Univerza *v Ljubljani*
Fakulteta *za elektrotehniko*

EMBEDDED SYSTEM:

LABORATORY EXERCISES

Založba FE

JANEZ PUHAN

University of Ljubljana
Faculty of electrical engineering

Embedded systems: Laboratory exercises

Janez Puhan

Ljubljana, 2019

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

COBISS.SI-ID=299479040

ISBN 978-961-243-380-2 (pdf)

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

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

Contents

Preface **v**

1 Installing IDE **1**

2 Simple electronic lock **13**

3 Watchdog timer **19**

4 UART **23**

5 µC initialization **29**

6 Timer **37**

7 LCD driver **47**

8 ADC and DACC **51**

9 Ramp application **59**

Bibliography **65**

Preface

The laboratory exercises described in this script are part of the Embedded systems course. The course is held in the fifth semester of the 1st Cycle Professional Study Programme in Applied Electrical Engineering, study programme option Electronics, at the Faculty of electrical engineering of the University of Ljubljana, Slovenia. The students are introduced into embedded system programming through nine laboratory exercises. A µC[1] system with ARM[2] Cortex based processor core is used. The Arduino Due board with Olimex ARM-USB-OCD-H[4] JTAG[5] interface serves as a hardware platform. The Eclipse IDE[6] for C/C++ Developers is used as a graphical interface to the GNU[7] tools (i.e., compiler, linker, debugger, etc.). The environment is installed on a PC[8] with installed Linux operating system. A solid knowledge of the C programming language is a required prerequisite.

---

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

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

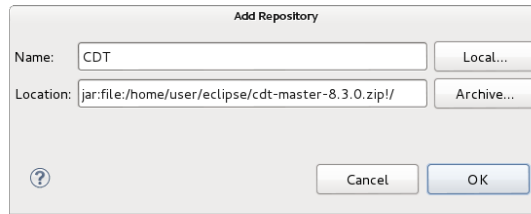


Figure 1.1: The *Add Repository* dialog box

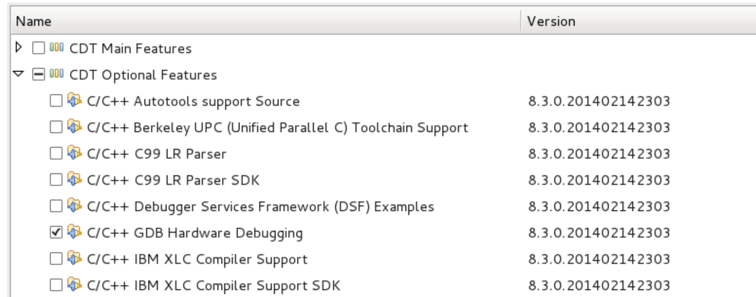


Figure 1.2: Select C/C++ GDB Hardware Debugging plug-in in the *Install* dialog box

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.

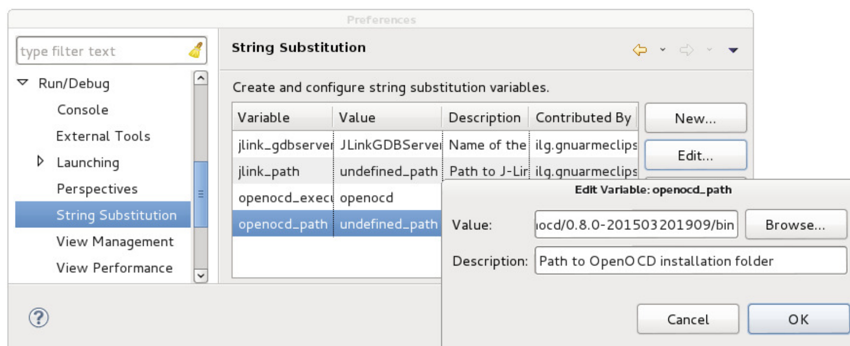


Figure 1.3: Specifying absolute path to the OpenOCD binary directory

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.

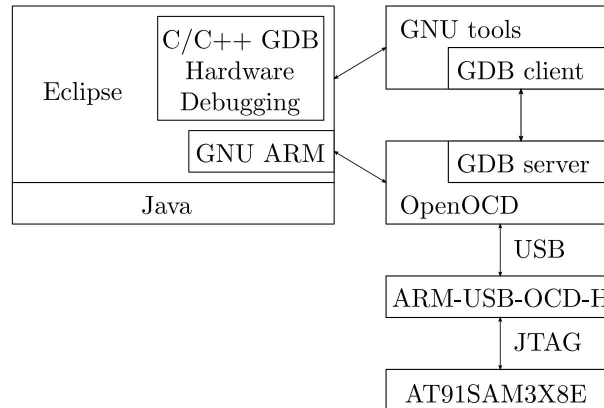


Figure 1.4: Working environment

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.

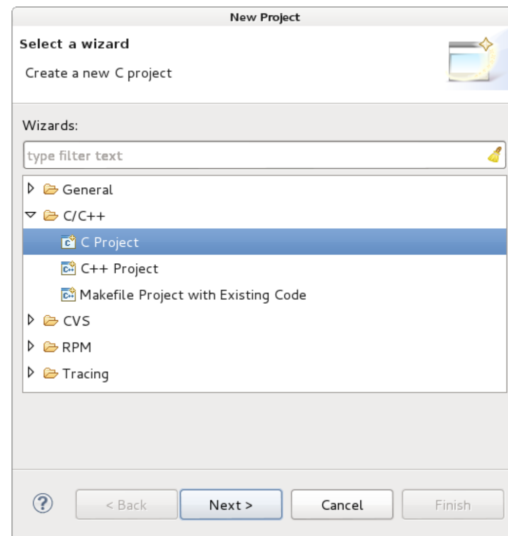


Figure 1.5: The *New Project* dialog box

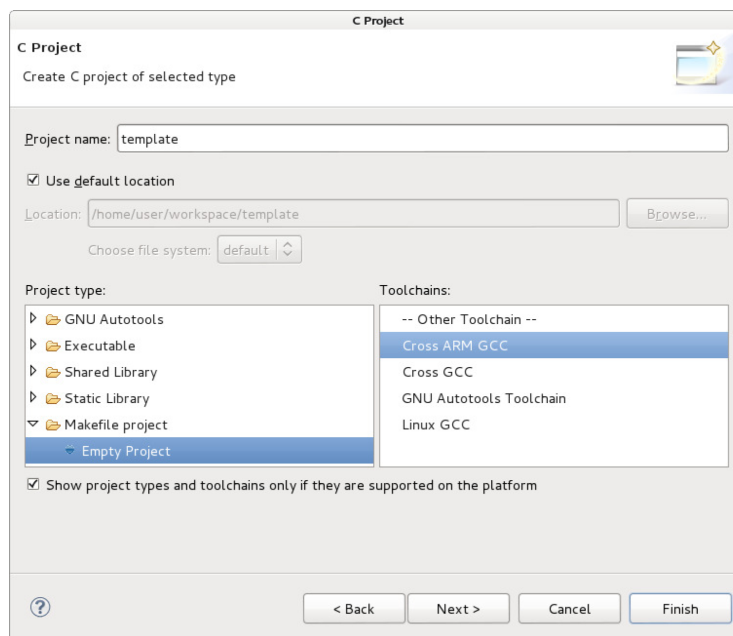


Figure 1.6: The *C Project* dialog box

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)

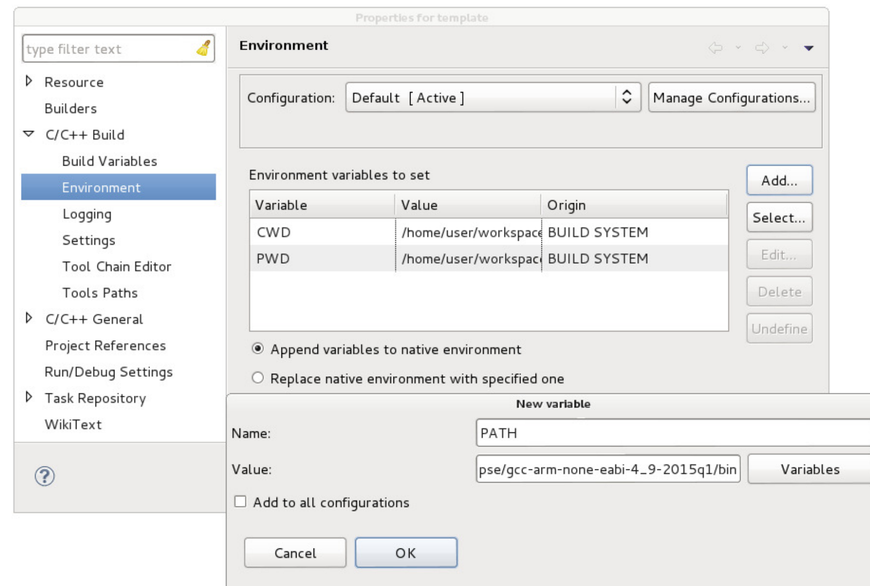


Figure 1.7: Path to the GNU tools directory

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

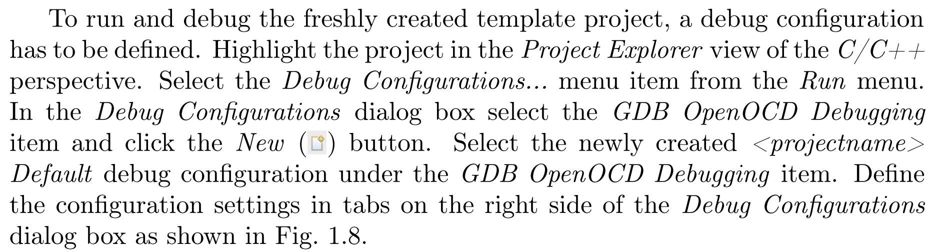
vector loaded into the program counter register at power-up. The reset vector refers to the `Reset_Handler()` function also defined in this file. Thus, the µC starts in `Reset_Handler()` which after some basic initialization[25] calls the `main()` function. The `main()` function is considered as the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



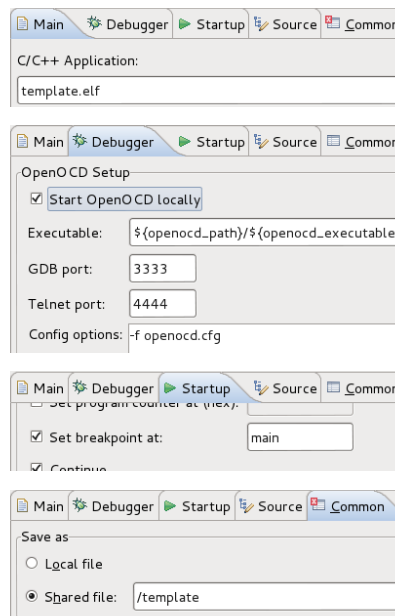


Figure 1.8: Debug configuration settings

[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 can 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 Exercise 4). 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`.

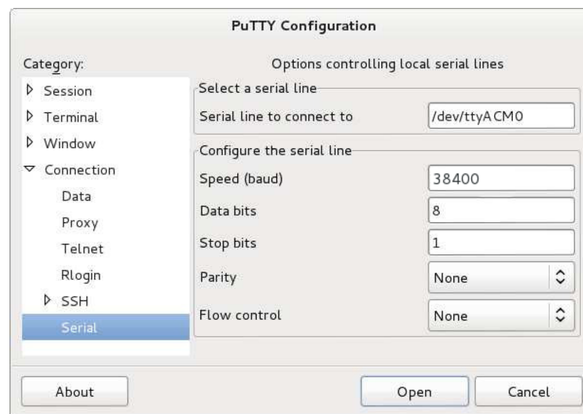


Figure 1.9: An example of the PuTTY serial settings

Exercise 2

Simple electronic lock

Write a software for the Arduino Due board simulating a simple electronic lock. Use button keys and LEDs[1] available on the external board to enter the opening combination and to signal the lock status, respectively. On typing the right combination the lock should open for a limited amount of time. The external boards are available in the faculty laboratory.

Explanation

First, a user interface should be decided. One possible arrangement is to use the external board LED1 to signal key pushes, and LED4 to signal the lock status. The LED1 is on when any of the external board keys is pressed, and the LED4 is on when the lock is locked. The external board keys T1, T2 and T3 are used to enter the opening combination, and key T4 is an Enter key. The user enters the opening combination with T1, T2 and T3, then presses the Enter key. If the combination is right, the lock will open for a few seconds. The LED4 is turned off and, when the time is up, back on. If the combination is wrong, the keys will freeze for the same amount of time to prevent fast combination guessing.

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.

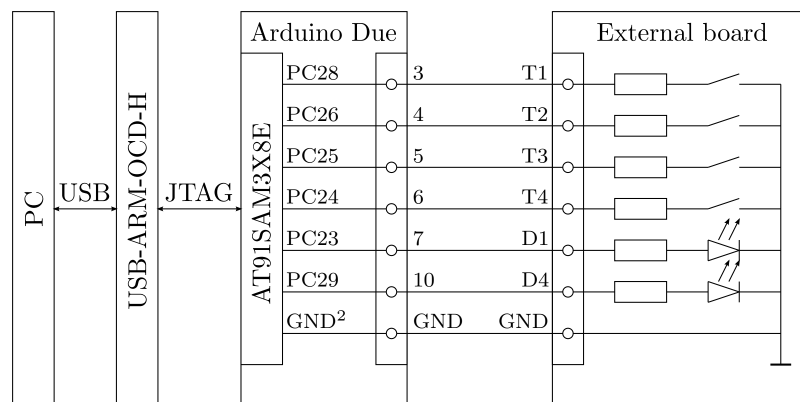


Figure 2.1: PC / Olimex ARM-USB-OCD-H / Arduino Due / External board connection for Exercise 2

[1] LED ... Light Emitting Diode
[2] GND ... Ground

Initialization of the µC and the Arduino Due board

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.

Initialization of the I/O[3] pins

The key and LED controlling pins have to be configured as GPIO[4] pins managed by the PIO[5] controllers. There are four PIO 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 in the `board_init()` function call.

