

GE863-PRO³ Linux Software User Guide

1v0300781 Rev.5 – 2010-01-25



DISCLAIMER

The information contained in this document is the proprietary information of Telit Communications S.p.A. and its affiliates (“TELIT”).

The contents are confidential and any disclosure to persons other than the officers, employees, agents or subcontractors of the owner or licensee of this document, without the prior written consent of Telit, is strictly prohibited.

Telit makes every effort to ensure the quality of the information it makes available. Notwithstanding the foregoing, Telit does not make any warranty as to the information contained herein, and does not accept any liability for any injury, loss or damage of any kind incurred by use of or reliance upon the information.

Telit disclaims any and all responsibility for the application of the devices characterized in this document, and notes that the application of the device must comply with the safety standards of the applicable country, and where applicable, with the relevant wiring rules.

Telit reserves the right to make modifications, additions and deletions to this document due to typographical errors, inaccurate information, or improvements to programs and/or equipment at any time and without notice.

Such changes will, nevertheless be incorporated into new editions of this document.

Copyright: Transmittal, reproduction, dissemination and/or editing of this document as well as utilization of its contents and communication thereof to others without express authorization are prohibited. Offenders will be held liable for payment of damages. All rights are reserved.

Copyright © Telit Communications S.p.A. 2010.



Applicable Products



Linux SW Version	Recommended U-Boot version
06.0006	21.00.0000
06.1006	211.00.0000



Contents

Contents	4
1. Introduction	8
1.1. Scope	8
1.2. Audience	8
1.3. Contact Information, Support	8
1.4. Open Source Licenses	8
1.5. Product Overview	9
1.6. Document Organization	9
1.7. Text Conventions	10
1.8. Related Documents	10
1.9. Document History	11
2. GE863-PRO³ Architecture	12
2.1. Hardware	12
2.2. Software	15
2.2.1. Telit Bootloader	16
2.2.2. Telit customized U-boot	16
2.2.3. Linux kernel	17
2.2.4. Filesystem	21
3. System Startup	25
3.1. Startup process	25
3.2. The Linux shell	26
3.3. Loading a module	27
3.4. Auto-Setup at system startup	28
3.5. Time-based scheduling service	29
3.6. Downloading a file into GE863-PRO ³	30
3.6.1. Downloading a file using the Ethernet connection	30
3.6.2. Downloading a file using an USB mass storage device	32
3.6.3. Downloading/Uploading a file using a serial port	33
3.7. Version statistics	34
4. Device Drivers	35
4.1. Serial port	35



4.1.1.	open().....	36
4.1.2.	read().....	37
4.1.3.	write().....	38
4.1.4.	close().....	38
4.1.5.	termios interface.....	39
4.2.	I²C.....	51
4.2.1.	Loading i2c modules.....	52
4.2.2.	open().....	53
4.2.3.	ioctl().....	54
4.2.4.	read().....	61
4.2.5.	write().....	62
4.2.6.	close().....	62
4.2.7.	A Test Program.....	63
4.3.	SPI.....	65
4.3.1.	Loading the SPI module.....	65
4.3.2.	open().....	66
4.3.3.	ioctl().....	66
4.3.4.	read().....	69
4.3.5.	write().....	70
4.3.6.	close().....	70
4.3.7.	A Test Program.....	71
4.4.	GPIO.....	73
4.4.1.	Loading the GPIO module.....	74
4.4.2.	open().....	74
4.4.3.	read().....	74
4.4.4.	write().....	75
4.4.5.	close().....	76
4.5.	Ge863pro3_GPIO.....	77
4.5.1.	Interrupt description.....	77
4.5.2.	Loading the GPIO module.....	77
4.5.3.	open().....	78
4.5.4.	ioctl().....	79
4.5.5.	read().....	81
4.5.6.	write().....	82
4.5.7.	Interrupt routine customization.....	83
4.5.8.	close().....	84
4.5.9.	A Test Program.....	84
4.6.	ADC.....	86
4.6.1.	Loading the ADC Module.....	86
4.6.2.	open().....	86
4.6.3.	ioctl().....	87
4.6.4.	read().....	94
4.6.5.	close().....	95
4.7.	SSC.....	96
4.7.1.	Loading the SSC module.....	96



4.7.2.	open()	97
4.7.3.	ioctl()	97
4.7.4.	read()	100
4.7.5.	write()	101
4.7.6.	close()	102
4.7.7.	A Test Program	102
4.8.	Watchdog	104
4.8.1.	open()	104
4.8.2.	ioctl()	105
4.8.3.	close()	107
4.9.	Power Management	108
4.9.1.	open()	109
4.9.2.	write()	109
4.9.3.	close()	110
4.10.	Real Time Clock (RTC)	111
4.10.1.	open()	111
4.10.2.	ioctl()	112
4.10.3.	close()	116
4.10.4.	A Test Program	117
4.11.	SD/MMC	120
4.12.	Ethernet	120
4.13.	USB	122
4.13.1.	USB Mass Storage	122
4.13.2.	USB device (Ethernet Gadget)	122
4.14.	Timer Counter	123
4.14.1.	Loading the Timer Counter Module	123
4.14.2.	open()	124
4.14.3.	ioctl()	125
4.14.4.	close()	140
5.	ISO7816 – Smartcard Reader	141
5.1.	ISO-7816 APIs	141
5.1.1.	Defines	142
5.1.2.	Types	142
5.1.3.	Enums	142
5.1.4.	Functions	143
6.	CMUX	150
6.1.	Code example	151
7.	Using external flash memories	153
7.1.	Supported external flash memories	153
7.2.	Erasing the flash memory	154



1. Introduction

1.1. Scope

This user guide serves the following purpose:

- Provides details about the GE863-PRO³ software architecture.
- Describes how software developers can use the functions of Linux device drivers to configure, manage and use GE863-PRO³ hardware resources and system peripherals.

1.2. Audience

This User Guide is intended for software developers who develop applications on the ARM processor of GE863-PRO³ module.

1.3. Contact Information, Support

For general contact, technical support, to report documentation errors and to order manuals, contact Telit's Technical Support Center (TTSC) at:

TS-EMEA@telit.com
TS-NORTHAMERICA@telit.com
TS-LATINAMERICA@telit.com
TS-APAC@telit.com

Alternatively, use:

<http://www.telit.com/en/products/technical-support-center/contact.php>

For detailed information about where you can buy the Telit modules or for recommendations on accessories and components visit:

<http://www.telit.com>

To register for product news and announcements or for product questions contact Telit's Technical Support Center (TTSC).

Our aim is to make this guide as helpful as possible. Keep us informed of your comments and suggestions for improvements.

Telit appreciates feedback from the users of our information.

1.4. Open Source Licenses

Linux system is made up of many Open Source device drivers licensed as follows:



GNU GENERAL PUBLIC LICENSE
Version 2, June 1991

Copyright (C) 1989, 1991 Free Software Foundation, Inc.
675 Mass Ave, Cambridge, MA 02139, USA
Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.

Please refer to the following web page for the full text of the license:

<http://www.gnu.org/licenses/gpl-2.0.html>

1.5. Product Overview

The GE863-PRO³ module contains a fully featured GSM/GPRS communications section, compatible with the other Telit GSM/GPRS modules, but also incorporates a standalone ARM9 CPU and memories, dedicated to user applications.

This eliminates the need for an external host CPU in many applications, bringing true real-time and multi tasking capabilities to an embedded module.

1.6. Document Organization

This manual contains the following chapters:

- “Chapter 1, Introduction” provides a scope for this manual, target audience, technical contact information, and text conventions.
- Chapter 2, GE863-PRO3 Architecture” provides an overview on GE863-PRO3 hardware and software architecture describing the main software components: Telit Bootloader, Telit customized U-Boot, Linux kernel 2.6.24 and Filesystem.
- “Chapter 3, System Startup” describes how to perform the start up process, how to download a file onto GE863-PRO³ and how to load a kernel module.
- “Chapter 4, Device Drivers” details Linux device drivers and shows how software developers can use them to interact with GE863-PRO³ hardware resources and peripherals.
- “Chapter 5, ISO7816 – Smartcard Reader” details the ISO-7816 library and shows how software developers can use it to write applications for managing smartcards.



- “Chapter 6, CMUX” describes GSM 7.10 multiplexing protocol implemented by GE863-PRO³ and its use.
- “Chapter 7, Using external flash memories”, describes how external flash memories can be connected to GE863-PRO³ and managed.

How to Use

If you mainly use this document as reference, the main chapters of interest are Chapter 3, System Startup and Chapter 4, Device Drivers.

If you are new to this product, it is recommended to start by reading through TelitGE863PRO3 EVK User Guide 1V0300776, TelitGE863PRO3_Linux_Development 1V0300780 and this document in their entirety in order to understand and the concepts and specific features provided by the built in software of the GE863-PRO³.

1.7. Text Conventions

This section lists the paragraph and font styles used for the various types of information presented in this user guide.

Format	Content
Courier	Linux shell commands, filesystem paths and example C source code

1.8. Related Documents

The following documents are related to this user guide:

- [1] TelitGE863PRO3 Hardware User Guide 1v0300773a
- [2] TelitGE863PRO3 U-BOOT Software User Guide 1v0300777
- [3] TelitGE863PRO3 EVK User Guide 1V0300776
- [4] TelitGE863PRO3 Linux Development Environment 1V0300780
- [5] TelitGE863PRO3 Linux GSM Library User Guide 1v0300782
- [6] TelitGE863PRO3 Product Description 80285ST10036a
- [7] Telit AT Commands Reference Guide 80000ST10025a

All documentation can be downloaded from Telit’s official web site www.telit.com if not otherwise indicated.



1.9. Document History

Revision	Date	Changes
ISSUE #0	06-04-2008	First Release
ISSUE #1	07-31-2008	Added the following paragraphs: 4.5 Ge863pro3_GPIO 4.6 ADC 4.7 SSC 4.9 Power Management 4.10 RTC 5 ISO7816 – Smartcard Reader 7 Using external flash memories
ISSUE #2	05-21-2009	Updated write() function examples in paragraphs: 4.1.3, 4.2.5, 4.3.5, 4.4.4, 4.5.6, 4.7.5, and 4.9.2. 4.1.5 TERMIOS interface: Added Describes how tty devices can be configured. Modified the following paragraph: 4.6.3 ADC ioctl() 4.6.4 ADC read() (example) Added the following paragraphs: 3.5 Time-based scheduling service 4.1 Serial Port ttyS6 added to list 4.14 Timer Counter 6 CMUX escape command sequence Preface: recommended U-Boot version
ISSUE #3	08-06-2009	Updated: Applicable Products 2.1 Hardware
ISSUE #4	12-04-2009	Updated: 3.6.3 Downloading/Uploading a file using a serial port 3.7 Version statistics 5.1.4.3 iso7816_reset() 6 CMUX -v command line option Applied new template
ISSUE #5	01/25/2010	Updated: 4.11 SD/MMC



2. GE863-PRO³ Architecture

2.1. Hardware

The GE863-PRO³ is an innovation to the quad-band, RoHS compliant GE863 product family which includes a powerful ARM9™ processor core exclusively dedicated to customer applications. The GE863-PRO³ incorporates much of the necessary hardware for communicating microcontroller solutions, including the critical element of memory, significant simplification of the bill of material, vendor management, and logistics effort are achieved.

The Telit GE863-PRO3 comes in three main variants regarding the available flash and RAM memory. All variants share the same high level architecture and most of the concepts apply to all variants. Table below details the available variants and main features.

Variant	4/8	4/64	128/64
Flash Memory	4 MB	4 MB	128 MB
Flash Memory type	NOR	NOR	NAND
Flash Memory access	Serial	Serial	Parallel
SDRAM Memory	8 MB	64 MB	64 MB
U-Boot version	21.00.0000	21.00.0000	211.00.0000
Linux FW Version (1)	21.06.0006	21.06.0006	211.06.1006

(1)Optional



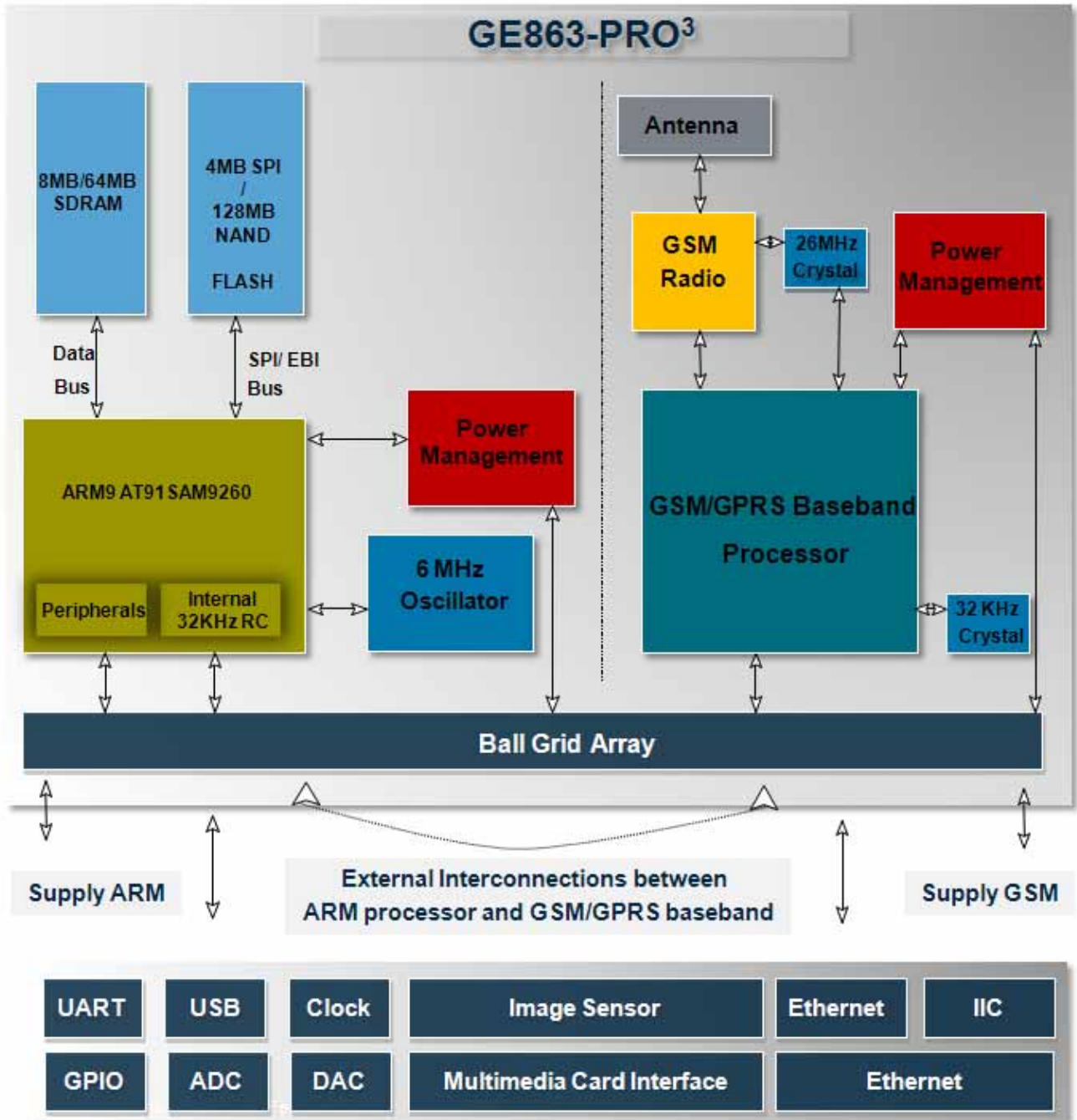
Note – Versions 4/8 and 4/64 are practically equivalent for what is concerned U-Boot, memory management and addressing. For sake of simplicity, we will differentiate between 4/64 and 128/64 versions, as 4/8 version is a subset of 4/64 one.

