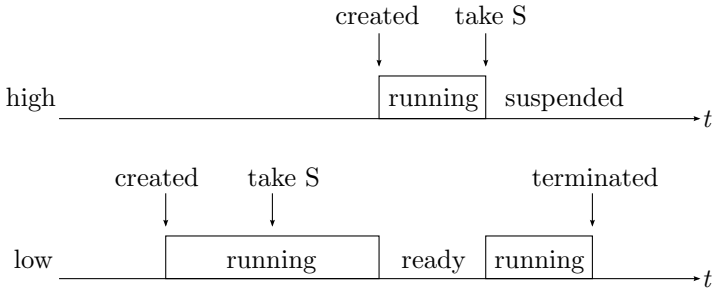


Figure 8.3: Termination deadlock (S ... semaphore)

Circular deadlock

Each task in a circular deadlock took one semaphore and wait for the other taken by another task. A circular deadlock occurs when two or more tasks develop a circular dependency. A two tasks circular deadlock is shown in Fig. 8.4.

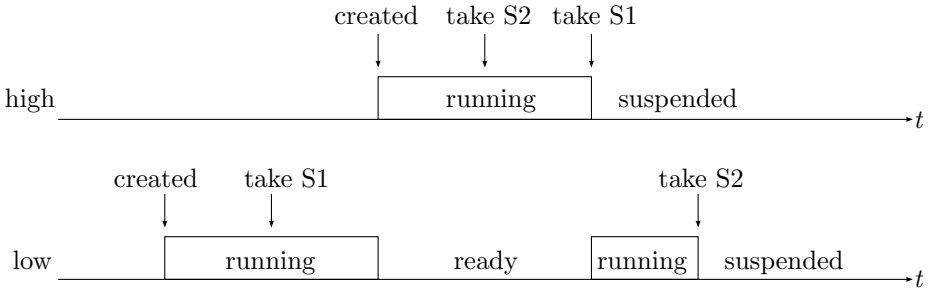


Figure 8.4: Circular deadlock (S1, S2 ... semaphores)

A circular deadlock will be avoided if all the tasks involved follow the same semaphore order. Before a task takes a semaphore, it must take all the preceding semaphores regarding the order, e.g., task high in Fig. 8.4 should take S1 before S2.

Priority inversion

Priority inversion is not a deadlock. However, it is a harmful scheduling effect resulting in a high priority task waiting for a middle priority task to finish. Priority inversion occurs when a medium priority task preempts low priority task that took

a semaphore which high priority task waits for (see Fig. 8.5). The scenario breaks the priority policy.

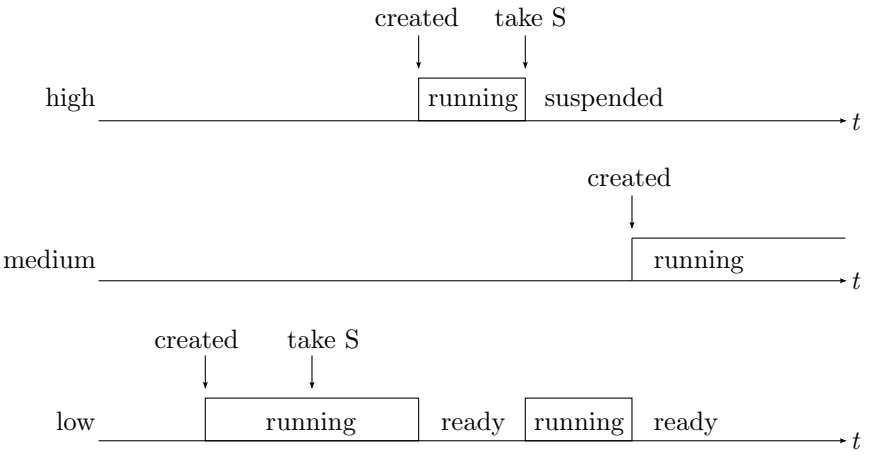


Figure 8.5: Priority inversion (S ... semaphore)

Mutex

Mutex (mutual exclusion) is a binary semaphore with ownership. A mutex is owned by the task that took it. It can be given only by its owner. A mutex is created by the `xSemaphoreCreateMutex()` function[13] [15] [16]. The declaration of the function is:

```
SemaphoreHandle_t xSemaphoreCreateMutex( void );
```

The function returns a handle to the created mutex on success. The mutex is created in a given (unlocked) state. The function will return `NULL` if there is not enough heap memory available to allocate the mutex data structure. The following `xSemaphoreCreateMutex()` function call creates a mutex:

```
SemaphoreHandle_t xMut = xSemaphoreCreateMutex();
```

A mutex is taken (locked) and given (unlocked) in by the same `xSemaphoreTake()` and `xSemaphoreGive()` functions as a binary semaphore.