To configure one or more I/O pins, the `pio_configure()` function can be used. The declaration of the function is:

```
uint32_t pio_configure(Pio *p_pio, pio_type_t ul_type,
                            uint32_t ul_mask, uint32_t ul_attribute);
```
[6]

The function returns zero if the pin type `ul_type` is unknown, and one otherwise. The arguments are:

`p_pio`	...	PIOx registers base address (e.g., `PIOC` for PIOC peripheral device)
`ul_type`	...	pin(s) type (e.g., `PIO_INPUT` for input pin(s))
`ul_mask`	...	mask of the pin(s) to be configured (e.g., `PIO_PC28` for pin PC28 of the PIOC device)
`ul_attribute`	...	pin(s) attributes (e.g., `PIO_PULLUP` to enable the pull-up resistor(s))

E.g., Six pins of the PIOC peripheral device are used in this exercise (see Fig. 2.1). Pins PC23 and PC29 control the LEDs and are therefore output pins. On the other hand, the pins PC24, PC25, PC26 and PC28 are input pins connected to the keys. A pull-up resistor is required at each input pin or else an input pin is floating when the corresponding key is not pressed. To configure the six pins as required, the `pio_configure()` function with the appropriate argument values must be called for the output and input pins:

```
pio_configure(PIOC, PIO_OUTPUT_0, PIO_PC23 | PIO_PC29, 0);
pio_configure(PIOC, PIO_INPUT, PIO_PC24 | PIO_PC25 | PIO_PC26 |
                              PIO_PC28, PIO_PULLUP | PIO_DEBOUNCE);
```
[7] [8]

To avoid key debouncing, the hardware debounce filters are enabled at the input pins. To configure one or more debounce filters, the `pio_set_debounce_filter()` function can be used. The declaration of the function is:

---

[3] I/O ... Input / Output

[4] GPIO ... General Purpose I/O

[5] PIO ... Parallel I/O

[6] `uint32_t` is a 32-bit unsigned integer type.
`Pio` is a structure of 32-bit integers representing the AT91SAM3X8E PIO hardware register addresses. If the pointer to such a structure refers to the top of the PIO device address space, the structure elements directly represent the registers.
`pio_type_t` is an enumerated type with PIO pin types.

[7] Type `PIO_OUTPUT_0` defines an output pin with initial value set to zero.

[8] Attribute `PIO_DEBOUNCE` enables a debounce filter at the input pin.

```
void pio_set_debounce_filter(Pio *p_pio, uint32_t ul_mask,
                                          uint32_t ul_cut_off));
```

The function arguments are:

`p_pio`	...	PIOx registers base address (e.g., `PIOC` for PIOC peripheral device)
`ul_mask`	...	mask of the pin(s) to be configured (e.g., `PIO_PC28` for pin PC28 of the PIOC device)
`ul_cut_off`	...	debounce filter cutoff frequency in Hz

E.g., To set the debounce cutoff frequencies of all four key input pins to 20Hz, the `pio_set_debounce_filter()` function with the appropriate argument values must be called:

```
pio_set_debounce_filter(PIOC, PIO_PC24 | PIO_PC25 | PIO_PC26 |
                                                  PIO_PC28, 20);
```

The declarations of the `pio_configure()` and `pio_set_debounce_filter()` functions reside in the `pio.h` header file, which has to be included.

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

I/O[9] pin usage

One or more output pins can be set, i.e., set their output voltage level to high (3.3V), with the `pio_set()` function. The declaration of the function is:

```
void pio_set(Pio *p_pio, uint32_t ul_mask);
```

The function arguments are:

`p_pio`	...	PIOx registers base address (e.g., `PIOC` for PIOC peripheral device)
`ul_mask`	...	mask of the pin(s) to be configured (e.g., `PIO_PC23` for pin PC23 of the PIOC device)

E.g., To set the output pin PC23 of the PIOC peripheral device to a high voltage level, i.e., to turn LED1 on the external board on, the `pio_set()` function with the appropriate argument values must be called:

```
pio_set(PIOC, PIO_PC23);
```

Similarly, the `pio_clear()` function can be used to clear one or more output pins, i.e., set their output voltage level to low (0V). The declaration of the function is:

```
void pio_clear(Pio *p_pio, uint32_t ul_mask);
```

The function arguments are:

`p_pio`	...	PIOx registers base address (e.g., `PIOC` for PIOC peripheral device)
`ul_mask`	...	mask of the pin(s) to be configured (e.g., `PIO_PC23` for pin PC23 of the PIOC device)

E.g., To clear the output pin PC23 of the PIOC peripheral device to a low voltage level, i.e., to turn LED1 on the external board off, the `pio_clear()` function with the appropriate argument values must be called:

```
pio_clear(PIOC, PIO_PC23);
```

A voltage level on one or more input or output pins can be obtained by the `pio_get()` function. The declaration of the function is:

```
uint32_t pio_get(Pio *p_pio, pio_type_t ul_type, uint32_t ul_mask);
```

The function obtains the actual voltage level at the specified pin(s). It returns one if at least one of the specified pins is high. Otherwise, when all the specified pins are low, zero is returned. It can be used to read the input pin(s), or to verify the output pin(s) that was(were) previously set or cleared by the `pio_set()` or `pio_clear()` function. The arguments of the `pio_get()` function are:

`p_pio`	...	PIOx registers base address (e.g., `PIOC` for PIOC peripheral device)
`ul_type`	...	pin(s) type (e.g., `PIO_INPUT` for input pin(s))
`ul_mask`	...	mask of the pin(s) to be read (e.g., `PIO_PC28` for pin PC28 of the PIOC device)

E.g., To read the input pin PC28 of the PIOC peripheral device, i.e., to obtain T1 key state, the `pio_get()` function with the appropriate argument values must be called:

```
pio_get(PIOC, PIO_INPUT, PIO_PC28);
```

The program

First, the hardware has to be initialized. Besides the µC and the board, the input and output pins have to be initialized. The lock is initially locked. Then enter into an endless loop. In each iteration, check all the keys and react accordingly. If any of the keys is pressed, set the key down signal and add the key to the entered combination. If the enter key is pressed, check the combination and unlock the lock in case the combination is right. After a delay, lock the lock. The pseudo code of the described algorithm is as follows:

```
initialize hardware (PIO pins)
lock the lock
while forever
    set down and enter variables to false
    for each key
        if the key is pressed
            set down to true
            if the key was not pressed in the previous iteration
                if this is the enter key
                    set enter to true
                else
                    add the key to the combination
    if enter is true
        if the combination is right
            unlock the lock
        delay
        lock the lock
    else if down is true
        set key down signal
    else
        clear key down signal
```

Exercise 3

Watchdog timer

Configure the WDT[1] of the AT91SAM3X8E µC. Write a testing program that drives two flashing LEDs on the external board simulating the railway crossing traffic lights. The program should regularly restart the WDT. Simulate a deadlock situation by pressing an external board key. The WDT should reset the system. The external boards 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, and to the external board button key and LEDs, as shown in Fig. 3.1. Initialize the key and LED I/O pins as explained in Exercise 2. Since the T1 key will be used only once to indicate a deadlock, the debouncing filter at the key input pin PC28 is not required.

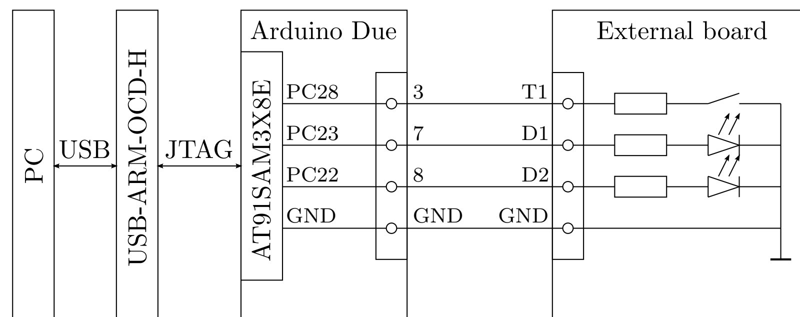


Figure 3.1: PC / Olimex ARM-USB-OCD-H / Arduino Due / External board connection for Exercise 3

Initialization of the WDT device

The WDT is disabled during the Arduino Due board initialization in the `board_init()` function call. Since the WDT cannot be reconfigured, the disabling in the `board_init()` must be avoided. This can be achieved by defining the `CONF_BOARD_KEEP_WATCHDOG_AT_INIT` macro in the `src/conf_board.h` file:

```
#define CONF_BOARD_KEEP_WATCHDOG_AT_INIT
```

[1]WDT ... WatchDog Timer

When the WDT is enabled, it has to be properly configured. The `wdt_init()` function can be used. The declaration of the function is:

```
void wdt_init(Wdt *p_wdt, uint32_t ul_mode, uint16_t us_counter,
                                                uint16_t us_delta);
```
[2]

Note, that WDT cannot be reconfigured. The `wdt_init()` function can be called only once. The function arguments are:

`p_wdt`	...	WDT registers base address (e.g., `WDT` for WDT peripheral device)
`ul_mode`	...	WDT configuration bitmask (e.g., `WDT_MR_WDRSTEN` to enable µC reset by WDT)
`us_counter`	...	WDV[3]
`us_delta`	...	WDD[4]

The restart value of the WDT counter is WDV. The counter is constantly decreased. A WDT fault occurs when the counter reaches zero. To avoid the fault, e.g., the µC reset, the WDT must be restarted before the counter reaches zero. However, the WDT cannot be restarted at any moment, but only when the counter is below the WDD. By setting WDD $<$ WDV, a time window when WDT restart is possible can be specified.

To configure the WDT, the integer values WDV and WDD are required. To convert a time interval into the corresponding integer value, the `wdt_get_timeout_value()` function can be used. The declaration of the function is:

```
uint32_t wdt_get_timeout_value(uint32_t ul_us, uint32_t ul_sclk);
```

The function returns the corresponding integer value. The arguments are:

`ul_us`	...	time interval in µs
`ul_sclk`	...	the SCLK[5] frequency in Hz (e.g., `BOARD_FREQ_SLCK_XTAL` for on board 32kHz XTAL[6])

E.g., Consider the following WDT configuration: 2s available for WDT restart[7], WDT restart always possible, reset µC on fault, stop when the µC is in debug mode or idle. The configuration can be set by appropriate use of the `wdt_get_timeout_value()` and `wdt_init()` functions:

```
ulTimeoutValue = wdt_get_timeout_value(2e6, BOARD_FREQ_SLCK_XTAL);
wdt_init(WDT, WDT_MR_WDDBGHLT | WDT_MR_WDIDLEHLT | WDT_MR_WDRSTEN,
                                 ulTimeoutValue, ulTimeoutValue);
```
[8]

The declarations of the `wdt_get_timeout_value()` and `wdt_init()` functions reside in the `wdt.h` header file, which has to be included.

```
#include <wdt.h>
```

[2] `Wdt` is a structure of 32-bit integers representing the AT91SAM3X8E WDT hardware register addresses. If the pointer to such a structure refers to the top of the WDT device address space, the structure elements directly represent the registers.
`uint16_t` is a 16-bit unsigned integer type.

[3] WDV ... WatchDog Value

[4] WDD ... WatchDog Delta

[5] SCLK ... Slow Clock (see Exercise 5)

[6] XTAL ... Crystal

[7] Maximal WDT restart period $T_{\max} = \frac{128(2^n-1)}{f_{\mathrm{SCLK}}}$, where n is the number of WDV bits. $T_{\max} < 16s$ at $f_{\mathrm{SCLK}} = 32768$Hz and $n = 12$.

[8] `WDT_MR_WDDBGHLT` mode stops the WDT when the µC is in debug mode.
`WDT_MR_WDIDLEHLT` mode stops the WDT when the µC is idle.

Restarting the WDT

To avoid the fault, the WDT counter must be restarted, i.e., set to WDV, before it reaches zero. The WDT can be restarted by the `wdt_restart()` function. The declaration of the function is:

```
void wdt_restart(Wdt *p_wdt);
```

The function argument is:

`p_wdt`	...	WDT registers base address (e.g., `WDT` for WDT peripheral device)

Therefore, the WDT can be restarted by a simple `wdt_restart()` call:

```
wdt_restart(WDT);
```

The program

After hardware initialization, the algorithm drives the LEDs in a railway crossing traffic lights simulation mode. Both LEDs are alternatively turned on and off. Between the turns, a delay takes place. In each iteration of the simulation, the WDT is restarted to avoid µC reset. The delay can be implemented in a `for` loop. During the delay, the external board button key is checked. If, or when the key is pressed, the algorithm enters into a deadlock, i.e., an endless loop. The program freezes, traffic lights stop blinking. However, after a certain amount of time, the WDT resets the µC. The railway traffic lights recover. The pseudo code of the described algorithm is as follows:

```
initialize hardware (PIO pins, WDT)
turn one LED on and the other off
while forever
        toggle both LEDs
        for (during) the delay time
                if the key is pressed
                        deadlock
        restart WDT
```

Toggling of the LEDs can be done by the `pio_set()` and `pio_clear()` functions introduced in Exercise 2. However, using a function `pio_toggle_pin()` is more convenient in this case. The declaration of the function is:

```
void pio_toggle_pin(uint32_t ul_pin);
```

The function argument is:

`ul_pin`	...	pin index (e.g., `PIO_PC23_IDX` for pin PC23 of the PIOC device)

Note that pin index (e.g., `PIO_PC23_IDX`) is not the same as pin mask (e.g., `PIO_PC23`). To toggle the output pin PC23 of the PIOC peripheral device, the `pio_toggle_pin()` function with a corresponding pin index has to be called:

```
pio_toggle_pin(PIO_PC23_IDX);
```

Exercise 4

UART

Write a program for the Arduino Due board implementing echo functionality on the UART peripheral device. A received character should be immediately transmitted back. This functionality should be maintained until an Esc character arrives. It switches the transmission off, while the reception normally continues. The next Esc character should switch the transmission back on, thus returning the program into the initial state. Implement the required task with and without using interrupts. Test the program with an arbitrary serial terminal program running on a PC. Note that the terminal program settings must be the same as the UART device settings.