Below there is a simple plot and a list of the GE863-PRO³ key elements:

- Atmel AT91SAM9260 microcontroller (220 MIPS at 200Mhz, MMU, EmbeddedICE Debug Communication Channel Support).
- RAM:
 - 8MB or 64MB¹ SDRAM Mobile for GE863-PRO³4/8 and 4/64
 - 64MB SDRAM Mobile Micron, Numonyx or Samsung for GE863-PRO³ 128/64
- Flash:
 - 4MB Serial Flash Atmel via SPI interface for GE863-PRO³ 4/8 and 4/64
 - 128MB Nand Parallel Flash Micron, Numonyx via EBI interfacefor GE863-PRO³ 128/64

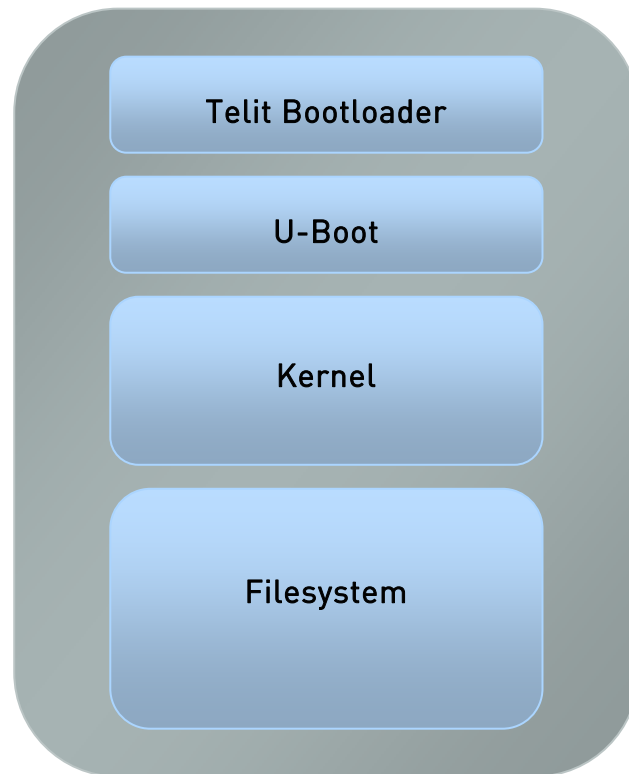
¹ Depends of product configuration that client choses





The **GE863-PRO³** is offered in a Ball-Grid-Array (BGA) package enabling a very low profile and small product size required for the design of extremely compact





2.2.1. Telit Bootloader

Telit Bootloader is a small binary used for hardware-related management: its main task is to load and start Telit customized U-Boot. For further details refer to [2] .

2.2.2. Telit customized U-boot

U-Boot is the Open Source "universal" cross-platform bootloader supporting hundreds of embedded boards and a wide variety of microcontrollers (the Atmel AT91SAM9260 included).

Some of the features provided by U-Boot are:

- Hardware initialization.
- Providing boot parameters for the Linux kernel.
- Starting the Linux kernel.
- Reading and writing memory locations.
- Uploading new binary images to the board's RAM.
- Copying binary images from RAM to FLASH memory.



- Storing environment variables which can be used to configure the system.

The GE863-PRO³ has a customized version of U-Boot to get the most out of the board. For further details refer to [2].
Telit recommends running Telit Linux along with U-Boot images.

2.2.3. Linux kernel

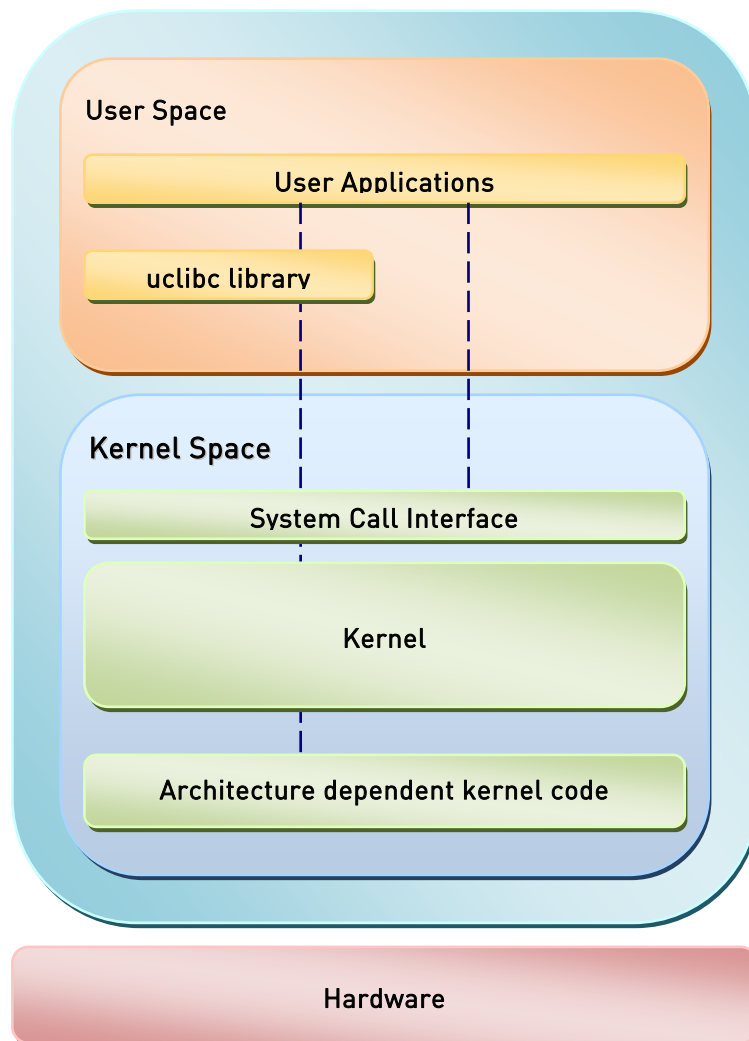
2.2.3.1. Overview

The kernel is the central part of the GNU/Linux operating system: its main task is to manage system's resources in order to make the hardware and the software communicate. A kernel usually deals with process management (including inter-process communication), memory management and device management.

The Linux kernel belongs to the family of Unix-like operating system kernel; created in 1991, it has been developed by a huge number of contributors worldwide during these years, becoming one of the most common and versatile kernel for embedded systems.

Below there is a schematic representing, from a high level perspective, the architecture of a GNU/Linux operating system:





Two regions can be identified:

1. User space: where the user applications are executed.
2. Kernel space: where the kernel (with all its components such as device drivers) works.

These two regions are separated and have different memory address spaces; there are several methods for user/kernel interaction:

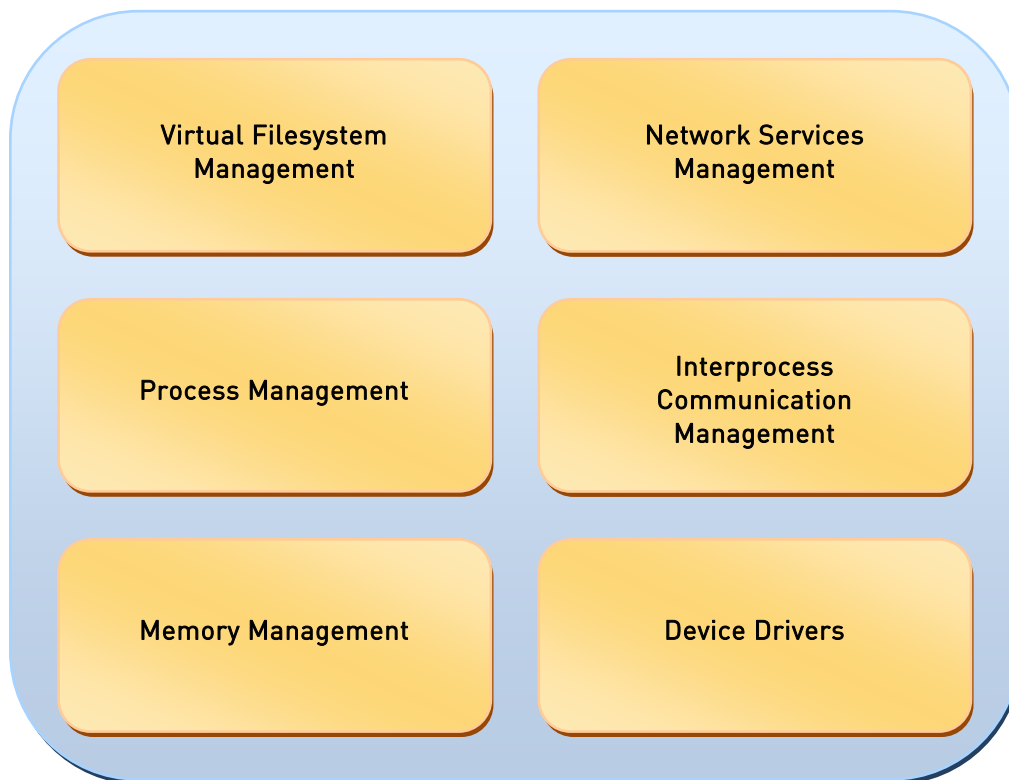
- Using the System Call Interface that connects user to the kernel and provides the mechanism to communicate between the user-space application and the kernel through the C library.
- Using kernel calls directly from application code leaping over the C library.
- Using the virtual filesystem.



The ordinary C library in Linux system is the glibc. Uclibc is a C library mainly targeted for developing embedded Linux systems; despite being much smaller than the glibc it has almost all its features (including shared libraries and threading), making easy to port applications from glibc to uclibc.

The Linux kernel architecture-independent code stays on the top of platform specific code for the GE863-PRO³ board: this code allows exploiting all hardware features of the GE863-PRO³.

Inside the kernel we can find, among the others, the following fundamental components:



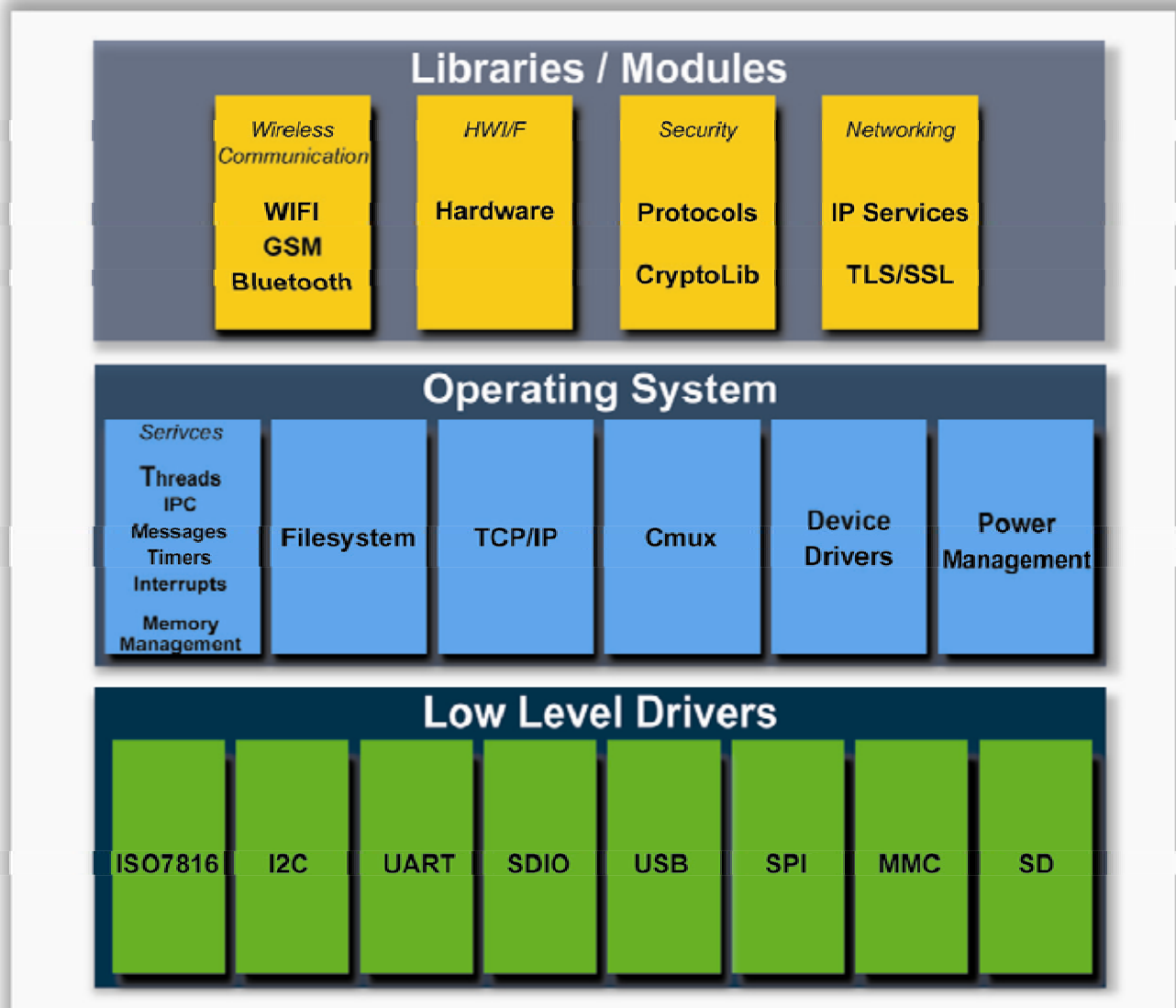
The Linux kernel is a monolithic one (i.e. all OS services run along with the main kernel thread, thus also residing in the same memory area), but it has the capability to dynamically load/unload some of its components called modules.



A kernel module is a compiled piece of code which can be dynamically linked to the kernel when is needed (becoming part of the kernel as the other normal kernel code) then removed when it is no longer needed; modules are mainly used for device drivers.