Besides ownership, a priority inheritance mechanism is also implemented in the FreeRTOS mutexes. If a high priority task waits for a mutex taken by a low priority task, the priority of the low task will be temporarily raised to high until the mutex is given. The low task inherits the priority of the high task while the high task is waiting. The priority inheritance mechanism solves the priority inversion problem. The event scenario from Fig. 8.5 will modify into Fig. 8.6 if mutex is used instead of a binary semaphore.

Since mutexes are owned, the operating system has an information which task took which mutex. Therefore, when mutexes are used, the operating system could deal with the termination deadlock. However, the FreeRTOS does not provide giving of mutexes left behind.

[13] The `configUSE_MUTEXES` option must be set to one and the `configSUPPORT_DYNAMIC_ALLOCATION` option must not be set to zero in the src/FreeRTOSConfig.h configuration file to make the `xSemaphoreCreateMutex()` function available.

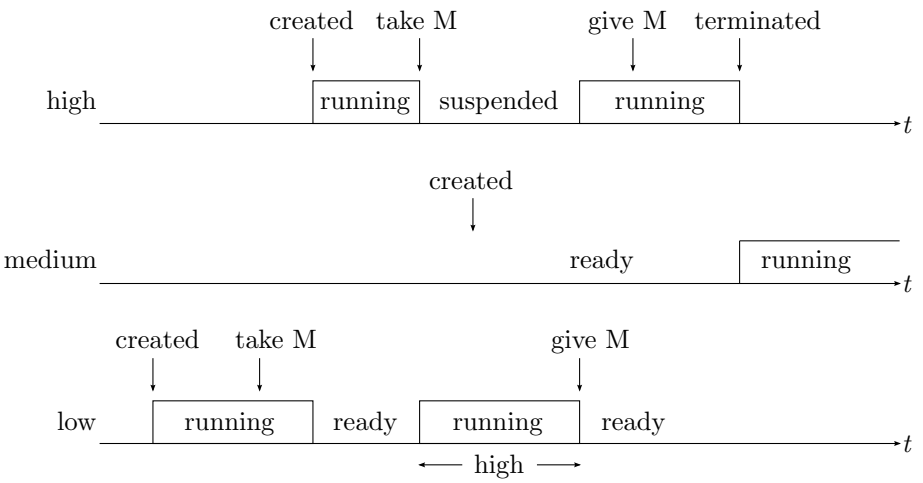


Figure 8.6: Priority inversion scenario solved by priority inheritance (M ... mutex)

Recursive mutex

Besides an ordinary mutex, the FreeRTOS also provides a recursive mutex. A recursive mutex has the same properties as an ordinary mutex, i.e., ownership and priority inheritance, but can be taken more than once. It becomes available (unlocked) when given the same number of times as taken. A recursive mutex is used in the same way as a binary semaphore or a mutex. However, the recursion implementing `xSemaphoreCreateRecursiveMutex()`[14], `xSemaphoreTakeRecursive()` and `xSemaphoreGiveRecursive()` functions [15] [16] have to be used. The declarations of the functions are:

```
SemaphoreHandle_t xSemaphoreCreateRecursiveMutex( void );
BaseType_t xSemaphoreGiveRecursive( SemaphoreHandle_t xMutex );
BaseType_t xSemaphoreTakeRecursive( SemaphoreHandle_t xMutex,
                                    TickType_t xTicksToWait );
```

The functions are analogous to `xSemaphoreCreateMutex()`, `xSemaphoreGive()` and `xSemaphoreTake()` functions. A recursive mutex is created, waited for until available, taken, and given by the following function calls:

```
SemaphoreHandle_t xRecMut = xSemaphoreCreateRecursiveMutex();
xSemaphoreTakeRecursive(xRecMut, portMAX_DELAY);
xSemaphoreGiveRecursive(xRecMut);
```

Since the recursive mutex can be taken more than once, it solves the recursive deadlock from Fig. 8.2. Using a recursive mutex, the task cannot lock itself (see Fig. 8.7).

Priority ceiling

In priority ceiling, a priority is assigned to each semaphore. A semaphore priority has to be equal or higher than the priority of any task using the semaphore. When

[14]The `configUSE_MUTEXES` and `configUSE_RECURSIVE_MUTEXES` options must be set to one and the `configSUPPORT_DYNAMIC_ALLOCATION` option must not be set to zero in the src/FreeRTOSConfig.h configuration file to make the `xSemaphoreCreateRecursiveMutex()` function available.