Explanation

Create a new empty project in the Eclipse working environment as explained in Exercise 1. Connect the Programming USB port on the Arduino Due board to the PC as shown in Fig. 4.1. The additional on-board ATMEGA16U2 µC acts as an UART to USB converter.

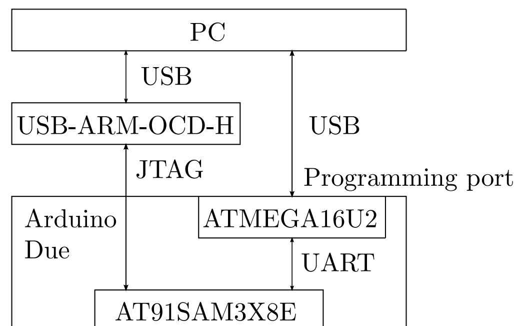


Figure 4.1: PC / Olimex ARM-USB-OCD-H / Arduino Due connection for Exercise 4

Initialization of the UART device

The AT91SAM3X8E µC has four PIOs (i.e., PIOA, PIOB, PIOC and PIOD) each controlling up to 32 I/O pins. An I/O pin can be configured as a GPIO pin like in Exercises 2 and 3, 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[1] and UTXD[2] 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 I/O pin needs to be reconfigured. The pull-up resistor reconfiguration is performed by the `pio_pull_up()` function declared as:

```
void pio_pull_up(Pio *p_pio, uint32_t ul_mask,
                                        uint32_t ul_pull_up_enable);
```

The function arguments are:

`p_pio`	...	PIOx registers base address (e.g., `PIOA` for PIOA peripheral device)
`ul_mask`	...	mask of pin(s) to be configured (e.g., `PIO_PA8` for pin PA8 of the PIOA device)
`ul_pull_up_enable`	...	pull-up resistor enable/disable flag (e.g., `PIO_PULLUP` to enable the pull-up resistor)

To reconfigure the PA8 I/O pin pull-up resistor, the `pio_pull_up()` function with the appropriate argument values must be called:

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

The declaration of the `pio_pull_up()` function resides in the `pio.h` header file, which has to be included.

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

The stdio[3] and UART peripheral device configuration is performed by the `stdio_serial_init()` function declared as:

```
void stdio_serial_init(void *usart, usart_serial_options_t *opt);
```
[4]

The function arguments are:

[1] URXD ... UART Receive Data

[2] UTXD ... UART Transmit Data

[3] stdio ... Standard I/O

[4] The `usart_serial_options_t` type is a structure defining UART properties:

```
typedef struct {
    uint32_t baudrate;   /* baud rate                                    */
    uint32_t paritytype; /* parity type                                  */
        ...
} usart_serial_options_t;
```

`usart`	...	UART or USART[5] registers base address (e.g., `UART` for UART peripheral device)
`opt`	...	pointer to a structure with UART or USART serial option values

To configure the stdio in serial mode and initialize the UART device, the `stdio_serial_init()` function with the appropriate argument values must be called:

```
usart_serial_options_t xUARTconf = {.baudrate = 38400,
                                    .paritytype = UART_MR_PAR_NO};[6]
stdio_serial_init(UART, &xUARTconf);[7]
```

The declaration of the `stdio_serial_init()` function resides in the `stdio_serial.h` header file, which has to be included.

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

At this point, the UART peripheral device is initialized and ready. 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 program

As usual, hardware initialization comes first. The characters are read from the UART device in an endless loop. Regarding the transmission mode, they are immediately written back or discarded. The pseudo code of the described algorithm is as follows:

```
initialize hardware (UART)
set transmission to true
while forever
    read a character from UART
    if character is Esc
        toggle transmission
    else if transmission is true
        write the character to UART
```

Interrupt configuration

The interrupt requests are handled by the NVIC[9]. In order to execute the interrupt handler code when requested, the NVIC has to be appropriately configured. The NVIC configuration is carried out after the UART peripheral device configuration performed in the `stdio_serial_init()` function.

The UART peripheral device should be disabled during the NVIC configuration. It is disabled by the `uart_disable()` function declared as:

[5] USART ... Universal Synchronous Asynchronous Receiver Transmitter

[6] Value `UART_MR_PAR_NO` defines no parity check mode.

[7] Note that the speed of the UART peripheral device is calculated regarding the MCK[8] settings in the `src/conf_clock.h` file. If the `sysclk_reinit()` function initializes MCK differently, UART reinitialization with the `uart_init()` function is required (see Exercise 5).

[8] MCK ... Master Clock

[9] NVIC ... Nested Vectored Interrupt Controller

```
void uart_disable(Uart *p_uart);
```
[10]

The function argument is:

`p_uart`	...	UART registers base address (e.g., `UART` for UART peripheral device)

To disable the UART device, the `uart_disable()` function with the UART base address must be called:

```
uart_disable(UART);
```

Further, all the UART interrupt request sources (e.g., request an interrupt when data is received, etc.) should be disabled. They are disabled by the `uart_disable_interrupt()` function declared as:

```
void uart_disable_interrupt(Uart *p_uart, uint32_t ul_sources);
```

The function arguments are:

`p_uart`	...	UART registers base address (e.g., `UART` for UART peripheral device)
`ul_sources`	...	mask of the interrupt sources to be disabled

To disable all the UART interrupt sources, the `uart_disable_interrupt()` function with the appropriate argument values must be called:

```
uart_disable_interrupt(UART, 0xffff);
```

Now, the interrupt request issued by the UART peripheral device can be enabled in the NVIC. This is performed by the CMSIS `NVIC_EnableIRQ()` function declared as:

```
void NVIC_EnableIRQ(IRQn_Type IRQn);
```
[11]

The function argument is:

`IRQn`	...	peripheral device identifier (e.g., `ID_UART` for UART peripheral device)

To enable the UART peripheral device interrupt request, the `NVIC_EnableIRQ()` function with the UART identifier must be called:

```
NVIC_EnableIRQ(ID_UART);
```

The UART interrupt sources that cause the interrupt request are enabled by the `uart_enable_interrupt()` function declared as:

```
void uart_enable_interrupt(Uart *p_uart, uint32_t ul_sources);
```

The function arguments are:

[10] `Uart` is a structure of 32-bit integers representing the AT91SAM3X8E UART hardware register addresses. If the pointer to such a structure refers to the top of the UART device address space, then the structure elements directly represent the registers.

[11] `IRQn_Type` is an enumerated type with peripheral device identifiers [8].

`p_uart`	...	UART registers base address (e.g., `UART` for UART peripheral device)
`ul_sources`	...	mask of the interrupt sources to be enabled (e.g., `UART_IER_RXRDY` for requesting an interrupt when data is received)

To enable UART interrupt request on the data received event, the `uart_enable_interrupt()` function with the appropriate argument values must be called:

```
uart_enable_interrupt(UART, UART_IER_RXRDY);
```

Finally, the UART device is enabled by the `uart_enable()` function declared as:

```
void uart_enable(Uart *p_uart);
```

The function argument is:

`p_uart`	...	UART registers base address (e.g., `UART` for UART peripheral device)

To enable the UART device, the `uart_enable()` function with the UART base address must be called:

```
uart_enable(UART);
```

The ISR[12]

As configured, the UART device requests an interrupt when it receives a character. The NVIC gets the UART request and starts the corresponding ISR code starting at the address given in the vector table defined in the `sam/utils/cmsis/sam3x/source/templates/gcc/startup_sam3x.c` file of the ASF. As defined in the vector table, the UART ISR is the `UART_Handler()` function. It is declared as a weak symbol[13] and as such can be overridden. Therefore a new definition of the `UART_Handler()` function is required in the user code:

```
void UART_Handler(void) {
	...
}
```

The `UART_Handler()` function is called every time the UART requests an interrupt, i.e., on each received character. Therefore, the algorithm of the function is reduced to the body of the endless loop from page 25 where the `transmission` is a global or `static` variable:

```
read a character from UART
if character is Esc
	toggle transmission
else if transmission is true
	write the character to UART
```

[12]ISR ... Interrupt Service Routine

[13]The `UART_Handler` symbol is declared as a weak alias of the `Dummy_Handler` symbol:

```
void Dummy_Handler(void) {
  while(1);
}
void UART_Handler(void) __attribute__((weak, alias("Dummy_Handler")));
```

After the received character is handled, the ISR must return. It must not wait for another character.

Exercise 5

µC initialization

Write a program printing the information about the current AT91SAM3X8E µC settings (i.e., PMC[1] clock generator related settings) to the console. Use UART peripheral device as the stdio, and an arbitrary serial terminal program as a console. Beside printing the information, the program should also implement a user interface allowing µC settings modification.

Explanation

Connect the Arduino Due board to the host PC as shown in Fig. 4.1. Create a new empty project in the Eclipse working environment as explained in Exercise 1. Initialize the UART peripheral device and configure the stdio in serial mode as explained in Exercise 4.

µC settings at startup

As already mentioned in Exercise 1, at power-up, the µC loads the reset vector into the program counter register, and starts with program execution. The reset vector is the second entry in the vector table in the `sam/utils/cmsis/sam3x/source/templates/gcc/startup_sam3x.c` file, refering to the `Reset_Handler()` function. The `Reset_Handler()` function performs some basic preparations (see footnote[25] in Exercise 1) before the `main()` function is called.

The `main()` function does the µC initialization by calling the `sysclk_reinit()`[2] function. It initializes the PMC clock generator settings to be printed and reconfigured in this exercise. Besides the clock generator configuration, the `sysclk_reinit()` function also sets the `SystemCoreClock` global variable containing the processor MCK frequency (f_{MCK}), and, according to the f_{MCK}, the number of FWSs[3] in the EEFC[4]. Since the embedded flash memory access time is approximately constant, the FWS value increases with the f_{MCK} (see Tab. 5.1).

Clock generation

The PMC clock generator is configured by bit values in several registers. The MCK generation is best explained in Fig. 5.1 showing the registers and bit values

[1] PMC ... Power Management Controller

[2] The `main()` function actually calls the `prvSetupHardware()` function which performs µC and other hardware initializations. The `sysclk_reinit()` function is called from the `prvSetupHardware()` function. The `sysclk_reinit()` function is not a part of the ASF, although it is added into the ASF in [13]. It is a rewrite of the original `sysclk_init()` function which initializes the PMC clock generator according to the macros in the `conf_clock.h` file.

[3] FWS ... Flash Wait State

[4] EEFC ... Enhanced Embedded Flash Controller

FWS	$f_{\mathrm{MCK}_{\max}}$ [MHz]
0	19
1	50
2	64
3	80
4	90

Table 5.1: Maximum MCK frequency ($f_{\mathrm{MCK}_{\max}}$) regarding to the FWS value at VDDCORE = 1.8V (core chip power supply)

causing a particular configuration (e.g., the `MOSCXTBY`, `MOSCSEL` and `MOSCRCF` bits in the `CKGR_MOR` register select the MAINCK[5] source). There are some restraints when PLLA[6] or UPLL[7] is selected as MCK source. The PLLA circuitry demands an input from 8MHz to 16MHz, and the output range is from 84MHz to 192MHz. Thus the embedded 4MHz RC oscillator cannot be selected as MAINCK source, and the `MULA` must be set according to the output range. The UPLL on the other hand requires a 12MHz input generated by the external crystal oscillator. Note that the Arduino Due board is equipped with the 32768Hz and 12MHz external crystal oscillators [9]. The lowest MCK is around 500Hz (512Hz at SLCK = 32768Hz), the highest is 84MHz which can be generated only by PLLA.

The default JTAG clock frequency defined in the AT91SAM3X8E configuration file included from the `openocd.cfg` is 500kHz. The fastest JTAG clock the processor allows is one sixth of the MCK [6] (5.1).

$$f_{\mathrm{JTAG}} < \frac{f_{\mathrm{MCK}}}{6} \tag{5.1}$$

Since the MCK is generated by the embedded 4MHz RC oscillator at the reset, the fastest JTAG clock could be 666.66kHz. Choosing 500kHz is safely below the limit. Although the processor can operate at very low MCK, the frequencies below 4MHz cannot be used when debugging over the 500kHz JTAG connection. The JTAG frequency can be lowered by adding the `adapter_khz` command in the `openocd.cfg` file (see Exercise 1). The command

```
adapter_khz 150
```
[9]

sets the f_{JTAG} to 150kHz, thus allowing $f_{\mathrm{MCK}} > 900$kHz (or ≥ 1MHz to be safe).

Get current settings

Information about the current clock generation configuration can be extracted from the `SUPC_CR`, `SUPC_MR`, `CKGR_MOR`, `CKGR_PLLAR` and `PMC_MCKR` registers [8] (see Fig. 5.1). Since only frequencies above 1MHz are to be considered, the SLCK should not be used as the MCK source in this exercise. Thus, the `SUPC_CR` and `SUPC_MR` registers become irrelevant. Bypassing the external crystal oscillator by providing an external clock signal on the `XIN` pin is also not available since there is no additional oscillator on the Arduino Due board. Therefore, the three

[5] MAINCK ... Main Clock

[6] PLLA ... Phase-Locked Loop A

[7] UPLL ... UTMI[8] Phase Lock Loop

[8] UTMI ... USB 2.0 Transceiver Macrocell Interface

[9] It turns out that the GDB cannot connect to the target at f_{JTAG} below 3kHz.