2.2.3.2. The GE863-PRO³ Linux kernel

The Linux kernel on the GE863-PRO³ is based on version number 2.6.24. Below there is a picture showing the kernel main components:



The GE863-PRO³ Linux kernel comes with some of its features linked statically; while others are compiled as modules (see the table below).

At the moment, available drivers are:

Driver Name	Module	Module Name	Loaded by default
GPIO	Y	at91sam9260_gpio	Y
GE863PRO3_GPIO	Y	ge863pro3_gpio	N
MMC/SD	N	N/A	Y
Ethernet	Y	macb	N
Support for Host-side USB	Y	ohci-hcd	N
Support for USB Mass Storage	Y	usb-storage	N
USB Ethernet Gadget	Y	g_ether	N
I2C	Y	i2c-core i2c-algo-bit i2c-dev i2c-gpio	N
SPI	Y	spidev	N
ADC	Y	adc_driver	N
SSC	Y	ge863pro3_ssc	N

2.2.4. Filesystem

2.2.4.1. Overview

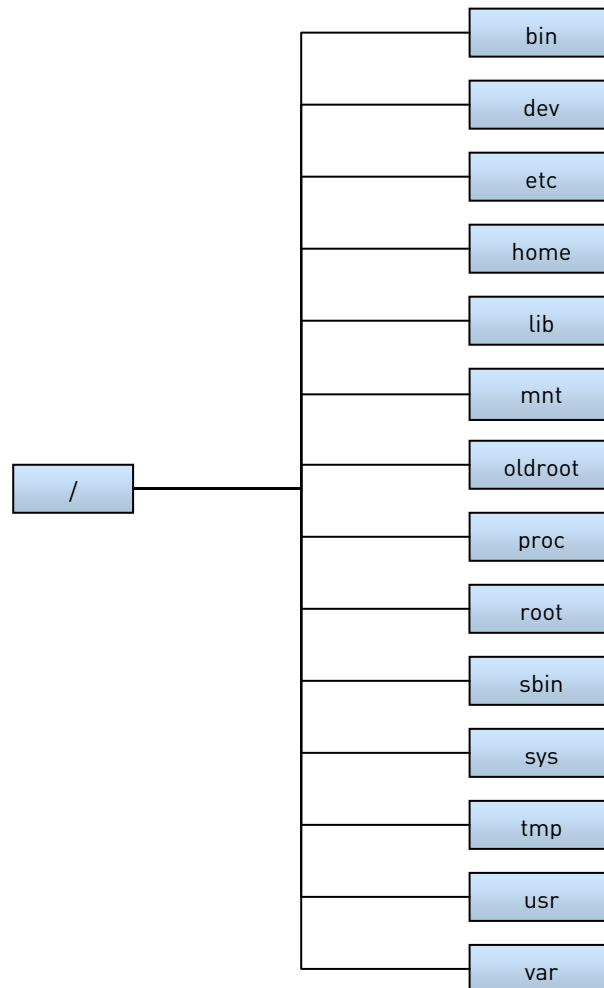
A filesystem is the entity where the files are placed logically for storage and retrieval. The filesystem also specifies conventions for naming files (e.g. maximum number of characters for the file name). Usually the file system has a hierarchical-tree structure: files are placed in directories inside this tree structure.

The GE863-PRO³ has the JFFS2, a log-structured file system specifically for use on flash devices in embedded systems. It implements file compression with the following formats: Zlib, LZO, Real time.

2.2.4.2. The filesystem structure

The image below shows the filesystem root tree.





`/bin`

This directory contains essential tools and other programs (so called binaries): note that, by default in the GE863-PRO³, a most of these files are symbolic link that depend on Busybox (see paragraph 2.2.4.3). Moreover the Busybox binary is also placed under this directory.

`/dev`

This directory contains files representing the system's various hardware devices. Here, for example, can be found all the devices representing the various GPIOs.

`/etc`

This directory contains system configuration files, startup files other. Here, for example, can be found the file `fstab` used for mounting the various devices in the filesystem.

`/home`



3. System Startup

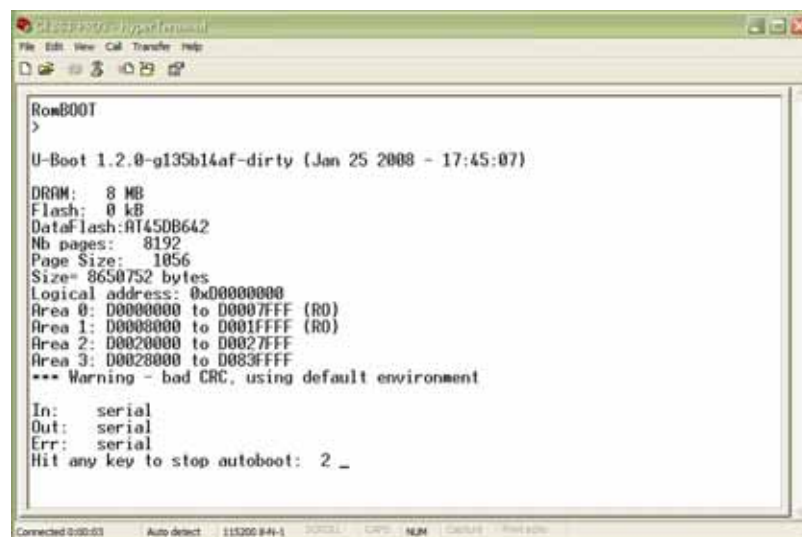
3.1. Startup process

Connect the GE863-PRO³ to your host system via serial cable (use Debug port of the EVK, for further details refer to document [3]). In your host system open a terminal program (such as Hyper Terminal) and use the following parameters for the connection:

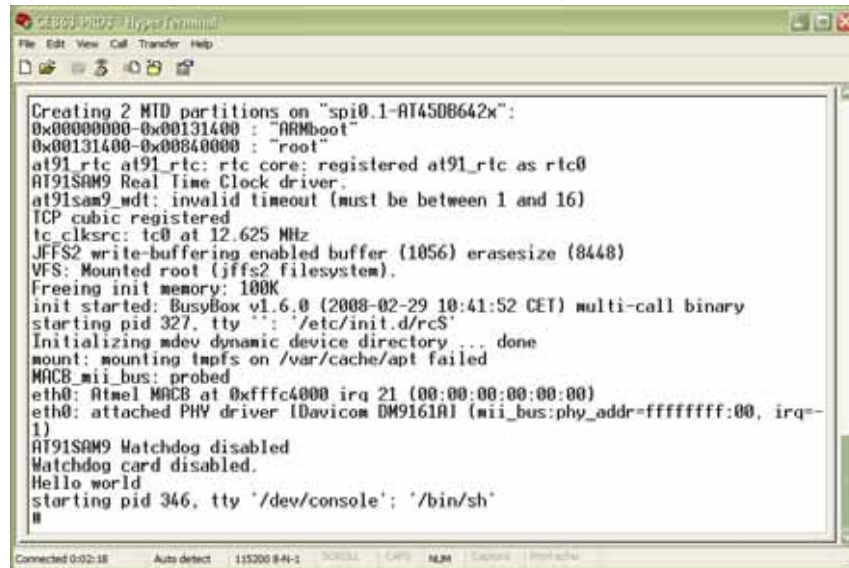
Bits per second: 115200
Data bits: 8
Parity: None
Stop bits: 1
Flow Control: None

After turning on the GE863-PRO³ the following operations occur:

- The Telit Bootloader (refer to 2.2.1) starts and after initializing the hardware it loads the U-Boot image from flash into RAM and runs it.
- Telit customized U-Boot (refer to 2.2.2) starts and in the terminal you should see a countdown. If you wish to enter in U-Boot command mode (for further details refer to document [2]) press any key, otherwise the startup process will continue within a few seconds: U-Boot loads the kernel image in RAM then starts it.



- The kernel image decompresses itself and starts. After the initial sequence the control is passed to the shell (refer to 3.2) which can be used to interact with the target. At this point the system is ready to be used.



3.2. The Linux shell

The Linux shell is a user program that allows you to interact with the target by entering commands from the keyboard; the shell parses the command, executes it and display the output of the command on the screen.

When the target has finished booting, in the terminal the shell prompt will appear:

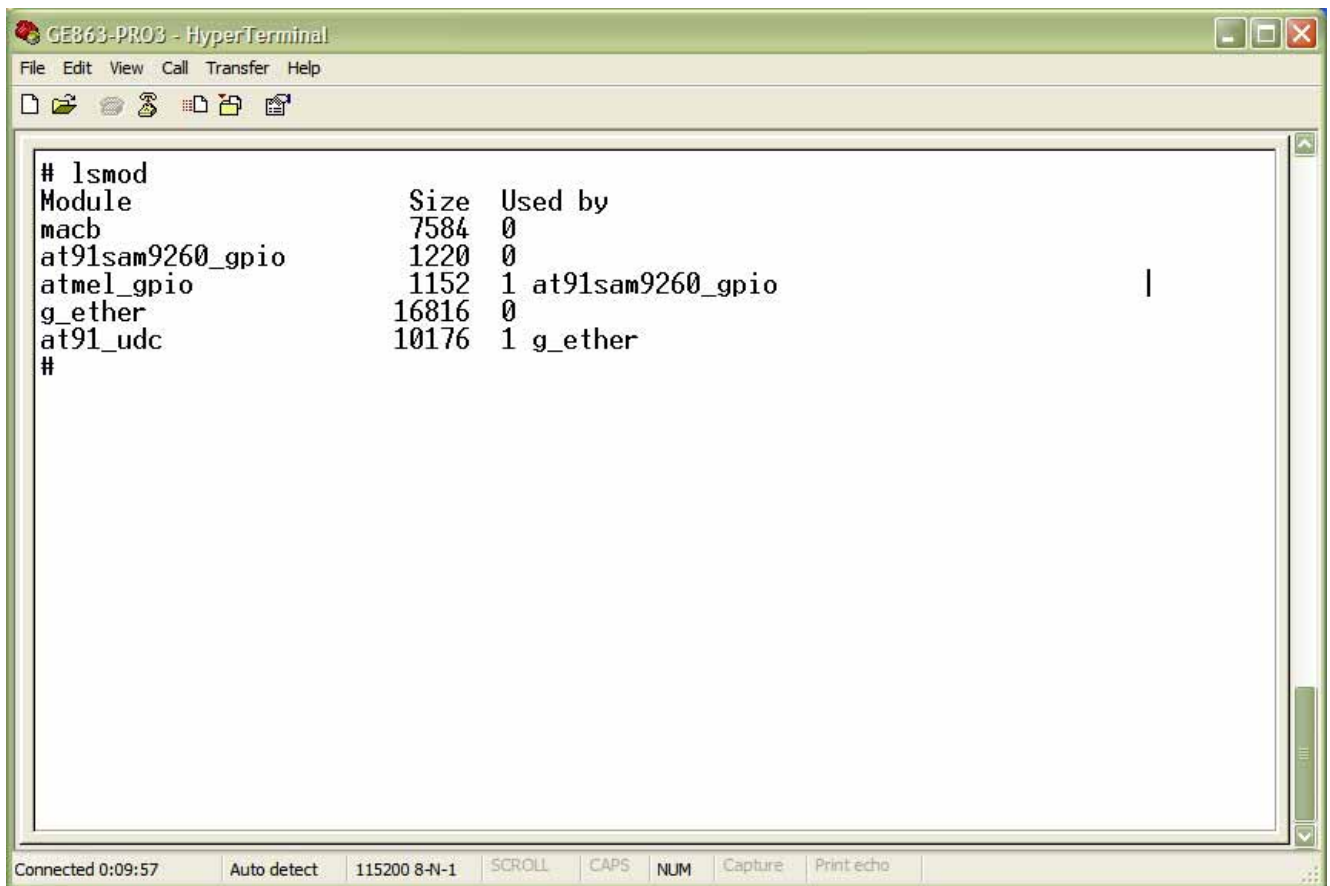
#

This means that the shell is ready to accept command: you can type in the terminal the command you want to execute; for example:

ls

You should see the listing of the “/” directory as in the image below.





```

# lsmod
Module                Size  Used by
macb                   7584  0
at91sam9260_gpio      1220  0
atmel_gpio             1152  1 at91sam9260_gpio
g_ether                16816 0
at91_udc              10176 1 g_ether
#
  
```

To unload a module type in the terminal:

```
# rmmod <module name>
```

Beware that `rmmod` removes only modules which are not currently used or are not needed by other modules as a dependency.

3.4. Auto-Setup at system startup

To automatically launch commands after boot create a script file in `/etc/init.d/` called `Sxy` (where `xy` is a number starting from 00 to 99). Inside this file put all the commands you need to be started after boot. For example, suppose you want to load the Ethernet on USB module and to configure this interface: create the file `/etc/init.d/S00` and edit it with the following content:

```
modprobe g_ether
ifconfig usb0 192.168.1.3 netmask 255.255.255.0
```

At the next reboot the script should be executed automatically.



3.5. Time-based scheduling service

`crond` is a daemon that allows to execute commands or scripts automatically at a specified time/date. It wakes up every minute, examining all stored `crontab` files, checking each command to see if it should be run in the current minute.

To enable `crond` daemon just type:

```
# crond
```

`crontab` is the program used to install or list the `crontab` files used to drive the `crond`.

To edit the current `crontab` file using the `vi` editor type:

```
# crontab -e
```

After you exit from the editor, the modified `crontab` file will be installed automatically.

The current `crontab` file will be displayed on standard output typing:

```
# crontab -l
```

With regard to `crontab` file syntax, it has the following format as a series of fields, separated by spaces and/or tabs where every line represents a job.

Job fields					
Minute (0-59)	Hour (0-23)	Day of Month (1-31)	Month (1-12)	Day of Week (0-6) (Sunday=0)	Command to be executed

The following ways allow to specify multiple date/time values in a field:

- The comma (',') operator specifies a list of values, for example: "1, 3, 4, 7, 8"
- The dash ('-') operator specifies a range of values, for example: "1-6", which is equivalent to "1, 2, 3, 4, 5, 6"
- The asterisk ('*') operator specifies all possible values for a field. For example, an asterisk in the hour time field would be equivalent to 'every hour' (subject to matching other specified fields).



For example, the following job will execute `test.sh` script at one minute past midnight each day:

```
01 00 * * * ./test.sh
```

3.6. Downloading a file into GE863-PRO³

There are several ways to download a file in the target. In the following paragraphs you can find explained three of these methods; for further details refer to document [4].

3.6.1. Downloading a file using the Ethernet connection

Be sure to have the Telit Development Environment correctly installed (with an Ethernet connection up) and coLinux started (refer to document [4] for further details). In the host system go to **Start** → **Telit Development Platform** → **Console**: and the Linux console will be opened. Type:

```
# cd /mnt/windows
```

Now the current directory is where the Windows partition has been mounted.