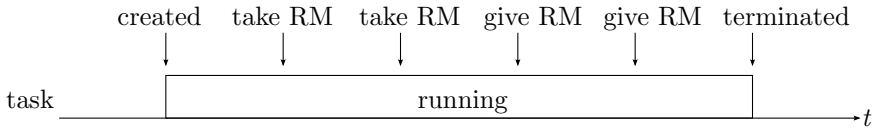


Figure 8.7: Recursive mutex (RM) usage

a task takes the semaphore, its priority is boosted to the semaphore priority until the semaphore is given. Therefore, while holding the semaphore, the task cannot be preempted by another task using the same semaphore. The priority ceiling mechanism solves the circular deadlock and the priority inversion problem. The event scenarios from Figs. 8.4 and 8.5 will modify into Figs. 8.8 and 8.9 if the priority ceiling mechanism is used.

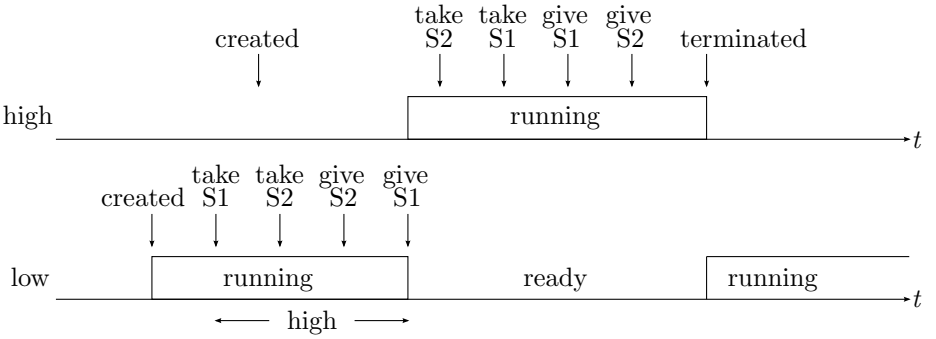


Figure 8.8: Circular deadlock scenario solved by priority ceiling (S1, S2 ... semaphores)

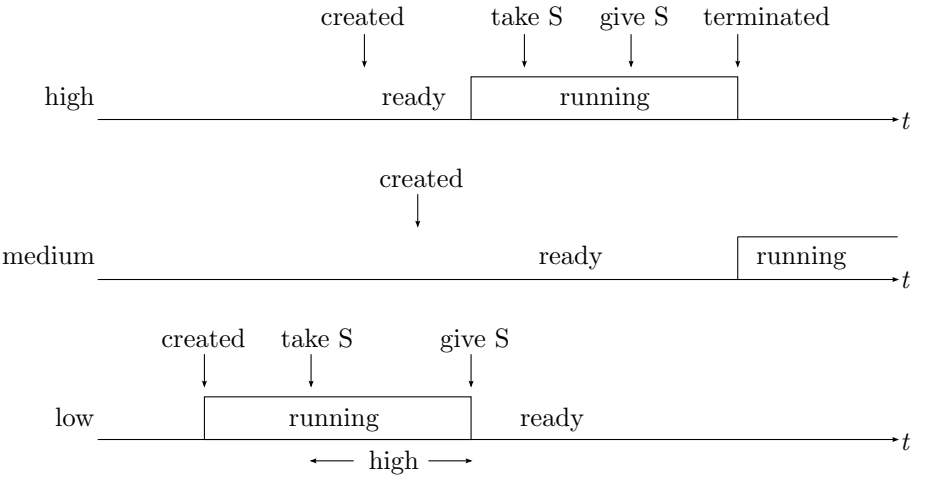


Figure 8.9: Priority inheritance problem solved by priority ceiling (S ... semaphore)

Although the priority ceiling mechanism solves the circular deadlock and the priority inversion problem, a high priority task can be repeatedly delayed because of it. If a low priority task takes a semaphore very often for a relatively long time intervals, then it will most of the time run at a boosted priority. Whenever due, the high priority task will likely has to wait instead of immediately preempting the low task.

The priority ceiling mechanism is not implemented in the FreeRTOS. However, it can be applied manually by raising task priority before semaphore take, and setting it back to the original level after semaphore give.

Exercise 9

Ramp application

Write a software for the Arduino Due board that drives a ramp model. Use FreeRTOS. The software should read the password from the `stdin`. On right password, the ramp should open, stay opened for a predefined time interval, and close. In case an obstacle is detected while the ramp is closing, the ramp should reopen. Ramp models are available in the faculty laboratory.

Explanation

Create a new empty project in the Eclipse working environment as explained in Exercise 1. Connect the Arduino Due board to the host PC over the Olimex ARM-USB-OCD-H interface as shown in Fig. 4.1. Configure the UART peripheral device and the stdio in serial mode as explained in Appendix A.