[10] PLLACK ... PLLA Clock

[11] UPLLCK ... UPLL Clock

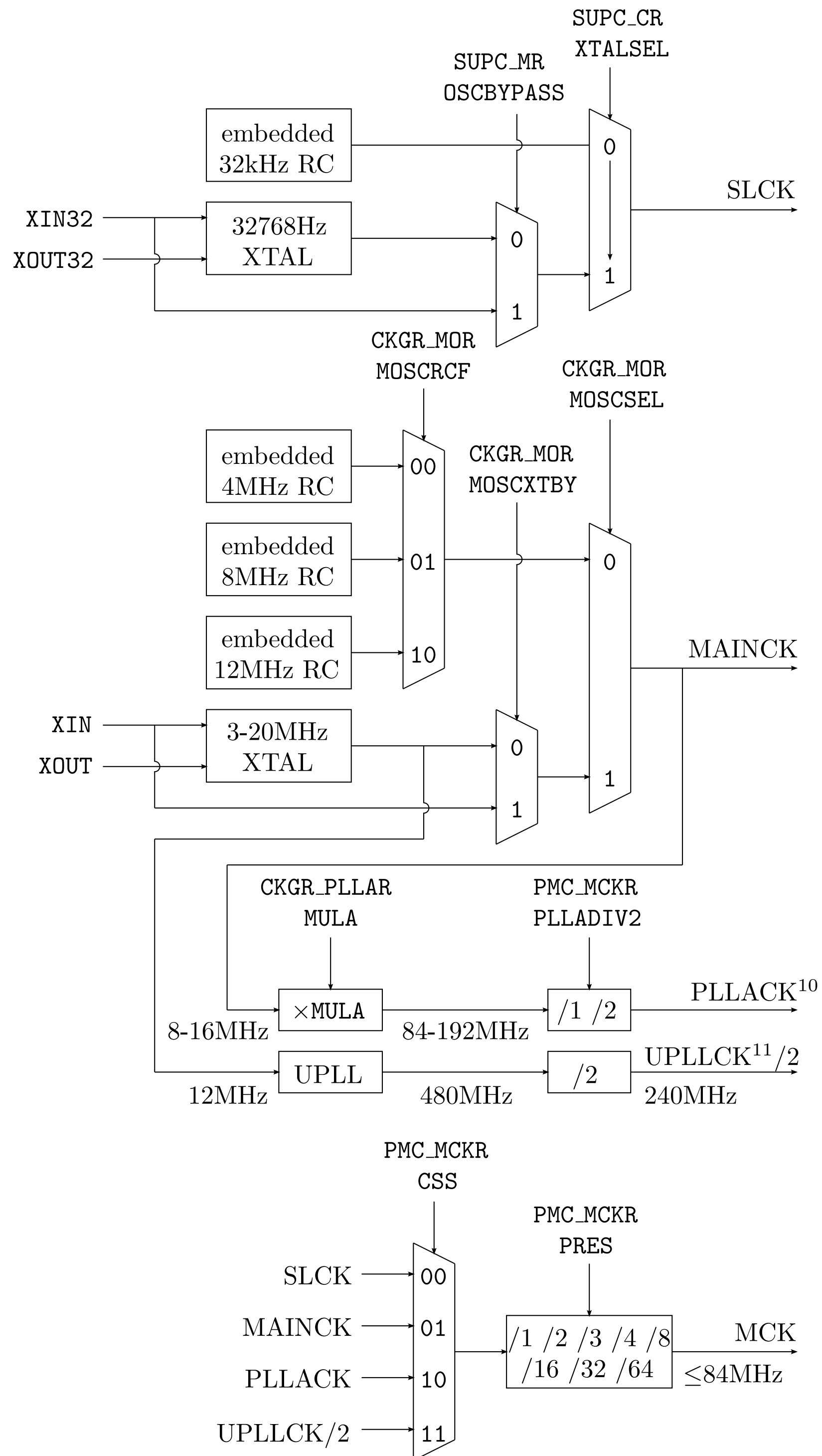


Figure 5.1: MCK generation mechanism

embedded fast RC oscillators, external crystal oscillator and both PLLs can be used as possible MCK sources in this exercise.

With SLCK never used, the current configuration can be retrieved from the three PMC registers `CKGR_MOR`, `CKGR_PLLAR` and `PMC_MCKR` (see Fig. 5.1). The registers can be easily accessed through the `PMC`[12] global pointer provided by the ASF. The bit mask and value macros are also defined. The registers, masks and values needed in this exercise are in Tab. 5.2.

if `(PMC->CKGR_MOR & CKGR_MOR_MOSCSEL)`

is	then
(un)set	external XTAL oscillator is (not) selected

if `(PMC->CKGR_MOR & CKGR_MOR_MOSCRCF_Msk)`

is	then
`CKGR_MOR_MOSCRCF_4_MHz`	embedded 4MHz RC oscillator is selected
`CKGR_MOR_MOSCRCF_8_MHz`	embedded 8MHz RC oscillator is selected
`CKGR_MOR_MOSCRCF_12_MHz`	embedded 12MHz RC oscillator is selected

if `(PMC->PMC_MCKR & PMC_MCKR_PRES_Msk)`

is	then
`PMC_MCKR_PRES_CLK_1`	division by 1 is selected
`PMC_MCKR_PRES_CLK_2`	division by 2 is selected
`PMC_MCKR_PRES_CLK_3`	division by 3 is selected
`PMC_MCKR_PRES_CLK_4`	division by 4 is selected
`PMC_MCKR_PRES_CLK_8`	division by 8 is selected
`PMC_MCKR_PRES_CLK_16`	division by 16 is selected
`PMC_MCKR_PRES_CLK_32`	division by 32 is selected
`PMC_MCKR_PRES_CLK_64`	division by 64 is selected

if `(PMC->PMC_MCKR & PMC_MCKR_CSS_Msk)`

is	then
`PMC_MCKR_CSS_MAIN_CLK`	MAINCK is selected
`PMC_MCKR_CSS_PLLA_CLK`	PLLACK is selected
`PMC_MCKR_CSS_UPLL_CLK`	UPLLCK/2 is selected

if `(PMC->PMC_MCKR & PMC_MCKR_PLLADIV2)`

is	then
unset / set	division by one / two is selected

`MULA` − 1 = `(PMC->CKGR_PLLAR & CKGR_PLLAR_MULA_Msk) >> 16`

Table 5.2: ASF provided registers, masks and belonging values

Set new settings

The µC clock reconfiguration to a new settings is performed by the `sysclk_reinit()` function in the same way as at the reset. The function declaration is:

```
uint32_t sysclk_reinit(uint32_t osc, uint32_t pres, uint32_t mck,
                                     uint32_t mula, uint32_t plladiv2);
```

[12] The `PMC` global pointer refers to a structure with PMC registers. Since it is pointed to the beginning of the PMC address space, the PMC registers can be easily accessed, e.g., the registers mentioned above can be accessed by `PMC->CKGR_MOR`, `PMC->CKGR_PLLAR` and `PMC->PMC_MCKR`.

The function returns zero on success, and one in case the clock reconfiguration to the specified settings fails. The arguments are:

`osc`	...	oscillator selection (has to be one of the following self-explaining enumerated type values[13]: `OSC_SLCK_32K_RC`, `OSC_SLCK_32K_XTAL`, `OSC_SLCK_32K_BYPASS`, `OSC_MAINCK_4M_RC`, `OSC_MAINCK_8M_RC`, `OSC_MAINCK_12M_RC`, `OSC_MAINCK_XTAL` and `OSC_MAINCK_BYPASS`)
`pres`	...	prescaler value (has to be one of the following self-explaining bit values: `PMC_MCKR_PRES_CLK_1`, `PMC_MCKR_PRES_CLK_2`, `PMC_MCKR_PRES_CLK_3`, `PMC_MCKR_PRES_CLK_4`, `PMC_MCKR_PRES_CLK_8`, `PMC_MCKR_PRES_CLK_16`, `PMC_MCKR_PRES_CLK_32`, and `PMC_MCKR_PRES_CLK_64`)
`mck`	...	MCK selection (has to be one of the following self-explaining bit values[14]: `PMC_MCKR_CSS_SLOW_CLK`, `PMC_MCKR_CSS_MAIN_CLK`, `PMC_MCKR_CSS_PLLA_CLK`, and `PMC_MCKR_CSS_UPLL_CLK`)
`mula`	...	PLLA multiplier value[15] (considered only when the `mck` argument is set to `PMC_MCKR_CSS_PLLA_CLK`)
`plladiv2`	...	additional PLLA divisor (possible values are `1` and `2`; considered only when the `mck` argument is `PMC_MCKR_CSS_PLLA_CLK`)

For instance, the `sysclk_reinit()` function is called with the following argument values at the reset:

```
sysclk_reinit(OSC_MAINCK_XTAL, PMC_MCKR_PRES_CLK_16,
                                        PMC_MCKR_CSS_PLLA_CLK, 14, 1);
```

The 12MHz external crystal oscillator is selected as the MAINCK source. It is the input of the PLLA providing the MCK. The MCK frequency is $f_{\mathrm{MCK}} = 12\mathrm{MHz} \times 14 / 1 / 16 = 10.5\mathrm{MHz}$ which can be confirmed by the `SystemCoreClock` global variable.

UART reinitialization

Every time the MCK changes, the UART baud rate is modified as well. It is defined by the `UART_BRGR` register (5.2).

$$baud\ rate = \frac{f_{\mathrm{MCK}}}{16 \times \texttt{UART_BRGR}} \tag{5.2}$$

With every MCK change, the serial terminal program on the PC should be restarted to utilize the new baud rate. To avoid the baud rate modifications, and consequently the serial terminal program restarts, the UART device can be reinitialized to the previous baud rate after every MCK change. The UART reinitialization is performed by the `uart_init()` function declared as:

[13] Since the SLCK cannot be used because the MCK should be above 1MHz, and the external crystal oscillator cannot be not bypassed because the Arduino Due board does not have an additional external oscillator, the `OSC_SLCK_32K_RC`, `OSC_SLCK_32K_XTAL`, `OSC_SLCK_32K_BYPASS` and `OSC_MAINCK_BYPASS` values will not be used in this exercise.

[14] Selection `PMC_MCKR_CSS_SLOW_CLK` will not be used in this exercise.

[15] The relation $84\mathrm{MHz} \leq \texttt{mula} \times f_{\mathrm{PLLA_{input}}} \leq 192\mathrm{MHz}$ must hold.

```
uint32_t uart_init(Uart *p_uart, sam_uart_opt_t *p_uart_opt);
```
[16]

The function returns zero on success, and one in case the baud rate cannot be realized. The function arguments are:

`p_uart`	...	UART registers base address (e.g., `UART` for UART peripheral device)
`p_uart_opt`	...	pointer to a structure with UART initialization options

To reinitialize the UART device to the same baud rate and parity type, the `uart_init()` function with the appropriate argument values must be called. The UART device options were already defined by the `usart_serial_options_t` type structure passed to the `stdio_serial_init()` function during the initialization of the stdio in serial mode (see page 25 in Exercise 4). Thus, the element values of the `usart_serial_options_t` structure has to be copied into the `sam_uart_opt_t` structure:

```
sam_uart_opt_t xUARTsettings;
xUARTsettings.ul_baudrate = xUARTconf.baudrate;
xUARTsettings.ul_mode = xUARTconf.paritytype;
      ...
xUARTsettings.ul_mck = SystemCoreClock;
uart_init(UART, &xUARTsettings);
```

Because of (5.2), an arbitrary baud rate cannot be obtained with an arbitrary MCK. For instance, the speed of 38400 bauds cannot be realized with $f_{\mathrm{MCK}} =$ 1MHz with sufficient accuracy. The `UART_BRGR` should be 1.63, which is not possible. For `UART_BRGR = 1`, 62500 bauds are obtained, and 31250 bauds for `UART_BRGR = 2`, both to far away from the 38400 baud target. Thus, a UART speed that can be realized with sufficient accuracy for a range of MCK frequencies from 1MHz to 84MHz has to be picked. It turns out that the 2400 bauds is one of such speeds.

Yet another UART speed effect can emerge. A stdio function (e.g., `printf()` or similar) can be used to send a string of characters to the UART to be transmitted. The function delivers the characters to the UART device and returns before the characters actually get transmitted. If the MCK is changed immediately after the function call, the baud rate can be changed during the ongoing transmission. The effect becomes more probable with a low baud rate (i.e., slow transmission) and a high MCK (i.e., fast execution). To avoid it, the transmission must complete before the MCK change. The function `uart_is_tx_empty()` checking whether the transmission is completed can be used. Its declaration is:

```
uint32_t uart_is_tx_empty(Uart *p_uart);
```

The function returns one if the transmitter is empty (i.e., transmission is completed), and zero otherwise. The function argument is:

`p_uart`	...	UART registers base address (e.g., `UART` for UART peripheral device)

To wait for UART transmission to complete, a while loop can be used:

---

[16] The `sam_uart_opt_t` type is a structure defining UART initialization options:

```
typedef struct {
      uint32_t ul_mck;      /* MCK                      */
      uint32_t ul_baudrate; /* baud rate                */
      uint32_t ul_mode;     /* UART mode register value */
} sam_uart_opt_t;
```

```
while(!uart_is_tx_empty(UART));
```

The program

The realization of the exercise takes some C programming. To program the console user interface, the stdio functions (e.g., `printf()`, `getchar()`, etc.) can be used. Use `sysclk_reinit()` function to reinitialize the MCK, and the `uart_init()` and `uart_is_tx_empty()` functions to address the UART speed issue.

After hardware initialization, the algorithm enters into an endless loop. The current MCK settings are printed, new ones obtained, and µC reinitialized in each iteration. Note that the MCK must never be initialized below 1MHz in order to keep the JTAG connection alive. The pseudo code of the described algorithm is as follows:

```
initialize hardware (UART)
while forever
        print current configuration according to PMC register values
        obtain new settings (code some user interface)
        wait for transmission to complete
        reinitialize MCK
        reinitialize UART
```

Exercise 6

Timer

Write a program for the Arduino Due board implementing a pulse dialing generator. The program should read a number from the `stdin` and generate the corresponding dialing signal at a pin connected to the on-board LED. Use UART peripheral device as stdio. Implement the required task with and without using interrupts.

Explanation

Connect the Arduino Due board to the host PC as shown in Fig. 4.1. Create a new empty project in the Eclipse working environment as explained in Exercise 1. Initialize the UART peripheral device and configure the stdio in serial mode as explained in Exercise 4.