Once identified the file to be copied, use the `cp` command in the following form:

```
# cp -r <path where the file is>/<file name> /var/www/apache2-
default/
```



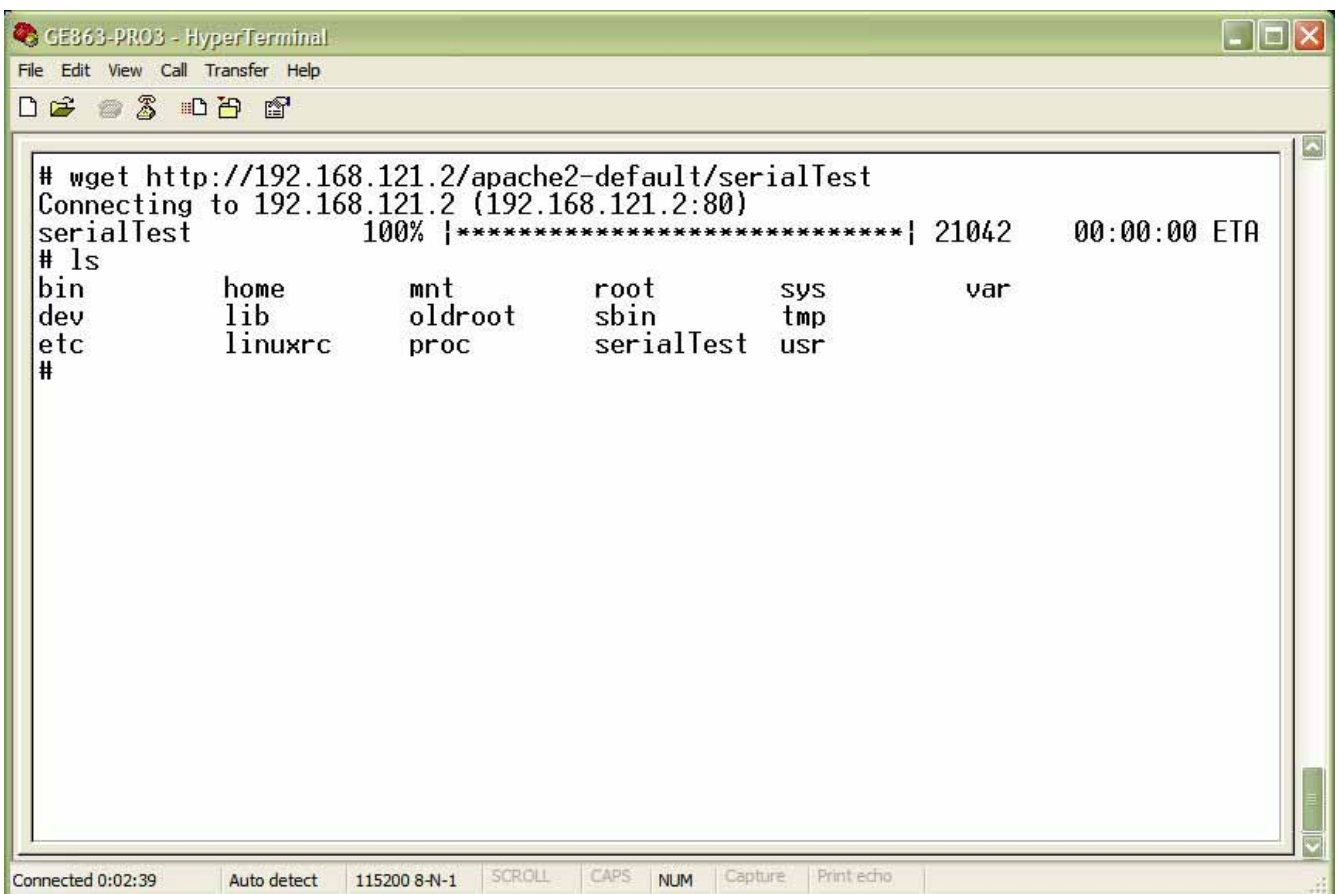
For example:

```
# cp -r ARM/binaries/executable /var/www/apache2-default/
```

The file is ready to be uploaded in the target; open the terminal software (as explained in 3.1) and, when the shell is ready, within the directory where you want to place the uploaded file, type:

```
# wget http://192.168.121.2/apache2-default/<file name>
```

You should see an output similar to that in the image below (supposing that the file is called serialTest):



The `ls` command shows that the file has been downloaded.

To remove the file from the development environment type in the Linux console:

```
# rm -rf /var/www/apache2-default/<file name>
```



3.6.2. Downloading a file using an USB mass storage device

Follow the procedure described in 4.13.1 for mounting the device (for example an USB memory key). Then, type in the terminal:

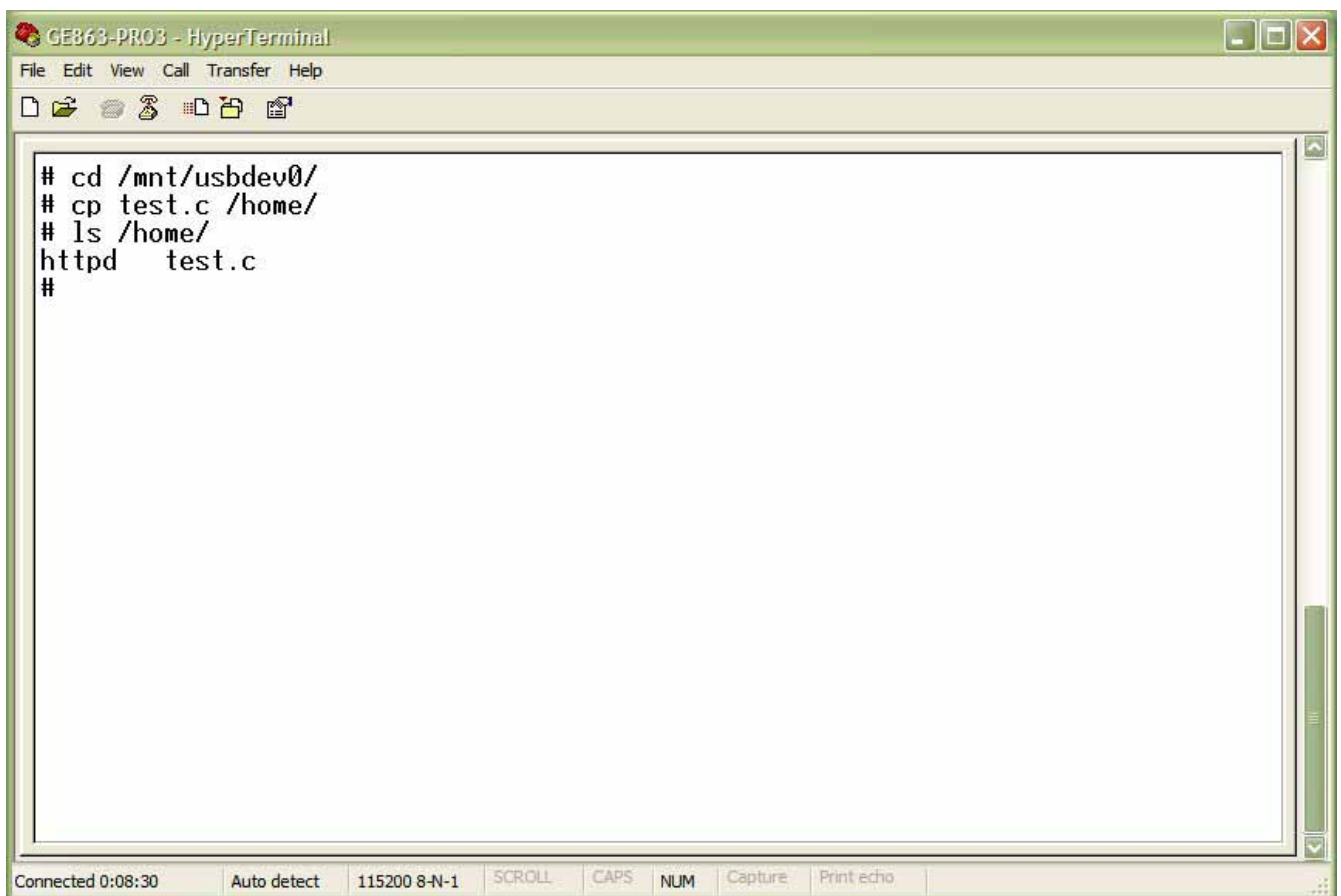
```
# cd /mnt/usbdev0
```

in order to have the current working directory in the USB mass storage device. Now the file can be copied using the cp command:

```
# cp <file name> <directory where the file is to be copied>
```

For example:

```
# cp test.c /home/
```



The ls command shows that the file has been copied.



3.6.3. Downloading/Uploading a file using a serial port

LRZSZ package allows file downloading and uploading by the means of a serial port (see 4.1).

The supported protocols are YModem and ZModem.

The `rz` command downloads a file into GE863-PRO³ through a `ttySX` device. Below the command synopsis:

```
# rz protocol </dev/ttySX >/dev/ttySX
```

The `protocol` parameter can be one of the following:

```
--ymodem    to use YModem protocol
--zmodem    to use ZModem protocol
```

For example, to receive in the current directory a file from `ttyS1` serial port by YModem or ZModem protocol, type respectively:

```
# rz --ymodem </dev/ttyS1 >/dev/ttyS1
# rz --zmodem </dev/ttyS1 >/dev/ttyS1
```

To see a description of all `rz` command parameters, just type:

```
# rz -h
```

The `sz` command uploads a file named `infilename` from GE863-PRO³ through a `ttySX` device:

```
# sz protocol infilename </dev/ttySX >/dev/ttySX
```

For example, to send the `test` file, from the current directory, to `ttyS1` serial port by YModem or ZModem protocol, type respectively:

```
# sz --ymodem test </dev/ttyS1 >/dev/ttyS1
# sz --zmodem test </dev/ttyS1 >/dev/ttyS1
```

To see a description of all `sz` command parameters, just type:

```
# sz -h
```



3.7. Version statistics

Current versions of Telit bootloader, Telit customized U-boot and Linux installed onto the target are shown by the `ver` command:

```
# ver

Bootloader version 21

U-boot version 21

Linux version Telit-06.0006

#
```



4. Device Drivers

Under Linux OS devices distinguish among three fundamental types: *char*, *block* and *network*. Each kernel module (see paragraph 2.2.3) usually implements one of these types, and thus is classifiable as a *char module*, a *block module*, or a *network module*. A *char* (character) device is one that can be accessed as a stream of bytes (like a file); serial ports (e.g. `/dev/ttyS0`) are examples of char devices. A char driver usually implements at least the *open*, *close*, *read*, and *write* system calls allowing this type of communication.

Like *char* devices, *block* devices are accessed by filesystem nodes in the `/dev` directory. A block device is a device (e.g., a disk) that can host a filesystem. *Block* and *char* devices differ only in the way data is managed internally by the kernel, thus *block* drivers have a completely different interface to the kernel than *char* drivers.

A *network* interface is a device that is able to exchange data with other hosts and is controlled by a *network* driver. Not being a stream-oriented device, a *network* interface doesn't have a corresponding entry in the filesystem.

Some types of drivers work with additional layers of kernel support functions for dedicated device interface: for example USB devices are driven by a USB module that works with the USB subsystem, but the devices themselves can be char devices (USB serial ports), block devices (USB memory cards), or network devices (USB Ethernet interfaces).

4.1. Serial port

As discussed above serial ports are *char* devices that can be accessed through the following filesystem nodes available on the GE863-PRO³ (corresponding peripherals of ARM9 processor are in parentheses):

ttyS0	used by the shell (DEBUG port)
ttyS1	available (UART0)
ttyS2	available (UART1)
ttyS3	available (UART2)
ttyS4	available (UART3)
ttyS5	available (UART4)
ttyS6	available (UART5)

Refer to document [3] for further information about hardware setup of serial ports before using them.

If you want to test the ttyS1 serial port you can, for example, connect with a serial cable your host pc and the GE863-PRO³ on ttyS1.

On GE863-PRO³' shell type:



```
# cat /dev/ttyS1 &
```

in order to print on standard output any character received over this serial port. Now open a terminal on you host pc (by default configured for 9600 data rate) and type, for example,

```
hello<carriage return>
```

to write the “hello” test string over ttyS1.

If you want to set serial port baud rate, type on GE863-PRO³ shell:

```
# stty baudrate </dev/ttySx
```

For example, to set 115200 bps data rate for ttyS1, type:

```
# stty 115200 </dev/ttyS1
```

The following subparagraphs show all the functions that can be used from C source code to perform read/write operations on the serial devices.

4.1.1. open()

The *open()* function establishes the connection between a file and a file descriptor. The file descriptor³ is used by other I/O functions to refer to that file.

Header file:
fcntl.h

Prototype:
int open(const char *pathname, int flags)

Parameters:
pathname – file name with its own path

flags – an *int* specifying file opening mode: is one of O_RDONLY, O_WRONLY or O_RDWR which request opening the file read-only, write-only or read/write, respectively.

In addition, zero or more file flags can be set, *e.g.*:

- O_NOCTTY: If *pathname* refers to a terminal device it will not become the process's controlling terminal even in case the process does not have one.
- O_NDELAY or O_NONBLOCK: when possible, the file is opened in non-blocking mode

³ Identifies ID of the file



Returns:

The new file descriptor *fildes* if operation is completed successfully, otherwise it is -1

Example:

Open the */dev/ttyS1* device.

```
int fd: // file descriptor for /dev/ttyS1 entry

if ((fd = open("/dev/ttyS1", O_RDWR | O_NOCTTY | O_NDELAY)) < 0)
{
    /* ERROR MANAGEMENT ROUTINE */
} else {
    /* SERIAL PORT OPENED */
}
```

4.1.2. read()

The *read()* function reads *nbyte* bytes from the file associated with the open file descriptor, *fildes*, and copies them in the buffer that is pointed to by *buf*.

Header file:

unistd.h

Prototype:

ssize_t read(int fildes, void *buf, size_t nbyte)

Parameters:

- filides – file descriptor
- buf – destination buffer pointer
- nbyte – number of bytes that read() attempts to read

Returns:

The number of bytes actually read if operation is completed successfully, otherwise it is -1

Example:

Read *sizeof(read_buff)* bytes from the file associated with *fd* and stores them into *read_buff*.

```
char read_buff[BUFF_LEN];

if(read(fd, read_buff, sizeof(read_buff)) < 0)
{
    /* Error Management Routine */
} else {
```



```

        /* Value Read */
    }

```

4.1.3. write()

The `write()` function writes *nbyte* bytes from the buffer that are pointed by *buf* to the file associated with the open file descriptor, *fildes*.