The ramp model has seven controlling pins, four driving the ramp, and another three reporting back the ramp status (Tab. 9.1). Connect the ramp model to the Arduino Due board as shown in Fig. 9.1. Provide the 38kHz signal, required at the OBST_TX ramp pin, using a PWM[1] channel. Configure the rest of the ramp pins as input and output GPIO pins. Neither pull-up resistors nor debounce filters are required on input pins. See Appendix A for PWM and GPIO pin configuration and usage. The software required in this exercise has to appropriately drive the pins to achieve the desired behavior.

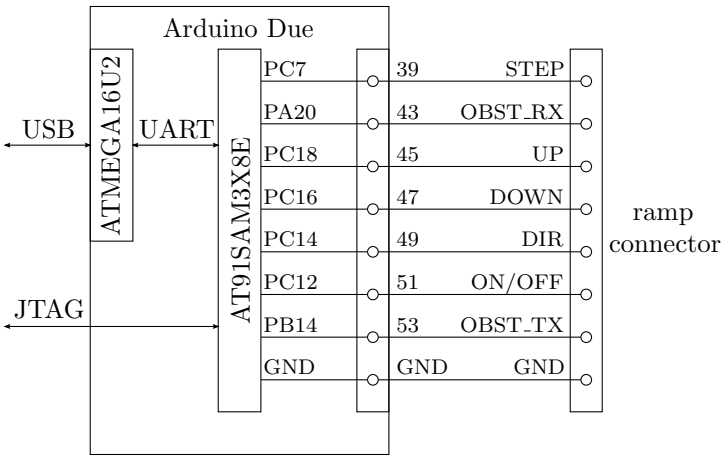


Figure 9.1: Ramp model to Arduino Due board connection

[1]PWM ... Pulse Width Modulation

ramp pin	direction*	description
STEP	in	motor step (< 300Hz, 50% duty cycle)
OBST_RX	out	obstacle sensor (0 ... no obstacle / 1 ... obstacle)
UP	out	ramp open sensor (0 ... fully open / 1 ... not fully open)
DOWN	out	ramp closed sensor (0 ... fully closed / 1 ... not fully closed)
DIR	in	ramp direction (0 ... up / 1 ... down)
ON/OFF	in	motor switch (0 ... off / 1 ... on)
OBST_TX	in	IR sensor (38kHz, 50% duty cycle)
GND		ground

*from the ramp perspective

Table 9.1: Ramp pins

Application organization

The application can be coded in different ways. The realization described here is of course not the only one possible. The proposed application organization is shown in Fig. 9.2. There are four FreeRTOS tasks. The UARTdrv task drives the UART, i.e., stdio. It sends the received characters to the RX queue and transmits the characters received from the TX queue. The check task receives the password from the RX queue. It verifies the password and notices the ramp task when to open the ramp. The ramp task drives the ramp. It sets and reads the ramp input and output pins to open and close the ramp. The ramp task also provides an information to the step task when the step motor signal is needed. The step task provides the step motor signal on the ramp STEP pin. All the tasks have the same priority. An exception is the step task whose priority must be higher to ensure a steady motor step.

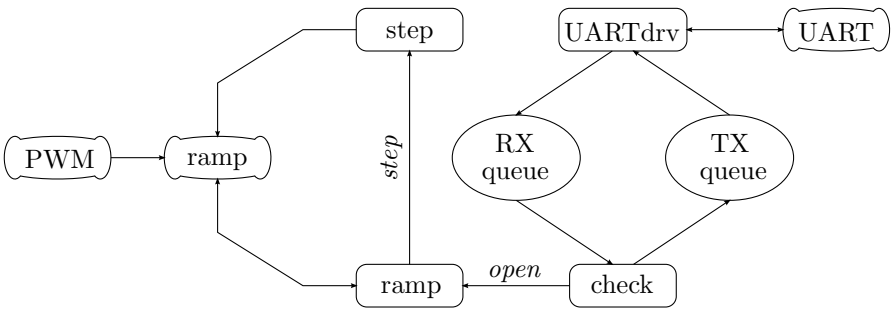


Figure 9.2: Ramp application organization

The pseudo code of the application initialization is:

create RX and TX queue
create ramp, check, and UARTdrv tasks with lower priority
create step task with higher priority
set ON/OFF ramp pin to on
start scheduler

Tasks

UARTdrv task

The UARTdrv task is an UART driver. It translates the UART device into an RX and TX queues for the rest of the software. The characters read from the UART device are send to the RX queue, and the characters received from the TX queue are written to the UART device. The pseudo code of the UARTdrv task is:

```
while forever
        if reading from UART is successful
                send character to RX queue
        if receiving from TX queue is successful
                do write character to UART
                while write is not successful
```