Pulse dialing

A device used to produce pulse trains encoding a telephone number is the rotary dial. There are several versions of digit encoding. In the most common version, the non-zero digits are encoded by the equivalent number of pulses, and the zero digit is encoded as ten pulses. The pulse frequency, duty cycle and pause between the two trains are not strictly set. Ten pulses per second, 39% duty cycle, and cca. one second pause are generally used (Fig. 6.1). However, pulse dialing is obsolete and only rarely used today.

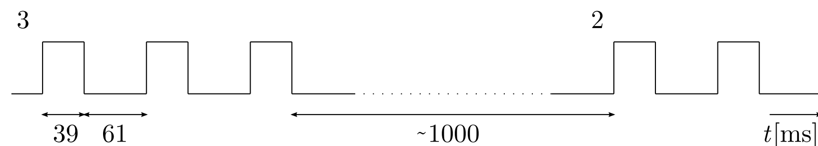


Figure 6.1: Numbers as trains of pulses in pulse dialing

Initialization of the TC[1] device

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.

[1]TC ... Timer Counter

A TC channel has to be clocked first. The clock is provided through the PMC. To enable a peripheral device clock, the `pmc_enable_periph_clk()` function can be used. Its declaration is:

```
uint32_t pmc_enable_periph_clk(uint32_t ul_id);
```

The function returns zero on success, and one in case of invalid peripheral identifier. The argument is:

`ul_id`	...	peripheral device identifier (e.g., `ID_TC0`[2] for TC channel 0)

To enable the peripheral clock for the TC channel 0, the channel's identifier must be passed to the `pmc_enable_periph_clk()` function:

```
pmc_enable_periph_clk(ID_TC0);
```

Similarly, the `pmc_enable_periph_clk()` function is used in the `board_init()` and `stdio_serial_init()` calls to enable the PIO and UART peripheral devices, respectively.

With TC channel clocked, it becomes available. Configuration of a TC channel is performed by the `tc_init()` function declared as:

```
void tc_init(Tc *p_tc, uint32_t ul_channel, uint32_t ul_mode);
```
[3]

The function arguments are:

`p_tc`	...	TC module registers base address (e.g. `TC0`[4] for the first TC module)
`ul_channel`	...	channel index[5]
`ul_mode`	...	`TC_CMRx`[6] value

The `TC_CMRx` defines the operating mode of the specified TC channel (see [8] for detailed information). Each channel can operate independently in two different modes: capture mode provides measurement on signals, and waveform mode provides wave generation. Fig. 6.2 shows a free up-running TC channel in a waveform mode generating an output signal. The counter increases with the maximum counting frequency $\frac{f_{\mathrm{MCK}}}{2}$, and is automatically reset on RCx[7] compare event. The channel 0 of the TC0 module can be configured as shown in Fig. 6.2 with the following `tc_init()` call:

```
tc_init(TC0, 0, TC_CMR_EEVT_XC0 | TC_CMR_WAVSEL_UP_RC | TC_CMR_WAVE
                        | TC_CMR_BCPB_SET | TC_CMR_BCPC_CLEAR);
```
[8]

[2] TC channel identifiers are `ID_TCx`, where `x` represents channel number from 0 to 8.

[3] `Tc` is a structure of 32-bit integers representing the AT91SAM3X8E TC hardware register addresses. If the pointer to such a structure refers to the top of the TC device address space, then the structure elements directly represent the registers.

[4] TC module identifiers are `TCx`, where `x` represents module number from 0 to 2.

[5] Channel index (0, 1, or 2) in a module, e.g., TC channel 4 is channel 1 of `TC1` module.

[6] `TC_CMRx` ... TC channel x Mode Register

[7] RCx ... TC channel x Register C

[8] `TC_CMR_EEVT_XC0` mask selects the signal XC0 as an external event.
`TC_CMR_WAVSEL_UP_RC` mask selects up counting mode with reset on RCx compare event.
`TC_CMR_WAVE` mask selects waveform mode.
`TC_CMR_BCPB_SET` mask selects setting of the TIOBx[9] output on RBx[10] compare event.
`TC_CMR_BCPC_CLEAR` mask selects clearing of the TIOBx output on RCx compare event.
Numerous of other `TC_CMRx` masks can be found in the `sam/utils/cmsis/sam3x/include/component/component_tc.h` file.

[9] TIOBx ... TC channel x I/O line B

[10] RBx ... TC channel x Register B

Each channel controls two I/O lines, TIOAx[11] and TIOBx[12]. Since the `tc_init()` call above initializes the TC channel 0 in a waveform mode generating the output signal at TIOB0, the TIOB0 line must not be configured as an external event input, which is the default. Although not used, the external event input is shifted to the XC0 line by the `TC_CMR_EEVT_XC0` mask.

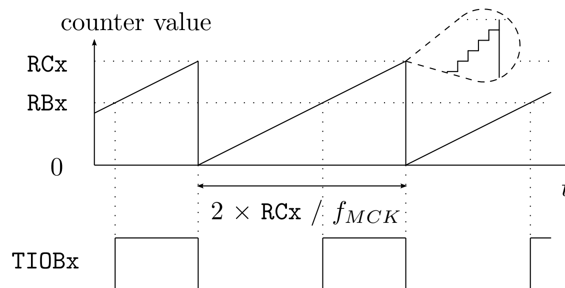


Figure 6.2: Timer configuration

The signal at TIOBx output is controlled by the RBx an RCx register values. The registers can be set with the `tc_write_rb()` and `tc_write_rc()` functions declared as:

```
void tc_write_rb(Tc *p_tc, uint32_t ul_channel, uint32_t ul_value);
void tc_write_rc(Tc *p_tc, uint32_t ul_channel, uint32_t ul_value);
```

The arguments are:

`p_tc`	...	TC module registers base address (e.g., `TC0` for the first TC module)
`ul_channel`	...	channel index
`ul_value`	...	value to be written to the register

For the purpose of this exercise, a signal with 100ms period and 39% duty cycle is required. Such a signal can be generated with the following RB0 and RC0 settings:

```
tc_write_rb(TC0, 0, (uint32_t)(0.061 * SystemCoreClock / 2));
tc_write_rc(TC0, 0, (uint32_t)(0.1 * SystemCoreClock / 2));
```

Note that the `SystemCoreClock` global variable contains the f_{MCK} value (see Exercise 5). For the described TC configuration (i.e., set TIOBx on RBx, clear TIOBx on RCx), the combination RBx > RCx causes TIOBx to be always low.

With TC channel configuration completed, the channel is ready to be started. To start the clock on the specified TC channel, the `tc_start()` function can be used. Its declaration is:

```
void tc_start(Tc *p_tc, uint32_t ul_channel);
```

The function arguments are:

`p_tc`	...	TC module registers base address (e.g., `TC0` for the first TC module)
`ul_channel`	...	channel index

To start the channel 0 of the TC0 module, use:

[11]TIOAx ... TC channel x I/O line A
[12]TIOBx ... TC channel x I/O line B

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

The declarations of the tc_init(), tc_write_rc(), tc_write_rc() and tc_start() functions reside in the tc.h header file, which has to be included.

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

There are four PIOs in the AT91SAM3X8E µC, each controlling up to 32 I/O pins. An I/O pin can be configured as a GPIO pin, or as a pin hardwired to a particular peripheral device. However, any pin cannot be hardwired to any peripheral device. Each pin has up to two predefined peripheral devices, A and B, that can be hardwired to it. The TIOB0 output line is available to be hardwired to pin PB27 of the PIOB controller as a peripheral device B. Use pio_set_peripheral() function to hardwire the PB27 pin to its B device (i.e., hardwire the PB27 pin to the TIOB0 output). The declaration of the function is:

```
void pio_set_peripheral(Pio *p_pio, pio_type_t ul_type,
                                                  uint32_t ul_mask);
```

The function arguments are:

p_pio	...	PIOx registers base address (e.g., PIOB for PIOB peripheral device)
ul_type	...	pin(s) type (e.g., PIO_PERIPH_B for B device controlled pin)
ul_mask	...	mask of the pin(s) to be configured (e.g., PIO_PB27 for pin PB27 of the PIOB device)

To make the PB27 I/O pin controlled by its B device, i.e., the TIOB0 output line of the TC channel 0, the pio_set_peripheral() function with appropriate argument values must be called:

```
pio_set_peripheral(PIOB, PIO_PERIPH_B, PIO_PB27);
```

The TIOB0 output of the TC channel 0 is not selected by chance. The on-board LED is connected to the PB27 I/O pin of the PIOB. Thus, the generated pulses can be visualized by the blinking LED.

Similarly, the pio_set_peripheral() function is used in the board_init() call when the CONF_BOARD_UART_CONSOLE macro is defined to configure the PA8 and PA9 pins of the PIOA controller as UART pins (see Exercise 4).

The program

The pseudo code of the algorithm implementing this exercise without interrupts is as follows:

```
initialize hardware (UART, TC0)
set 0% duty cycle (i.e, set RB0 beyond 100% of timer period)
set slice to zero
while forever
    read number from UART
    echo it back (optional)
    clear TC0 events (i.e., read TC_SR0 register)
    do
        wait for TC0 counter reset (i.e., RC0 compare event)
        if slice is zero
            slice = number + number of slices in 1s pause
            set 39% duty cycle (i.e., set RB0 to 61% of timer period)
        if only pause left in slice
            set 0% duty cycle (i.e, set RB0 beyond 100% of timer period)
        decrement slice
    while slice is not zero
```

The algorithm requires some additional explanation. UART and TC0 peripheral devices are configured in hardware initialization. The TC channel 0 is configured as described in previous section, and can therefore generate a train of pulses at the TIOB0 output line. The duty cycle of the pulse train is set by the RB0 register value (see Fig. 6.2). The stdio is configured in serial mode at the UART peripheral device. It will serve as a console. The usual endless loop follows.

One `number` is dialed in each iteration of the endless loop. The `number` is obtained from the console using the stdio functions (e.g., `getchar()`). The dialing is then performed in the internal `do`/`while` loop. In each iteration, it waits for a reset of the TC channel 0 counter, i.e., RC0 compare event (see Fig. 6.2), which occurs every 100ms. The `do`/`while` loop is therefore carried out exactly ten times per second. The `slice` variable is used to count the 100ms intervals or slices. To dial a `number` with one second pause included, a `number` + 10 slices are required; `number` slices for the pulses and 10 slices for the one second pause between the two pulse trains. Note that number zero is represented by ten pulses. The pulses are generated at the TIOB0 output of the TC channel 0 by setting the appropriate RB0 value. During the pause, the RB0 is set above the RC0, thus holding the TIOB0 low (see Fig. 6.2). After the `slice` countdown, the `do`/`while` loop ends, a new iteration of the endless loop starts.

To detect an RC0 compare event, an information about occurred events is needed. It is contained in the `TC_SRx` TC channel status register. The register can be read by the `tc_get_status()` function declared as:

```
uint32_t tc_get_status(Tc *p_tc, uint32_t ul_channel);
```

The function returns the selected `TC_SRx` status register value. The arguments are:

`p_tc`	...	TC module registers base address (e.g., `TC0` for the first TC0 module)
`ul_channel`	...	channel index

The `TC_SRx` register is cleared on every read. Waiting for the RC0 compare event can be realized with the following while loop:

```
while(!(tc_get_status(TC0, 0) & TC_SR_CPCS));
```
[13]

[13] To mask out the RCx compare event status from the `TC_SRx` status register, the `TC_SR_CPCS` bit mask is used.

Circular FIFO[14] buffer

A circular FIFO buffer (see Fig. 6.3) is represented by an array of n elements. Each element can hold one data item, e.g., one number. The indexes `ucBegin` and `ucEnd` define the current state. The `ucBegin` index refers to the data item to be first read from the buffer, and the *ucEnd* index refers to the vacant element for the next incoming data to be written to. In other words, the `ucEnd` is an entry point, and the `ucBegin` is a getting out point. Both indexes are incremented in a circular manner. If the `ucBegin` and `ucEnd` are the same, the buffer is empty. That implicates that the buffer is full when the `ucBegin` equals to `ucEnd` + 1 in circular manner, and that at most n - 1 pieces of data can be stored.

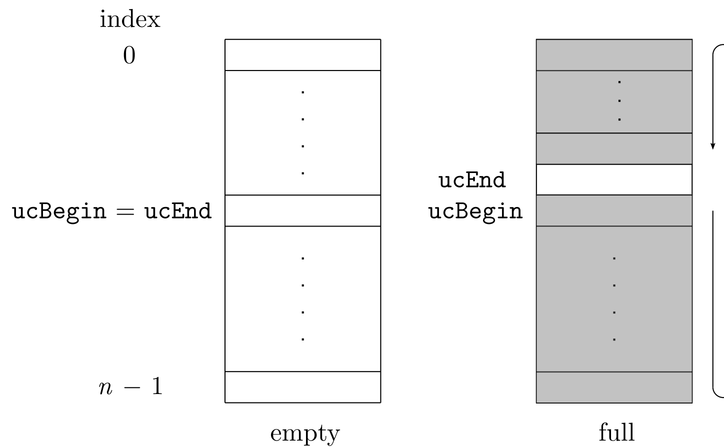


Figure 6.3: Circular FIFO buffer

A circular FIFO buffer `buf` with n data items of a particular *type* is defined by:

```
uint8_t ucBegin = 0, ucEnd = 0;[15]
type buf[n];
```

Writing one data *item* into the circular FIFO buffer `buf` is performed by the code below. Before write, the buffer is checked if it is full:

```
uint8_t ucTmp = ucEnd + 1;
if(ucTmp == n) ucTmp = 0;
if(ucTmp != ucBegin) {
      buf[ucEnd] = item;
      ucEnd = ucTmp;
}
```

Reading one data *item* from the circular FIFO buffer `buf` is performed by the code below. Before read, the buffer is checked if it is empty:

```
if(ucBegin != ucEnd) {
      item = buf[ucBegin];
      ucBegin = ucBegin + 1;
      if(ucBegin == n) ucBegin = 0;
}
```

[14]FIFO ... First In First Out

[15]`uint8_t` is an 8-bit unsigned integer type.