Header file:

unistd.h

Prototype:

```
ssize_t write(int fildes, const void *buf, size_t nbyte)
```

Parameters:

fildes – file descriptor

buf – destination buffer pointer

nbyte – number of bytes that write() attempts to write

Returns:

The number of bytes actually written if operation is completed successfully (this number shall never be bigger than *nbyte*), otherwise it is -1

Example:

Write `strlen(value_to_be_written)` bytes from the buffer pointed to by `value_to_be_written` to the file associated with the open file descriptor, *fd*.

```

char value_to_be_written[] = "Hello world\r\n";

if (write(fd, value_to_be_written, strlen(value_to_be_written))
    < 0)
{
    /* Error Management Routine */
} else {
    /* Value Written */
}

```

4.1.4. close()

The `close()` function deallocates the file descriptor indicated by *fildes*. To deallocate means to make the file descriptor available for subsequent calls to `open()` or other functions that allocate file descriptors.

Header file:

unistd.h



Prototype:

int close(int fildes)

Parameters:

fildes – file descriptor

Returns:

0 if operation is completed successfully, otherwise it is -1

Example:

Close the /dev/ttyS1 device.

```
if(close(fd) < 0)
{
    /* Error Management Routine */
} else {
    /* File Closed */
}
```

4.1.5. termios interface

POSIX.1 defines a standard interface for querying and manage tty devices. This interface is called termios and is defined in the system header file <termios.h>. termios is essentially a data structure and a set of functions that manipulate it.

4.1.5.1. Structure and Types

The following data types (all unsigned int) are defined:

- **tcflag_t** – used for terminal modes
- **cc_t** – used for terminal special characters
- **speed_t** – used for terminal baud rates. The valid values for objects of type **speed_t**, supported by the underlying hardware, are:

Value	Description
B0	Hang-up
B9600	9600 baud
B19200	19200 baud
B38400	38400 baud
B57600	57600 baud
B115200	115200 baud



The termios structure is defined as shown below:

```
struct termios {
    tcflag_t c_iflag;           /* input mode flags */
    tcflag_t c_oflag;           /* output mode flags */
    tcflag_t c_cflag;           /* control mode flags */
    tcflag_t c_lflag;           /* local mode flags */
    cc_t c_cc[NCCS];           /* control characters */
};
```

The `c_iflag` member controls input processing options. It affects whether and how the terminal driver processes input before sending it to a program.

The `c_oflag` member controls output processing and determines if and how the terminal driver processes program output before sending it to the screen or other output device.

The `c_cflag` member controls the hardware characteristics of the terminal device.

The local mode flags, stored in `c_lflag` member, manipulate terminal characteristics, such as whether or not input characters are echoed on the screen.

The `c_cc` array contains values for special character sequences, such as `^I` (quit) and `^H` (delete), and how they behave.

Terminals operate in one of two modes, *canonical* (or cooked) mode, in which the terminal device driver processes special characters and feeds input to a program one line at a time, and *non-canonical* (or raw) mode, in which most keyboard input is unprocessed and unbuffered. The shell is an example of an application that uses canonical mode.

All the constant values, and their meaning, that can be set for the termios struct members `c_iflag`, `c_oflag`, `c_cflag` and `c_lflag` are listed below:

Member	Flag	Description
c_iflag	BRKINT	Signal interrupt on break
	ICRNL	Map CR to NL on input
	IGNBRK	Ignore break condition
	IGNCR	Ignore CR
	IGNPAR	Ignore characters with parity errors
	INLCR	Map NL to CR on input



	INPCK	Enable input parity check	
	ISTRIP	Strip character	
	IUCLC	Map upper-case to lower-case on input (LEGACY)	
	IXANY	Enable any character to restart output	
	IXOFF	Enable start/stop input control	
	IXON	Enable start/stop output control	
	PARMRK	Mark parity errors	
c_oflag	OPOST	Post-process output	
	OLCUC	Map lower-case to upper-case on output (LEGACY)	
	ONLCR	Map NL to CR-NL on output	
	OCRNL	Map CR to NL on output	
	ONOCR	No CR output at column 0	
	ONLRET	NL performs CR function	
	OFILL	Use fill characters for delay	
	NLDLY		
	NL0	Newline character type 0	Select newline delay
	NL1	Newline character type 1	
	CRDLY		Select carriage-return delays
	CR0	Carriage-return delay type 0	
	CR1	Carriage-return delay type 1	
	CR2	Carriage-return delay type 2	
	CR3	Carriage-return delay type 3	
TABDLY		Select horizontal-tab delays	
TAB0	Horizontal-tab delay type 0		
TAB1	Horizontal-tab delay type 1		
TAB2	Horizontal-tab delay type 2		

TAB1	Horizontal-tab delay type 1
TAB2	Horizontal-tab delay type 2
TAB3	Horizontal-tab delay type 3



	Expand tabs to spaces	
	BSDLY	
	BS0 Backspace-delay type 0	Select backspace delays
	BS1 Backspace-delay type 1	
	VTDLY	
	VT0 Vertical-tab delay type 0	Select vertical-tab delays
	VT1 Vertical-tab delay type 1	
	FFDLY	
	FF0 Form-feed delay type 0	Select form-feed delays
	FF1 Form-feed delay type 1	
c_cflag	CSIZE	
	CS5 5 bits	Character size
	CS6 6 bits	
	CS7 7 bits	
	CS8 8 bits	
	CSTOPB	
	CREAD	Enable receiver
	PARENB	Parity enable
	PARODD	Odd parity, else even
	HUPCL	Hang up on last close
	CLOCAL	Ignore modem status lines
	CRTSCTS	Enable RTS/CTS (hardware) flow control.



c_lflag	ECHO	Enable echo
	ECHOE	Echo erase character as error-correcting backspace
	ECHOK	Echo KILL
	ECHONL	Echo NL
	ICANON	Canonical input (erase and kill processing)
	IEXTEN	Enable extended input character processing
	ISIG	Enable signals
	NOFLSH	Disable flush after interrupt or quit
	TOSTOP	Send SIGTTOU for background output
	XCASE	Canonical upper/lower presentation (LEGACY)

The special control characters of the `c_cc` array with the relative symbolic indices (initial values) and meaning are:

Index	Value	Control Character	Description
VINTR	0x03	Ctrl-C	Send a SIGINT signal. Recognized when ISIG is set, and then not passed as input.
VQUIT	0x1C	Ctrl-\	Quit character. Send SIGQUIT signal. Recognized when ISIG is set, and then not passed as input.
VERASE	0x7F	DEL	Erase character. This erases the previous not-yet-erased character, but does not erase past EOF or beginning-of-line. Recognized when ICANON is set, and then not passed as input.
VKILL	0x15	Ctrl-U	Kill character. This erases the input since the last EOF or beginning-of-line. Recognized when ICANON is set, and then not passed as input.
VEOF	0x04	Ctrl-D	End-of-file character. More precisely: this character causes the pending tty buffer to be sent to the waiting user program without waiting for end-of-line. If it is the first character of the line, the <code>read()</code> in the user program returns 0, which signifies end-of-file. Recognized when ICANON is set, and then not passed as input.
VMIN	-	-	Minimum number of characters for non-canonical read.
VEOL	0x00	NULL	Additional end-of-line character. Recognized when ICANON is set.
VTIME	-	-	Timeout in deciseconds for non-canonical read.
VEOL2	0x00	NULL	Yet another end-of-line character. Recognized when ICANON is set.
VSTART	0x21	Ctrl-Q	Start character. Restarts output stopped by the Stop character. Recognized when IXON is set, and then not passed as input.
VSTOP	0x23	Ctrl-S	Stop character. Stop output until Start character typed. Recognized when IXON is set, and then not passed as input.
VSUSP	0x1A	Ctrl-Z	Suspend character. Send SIGTSTP signal. Recognized when ISIG is set, and then not passed as input.



VLNEXT	0x16	Ctrl-V	Literal next. Quotes the next input character, depriving it of a possible special meaning. Recognized when IEXTEN is set, and then not passed as input.
VWERASE	0x17	Ctrl-W	Word erase. Recognized when ICANON and IEXTEN are set, and then not passed as input.
VREPRINT	0x12	Ctrl-R	Reprint unread characters. Recognized when ICANON and IEXTEN are set, and then not passed as input.

4.1.5.2. Functions

4.1.5.2.1. tcgetattr()

This method gets the parameters associated with the object referred by *fd* and stores them in the *termios* structure referenced by *termios_p*.

Header file:
termios.h

Prototype:
int tcgetattr(int fd, struct termios *termios_p)

Parameters:
fd – filedescriptor relative to the tty device
termios_p – pointer to a termios structure

Returns:
0 if successful, -1 otherwise

Example:

```
if(tcgetattr(filedes, &serial_config) != 0)
    /* Error Management */
```

4.1.5.2.2. tcsetattr()

This method sets the parameters associated with the terminal and specifies when changes have to take effect.

Header file:
termios.h

Prototype:
int tcsetattr(int fd, int optional_actions, const struct termios *termios_p)



Parameters:

fd – filedescriptor relative to the tty device

optional_actions – flag which specifies when the changes have to take effect. The possible values are:

- **TCSANOW**: the change occurs immediately.
- **TCSADRAIN**: the change occurs after all output written to *fd* has been transmitted. This function should be used when changing parameters that affect output.
- **TCSAFLUSH**: the change occurs after all output written to the object referred by *fd* has been transmitted, and all input that has been received but not read will be discarded before the change is made.

termios_p – pointer to a termios structure

Returns:

0 if successful, -1 otherwise

Example:

```
if(tcsetattr(filedes, TCSANOW, &serial_config) != 0)
    /* Error Management */
```

4.1.5.2.3. **tcsendbreak()**

This method transmits a continuous stream of zero-valued bits for a specific duration, if the terminal is using asynchronous serial data transmission. If *duration* is zero, it transmits zero-valued bits for at least 0.25 seconds and no more than 0.5 seconds. If *duration* is not zero, it sends zero-valued bits for some implementation-defined length of time.

Header file:

termios.h

Prototype:

```
int tcsendbreak(int fd, int duration)
```

Parameters:

fd – filedescriptor relative to the tty device

duration – value used for transmit time

Returns:

0 if successful, -1 otherwise



Example:

```
if(tcsendbreak(filedes, d_time) != 0)
    /* Error Management */
```

4.1.5.2.4. **tcdrain()**

This method waits until all output written to the object referred to by *fd* has been transmitted.

Header file:

termios.h

Prototype:

```
int tcdrain(int fd)
```

Parameters:

fd – filedescriptor relative to the tty device

Returns:

0 if successful, -1 otherwise

Example:

```
if(tcdrain(filedes) != 0)
    /* Error Management */
```

4.1.5.2.5. **tcflush()**

This method discards data written to the object referred to by *fd* but not transmitted, or data received but not read, depending on the value of *queue_selector*.

Header file:

termios.h

Prototype:

```
int tcflush(int fd, int queue_selector)
```

Parameters:

fd – filedescriptor relative to the tty device

queue_selector – flag that indicates what action has to be performed. The possible values are:

- **TCIFLUSH:** flushes data received but not read.



- **TCOFLUSH**: flushes data written but not transmitted.
- **TCIOFLUSH**: flushes both data received but not read, and data written but not transmitted.

Returns:

0 if successful, -1 otherwise

Example:

```
if(tcflush(filedes, TCIFLUSH) != 0)
    /* Error Management */
```

4.1.5.2.6. tcflow()

This method suspends transmission or reception of data on the object referred to by *fd*, depending on the value of *action*.

Header file:

termios.h

Prototype:

```
int tcflow(int fd, int action)
```

Parameters:

fd – filedescriptor relative to the tty device

action – flag which specifies what action has to be performed. The possible values are:

- **TCOOFF**: suspends output.
- **TCOON**: restarts suspended output.
- **TCIOFF**: transmits a STOP character, which stops the terminal device from transmitting data to the system.
- **TCION**: transmits a START character, which starts the terminal device transmitting data to the system.

Returns:

0 if successful, -1 otherwise

Example:

```
if(tcflow(filedes, TCOON) != 0)
    /* Error Management */
```

4.1.5.2.7. cfmakeraw()



This method sets the flags stored in the `termios` structure to a state disabling all input and output processing, giving a "raw I/O path", as follows:

```
termios_p->c_iflag &= ~(IGNBRK | BRKINT | PARMRK | ISTRIP
    | INLCR | IGNCR | ICRNL | IXON);
termios_p->c_oflag &= ~OPOST;
termios_p->c_lflag &= ~(ECHO | ECHONL | ICANON | ISIG | IEXTEN);
termios_p->c_cflag &= ~(CSIZE | PARENB);
termios_p->c_cflag |= CS8;
```

Header file:
`termios.h`

Prototype:
`void cfmakeraw(struct termios *termios_p)`

Parameters:
`termios_p` – pointer to a `termios` structure

Returns:
None

Example:
`cfmakeraw(&serial_config);`

4.1.5.2.8. `cfgetispeed()`

This method returns the input baud rate stored in the `termios` structure.

Header file:
`termios.h`

Prototype:
`speed_t cfgetispeed(const struct termios *termios_p)`

Parameters:
`termios_p` – pointer to a `termios` structure

Returns:
A `speed_t` value indicating the input baud rate. Error conditions are not expected.

Example:



```
speed_t input_speed;  
input_speed = cfgetispeed(&serial_config);
```

4.1.5.2.9. cfgetospeed()

This method returns the output baud rate stored in the *termios* structure.

Header file:

termios.h

Prototype:

```
speed_t cfgetospeed(const struct termios *termios_p)
```

Parameters:

termios_p – pointer to a termios structure

Returns:

A speed_t value indicating the output baud rate. Error conditions are not expected.

Example:

```
speed_t output_speed;  
input_speed = cfgetospeed(&serial_config);
```

4.1.5.2.10. cfsetispeed()

This method sets the input baud rate stored in the *termios* structure to *speed*, which must be specified as one of the speed_t **Bnnn** constants listed above.

Header file:

termios.h

Prototype:

```
int cfsetispeed(struct termios *termios_p, speed_t speed)
```

Parameters:

termios_p – pointer to a termios structure

speed – the speed value to be set

Returns:

0 if successful, -1 otherwise

Example:



```
if(cfsetispeed(&serial_config, B115200) != 0)
    /* Error Management */
```

4.1.5.2.11. cfsetospeed()

This method sets the output baud rate stored in the *termios* structure to *speed*, which must be specified as one of the *speed_t Bnnn* constants listed above.