Note that the UARTdrv task must not get blocked. The UART read/write, the RX send, and the TX receive functions must return immediately. If the RX queue is not regularly emptied by the rest of the software, the characters read from the UART will be thrown away. On the other hand, all characters received from the TX queue are written to the UART. The only limit is the UART speed.

check task

The check task prompts the user to enter the password. The TX queue is used. Then obtains the password character by character from the RX queue until a newline character is received. For each received character, an asterisk is send back. If the obtained password is correct, the *open* global variable is set to notice the ramp task to open the ramp. The pseudo code of the check task is:

```
while forever
        initialize empty password
        send prompt string (character by character) to TX queue
        while forever
                while receiving from RX queue is not successful
                        do nothing
                if newline received
                        break
                else
                        add character to password
                        send asterisk to TX queue
        if password is correct
                set open global variable
```

The `strcmp()` string compare function declared in the `string.h` header file comes in handy for password verification.

ramp task

Every time the *open* global variable is set, the ramp task performs one lifting cycle. The ramp is opened, stay opened for a predefined amount of time, and closed. During the closing part, the obstacle sensor is checked. If an obstacle is detected, the ramp will be reopened. Essentially, the ramp task just appropriately sets and reads the ramp input and output pins to achieve the desired behavior. The pseudo code of the ramp task is:

```
while forever
    if open global variable is set
        reset open global variable
        while forever
            set DIR pin to up
            set step global variable
            while ramp is not fully open (use UP pin)
                do nothing
            reset step global variable
            go to blocked state for a predefined amount of time[2]
            set DIR pin to down
            set step global variable
            while ramp is not fully closed (use DOWN pin) and
                                there is no obstacle (use OBST_RX pin)
                do nothing
            reset step global variable
            if ramp is fully closed (use DOWN pin)
                break
```

The ramp is moving when the step motor signal at STEP pin is present, and stand still otherwise. The STEP pin is not directly controlled by the ramp task. The *step* global variable is used to inform the step task when the step motor signal is needed.

step task

The step task is a periodic task toggling the STEP pin when the step motor signal is needed, i.e., when the *step* global variable is set. The signal frequency is one half of the task's frequency rate. The pseudo code of the step task is:

```
get current tick
while forever
    if step global variable is set
        toggle STEP ramp pin
    go to blocked state until one half of signal period passes
```

The priority of step task must be higher than the priorities of other tasks. To ensure a steady step motor signal, the task must not miss its deadlines.

[2] Use `vTaskDelay()` function.

Appendix A

Peripheral device initialization and usage receipts

Initialization and usage code examples for some peripheral devices of the AT91SAM3X8E µC can be found in this appendix. The code is tailored to the exercises described in this collection. Of course, the peripheral devices can also be configured and used in many other ways not presented here. For detailed information about AT91SAM3X8E peripheral devices, see [8].

GPIO pins

There are four PIO[1] controllers (i.e., PIOA, PIOB, PIOC and PIOD) in the AT91SAM3X8E µC, each controlling up to 32 I/O pins. All four PIO controllers are enabled during the hardware initialization in the `board_init()` function call.

To configure the specified pins, e.g., `PC24`, `PC25`, `PC26` and `PC28`, as input pins with pull-up resistors and debouncing filters with cutoff frequency at 20Hz, use:

```
pio_configure(PIOC, PIO_INPUT, PIO_PC24 | PIO_PC25 | PIO_PC26 |
                               PIO_PC28, PIO_PULLUP | PIO_DEBOUNCE);
pio_set_debounce_filter(PIOC, PIO_PC24 | PIO_PC25 | PIO_PC26 |
                                                    PIO_PC28, 20);
```

If the `PULL_UP` attribute is omitted, the pull-up resistors are not enabled. The same goes for the `DEBOUNCE` attribute and the debouncing filters. Without `DEBOUNCE` attribute, the debouncing filters are not enabled. In that case, the `pio_set_debounce_filter()` function call is irrelevant.
To read the specified input pin, e.g., `PC24`, state, use:

```
uint32_t ulState = pio_get(PIOC, PIO_INPUT, PIO_PC24);
```

Regarding the input pin state, the function returns zero or one.