Interrupt configuration

The procedure of configuring the UART interrupt is described in Exercise 4. Configuring the TC interrupt is similar. The interrupt request issued by the TC channel is enabled by the CMSIS `NVIC_EnableIRQ()` function:

```
NVIC_EnableIRQ(ID_TC0);
```

Channel events that trig the TC channel interrupt request are enabled by the `tc_enable_interrupt()` function declared as:

```
void tc_enable_interrupt(Tc *p_tc, uint32_t ul_channel,
                                                uint32_t ul_sources);
```

The function arguments are:

`p_tc`	...	TC module registers base address (e.g. `TC0` for the first TC module)
`ul_channel`	...	channel index
`ul_sources`	...	bitmask of interrupt sources (e.g. `TC_IER_CPCS` for interrupt request on RCx compare event)

To enable an interrupt request on RC0 compare event, the `tc_enable_interrupt()` function with appropriate argument values must be called:

```
tc_enable_interrupt(TC0, 0, TC_IER_CPCS);
```

To avoid executing pending interrupt requests issued before the TC interrupt configuration, perform the configuration before starting the counter with the `tc_start()` function.

The ISRs

The NVIC receives the interrupt requests from the UART and TC0 devices. On each request, the NVIC starts the corresponding ISR code located at the address defined in the vector table defined in the `sam/utils/cmsis/sam3x/source/templates/gcc/startup_sam3x.c` file of the ASF. As defined in the vector table, the UART ISR is the `UART_Handler()` function, and the TC0 ISR is the `TC0_Handler()` function. Both are declared as weak symbols (see Exercise 4) and can be overridden. New definitions are required in the user code:

```
void UART_Handler(void) {
      ...
}
void TC0_Handler(void) {
      ...
}
```

The TC0 interrupt must be acknowledged in the `TC0_Handler()` function. The event causing the interrupt request has to be cleared from the `TC_SR0` status register. Otherwise the NVIC receives another interrupt request immediately after the `TC0_Handler()` function finishes. Since the status register is cleared on read, a dummy call of the `tc_get_status()` function does the task.

The `UART_Handler()` function is called every time the UART requests an interrupt, i.e., on each received character. The received number is written into

a circular buffer for TC0 ISR to pick it up. If the buffer is full, the UART ISR waits. Sooner or later the TC0 ISR will read from the buffer and make space for the number just received. The pseudo code of the `UART_Handler()` function is as follows:

```
read a number from UART
echo it back (optional)
while buffer is full
    do nothing
write the number into the buffer
```

The TC0 ISR is executed on every RC0 compare event, i.e., every 100ms. It reads the number from the buffer and starts to generate the pulses. To keep track of the 100ms slices, a zero initialized global variable `slice` is required. The pseudo code of the `TC0_Handler()` function is as follows:

```
clear RC0 event status
if slice is zero
    if buffer is empty
        return
    read number from buffer
    slice = number + number of slices in 1s pause
    set 39% duty cycle (i.e., set RB0 to 61% of timer period)
if only pause left in slice
    set 0% duty cycle (i.e, set RB0 beyond 100% of timer period)
decrement slice
```

The UART and TC0 ISRs operate as two independent tasks executed on request. The UART ISR is executed on data received event, the TC0 ISR is executed on every RC0 compare event, i.e., every 100ms. The communication between the two ISRs is one-way, form the UART to the TC0 handler. The circular FIFO buffer is used to transfer data from one ISR to the other.

Both ISRs have the highest priority by default, one cannot interrupt or preempt the other. An interrupt request remains pending until the ongoing ISR of the same or higher priority finishes. In case the buffer is full, the UART ISR waits for the TC0 ISR to read from the buffer. Therefore, the UART ISR must have a lower priority than the TC0 ISR, otherwise the latest cannot interrupt the first, and a deadlock[16] could occur. The priority of an interrupt request issued by a specified device is set with the CMSIS `NVIC_SetPriority()` function declared as:

```
void NVIC_SetPriority(IRQn_Type IRQn, uint32_t priority);
```

The function arguments are:

`IRQn`	...	peripheral device identifier (e.g., `ID_UART` for UART peripheral device)
`priority`	...	priority number from 0 (the highest) to 15 (the lowest priority)

To set the UART interrupt request priority to the lowest, the `NVIC_SetPriority()` function with appropriate argument values must be called:

```
NVIC_SetPriority(ID_UART, 15);
```

[16]A deadlock occurs when two or more processes indefinitely wait for one another to perform a required action. In our case, the UART ISR is waiting the TC0 ISR to read from buffer, while TC0 ISR is waiting for UART ISR to finish.

Since the TC0 ISR now solely has the highest priority, it will never wait for the UART ISR to finish. The TC0 ISR is guaranteed to execute in 100ms time intervals, thus assuring accurate pulse trains and pauses. With one ISR enabled to preempt the other, a race condition[17] on a shared resource (i.e., circular FIFO buffer) becomes possible. Therefore, the critical code, i.e., writing into the FIFO buffer in UART ISR, must be atomic[18]. The TC0 ISR must be temporary disabled.

To disable an interrupt request issued by a specified device, the CMSIS `NVIC_DisableIRQ()` function can be used. Its declaration is:

```
void NVIC_DisableIRQ(IRQn_Type IRQn);
```

The function arguments is:

`IRQn` ... peripheral device identifier
(e.g., `ID_TC0` for TC channel 0)

To disable the TC0 interrupt request at the beginning of the critical code, the `NVIC_DisableIRQ()` function with the TC channel 0 identifier must be called:

```
NVIC_DisableIRQ(ID_TC0);
```

At the end of the critical code, the TC0 interrupt request is reenabled by the `NVIC_EnableIRQ()` function:

```
NVIC_EnableIRQ(ID_TC0);
```

To ensure atomic writing into the circular buffer, the UART ISR pseudo code from page 44 should be supplemented by:

```
read a number from UART
echo it back (optional)
while buffer is full
        do nothing
disable TC0 ISR
write the number into the buffer
enable TC0 ISR
```

[17] A race condition occurs when the outcome depends on a sequence or timing of outside events. E.g., A process is preempted by another one in the middle of using a resource. If the interrupting process uses the same resource, the resource can become corrupted.

[18] Atomic code is a part of code that is guaranteed to be executed without interruption.

Exercise 7

LCD driver

Write a driver for the HD44780U dot matrix LCD[1] controller [14]. The driver should consist of a display memory and refresh function. The content to be displayed should be written into the display memory. The refresh function call should synchronize the LCD RAM with the display memory. The function should also receive one argument representing an optional LCD command. If the command is specified (the argument is not zero), the refresh function should, besides synchronizing, execute it. To test the driver, write a test application writing your name and surname in the first row, and the time elapsed from the application start in the second row of the LCD. The information should repeatedly travel from the right to left side of the LCD with the speed of three characters per second. The external boards with mounted LCDs are available at 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, and to the LCD on the external board, as shown in Fig. 7.1.

LCD and driver display memory initialization

The LCD used in this exercise is driven by the long-established HD44780U display controller. The initialization steps of the controller are described in [14], and are already programed in the `vLCDInit()` function. The declaration of the function is:

```
void vLCDInit(void);
```

Therefore, the LCD is initialized in a single line:

```
vLCDInit();
```

The declaration of the `vLCDInit()` function resides in the `lcd.h` header file, which has to be included.

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

The LCD used in this exercise has 2×40 bytes of display RAM. It contains the characters in two forty character long lines. Since the display driver synchronizes

[1] LCD ... Liquid-Crystal Display

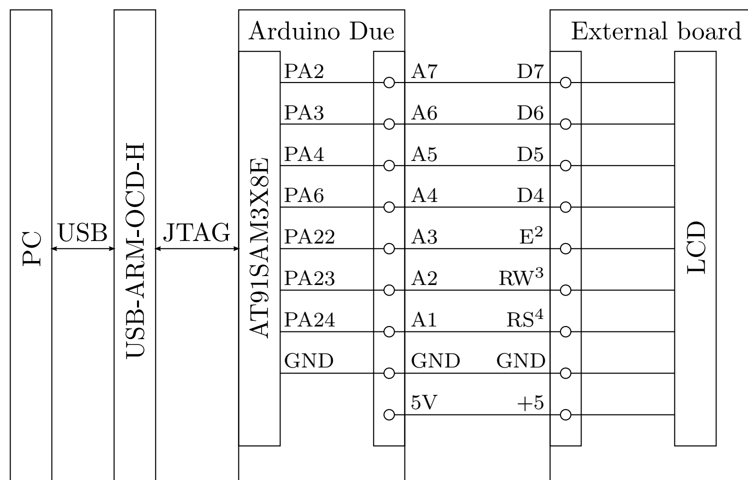


Figure 7.1: PC / Olimex ARM-USB-OCD-H / Arduino Due / External board connection for Exercise 7

the LCD RAM with its display memory, the display memory of the same size is required. Most conveniently the display memory is represented by two global character arrays:

```
uint8_t pucLine1[40];
uint8_t pucLine2[40];
```

To initialize the characters to spaces, i.e., blank LCD, a `for` loop can be used.

LCD commands

The commands can be sent to the HD44780U display controller with the `vLCDWrite()` function. The declaration of the function is:

```
void vLCDWrite(uint8_t ucRS, uint8_t ucByte);
```

The function arguments are:

`ucRS`	...	register select, i.e., command/data, flag (e.g., `COMM` for command)
`ucByte`	...	data or command (e.g., `DDRAM|`*address* to set the *address* in LCD RAM)

Available commands can be found in the `src/lcd.h` file. For the purpose of this exercise, a few useful examples of `vLCDWrite()` function calls follow:

Set LCD RAM address to the third position in the first line[5]:

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

Set LCD RAM address to the third position in the second line[6]:

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

[2] E ... Enable
[3] RW ... Read Write
[4] RS ... Register Select
[5] The first line has addresses from `0x00` to `0x27`.
[6] The second line has addresses from `0x40` to `0x67`.

Write character A at the current LCD RAM address:

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

Shift display window one character left[7]:

```
vLCDWrite(COMM, SHIFT | DISPLAY | LEFT);
```

Display driver

The display driver is consisted of two parts: the display memory and the refresh function. As already mentioned, the display memory is represented by two global character arrays. The refresh function synchronizes LCD RAM with the display memory. To achieve that, the refresh function could just copy the entire display memory into the LCD RAM. However, copying the entire display memory is in most cases not necessary. Cases, in which all the characters have been changed since last refresh function call, are rare. Therefore, the refresh function could read the LCD RAM, compare it to the display memory, and write back only the characters that have been changed. Such a solution seems advanced, but it is even slower than copying since reading LCD RAM takes the same amount of time as writing to it.

Instead of reading from the LCD RAM, a copy of it can be maintained in two additional global arrays. Thus, the refresh function can quickly find the differences without the LCD RAM reading, and write only the characters that have been changed. Of course, the two additional global arrays have to be updated also. The pseudo code of the refresh function of the display driver is as follows:

```
for each character in display memory
    if the character differs from its counterpart in the additional array
        set appropriate LCD RAM address
        write the character to the current LCD RAM address
        copy the character to the counterpart in the additional array
if command argument is not zero
    execute the command
```

The refresh function also receives one argument, i.e., an arbitrary display command. Besides refreshing, the refresh function should execute the command when one is given. This is implemented in the last `if` statement of the pseudo code.

Test application

To show your name and number of seconds elapsed from the application start on the LCD, the characters have to be copied into the display memory and the refresh function has to be called. Since the number of seconds is constantly changing, the display memory has to be updated every second. The refresh function has to be called after each update. Such a functionality can be achieved by using the TC channel interrupt (see Exercise 6). The display memory update and refresh function call is performed from the TC channel ISR.

So far, a regularly updated text is shown on the LCD at a fixed position. The text can be moved by issuing the shift display window command. Therefore, the TC channel ISR has to call the refresh function with the shift command. To obtain the speed of three characters per second, the TC channel should request an interrupt three times per second. However, the display memory update is needed only on every third request, i.e., once per second. Hence, a zero initialized global

[7]There are 2×40 characters space in LCD RAM, which means two lines with forty characters. However, only two lines with 16 characters are actually displayed. The displayed characters are specified by the position of the display window that can be moved left and right in cyclic manner.

`counter` is required to count the interrupts. The pseudo code of the TC channel ISR is as follows:

increment `counter`
`if` `counter` is greater than two
    reset `counter`
`if` `counter` is zero
    update display memory (increase number of seconds)
call refresh function with shift display window command

See Exercise 6 for appropriate TC channel configuration. Note, that LCD functions `vLCDInit()` and `vLCDWrite()` internally measure time intervals using TC channel 2 configured as a free running counter. Therefore, TC channel 2 must not be used.

Since everything is done in the TC channel ISR, only an endless loop is left in the `main()` function.

Exercise 8

ADC and DACC

Write a program for the Arduino Due board implementing a low frequency signal generator at DACC[1] output pin. The generator should be able to generate rectangular, triangular and sine shaped signals in frequency range from 50Hz to 100Hz. Use button keys on the external board to switch among the signal shapes. Connect analog potentiometer on the external board to the ADC[2] input pin and use it to set the signal frequency. Observe the generated signal with an oscilloscope. Implement the required task with and without using the TC channel interrupt. The external boards 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, and to the button keys and analog potentiometer on the external board, as shown in Fig. 8.1. Initialize the key I/O pins as explained in Exercise 2. Since the keys will be used only to switch among signal shapes, the debouncing filters at their input pins `PC28`, `PC26` and `PC25` are not required.