Header file:

termios.h

Prototype:

```
int cfsetospeed(struct termios *termios_p, speed_t speed)
```

Parameters:

termios_p – pointer to a termios structure

speed – the speed value to be set

Returns:

0 if successful, -1 otherwise

Example:

```
if(cfsetospeed(&serial_config, B115200) != 0)
    /* Error Management */
```

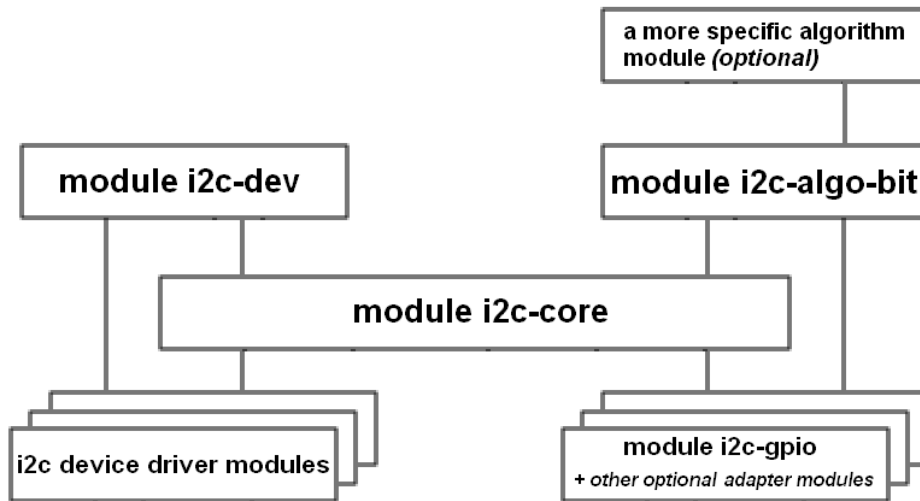
4.1.5.3. A Test Program

```
#include <termios.h>
#include <fcntl.h>
#include <string.h>

int SerOpen(char *serName, speed_t baudrate)
{
    int fd;
    struct termios serCfg;
    memset(&serCfg, 0, sizeof(serCfg));

    if((fd = open(serName, O_RDWR)) < 0)
        return -1; /* Can't open port */
    else
        /* Get the actual configuration */
        if(tcgetattr(fd, &serCfg) != 0)
            return -1; /* Can't get port parms */
```





So, in our system, in order to use the I²C Interface, the following steps shall be followed:

1. Load the i2c-core module;
2. Load the i2c-algo-bit module;
3. Load the i2c-dev module;
4. Load the i2c-gpio module and/or other adapter driver module(s);
5. Load your I²C device driver module(s).

However, you can find the list of all the available I²C loadable modules in the directory:
`/lib/modules/your_kernel_version/kernel/drivers/i2c`

4.2.1. Loading i2c modules

1. The module i2c-core is used by every other I²C modules, so it must be loaded first.

To load this module type the command

```
# modprobe i2c-core
```

2. The i2c-algo-bit module implements a generic algorithm for the communications on the bus. If you need a more specific algorithm, you can modify the i2c-algo-bit source file or implement another module that uses i2c-algo-bit.

To load the i2c-algo-bit module type the command



flags – an *int* specifying file opening mode: that can be O_RDONLY, O_WRONLY or O_RDWR which requests opening the file read-only, write-only or read/write, respectively

Returns:

The new file descriptor *fildev* if operation is completed successfully, otherwise it is -1

Example:

Open the /dev/i2c-0.

```
int fd; // file descriptor for the /dev/i2c-0 entry

if((fd = open("/dev/i2c-0", O_RDWR) < 0)
{
    /* Error Management Routine */
} else {
    /* I2C Device Opened */
}
```

4.2.3. ioctl()

The ioctl() function manipulates the underlying device parameters. In particular, many operating characteristics can be controlled with ioctl() requests.

Header file:

sys/ioctl.h

Prototype:

int ioctl(int fildev, int request, ...)

Parameters:

fildev – file descriptor

request – device-dependent request code.

The following ioctls request codes can be used for I²C:

- I2C_SLAVE, I2C_SLAVE_FORCE: to select the slave address
- I2C_TENBIT: to set the ten bit addressing
- I2C_PEC: to enable the packet error checking
- I2C_FUNCS: to get the adapter functionality mask
- I2C_RDWR: to do a combined read/write transaction
- I2C_SET_RATE, I2C_GET_RATE: to set/get the I²C clock rate
- I2C_SMBUS: to use SMBus functionalities
- I2C_RETRIES: to set the number of retries when not acknowledged
- I2C_TIMEOUT: to set the timeout value.



Gets the adapter functionality mask and puts it in *funcs.

Example:

```
unsigned long funcs;
user = ioctl(file_descriptor, I2C_FUNCS, &funcs);
```

After this call, the funcs mask should be equal to FFF800F, the hexadecimal value for 111111111111000000000001111, which indicates that the following adapter functionalities are available:

```
I2C_FUNC_I2C
I2C_FUNC_10BIT_ADDR
    I2C_FUNC_PROTOCOL_MANGLING
    I2C_FUNC_SMBUS_HWPEC_CALC
    I2C_FUNC_SMBUS_BLOCK_PROC_CALL
    I2C_FUNC_SMBUS_QUICK
    I2C_FUNC_SMBUS_READ_BYTE
    I2C_FUNC_SMBUS_WRITE_BYTE
    I2C_FUNC_SMBUS_READ_BYTE_DATA
    I2C_FUNC_SMBUS_WRITE_BYTE_DATA
    I2C_FUNC_SMBUS_READ_WORD_DATA
    I2C_FUNC_SMBUS_WRITE_WORD_DATA
    I2C_FUNC_SMBUS_PROC_CALL
    I2C_FUNC_SMBUS_READ_BLOCK_DATA
    I2C_FUNC_SMBUS_WRITE_BLOCK_DATA
    I2C_FUNC_SMBUS_READ_I2C_BLOCK
    I2C_FUNC_SMBUS_WRITE_I2C_BLOCK
    I2C_FUNC_SMBUS_BYTE
    I2C_FUNC_SMBUS_BYTE_DATA
    I2C_FUNC_SMBUS_WORD_DATA
    I2C_FUNC_SMBUS_BLOCK_DATA
    I2C_FUNC_SMBUS_I2C_BLOCK
```

Of course, if the funcs mask is different from FFF800F, the system will support different functionalities. See the i2c.h file in the linux source tree for the available functionalities.

- I2C_RDWR

Do a combined read/write transaction without break in between. This is valid only if the adapter has I2C_FUNC_I2C.

The difference between doing normal read/write calls and using an ioctl with the I2C_RDWR request is that in the second case you can do the several read or write using the repeated start condition between two messages. If you use multiple calls to read and write functions instead of the ioctl with the I2C_RDWR request, you'll send



messages with a stop condition between them. The argument of the `ioctl` call is a pointer to a

```

    struct i2c_rdwr_ioctl_data {
        struct i2c_msg *msgs; /* ptr to array of simple messages
*/
        int nmsgs;           /* number of messages to exchange
*/
    }

```

where `nmsgs` is the number of messages to exchange. These messages are contained in the `struct i2c_msg` pointer that points to an array of structures with following definition:

```

struct i2c_msg {
    __u16 addr;           /* slave address      */
    __u16 flags;         /*
    __u16 len;           /* msg length      */
    __u8 *buf;          /* pointer to msg data */
};

```

In the `addr` field you have to put the address of the slave device you want to communicate with. In the `flags` field you have to set the options among the ones listed below:

I2C_M_TEN: setting this option you indicate that the address in the `addr` field is composed by ten bits, so you'll use the ten bit address mode;

I2C_M_RD: if set, a read operation will be performed; if you don't set it, it'll automatically do a write operation;

I2C_M_NOSTART: if set, there won't be any start condition during the sending of this message with the i2c protocol;

I2C_M_REV_DIR_ADDR: you should set this flag if you want to do a write, but need to simulate the process of a Read instead of a normal Write, or vice versa;

I2C_M_IGNORE_NAK: Normally a message is interrupted immediately if there is NACK from the client. Setting this flag treats any NACK as an ACK, and all subsequent messages are sent.

I2C_M_NO_RD_ACK: in a read message, the ACK/NACK bit from the master is skipped.

The `struct i2c_msg` also contains a pointer into a data buffer. The function will write or read data to or from that buffer depending on whether the `I2C_M_RD` flag is set or not in the `flags` field of a particular message. Finally, in the `len` field you have to set the number of bytes you want to be read (written) to (from) the array pointed by `buf`.



In the following example we suppose that our slave device is an EEPROM. We want to read a data from a specific internal address of the EEPROM, in this case the address is 0x0. First we have to do a write with the internal address, and then a read of the data:

```
struct i2c_rdwr_ioctl_data work_queue;
char buf[1];
struct i2c_msg msgs[ 2 ];

buf[0]=0x0;    //this is the internal address

msgs[0].len    = 1;
msgs[0].flags  = 0;
msgs[0].addr   = 0x50; //this is the device address
msgs[0].buf    = buf;

msgs[1].addr = addr;
msgs[1].flags = I2C_M_RD;
msgs[1].len = 255;
msgs[1].buf = buf2;

work_queue.msgs = msgs;
work_queue.nmsgs = 2;

ret = ioctl(fd,I2C_RDWR,&work_queue);
if ( ret < 0 ) {
    printf("Error during I2C_RDWR ioctl with error
code: %d\n",ret);
}
else{
printf("The value read at address %X
is %s\n",buf[0],work_queue.msgs[1].buf);
}
```

- I2C_SET_RATE, I2C_GET_RATE

The I²C clock rate can be reduced setting a prescalar value that divide the master clock rate for the use on I²C. So, if we call "arg" the prescalar value, the I²C clock frequency is about $(500 / (arg + 1))$ kHz. To set this prescalar value, call the ioctl function with the I2C_SET_RATE option. The allowed prescalar values ranges from 0 (which means "fully master clock rate use") to I2C_LOW_RATE (which corresponds to 50). Any other value won't be accepted. In the following code we can see an example where we divide by two the master clock rate:

```
ret = ioctl(fd,I2C_SET_RATE, 1);
if ( ret < 0 ) {
```



```
    printf("Error during I2C_SET_RATE ioctl with error
code: %d\n",ret);
}
```

If you want to know the current prescalar value, use I2C_GET_RATE following this example:

```
int value;
ret = ioctl(fd,I2C_RDWR,&value);
if ( ret < 0 ) {
    printf("Error during I2C_GET_RATE ioctl with error
code: %d\n",ret);
}
```

- I2C_SMBUS

The SMBus protocol is essentially a subset of the I²C protocol, so it can be easily implement in our system. An ioctl with the I2C_SMBUS option can be used to do a SMBus transfer, using the following data structure:

```
struct i2c_smbus_ioctl_data {
    __u8 read_write;
    __u8 command;
    __u32 size;
    union i2c_smbus_data {
        __u8 byte;
        __u16 word;
        __u8 block[I2C_SMBUS_BLOCK_MAX + 2];
        // block[0] is used for length
        // and one more for user-space
compatibility
    } *data;
};
```

The read_write variable must be configured with I2C_SMBUS_READ, if you need to do a read operation or with I2C_SMBUS_WRITE if you need to do a write operation. If this field is set with a different value the ioctl call will fail.

The command field value is always valid (is not checked by the driver) because it can be set with a command specific of your SMBus slave device. It functionally corresponds to the first location of the buf field of a struct i2c_msg.

The size value specifies the data format you need for the transfer. It can be:

- I2C_SMBUS_QUICK: to perform a SMBus quick command. In this case the *data field must be set to NULL;



- I2C_SMBUS_BYTE: to perform a byte read or write, depending on the `read_write` field value. The `*data` field must be set to NULL. In the write case, the byte that will be wrote is the `command` field value;
- I2C_SMBUS_BYTE_DATA: to perform a byte data read or write, depending on the `read_write` field value. The I2C_SMBUS_BYTE_DATA option differs from the I2C_SMBUS_BYTE option because it sends to the slave device both a one byte command and a byte of data;
- I2C_SMBUS_WORD_DATA: it is the analog of the I2C_SMBUS_BYTE_DATA to read or write a word data;
- I2C_SMBUS_BLOCK_DATA: to perform a data block read or write. The maximum size allowed for the data block is 32 bytes;
- I2C_SMBUS_PROC_CALL: to perform a process call, that is a Write Word followed by a Read Word. In this case the `read_write` field value will be ignored by the driver, but however if you set it with a value different from I2C_SMBUS_READ or I2C_SMBUS_WRITE, the `ioctl` will fail;
- I2C_SMBUS_BLOCK_PROC_CALL: to perform a process call using a 32 byte block of data. If M is the number of bytes wrote in the process call and N is the number of bytes read, M+N must be minor or equal to 32;
- I2C_SMBUS_I2C_BLOCK, I2C_SMBUS_I2C_BLOCK_BROKEN: to perform "I²C block" read or write, depending on the `read_write` field value. The use of the first option or the "broken" version depends on your preferences, because in any case the driver will convert your call into the I2C_SMBUS_I2C_BLOCK version.

If the `size` field is set with a value different from the ones listed above, the `ioctl` call will fail.

For further information about the SMBus data formats and the specification of all the available commands, see the SMBus Documentation.

Finally, the `*data` field is a pointer to the data you want to send or receive through the SMBus transfer.

In the following example, we want to write a byte of data to the slave device:

```
struct i2c_smbus_ioctl_data mycmd;
union i2c_smbus_data tmp;

mycmd.read_write = I2C_SMBUS_WRITE;
mycmd.command = 0xF7;
mycmd.size = I2C_SMBUS_BYTE_DATA;
tmp.byte = 0xA0;
mycmd.data = &tmp;
ret = ioctl(f, I2C_SMBUS, &mycmd);
if (ret < 0)
    printf("An error occurred in ioctl");
```



- I2C_RETRIES

Sets the number of times a device address should be polled when not acknowledging.
Example:

```
unsigned long num_retries = 4;
res = ioctl(file_descriptor, I2C_RETRIES, num_retries);
```

- I2C_TIMEOUT

With this macro you can set the timeout value in jiffies.
Example:

```
unsigned long jiffies = 100;
res = ioctl(file_descriptor, I2C_TIMEOUT, jiffies);
```

4.2.4. read()

The *read()* function reads *nbyte* bytes from the file associated with the open file descriptor, *fildes*, and copies them in the buffer that is pointed to by *buf*.

Header file:
unistd.h

Prototype:
ssize_t read(int fildes, void *buf, size_t nbyte)

Parameters:
fildes – file descriptor
buf – destination buffer pointer
nbyte – number of bytes that read() attempts to read

Returns:
The number of bytes actually read if if operation is completed successfully, otherwise it is -1.

Example:
Read *sizeof(read_buff)* bytes from the file associated with *fd* and stores them into *read_buff*.

```
char read_buff[BUFF_LEN];

if(read(fd, read_buff, sizeof(read_buff)) < 0)
```



```
{
    /* Error Management Routine */
} else {
    /* Value Read */
}
```

4.2.5. write()

The *write()* function writes *nbyte* bytes from the buffer that are pointed by *buf* to the file associated with the open file descriptor, *fildes*.

Header file:
unistd.h

Prototype:
ssize_t write(int fildes, const void *buf, size_t nbyte)

Parameters:
fildes – file descriptor
buf – destination buffer pointer
nbyte – number of bytes that write() attempts to write

Returns:
The number of bytes actually written if operation is completed successfully (this number shall never be greater than *nbyte*), otherwise it is -1.