To configure the specified pins, e.g., `PC21`, `PC22`, `PC23` and `PC29`, as output pins with initial value zero, use:

```
pio_configure(PIOC, PIO_OUTPUT_0, PIO_PC21 | PIO_PC22 | PIO_PC23 |
                                                    PIO_PC29, 0);
```

To reset the specified output pin state, e.g., `PC21`, use:

[1] PIO ... Parallel I/O

```
pio_clear(PIOC, PIO_PC21);
```

To set the specified output pin state, e.g., `PC21`, use:

```
pio_set(PIOC, PIO_PC21);
```

To toggle the specified output pin state, e.g., `PC7`, use:

```
pio_toggle_pin(PIO_PC7_IDX);
```

To use the PIO functions, the `pio.h` header file has to be included.

```
#include <pio.h>
```

UART and stdio in serial mode

An I/O pin can be configured as a GPIO pin, or as a pin hardwired to a peripheral device. However, any pin cannot be hardwired to any peripheral device. Each pin has up to two predefined peripheral devices that can be hardwired to it. The UART peripheral device has URXD[2] and UTXD[3] lines hardwired to the PA8 and PA9 I/O pins of the PIOA.

The configuration of the PA8 and PA9 I/O pins as UART pins is performed on demand during the Arduino Due board initialization in the `board_init()` function. To request the UART configuration, define the `CONF_BOARD_UART_CONSOLE` macro in the `src/conf_board.h` file:

```
#define CONF_BOARD_UART_CONSOLE
```

The PA8 and PA9 I/O pins are connected to the additional ATMEGA16U2 µC on the Arduino Due board. The additional µC acts as an UART to USB converter to the on-board Programming USB port. Such a solution needs a pull-up resistor at the PA8 I/O pin of the PIOA (i.e., the URXD line). However, all pull-up resistors are disabled in the `board_init()` call. The PA8 pull-up resistor has to be enabled by:

```
pio_pull_up(PIOA, PIO_PA8, PIO_PULLUP);
```

To use the `pio_pull_up()` function, the `pio.h` header file has to be included.

```
#include <pio.h>
```

To configure the stdio in serial mode and initialize the UART device to 38400/8-N-1[4], the following code can be used:

```
usart_serial_options_t xUARTconf = {.baudrate = 38400,
                                    .paritytype = UART_MR_PAR_NO};
stdio_serial_init(UART, &xUARTconf);[5]
```

[2]URXD ... UART Receive Data
[3]UTXD ... UART Transmit Data
[4]38400 baud, 8 data bits, no parity, 1 stop bit.
[5]Note that the speed of the UART peripheral device is calculated regarding the MCK settings in the `src/conf_clock.h` file.

To use the `stdio_serial_init()` function and the `usart_serial_options_t` structure, the `stdio_serial.h` header file has to be included.

```
#include <stdio_serial.h>
```

With stdio configured, the stdio functions (e.g., `getchar()`, `putchar()`, etc.) can be used. The URXD line is used as `stdin`, and UTXD as `stdout`.

The stdio `getchar()` function waits for a character at `stdin`. It returns when a character is available, e.g., when a keyboard key is pressed and the character is received on the URXD line. Therefore, the `getchar()` function may wait indefinitely long. When waiting has to be avoided, the ASF provided `uart_read()` function can be used. The function returns zero on success. If a character on the URXD line is not available, one will be returned. To read a character from the URXD line and save it into the `ucCharacter` variable of `uint8_t`[6] type, use:

```
uint32_t ulError = uart_read(UART, &ucCharacter);
```

For writing to the UTXD line, the equivalent `uart_write()` function is available. The function returns zero on success. If writing fails, i.e., the UART transmit buffer is full, one will be returned. To write a character in the `ucCharacter`[7] variable to the UTXD line, use:

```
uint32_t ulError = uart_write(UART, ucCharacter);
```

TC as a free running counter

There are three TC modules available in the AT91SAM3X8E µC. Each module has three channels. Thus, there are nine independent TC channels available. The channels 0, 1 and 2 belong to the TC0 module, 3, 4, and 5 to the TC1 module, 6, 7, and 8 to the TC2 module.

A TC module has to be clocked first. The clock is provided through the PMC[8]. The TC0 module clock is enabled by:

```
pmc_enable_periph_clk(ID_TC0);
```

With TC0 module clocked, its three independent channels 0, 1 and 2 become available. The channel 0 of the TC0 module is configured as a free up-running counter with counting frequency $\frac{f_{\mathrm{MCK}}}{2}$ with:

```
tc_init(TC0, 0, TC_CMR_WAVE);
```

The µC master clock frequency f_{MCK} can be obtained in the `SystemCoreClock` global variable.