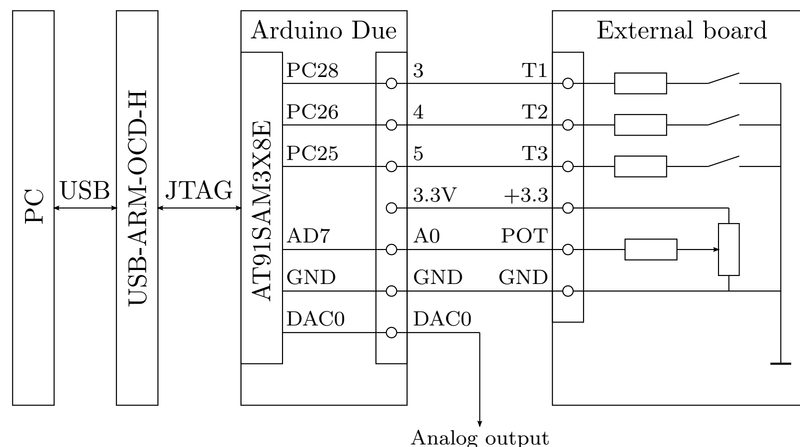


Figure 8.1: PC / Olimex ARM-USB-OCD-H / Arduino Due / External board connection for Exercise 8

[1]DACC ... Digital to Analog Converter Controller
[2]ADC ... Analog to Digital Converter

Initialization of the ADC device

The AT91SAM3X8E µC has one eight channel 12-bit ADC. Only one channel, e.g., channel AD7, will be used in this exercise. The ADC peripheral device has to be clocked first. The `pmc_enable_periph_clk()` function can be used (see Exercise 6). To enable the ADC device, its identifier `ID_ADC` must be passed to the `pmc_enable_periph_clk()` function:

```
pmc_enable_periph_clk(ID_ADC);
```

The configuration of the ADC begins with ADC clock and startup time initialization. The `adc_init()` function can be used. Its declaration is:

```
uint32_t adc_init(Adc *p_adc, uint32_t ul_mck,
                  uint32_t ul_adc_clock, adc_startup_time startup);
```
[3]

The function always returns zero. Its arguments are:

`p_adc`	...	ADC registers base address (e.g., `ADC` for ADC peripheral device)
`ul_mck`	...	MCK frequency f_{MCK} (see Exercise 5) (e.g., `SystemCoreClock` global variable)
`ul_adc_clock`	...	ADC conversion clock frequency f_{ADC}[4]
`startup`	...	mask defining ADC startup time[5] (e.g., `ADC_STARTUP_TIME_3` for 24 f_{ADC} periods)

For $f_{\mathrm{MCK}} = 10.5\mathrm{MHz}$, the ADC clock can be initialized to $f_{\mathrm{ADC}} = \frac{f_{\mathrm{MCK}}}{2} = 5.25\mathrm{MHz}$, and startup time to $t_{\mathrm{START}} = \frac{24}{f_{\mathrm{ADC}}} = 4.6\mu\mathrm{s}$ with the following `adc_init()` function call:

```
adc_init(ADC, SystemCoreClock, SystemCoreClock / 2,
                                                ADC_STARTUP_TIME_3);
```

ADC tracking time, settling time and transfer period [8] have to be configured next. The `adc_configure_timing()` function can be used. Its declaration is:

```
void adc_configure_timing(Adc *p_adc, uint8_t uc_tracking,
               adc_settling_time_t settling, uint8_t uc_transfer);
```
[6]

The function arguments are:

`p_adc`	...	ADC registers base address (e.g., `ADC` for ADC peripheral device)
`tracking`	...	`TRACKTIM` constant defining ADC tracking time
`settling`	...	mask defining ADC settling time[7] (e.g., `ADC_SETTLING_TIME_0` for 3 f_{ADC} periods)
`transfer`	...	`TRANSFER` constant defining ADC transfer period

[3] `Adc` is a structure of 32-bit integers representing the AT91SAM3X8E ADC hardware register addresses. If the pointer to such a structure refers to the top of the ADC device address space, then the structure elements directly represent the registers.

adc_startup_time is an enumerated type with ADC startup time masks used in the `ADC_MR` mode register.

[4] f_{ADC} must be picked inside an interval $1\mathrm{MHz} \leq f_{\mathrm{ADC}} \leq 22\mathrm{MHz}$, and the expression $\frac{f_{\mathrm{MCK}}}{2f_{\mathrm{ADC}}} - 1$ must yield an integer value from 0 to 255 [8].

[5] t_{START} must be picked inside an interval $4\mu\mathrm{s} \leq t_{\mathrm{START}} \leq 12\mu\mathrm{s}$ [8].

[6] adc_settling_time_t is an enumerated type with ADC settling time masks used in the `ADC_MR` mode register.

[7] Settling time must be $t_{\mathrm{S}} \geq 200\mathrm{ns}$ [8].

For further details about tracking time, settling time and transfer period, see [8][8]. At $f_{\mathrm{ADC}} = 5.25\mathrm{MHz}$, the ADC timings can be appropriately set with the following `adc_configure_timing()` function call:

```
adc_configure_timing(ADC, 0, ADC_SETTLING_TIME_0, 1);
```

The ADC device in the AT91SAM3X8E µC has eight channels. To enable a channel, the `adc_enable_channel()` function can be used. Its declaration is:

```
void adc_enable_channel(Adc *p_adc, adc_channel_num_t adc_ch);
```
[9]

The function arguments are:

`p_adc`	...	ADC registers base address (e.g., `ADC` for ADC peripheral device)
`adc_ch`	...	ADC channel number (e.g., `ADC_CHANNEL_7` for AD7)

The ADC channel AD7 is enabled with the following `adc_enable_channel()` function call:

```
adc_enable_channel(ADC, ADC_CHANNEL_7);
```

Finally, the conversion trigger has to be specified and free run mode has to be switched on or off. The `adc_configure_trigger()` function can be used. Its declaration is:

```
void adc_configure_trigger(Adc *p_adc, adc_trigger_t trigger,
                                                    uint8_t uc_freerun);
```
[10]

The function arguments are:

`p_adc`	...	ADC registers base address (e.g., `ADC` for ADC peripheral device)
`trigger`	...	conversion trigger (e.g., `ADC_TRIG_SW` to disable hardware triggers)
`uc_freerun`	...	free running mode flag (e.g., set the flag to 1 to never wait for any trigger)

When free running mode is specified, the conversion trigger selection is void. To set the free running mode, the following `adc_configure_trigger()` function call is required:

```
adc_configure_trigger(ADC, ADC_TRIG_SW, 1);
```

The declarations of the ADC initialization functions `adc_init()`, `adc_configure_timing()`, `adc_enable_channel()`, and `adc_configure_trigger()` reside in the `adc.h` header file, which has to be included.

```
#include <adc.h>
```

[8] The ADC timings section is confusing in [8]. Set `TRACKTIM` = 0 for $t_{\mathrm{TRACK}} = \frac{15}{f_{\mathrm{ADC}}}$, or `TRACKTIM` = 15 for $t_{\mathrm{TRACK}} = \frac{16}{f_{\mathrm{ADC}}}$, and set `TRANSFER` = 1 for $t_{\mathrm{TRANSFER}} = \frac{5}{f_{\mathrm{ADC}}}$ (recommended).

[9] adc_channel_num_t is an enumerated type with ADC channel numbers.

[10] adc_trigger_t is an enumerated type with ADC channel numbers.

Initialization of the DACC device

A similar procedure as with ADC is required for DACC initialization. The AT91SAM3X8E µC has one 12-bit DACC with two independent analog output lines. One output line, e.g., DAC0, will be used in this exercise. Again, the DACC peripheral device has to be clocked first. The `pmc_enable_periph_clk()` function does the trick. Of course, the DACC device identifier `ID_DACC` has to be passed as the argument:

```
pmc_enable_periph_clk(ID_DACC);
```

DACC needs to be reset before further configuration. The `dacc_reset()` function can be used. Its declaration is:

```
void dacc_reset(Dacc *p_dacc);
```
[11]

The function argument is:

`p_dacc`	...	DACC registers base address (e.g., `DACC` for DACC peripheral device)

The following `dacc_reset()` function call resets the DACC peripheral device:

```
dacc_reset(DACC);
```

DACC refresh period, speed mode and startup time [8] have to be configured next. The `dacc_set_timing()` function can be used. Its declaration is:

```
uint32_t dacc_set_timing(Dacc *p_dacc, uint32_t ul_refresh,
                              uint32_t ul_maxs, uint32_t ul_startup);
```

The function always returns zero. Its arguments are:

`p_dacc`	...	DACC registers base address (e.g., `DACC` for DACC peripheral device)
`ul_refresh`	...	`REFRESH` constant defining DACC refresh period[12] (e.g., set `REFRESH` = 0 for no refreshing)
`ul_maxs`	...	maximum speed mode switch[13] (e.g., set the switch to 0 to disable maximum speed mode)
`ul_startup`	...	constant defining DACC startup time[14] (e.g., set the constant to 3 for 24 f_{DACC} periods)

The DACC clock frequency f_{DACC} is halved MCK, $f_{\mathrm{DACC}} = \frac{f_{\mathrm{MCK}}}{2}$. For further details about refresh period, speed mode and startup time, see [8]. To disable refreshing, switch maximum speed mode off, and to set the startup time to 4.6µs at $f_{\mathrm{DACC}} = 5.25$MHz, the `dacc_set_timing()` function with the following arguments has to be called:

```
dacc_set_timing(DACC, 0, 0, 3);
```

[11] `Dacc` is a structure of 32-bit integers representing the AT91SAM3X8E DACC hardware register addresses. If the pointer to such a structure refers to the top of the DACC device address space, then the structure elements directly represent the registers.

[12] Automatic output voltage refresh period to avoid voltage decreasing over time due to leakage. It must be below 20µs and is calculated as: $t_{\mathrm{REFRESH}} = \frac{1024\ \mathtt{REFRESH}}{f_{\mathrm{DACC}}} \leq 20$µs, where $f_{\mathrm{DACC}} = \frac{f_{\mathrm{MCK}}}{2}$ [8]. Refresh has no effect in free running mode, and can be disabled by setting `REFRESH` to zero. The free running mode is the default DACC trigger mode.

[13] In maximum speed mode, the DACC does not wait for the end of conversion cycle signal before starting the next conversion [8].

[14] t_{START} must be picked inside an interval 2.5µs $\leq t_{\mathrm{START}} \leq$ 5µs [8].

The DACC device in the AT91SAM3X8E µC has two analog output lines. To enable a line, the `dacc_enable_channel()` function can be used. Its declaration is:

```
uint32_t dacc_enable_channel(Dacc *p_dacc, uint32_t ul_channel));
```

The function returns zero on success. Its arguments are:

`p_dacc`	...	DACC registers base address (e.g., `DACC` for DACC peripheral device)
`ul_channel`	...	DACC line number (e.g., zero for DAC0)

The DACC line DAC0 is enabled with the following `dacc_enable_channel()` function call:

```
dacc_enable_channel(DACC, 0);
```

Finally, the analog current settings influencing the DACC current consumption has to be specified. The `dacc_set_analog_control()` function can be used. Its declaration is:

```
uint32_t dacc_set_analog_control(Dacc *p_dacc,
                                          uint32_t ul_analog_control);
```

The function always returns zero. Its arguments are:

`p_dacc`	...	DACC registers base address (e.g., `DACC` for DACC peripheral device)
`ul_analog_control`	...	analog control settings [8]

Appropriate analog current settings can be achieved with the following `dacc_set_analog_control()` function call:

```
dacc_set_analog_control(DACC, DACC_ACR_IBCTLCH0(2) |
                                          DACC_ACR_IBCTLDACCORE(1));
```
[15]

The declarations of the DACC initialization functions `dacc_reset()`, `dacc_set_timing()`, `dacc_enable_channel()`, and `dacc_set_analog_control()` reside in the `dacc.h` header file, which has to be included.

```
#include <dacc.h>
```

ADC and DACC usage

The ADC converted value can be obtained by the `adc_get_channel_value()` function. Its declaration is:

```
uint32_t adc_get_channel_value(Adc *p_adc,
                                          adc_channel_num_t adc_ch);
```

The function returns current converted value at the specified channel. Its argu-

[15]For additional information on analog current settings and DACC current consumption, see [8].

ments are:

`p_adc`	...	ADC registers base address (e.g., `ADC` for ADC peripheral device)
`adc_ch`	...	ADC channel number (e.g., `ADC_CHANNEL_7` for AD7)

To read the ADC converted value of the channel AD7, the following `adc_enable_channel()` function has to be called:

```
value = adc_get_channel_value(ADC, ADC_CHANNEL_7);
```

In the opposite direction, the value to be converted by the DACC can be specified by the `dacc_write_conversion_data()` function. Its declaration is:

```
void dacc_write_conversion_data(Dacc *p_dacc, uint32_t ul_data);
```

The function arguments are:

`p_dacc`	...	DACC registers base address (e.g., `DACC` for DACC peripheral device)
`ul_data`	...	value to be converted

To write a `value` into the DACC conversion register, the following `dacc_write_conversion_data()` function has to be called:

```
dacc_write_conversion_data(DACC, value);
```

Sample tables

To generate a signal with DACC, 12-bit sample values over one period of the signal are needed. The generated signal is of course a stepwise approximation. The sample values can be calculated once and saved into a sample table. Since three waveforms, rectangular, triangular, and sine, are to be generated in this exercise, three sample tables are required. If there are N samples per period, the ith sample can be calculated by (8.1). The calculation for all three required signal shapes is given.

$$\begin{aligned} rect_i &= \begin{cases} 0 & i < \mathrm{N}/2 \\ \max & i \geq \mathrm{N}/2 \end{cases} \\ triang_i &= i\,\frac{\max}{\mathrm{N}-1} \\ sine_i &= \frac{\max}{2}\left(\sin\frac{2\pi i}{\mathrm{N}} + 1\right) \end{aligned} \tag{8.1}$$

Constant max is the maximum 12-bit sample value, $\max = 2^{12} - 1 = 4095$, and index $i \in \{0, 1, 2, \ldots, \mathrm{N}-1\}$. Do not exaggerate in picking the number of samples N. Larger N causes larger sample tables and shorter intervals between digital to analog conversions. $\mathrm{N} \leq 100$ will do. To calculate the required samples, a for loop can be used.