Example:
Write *strlen(value_to_be_written)* bytes from the buffer pointed by *value_to_be_written* to the file associated with the open file descriptor, *fd*.

```
char value_to_be_written[] = "dummy_write";

if (write(fd, value_to_be_written, strlen(value_to_be_written))
    < 0)
{
    /* Error Management Routine */
} else {
    /* Value Written */
}
```

4.2.6. close()

The *close()* function deallocates the file descriptor indicated by *fildes*. To deallocate means to make the file descriptor available for subsequent calls to *open()* or other functions that allocate file descriptors.



Header file:

unistd.h

Prototype:

int close(int fildes)

Parameters:

fildes – file descriptor

Returns:

0 if operation is completed successfully, otherwise it is -1

Example:

Close the I²C device.

```
if(close(fd) < 0)
{
    /* Error Management Routine */
} else {
    /* File Closed */
}
```

4.2.7. A Test Program

The following simple C program is useful to test the I²C interface. It opens the /dev/i2c interface and calls the write function in an infinite loop to write some random values on the device.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <linux/i2c.h>
#include <linux/i2c-dev.h>
```

```
int main()
{
    int fd;
    char val;
```



```

char buf[5];
int addr = 0x40; /* This I2C address is a random value
because no device is attached
on the bus; if you connect a device,
change this value with the correct one*/
fd = open("/dev/i2c-0",O_RDWR);
if (fd < 0) {
    fprintf(stderr, "Error during open\n");
    exit(1);
}

/* The following ioctl call sets in slave mode
the device located at the addr address */
if (ioctl(fd, I2C_SLAVE, addr) < 0) {
    fprintf(stderr, "Error during IOCTL\n");
    exit(1);
}
buf[0] = 'T';
buf[1] = 'e';
buf[2] = 's';
buf[3] = 't';

for(val=1;val=(val+1)%255) {
    buf[4] = val;
    if (write(fd, buf, 5) != 5) {
        fprintf(stderr, "Write error
(%d)\n",val);
    } else {
        fprintf(stderr, "Write OK (%d)\n",val);
    }
    usleep(500000);
}
return 0;
}

```

If you have some I²C device connected to the board, you'll have to set the *addr* variable with the correct address of the I²C device you want to write. Otherwise, you can initialize that variable with a random value. In this case, you can connect an oscilloscope to the SDA and SCL pins of the board to observe the behaviour of the signal on the pins.



If you connect an I²C device to the board the write call should be successful and the program should report "Write OK". Otherwise the program displays "Write error", but this is not a problem: it only means that there is no device connected on the bus.

4.3. SPI

The "Serial Peripheral Interface" (SPI) is a synchronous four wire serial link used to connect microcontrollers to sensors, memory, and peripherals.

The Serial Peripheral Interface is essentially a shift register that serially transmits data bits to other SPIs. During a data transfer, one SPI system acts as the "master" and controls the data flow, while the other devices act as "slaves".

The SPI system consists of two data lines and two control lines:

- Master Out Slave In (MOSI): This data line supplies the output data from the master shifted into the input(s) of the slave(s).
- Master In Slave Out (MISO): This data line supplies the output data from a slave to the input of the master. There may be no more than one slave transmitting data during any particular transfer.
- Serial Clock (SPCK): This control line is driven by the master and regulates the flow of the data bits. The master may transmit data at a variety of baud rates; the SPCK line cycles once for each bit that is transmitted.
- Slave Select (NSS): This control line allows slaves to be turned on and off by hardware (it is also called chipselect).

4.3.1. Loading the SPI module

The SPI driver is released to customer under in the form of a loadable module.

To load the SPI module type in the terminal:

```
# modprobe spidev
```

When loaded, the SPI driver will install four new devices named `spidev1.X`, under the `/dev/` directory. The first number in the "1.X" name extension represents the number of the selected SPI bus while the second represents the selected chipselect (from 0 to 3).

Once installed the SPI device can be used as a normal character device and can be accessed by any application running in userspace.



4.3.2. open()

The *open()* function shall establish the connection between a file and a file descriptor. The file descriptor is used by other I/O functions to refer to the opened file.

Header file:

fcntl.h

Prototype:

int open(const char *pathname, int flags)

Parameters:

pathname – file name with its own path

flags – is an *int* specifying file opening mode: is one of O_RDONLY, O_WRONLY or O_RDWR which request opening the file read-only, write-only or read/write, respectively

Returns:

The new file descriptor *fildev* if operation is completed successfully, otherwise it is -1.

Example:

Open the /dev/spidev1.1.

```
int fd; // file descriptor for the /dev/spidev1.1 entry

if((fd = open("/dev/spidev1.1", O_RDWR) < 0)
{
    /* Error Management Routine */
} else {
    /* SPI Device Opened */
}
```

4.3.3. ioctl()

The *ioctl()* function manipulates the underlying device parameters. In particular, many operating characteristics can be controlled with *ioctl()* requests.

Header file:

sys/ioctl.h
linux/spi/spidev.h

Prototype:

int ioctl(int fildev, int request, ...)



Parameters:

fildev – file descriptor

request – device-dependent request code.

The following ioctl request codes can be used for SPI:

- SPI_IOC_RD_MODE and SPI_IOC_WR_MODE to get/set the SPI mode
- SPI_IOC_RD_BITS_PER_WORD and SPI_IOC_WR_BITS_PER_WORD to get/set the number of bits per words exchanged
- SPI_IOC_RD_MAX_SPEED_HZ and SPI_IOC_WR_MAX_SPEED_HZ to get/set the maximum allowed speed
- SPI_IOC_MESSAGE to exchange data in full duplex mode

The third argument is a *void ** and depends on the ioctl request code used: see the examples below.

Returns:

0 if operation is completed successfully, otherwise it is -1.

Examples:

- SPI_IOC_RD_MODE and SPI_IOC_WR_MODE

Get (RD) or set (WR) the SPI mode. The third parameter is a pointer to a byte which will return (RD) or assign (WR) the SPI transfer mode.

The constant values defined in the “spidev.h” file will be used to set the SPI mode.

Please note that SPI_CS_HIGH, SPI_CPHA, SPI_CPOL and SPI_MODE_0..SPI_MODE_3 are the only constant values, listed in the “spidev.h” file, supported by the current SPI driver.

Example:

```
uint8_t mode;
mode = SPI_CS_HIGH|SPI_CPOL|SPI_CPHA; /* Beware! this is just an
example. Check your device before setting these fields*/
ret = ioctl(fd, SPI_IOC_WR_MODE, &mode);
if (ret == -1)
    printf("can't set SPI mode");
```

- SPI_IOC_RD_BITS_PER_WORD and SPI_IOC_WR_BITS_PER_WORD

Get (RD) or set (WR) the number of bits per word exchanged. The third parameter is a pointer to a byte which will return (RD) or assign (WR) the number of bits in each SPI transfer word.

Example:

```
uint8_t bits;
```



```
ret = ioctl(fd, SPI_IOC_WR_BITS_PER_WORD, &bits);
if (ret == -1)
    printf("can't set bits per word");
```

- SPI_IOC_RD_MAX_SPEED_HZ and SPI_IOC_WR_MAX_SPEED_HZ

Get (RD) or set (WR) the SPI max speed. The third parameter is a pointer to a uint32_t which will return (RD) or assign (WR) the maximum SPI transfer speed, in Hz. It is not required that the controller assigns specific clock speed.

Example:

```
uint32_t speed;
ret = ioctl(fd, SPI_IOC_WR_MAX_SPEED_HZ, &speed);
if (ret == -1)
    printf("can't set max speed hz");
```

- SPI_IOC_MESSAGE

Standard read() and write() operations are obviously only half-duplex, and the chipselect is deactivated between those operations. Full-duplex access, and composite operation without chipselect de-activation, is available using the SPI_IOC_MESSAGE(N) request.

Please note that the SPI_IOC_MESSAGE(N) request needs, as parameter, a pointer to struct spi_ioc_transfer whose fields speed_hz and bits_per_word must be set to 0.

Example:

```
uint8_t tx[] = {
    0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
    0x40, 0x00, 0x00, 0x00, 0x00, 0x95,
    0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
    0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
    0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
    0xDE, 0xAD, 0xBE, 0xEF, 0xBA, 0xAD,
    0xF0, 0x0D,
};
uint8_t rx[ARRAY_SIZE(tx)] = {0, };
struct spi_ioc_transfer tr = {
    .tx_buf = (unsigned long)tx,
    .rx_buf = (unsigned long)rx,
    .len = ARRAY_SIZE(tx),
    .delay_usecs = delay, /*Delay before chipselect
deactivation*/
```



4.3.5. write()

The *write()* function writes *nbyte* bytes from the buffer that are pointed by *buf* to the file associated with the open file descriptor, *fildes*.

Header file:

unistd.h

Prototype:

```
ssize_t write(int fildes, const void *buf, size_t nbyte)
```

Parameters:

fildes – file descriptor

buf – destination buffer pointer

nbyte – number of bytes that *write()* attempts to write

Returns:

The number of bytes actually written if operation is completed successfully, (this number shall never be greater than *nbyte*), otherwise it is -1

Example:

Write *strlen(value_to_be_written)* bytes from the buffer pointed by *value_to_be_written* to the file associated with the open file descriptor, *fd*.

```
char value_to_be_written[] = "dummy_write";

if (write(fd, value_to_be_written, strlen(value_to_be_written))
    < 0)
{
    /* Error Management Routine */
} else {
    /* Value Written */
}
```

4.3.6. close()

The *close()* function deallocates the file descriptor indicated by *fildes*. To deallocate means to make the file descriptor available for subsequent calls to *open()* or other functions that allocate file descriptors.

Header file:

unistd.h

Prototype:



```
int close(int fildes)
```

Parameters:

fildes – file descriptor

Returns:

0 if operation is completed successfully, otherwise it is -1

Example:

Close the SPI device.

```
if(close(fd) < 0)
{
    /* Error Management Routine */
} else {
    /* File Closed */
}
```

4.3.7. A Test Program

Below it is reported a simple piece of code that opens the device, writes some random data, reads some data if available and then closes the devices. Please note that the following code works for half duplex communication. For typical full duplex communication the SPI_IOC_MESSAGE ioctl will be used.

In order to receive data when executing the read() function a transmitting SPI peripheral must be connected.

```
#include <stdint.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <getopt.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <linux/types.h>
#include <linux/spi/spidev.h>

#define ARRAY_SIZE(a) (sizeof(a) / sizeof((a)[0]))
static const char *device = "/dev/spidev1.1";

static void myTransfer(int fd)
{
    int ret;
    int i = 0;
    uint8_t tx[] = {
        0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
```



```

        0x40, 0x00, 0x00, 0x00, 0x00, 0x95,
        0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
        0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
        0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
        0xDE, 0xAD, 0xBE, 0xEF, 0xBA, 0xAD,
        0xF0, 0x0D,
};
uint8_t rx[ARRAY_SIZE(tx)] = {0, };
if((write(fd, tx, ARRAY_SIZE(tx))!=ARRAY_SIZE(tx))
{
    printf("Write Failed.\n");
}
else
{
    printf("\nTransmitted Buffer.\n");
    for(i=0; i < ARRAY_SIZE(tx); i++)
    {
        printf("0x%x,",tx[i]);
        if((!(i%10)) &&(i!=0))
        {
            printf("\n");
        }
    }
}
if((read(fd, rx, ARRAY_SIZE(tx))!=ARRAY_SIZE(tx))
{
    printf("\nRead Failed.\n");
}
else
{
    printf("\nReceived Buffer.\n");
    for(i=0; i < ARRAY_SIZE(tx); i++)
    {
        printf("0x%x,",rx[i]);
        if((!(i%10)) &&(i!=0))
        {
            printf("\n");
        }
    }
}

return;
}

int main(int argc, char *argv[])
{
    int ret = 0;
    int fd;

```



```
fd = open(device, O_RDWR);  
if (fd < 0)  
{  
    printf("can't open device");  
    return -1;  
}  
myTransfer(fd);  
  
close(fd);  
  
return ret;  
}
```

4.4. GPIO

The GPIO driver creates a series of devices under the `/dev/` directory. Each device is named as `at91sam9260_gpio.<pin>`.

GPIO pins are numbered from 0 to 95. The first 32 pins refer to the PIO A controller, the second 32 pins refer to the PIO B controller and the others refer to the PIO C controller. For further information on GPIO pins (numbering, availability, etc.) please refer to [6] (paragraph 4.17) and [1] (paragraph 3.3).

GPIO pins can be read and written thorough simple read/write operations.

The read operation returns the status and the level of the pin if it is configured as a GPIO, otherwise it fails. The write operation can change the configuration and output value. The configuration can be changed using:

- for output enable with pull up
- for output enable without pull up
- l input enabled with pull up
- i input enabled without pull up
- 1 set the level up
- 0 set the level low

To test a gpio you can use the shell, for example (gpio number 92):

```
# echo 01 > /dev/at91sam9260_gpio.92
```

to raise up the signal.

The following subparagraphs show all the functions that can be used from C source code to perform read/write operations onto GPIOs pins.



4.4.1. Loading the GPIO module

Currently the GPIO is compiled as a module (*at91sam9260_gpio.ko*); it is automatically loaded at startup using the script *S02* placed in the folder */etc/init.d/*. The user can unload this module typing the following commands:

```
# rmmod at91sam9260_gpio
# rmmod atmel_gpio
```

To load the module the user shall type:

```
# modprobe at91sam9260_gpio
```

4.4.2. open()

The *open()* function establishes the connection between a file and a file descriptor. The file descriptor is used by other I/O functions to refer to the opened file.

Header file:

fcntl.h

Prototype:

```
int open(const char *pathname, int flags)
```

Parameters:

pathname – file name with its own path

flags – is an *int* specifying file opening mode: is one of O_RDONLY, O_WRONLY or O_RDWR which request opening the file read-only, write-only or read/write, respectively

Returns:

The new file descriptor *fildev* if operation is completed successfully, otherwise it is -1.

Example:

Open the GPIO device (for example number 92).

```
int fd; // file descriptor for GPIO /dev entries

if((fd = open("/dev/at91sam9260_gpio.92", O_RDWR)) < 0)
{
    /* Error Management Routine */
} else {
    /* Gpio Opened */
}
```

4.4.3. read()

The *read()* function reads *nbyte* bytes from the file associated with the open file descriptor, *fildev*, and copies them in the buffer that is pointed to by *buf*.



Header file:

unistd.h

Prototype:

ssize_t read(int fildes, void *buf, size_t nbyte)

Parameters:

fildes – file descriptor

buf – destination buffer pointer

nbyte – number of bytes that read() attempts to read

Returns:

The number of bytes actually read if operation is completed successfully, otherwise it is -1

Example:

Read *sizeof(read_buff)* bytes from the file associated with *fd* and stores them into *read_buff*.