After configuration, the counter has to be started. The TC0 module channel 0 is started by:

```
tc_start(TC0, 0);
```

The current counter value of the TC0 module channel 0 is read by:

[6] `uint8_t` is an 8-bit unsigned integer type.
[7] `ucCharacter` is an `uint8_t` type variable.
[8] PMC ... Power Management Controller

```
uint32_t ulVal = tc_read_cv(TC0, 0);
```

To use the TC functions, the `tc.h` header file has to be included.

```
#include <tc.h>
```

A free running counter can be used to measure time intervals. The counter is read at the beginning and at the end of the interval. Knowing the counting frequency, the passed time interval can be calculated. In the following code, the interval is calculated in microseconds:

```
uint32_t ulInterval_us, ulStop, ulStart = tc_read_cv(TC0, 0);
    ...
ulStop = tc_read_cv(TC0, 0);
ulInterval_us = 2e6 * (ulStop - ulStart) / SystemCoreClock;
```

Using PWM peripheral device as a signal generator

A PWM peripheral device in the AT91SAM3X8E µC has eight channels, each generating an independent waveform. It has to be clocked first. The clock is provided through the PMC. The PWM device clock is enabled by:

```
pmc_enable_periph_clk(ID_PWM);
```

The PWM device provides thirteen MCK based clock sources to its channels. Two of them are distinct and have to be additionally configured. To disable the special clock sources, use the following code:

```
pwm_clock_t xPWMClk = {0};
     ...
xPWMClk.ul_mck = SystemCoreClock;
pwm_init(PWM, &xPWMClk);
```

Note, that the `SystemCoreClock` global variable is set in the `sysclk_reinit()` function call, and therefore should not be used before the call.

To configure an individual channel, the channel has to be disabled. Channel two is disabled by:

```
pwm_channel_disable(PWM, PWM_CHANNEL_2);
```

A frequency, duty cycle, and a clock source has to be configured for an individual channel. To get a signal on channel two with *frequency*, 50% duty cycle, and using MCK as a clock source, use the following code:

```
pwm_channel_t xPWMChannel = {0};
  ...
xPWMChannel.channel      = PWM_CHANNEL_2;
xPWMChannel.ul_prescaler = PWM_CMR_CPRE_MCK;
xPWMChannel.ul_period    = SystemCoreClock / frequency;
xPWMChannel.ul_duty      = xPWMChannel.ul_period / 2;
pwm_channel_init(PWM, &xPWMChannel);
```

Again, be aware that the `SystemCoreClock` global variable is set in the `sysclk_re`

`init()` function call.
After configuration, the channel is enabled by:

```
pwm_channel_enable(PWM, PWM_CHANNEL_2);
```

To use the PWM functions, the `pwm.h` header file has to be included.

```
#include <pwm.h>
```

Finally, the configured and enabled PWM channel has to be hardwired to a GPIO pin to become available. Note that an individual peripheral device output cannot be hardwired to an arbitrary pin. The PWM channel two can be hardwired to the PB14 pin:

```
pio_set_peripheral(PIOB, PIO_PERIPH_B, PIO_PB14);
```

Appendix B

External board LCD

There is an LCD[1] available on the external board. This appendix provides a short LCD usage instructions, although the LCD is not explicitly used in the laboratory work. A detailed LCD description can be found in [22].

If the LCD is connected to the Arduino Due board as shown in Fig. B.1, the LCD functions in `src/lcd.h` file can be used. Of course, the `lcd.h` file has to be included.

```
#include <lcd.h>
```

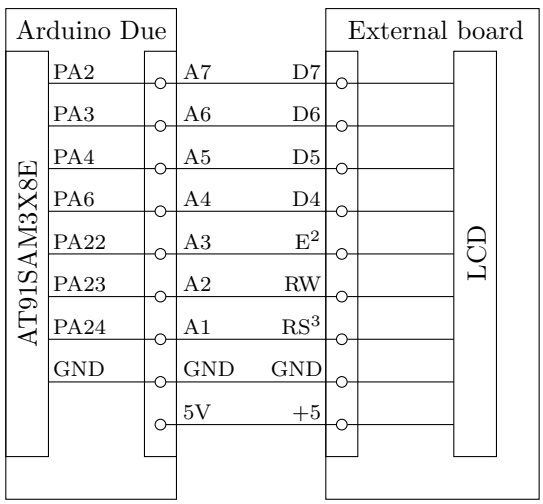


Figure B.1: LCD to Arduino Due board connection

The LCD is initialized by:

```
vLCDInit();
```

The commands and data can be sent to LCD using the `vLCDWrite()` function. For instance:

[1] LCD ... Liquid Crystal Display
[2] E ... Enable
[3] RS ... Register Select

Set LCD RAM address to the third position in the first line:

```
vLCDWrite(COMM, DDRAM | 0x02);
```

Set LCD RAM address to the third position in the second line:

```
vLCDWrite(COMM, DDRAM | (0x40 + 0x02));
```

Write character A at the current LCD RAM address:

```
vLCDWrite(DATA, 'A');
```

For a detailed description of the available LCD commands, see [22]. Predefined macros with command codes can be found in the `src/lcd.h` file.