The program

In version without interrupts, the TC channel has to be initialized as a free running counter (see Exercise 6). To read the current TC channel counter value, the `tc_read_cv()` function can be used. Its declaration is:

```
uint32_t tc_read_cv(Tc *p_tc, uint32_t ul_channel);
```

The function returns current specified TC channel counter value. The arguments of the function are:

`p_tc`	...	TC module registers base address (e.g. `TC0` for the first TC module)
`ul_channel`	...	channel index

To obtain the current counter value of channel 0 of the TC0 module, issue the following `tc_read_cv()` function call :

```
value = tc_read_cv(TC0, 0);
```

With ADC, DACC and TC channel initialized, and sample tables ready, the main algorithm can be coded. If there are N samples, and the frequency of the generated signal is f_{SIGNAL}, a new sample has to be converted on every $\frac{1}{N f_{\text{SIGNAL}}}$ seconds. If the TC channel counts with $\frac{f_{\text{MCK}}}{2}$, the interval elapses in $\frac{f_{\text{MCK}}}{2N f_{\text{SIGNAL}}}$ pulses, which is the basis to obtain equidistant time points. The algorithm runs in an endless loop. When a time point is detected, a new sample for conversion is provided to the DACC, and the ADC and key buttons are checked for any f_{SIGNAL} or signal shape modification. The pseudo code of the algorithm implementing this exercise without interrupts is as follows:

```
set point to zero
set i to zero
set table pointer to one of the sample tables
while forever
    obtain current TC channel counter value
    if point - current value is less than zero
        write i-th sample from the table into DACC conversion register
        increment i in cyclic manner regarding N
        check keys and set the table pointer appropriately
        read ADC and obtain new f_SIGNAL
        increase point for f_MCK/(2N f_SIGNAL)
```

Using TC channel interrupt

Instead of detecting the equidistant time points in an endless loop, the TC channel interrupt can be used. See Exercise 6 to configure the TC channel to request an interrupt on every $\frac{f_{\text{MCK}}}{2N f_{\text{SIGNAL}}}$ pulses. Use RCx compare event. The pseudo code of the TC channel ISR is as follows:

```
write i-th sample from the table into DACC conversion register
increment i in cyclic manner regarding N
check keys and set the table pointer appropriately
read ADC and obtain new f_SIGNAL
set RCx to f_MCK/(2N f_SIGNAL)
clear RCx event status
```

A zero initialized global index `i` is required to count samples. The `table` pointer selecting current sample table also has to be global. Since everything is done in the TC channel ISR, only an endless loop is left in the `main()` function.

Exercise 9

Ramp application

Write a software for the Arduino Due board that drives a ramp model. 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 as shown in Fig. 4.1. Initialize the UART peripheral device and configure the stdio in serial mode as explained in Exercise 4.

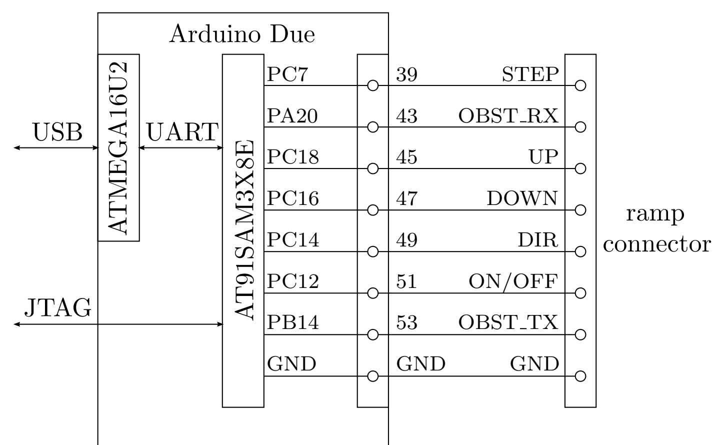


Figure 9.1: Ramp model to Arduino Due board connection

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. The software required in this exercise has to appropriately drive the pins to achieve the desired behavior.

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

Initialization of the PWM[1] device

A PWM peripheral device in the AT91SAM3X8E µC has eight channels, each generating an independent waveform. The OBST_TX ramp input needs a 38kHz 50% duty cycle signal, which can be generated by one of the PWM channels.

A PWM peripheral device has to be clocked first. To enable the peripheral clock of the PWM device, the device's identifier `ID_PWM` must be passed to the `pmc_enable_periph_clk()` function (see Exercise 6):

```
pmc_enable_periph_clk(ID_PWM);
```

The PWM peripheral device provides thirteen clock sources that can be used by any of its eight channels. Besides the eleven f_{MCK} based clocks (providing frequencies $\frac{f_{\text{MCK}}}{2^i}$, where $i = \{0, 1, \ldots 10\}$), two additional dividers are available (providing special frequencies $\frac{f_{\text{MCK}}}{2^j k}$, where $j = \{0, 1, \ldots 10\}$, and $k = \{1, 2, \ldots 255\}$). The frequencies of the two additional dividers has to be defined first. The `pwm_init()` can be used. Its declaration is:

```
uint32_t pwm_init(Pwm *p_pwm, pwm_clock_t *clock_config);
```
[2]

The function returns zero on success, and non-zero value in case the j and k parameters cannot be determined. The arguments are:

`p_pwm`	...	PWM device registers base address (e.g. `PWM` for PWM peripheral device)
`clock_config`	...	pointer to a structure with frequencies

To turn the two additional dividers off, the `pwm_init()` function with appropriately set clock stucture must be called:

[1] PWM ... Pulse Width Modulation

[2] `Pwm` is a structure of 32-bit integers representing the AT91SAM3X8E PWM device hardware register addresses. If the pointer to such a structure refers to the top of the PWM device address space, the structure elements directly represent the registers.

The `pwm_clock_t` type is a structure containing the two special frequencies and the f_{MCK} value:

```
typedef struct {
      uint32_t ul_clka; /* clock A frequency (set to 0 to turn it off) */
      uint32_t ul_clkb; /* clock B frequency (set to 0 to turn it off) */
      uint32_t ul_mck;  /* MCK                                         */
} pwm_clock_t;
```

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

Note, that the `SystemCoreClock` global variable is set in the `sysclk_reinit()` function call (see Exercise 5), and therefore should not be used before the call.

With a PWM peripheral device clocked and initialized, its eight independent channels become available. To initialize or reconfigure a PWM channel, it has to be disabled. The PWM channel is disabled by the `pwm_channel_disable()` function declared as:

```
void pwm_channel_disable(Pwm *p_pwm, uint32_t ul_channel);
```

The function arguments are:

`p_pwm`	...	PWM device registers base address (e.g. `PWM` for PWM peripheral device)
`ul_channel`	...	channel number (e.g. `PWM_CHANNEL_2` for PWM channel two)

To disable the PWM channel two, the following `pwm_channel_disable()` function has to be called:

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

When disabled, the PWM channel can be initialized. The initialization of a PWM channel is performed by the `pwm_channel_init()` function. It is declared as:

```
uint32_t pwm_channel_init(Pwm *p_pwm, pwm_channel_t *p_channel);
```
[3]

The function returns zero on success. The arguments are:

`p_pwm`	...	PWM device registers base address (e.g. `PWM` for PWM peripheral device)
`p_channel`	...	pointer to a structure with channel configuration

To generate a 38kHz 50% duty cycle signal, the f_{MCK} can be used as a PWM channel clock source (see Fig. 9.2). To obtain the 38kHz signal at the PWM channel two, the `pwm_channel_init()` function with the following arguments has to be called:

[3]The `pwm_channel_t` type is a structure containing channel configuration:

```
typedef struct {
      uint32_t channel;      /* channel number     */
      uint32_t ul_prescaler; /* clock selector     */
            ...
      uint32_t ul_duty;      /* duty cycle value   */
      uint32_t ul_period;    /* period cycle value */
            ...
} pwm_channel_t;
```

```
pwm_channel_t pwm_channel = {0};
  ...
pwm_channel.channel      = PWM_CHANNEL_2;
pwm_channel.ul_prescaler = PWM_CMR_CPRE_MCK;
pwm_channel.ul_duty      = SystemCoreClock / (38000 * 2);
pwm_channel.ul_period    = SystemCoreClock /  38000;
pwm_channel_init(PWM, &pwm_channel);
```
[4]

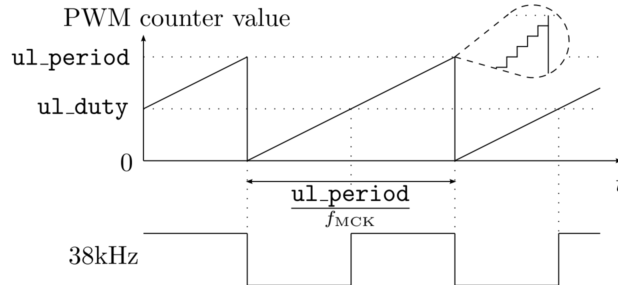


Figure 9.2: Obtaining 38kHz 50% duty cycle signal with PWM peripheral device (PWM clock source = MCK, ul_period $= \frac{f_{\mathrm{MCK}}}{38\mathrm{kHz}}$, ul_duty $= \frac{\texttt{ul_period}}{2}$)

Note, that the SystemCoreClock global variable is set in the sysclk_reinit() function call (see Exercise 5), and therefore should not be used before the call.

After initialization, the PWM channel can be enabled. To enable the channel, the pwm_channel_enable() function can be used. Its declaration is:

```
void pwm_channel_enable(Pwm *p_pwm, uint32_t ul_channel);
```

The function arguments are:

p_pwm	...	PWM device registers base address (e.g. PWM for the PWM peripheral device)
ul_channel	...	channel number (e.g. PWM_CHANNEL_2 for PWM channel two)

To enable the PWM channel two, the following pwm_channel_enable() function has to be called:

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

The declarations of the pwm_init(), pwm_channel_disable(), pwm_channel_init() and pwm_channel_enable() functions reside in the pwm.h header file which has to be included.

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

The 38kHz 50% duty cycle signal is required at OBST_TX ramp input, or, according to Fig. 9.1, at pin PB14. Most PIO pins can be hardwired to one of two predefined devices available for the pin (see Exercise 6). Conveniently, the PWM channel two is available to be hardwired to pin PB14 of the PIOB controller as a peripheral device B. To hardwire the PB14 to its B device, i.e., the PWM channel two, the following pio_set_peripheral() function call (see Exercise 6) can be used:

[4]Mask PWM_CMR_CPRE_MCK sets PWM channel clock frequency to f_{MCK}.

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

GPIO pins

The ramp controlling pins beside the PB14 have to be configured as GPIO pins managed by the PIO controllers. To initialize the pins from Fig. 9.1, a few `pio_configure()` function calls (see Exercise 2) are required:

```
pio_configure(PIOA, PIO_INPUT, PIO_PA20, 0);
pio_configure(PIOC, PIO_INPUT, PIO_PC16 | PIO_PC18, 0);
pio_configure(PIOC, PIO_OUTPUT_1, PIO_PC7 | PIO_PC12 | PIO_PC14,
                                                                   0);[5]
```

No pull-up resistors or debounce filters are needed at input pins. Use the `pio_get()` function to read an input pin, and the `pio_set()`, `pio_clear()` and `pio_toggle_pin()` functions to drive an output pin (see Exercises 2 and 3).

TC channel

The TC channel interrupt will be used to generate the 100Hz[6] signal at the STEP input when the ramp is moving. To generate a 100Hz signal at PC7 pin, the pin has to be toggled on every five milliseconds. An interrupt on RCx compare event is the most suitable for that purpose. See Exercise 6 for the TC channel configuration. Conveniently, the TC channel interrupt can also be used for counting the 'time' in the delay when the ramp is fully open. The pseudo code of the TC channel ISR is as follows:

```
if moving
        toggle STEP ramp pin
increment time
```

The ISR communicates with the rest of the code through two global variables, `moving` and `time`. The `moving` variable contains the information wheather the ramp is currently moving or not. The `time` variable counts the TC channel interrupts, i.e., five millisecond intervals, and can therefore be used for time measuring.

The program

The ramp input signals OBST_TX and STEP are handled by the PWM device and the TC channel. The remaining ramp pins will be driven in the main endless loop. First, the password is obtained from the UART device. The password string can be assembled character by character using the `getchar()` function. The `strcmp()` string compare function declared in the `string.h` header file comes in handy for password verification. When the correct password is typed in, the ramp open/close procedure is started. Since the obstacle can cause the procedure to be repeated for the unknown number of times, it is wrapped in an internal endless loop. The pseudo code of the described algorithm is as follows:

[5] Type `PIO_OUTPUT_1` defines an output pin with initial value set to one.
[6] According to table 9.1, any frequency below 300Hz is suitable.

```
while forever
    obtain password from UART
    if the password is correct
        while forever
            set DIR pin to up
            set moving
            while ramp is not fully open (use UP pin)
                do nothing
            clear moving
            wait for delay time (use time)
            set DIR pin to down
            set moving
            while ramp is not fully closed (use DOWN pin) and
                        there is no obstacle (use OBST_RX pin)
                do nothing
            clear moving
            if ramp is fully closed (use DOWN pin)
                break
```

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] *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

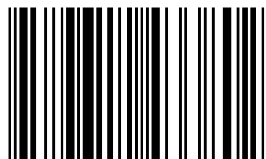
EMBEDDED SYSTEM: LABORATORY EXERCISES

The script contains instructions and detailed explanation of laboratory exercises covered in the Embedded Systems course that is held in the fth semester of the 1st Cycle Professional Study Programme in Applied Electrical Engineering, study programme option Electronics, at the Faculty of electrical engineering of the University of Ljubljana, Slovenia. The laboratory exercises focus on usage of modern 32-bit microcontroller features such as: General Purpose In-put/Output pins (GPIO), WatchDog Timer (WDT), Universal Asynchronous Receiver/ Transmitter (UART), Timers, Analog to Digital and Digital to Analog Conversion (ADC and DAC), etc., in embedded applications.

KEYWORDS

Embedded programming, C language, Microcontroller peripherals, ARM Cortex-M3, AT91SAM3X8E

ISBN 978-961-243-380-2



ZALOŽBA FAKULTETE ZA ELEKTROTEHNIKO