Bibliography

[1] The Eclipse Foundation open source community website, *https://eclipse.org*, Apr. 2015

[2] Oracle website, *http://www.oracle.com/index.html*, Apr. 2015

[3] R.M. Stallman and the GCC Developer Community, *Using the GNU Compiler Collection*, GNU Press, 2014, Free Software Foundation, *https://gcc.gnu.org/onlinedocs/gcc-4.9.2/gcc.pdf*, Apr. 2015

[4] Launchpad website, *https://launchpad.net*, Apr. 2015

[5] *ARM-USB-OCD-H, ARM-USB-OCD User's Manual*, Olimex, 2015, Olimex, *https://www.olimex.com/Products/ARM/JTAG/_resources/ARM-USB-OCD_and_OCD_H_manual.pdf*, Apr. 2015

[6] *Open On-Chip Debugger: OpenOCD User's Guide*, 2014, The OpenOCD Project, *http://sourceforge.net/projects/openocd/files/openocd/0.8.0/openocd.pdf/download*, Apr. 2015

[7] Sourceforge website, *http://sourceforge.net*, Apr. 2015

[8] *ATMEL SAM3X / SAM3A Series Datasheet*, Atmel, 2015, Atmel, *http://www.atmel.com/Images/Atmel-11057-32-bit-Cortex-M3-Microcontroller-SAM3X-SAM3A_Datasheet.pdf*, Apr. 2015

[9] Arduino Due board website, *http://www.arduino.cc/en/Main/ArduinoBoardDue*, Apr. 2015

[10] The ARM Ltd. website, *http://www.arm.com*, Apr. 2015

[11] The Atmel Corporation website, *http://www.atmel.com*, Apr. 2015

[12] R.M. Stallman, R. McGrath, P.D. Smithand, *GNU Make*, Free Software Foundation, 2014, Free Software Foundation, *http://www.gnu.org/software/make/manual/make.pdf*, Apr. 2015

[13] AT91SAM3X8E source files, makefiles and linker script from ASF for laboratory exercises, *http://fides.fe.uni-lj.si/~janezp/embedded_systems/asf.zip*, Oct. 2017

[14] The FreeRTOS website, *http://www.freertos.org*, Apr. 2017

[15] R. Barry, *Mastering the FreeRTOS Real Time Kernel*, Real Time Engineers Ltd., 2016, *http://www.freertos.org/Documentation/161204_Mastering_the_FreeRTOS_Real_Time_Kernel-A_Hands-On_Tutorial_Guide.pdf*, Apr. 2017

[16] *The FreeRTOS Reference Manual*, Real Time Engineers Ltd., 2016, *http://www.freertos.org/Documentation/FreeRTOS_Reference_Manual_V9.0.0.pdf*, Apr. 2017

[17] A. Silberschatz, P.B. Galvin, G. Gagne, *Operating System Concepts*, 9th ed., Willey, 2013

[18] *Cortex-M3 Technical Reference Manual*, ARM Ltd., 2010, *http://infocenter.arm.com/help/topic/com.arm.doc.ddi0337e/DDI0337E_cortex_m3_r1p1_trm.pdf*, Apr. 2015

[19] *ARM Procedure Call Standard for the ARM® Architecture*, ARM Ltd., 2012, *http://infocenter.arm.com/help/topic/com.arm.doc.ihi0042e/IHI0042E_aapcs.pdf*, Apr. 2015

[20] D. Elsner, J. Fenlason & friends, *Using as / The GNU Assembler*, Free Software Foundation, 2009, HTML version: *https://sourceware.org/binutils/docs/as/index.html*, Apr. 2015

[21] *ARMv7-M Architecture Reference Manual*, ARM Ltd., 2014

[22] *HD44780U (LCD-II) (Dot Matrix Liquid Crystal Display Controller/Driver)*, Hitachi, 1998, Alldatasheet, *http://pdf1.alldatasheet.com/datasheet-pdf/view/63663/HITACHI/HD44780U.html*, Apr. 2017

The script contains instructions and detailed explanation of laboratory exer-cises covered in the Real-time operating systems course that is held in the third semester of the 2nd Cycle Postgraduate Study Programme in Electrical Engi-neering, study programme option Electronics, at the Faculty of electrical engi-neering of the University of Ljubljana, Slovenia. The laboratory exercises focus on usage of operating system features such as: task scheduling, scheduling al-gorithms, memory protection, stack and heap management, semaphores and mutexes, etc., in embedded applications.

REAL-TIME OPERATING SYSTEMS: LABORATORY EXERCISES

Operating system, Real-time, C language, FreeRTOS, ARM Cortex-M3, AT91SAM3X8E

KEYWORDS

ISBN 978-961-243-381-9

ZALOŽBA FAKULTETE ZA ELEKTROTEHNIKO