```
char read_buff[BUFF_LEN];

if(read(fd, read_buff, sizeof(read_buff)) < 0)
{
    /* Error Management Routine */
} else {
    /* Value Read */
}
```

4.4.4. write()

The *write()* function writes *nbyte* bytes from the buffer that are pointed by *buf* to the file associated with the open file descriptor, *fildes*.

Header file:

unistd.h

Prototype:

ssize_t write(int fildes, const void *buf, size_t nbyte)

Parameters:

fildes – file descriptor

buf – destination buffer pointer

nbyte – number of bytes that write() attempts to write



Returns:

The number of bytes actually written if operation is completed successfully (this number shall never be greater than *nbyte*), otherwise it is -1.

Example:

Write *strlen(value_to_be_written)* bytes from the buffer pointed by *value_to_be_written* to the file associated with the open file descriptor, *fd*.

```
char value_to_be_written[] = "01";

if (write(fd, value_to_be_written, strlen(value_to_be_written))
    < 0)
{
    /* Error Management Routine */
} else {
    /* Value Written */
}
```

4.4.5. close()

The *close()* function deallocates the file descriptor indicated by *filides*. To deallocate means to make the file descriptor available for subsequent calls to *open()* or other functions that allocate file descriptors.

Header file:

unistd.h

Prototype:

int close(int filides)

Parameters:

filides – file descriptor

Returns:

0 if operation is completed successfully, otherwise it is -1.

Example:

Close the GPIO device.

```
if(close(fd) < 0)
{
    /* Error Management Routine */
} else {
    /* File Closed */
}
```



4.5. Ge863pro3_GPIO

Currently, the Linux kernel has a GPIO driver loaded by default as module. The loaded GPIO module, described in the previous paragraph, does not support interrupt management. When gpio interrupt handling is needed, the default GPIO module shall be unloaded typing the following commands:

```
# rmmmod at91sam9260_gpio
# rmmmod atmel_gpio
```

and ge863pro3_gpio module shall be loaded as described in the chapter 4.5.2.

With the ge863pro3_gpio driver it is possible to configure each GPIO in three different ways:

- No interrupt
- Continuous interrupt
- Interrupt notified by read function

By default the module starts with the “No interrupt” behavior. When enabling GPIO interrupt it is possible to associate a different function (interrupt routine) to each GPIO. Please note that if the user needs a specific function he shall modify the driver source code by adding his personal routine. The user shall the recompile and reload the module.

Enabling or disabling GPIO interrupt it changes the behavior of other system calls as *read* and *write*.

4.5.1. Interrupt description

The interrupt management allows two different behaviors:

1. Continuous interrupt
2. Interrupt notified by read function

With “continuous interrupt” the driver manages each interrupt incrementing a counter. When the user wants to get the counter value he shall use the GPIO_IOCTL_RET_VAL ioctl (as explained in the following paragraphs 4.5.4).

With interrupt notified by read function, the user shall call the read function every time he is waiting for an interrupt. When the interrupt occurs the thread waiting on the read function is awoken.

4.5.2. Loading the GPIO module

As already stated the ge863pro3_gpio driver is released to customer under the shape of a loadable module.



4.5.4. ioctl()

The `ioctl()` function manipulates the underlying device parameters. In particular, many operating characteristics can be controlled with `ioctl()` requests.

Header file:

`sys/ioctl.h`
`linux/gpio/ge863pro3_gpio_ioctl.h`

Prototype:

`int ioctl(int fildes, int request, ...)`

Parameters:

`fildes` – file descriptor
`request` – device-dependent request code.

The following `ioctl`s request codes can be used for SSC:

- `GPIO_IOCTL_EN_IRQ_CONTINUOUS` enables the continuous interrupt behavior
- `GPIO_IOCTL_EN_IRQ_BY_READ` enables the interrupt managed by the read system call
- `GPIO_IOCTL_DIS_IRQ` disables the interrupt
- `GPIO_IOCTL_RET_VAL`, if continuous interrupt is enabled, it allows the user to read data managed by interrupt routine
- `GPIO_IOCTL_WAIT_READ`, if the interrupt is associated to the read system call, sets the wait timeout of the read function.

The third argument is a *void ** and depends on the `ioctl` request code used: see the examples below.

Returns:

0 if operation is completed successfully, otherwise it is -1.

Examples:

- `GPIO_IOCTL_EN_IRQ_CONTINUOUS`

This `ioctl` starts the `gpio` interrupt routine after setting the specific `gpio` as input `gpio` and enabling deglitch filter. The interrupt occurs both in raising and falling edge. Using this configuration the interrupt routine updates a counter value every time an interrupt occurs. To read the value of the counter the `GPIO_IOCTL_RET_VAL` `ioctl` shall be used.

Example:

```
if (ioctl(fd, GPIO_IOCTL_EN_IRQ_CONTINUOUS, NULL))
{
```



```
    printf("Error GPIO_IOCTL_EN_IRQ_CONTINUOUS\n");
    return -1;
}
```

- **GPIO_IOCTL_EN_IRQ_BY_READ**

This ioctl starts the gpio interrupt routine after setting the specific gpio as input gpio and enabling deglitch filter. The interrupt occurs both in raising and falling edge. When the user calls the read system call, the function wait until an interrupt occurs or the timeout expires; the timeout is set using the GPIO_IOCTL_WAIT_READ, by default the timeout is "wait forever".

Example:

```
if (ioctl(fd, GPIO_IOCTL_EN_IRQ_BY_READ, NULL))
{
    printf("Error GPIO_IOCTL_EN_IRQ_BY_READ\n");
    return -1;
}
```

- **GPIO_IOCTL_DIS_IRQ**

By default the system has the interrupt disabled. To return to the default status the following ioctl shall be used.

Example:

```
if (ioctl(fd, GPIO_IOCTL_DIS_IRQ, NULL))
{
    printf("disableGpioInterrupt - Error\n");
    return -1;
}
```

- **GPIO_IOCTL_RET_VAL**

This ioctl is used to retrieve the interrupt counter value when the driver is configured in "continuous interrupt" mode. In all the other cases the ioctl returns an error code.

Example:

```
int val;

if (ioctl(fd, GPIO_IOCTL_RET_VAL, &val))
{
    printf("GPIO_IOCTL_RET_VAL Error\n");
    return -1;
}
```

- **GPIO_IOCTL_WAIT_READ**

This ioctl is used to configure the wait timeout associated to the read function when the driver is configured in "interrupt notified by read function" mode. In all the other cases the ioctl returns an error code.



Example:

```
long time;

time = 1000; /*ms*/
if (ioctl(fd, GPIO_IOCTL_WAIT_READ, time))
{
    printf("Error GPIO_IOCTL_WAIT_READ\n");
    return -1;
}
```

4.5.5. read()

The `read()` function reads *nbyte* bytes from the file associated with the open file descriptor, *fildev*, and copies them in the buffer that is pointed to by *buf*.

In this particular case the read system call has two different behaviors depending on the driver configuration. When the driver is configured either in “No Interrupts” or in “Continuous interrupt” mode the read function returns the configuration of GPIO: Input, Output, zero, one, pull up, pull down. If the pin is configured as peripheral the read function returns -1.

Characters returned into the buffer by the read function:

- ‘0’ = output with pull up
- ‘o’ = output without pull up
- ‘l’ = input with pull up
- ‘i’ = input without pull up
- ‘1’ = set the level up
- ‘0’ = set the level down

Example:

‘01’ = GPIO configured as output with pull up and level up

When the driver is configured in “interrupt notified by the read function” mode, the read function returns the interrupt counter value, but if the timeout expires it will return an error code.

Header file:

unistd.h

Prototype:

ssize_t read(int fildev, void *buf, size_t nbyte);

Parameters:

fildev – file descriptor

buf – destination buffer pointer

nbyte – number of bytes that read() attempts to read



Returns:

The number of bytes actually read if operation is completed successfully, otherwise it is -1.

Example:

Read *sizeof(val)* bytes from the file associated with *fd* and stores them in *val*.

```
int val;

if(read(fd, &val, sizeof(val)) < 0)
{
    /* Error Management Routine */
} else {
    /* Value Read */
}
```

4.5.6. write()

The *write()* function writes *nbyte* bytes from the buffer that are pointed by *buf* to the file associated with the open file descriptor, *filides*. In this particular case the use of this system call is allowed only if the driver is configured in “No interrupt” mode. The user can configure the GPIO as input/output, the GPIO value one (up) or zero (down) and pull up or not pull up.

Characters to put into the buffer to configure the GPIO:

- '0' = output with pull up
- 'o' = output without pull up
- 'l' = input with pull up
- 'i' = input without pull up
- '1' = set the level up
- '0' = set the level down

Header file:

unistd.h

Prototype:

ssize_t write(int filides, const void *buf, size_t nbyte);

Parameters:

filides – file descriptor
buf – destination buffer pointer
nbyte – number of bytes that write() attempts to write

Returns:

The number of bytes actually written if operation is completed successfully, (this number shall never be greater than *nbyte*), otherwise it is -1



Example:

Write *strlen(value_to_be_written)* bytes from the buffer pointed by *value_to_be_written* to the file associated with the open file descriptor, *fd*.

```
char value_to_be_written[] = "01"; /*output - one - pullup*/

if (write(fd, value_to_be_written, strlen(value_to_be_written))
    < 0)
{
    /* Error Management Routine */
} else {
    /* Value Written */
}
```

4.5.7. Interrupt routine customization

Currently the interrupt routine code is the following:

```
static irqreturn_t ge863pro3_handler(int irq, void *dev_id)
{
    int retval = IRQ_NONE;
    struct ge863pro3_gpio_irq *irq_gpio=(struct ge863pro3_gpio_irq*)
    dev_id;

    spin_lock(&ge863pro3_gpio_spinlock);
    if(irq_gpio->is_irq_enable == GPIO_EN_IRQ_CONTINUOUS)
    {
        irq_gpio->counter+=1;
        retval = IRQ_HANDLED;
    }
    else if(irq_gpio->is_irq_enable == GPIO_EN_IRQ_BY_READ)
    {
        irq_gpio->counter+=1;
        complete(&irq_gpio->irq_completion);
        retval = IRQ_HANDLED;
    }
    spin_unlock(&ge863pro3_gpio_spinlock);

    return retval;
}
```

When the user needs a specific behavior of the interrupt routine, he shall change the above code. He shall pay attention to the two “if-else” scopes. The first is executed when the driver is set in “Continuous interrupt” mode, the second one is executed when the driver is configured in “Interrupt notified by read function” mode.



In the specific case of “Interrupt notified by read function” mode, the user modification shall keep the code line `complete(&irq_gpio->irq_completion)` at the end of the interrupt routine.

This function awakes the read system call previously suspended.

If the user needs to manage a most complex data structure, in our case is a simple integer counter, he shall change the data structure `ge863pro3_gpio_irq`.

4.5.8. close()

The `close()` function deallocates the file descriptor indicated by `filides`. To deallocate means to make the file descriptor available for subsequent calls to `open()` or other functions that allocate file descriptors.

Header file:

unistd.h

Prototype:

int close(int filides);

Parameters:

filides – file descriptor

Returns:

0 if operation is completed successfully, otherwise it is -1

Example:

Close the GPIO device.

```
if(close(fd) < 0)
{
    /* Error Management Routine */
} else {
    /* File Closed */
}
```

4.5.9. A Test Program

Below it is reported a simple piece of code that first opens the device and then reads some data.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
```



```
#include <unistd.h>

#include <linux/gpio/ge863pro3_gpio_ioctl.h>

int main(int argc, char *argv[])
{
    int i;
    int val;
    int fd;
    long time;

    fd = open("/dev/ge863pro3_gpio.66",O_RDWR);
    if(fd == NULL)
    {
        printf("ERROR OPEN\n");
        return -1;
    }

    if (ioctl(fd, GPIO_IOCTL_EN_IRQ_BY_READ, NULL))
    {
        printf("Error GPIO_IOCTL_EN_IRQ_BY_READ\n");
        return -1;
    }

    time = 10000; /*10 sec of timeout*/

    if (ioctl(fd, GPIO_IOCTL_WAIT_READ, time))
    {
        printf("Error GPIO_IOCTL_WAIT_READ\n");
        return -1;
    }

    i = 0;

    while(i<10)
    {
        if(read(fd,&val, sizeof(int))<0)
        {
            printf("Error in read\n");
        }
        else
        {
            printf("Read : %d\n", val);
        }

        i++;
    }
}
```



```
close(fd);
return 0;
}
```

4.6. ADC

The GE863-PRO³ is provided with a 10-bit Analog-to-Digital Converter (ADC) based on a Successive Approximation Register (SAR). It makes possible the conversion of up to 4 analog lines, because it integrates a 4-to-1 analog multiplexer.

The user can configure many ADC features, for example: 8-bit or 10-bit resolution mode; Normal or Sleep Mode to save power consumption, Sample and Hold Time, Startup Time, and other explained later.

In this technical note we'll see how to use the Telit ADC Driver and the options it provides.

4.6.1. Loading the ADC Module

To use the ADC device through the `/dev` interface of the Linux Operating System, you have to load the `adc_driver.ko` module.

To do this, select the folder containing that file and type
`insmod ./adc_driver.ko`

After the loading, in the `/dev` folder will be created one device called `adc`. The ADC default settings after the loading are: 10-bit resolution mode, normal mode, `PRESCAL=0x9`, `STARTUP=0x7`, and `SHTIM=0x3`.

4.6.2. `open()`

The `open()` function shall establish the connection between a file and a file descriptor. The file descriptor is used by other I/O functions to refer to that file.

Header file:
`fcntl.h`

Prototype:
`int open(const char *pathname, int flags)`

Parameters:
`pathname` – file name with its own path



flags – is an *int* specifying file opening mode: is one of O_RDONLY, O_WRONLY or O_RDWR which request opening the file read-only, write-only or read/write, respectively

Returns:

The new file descriptor *fildev* if successful, -1 otherwise

Example:

Open the /dev/adc.

```
int fd; // file descriptor for the /dev/adc entry

if((fd = open("/dev/adc", O_RDONLY) < 0)
{
    /* Error Management Routine */
} else {
    /* ADC Device Opened */
}
```

4.6.3. ioctl()

The ioctl() function manipulates the underlying device parameters of special files. In particular, many operating characteristics of character special files may be controlled with ioctl() requests.

Header file:

sys/ioctl.h
linux/adc/adc.h

Prototype:

int ioctl(int fildev, int request, ...)

Parameters:

fildev – file descriptor

request – device-dependent request code.

The following ioctls request codes can be used for ADC:

- ADC_SET_RESOLUTION, ADC_GET_RESOLUTION: to set the ADC resolution mode
- ADC_SET_TYPE, ADC_GET_TYPE: to set/get the interrupt or the trigger mode
- ADC_SET_MAX_WAIT, ADC_GET_MAX_WAIT: to set/get the maximum wait for an interrupt
- ADC_ECHLS: to enable the chosen ADC lines
- ADC_DCHLS: to disable the chosen ADC lines



- ADC_SCHLS: to know the currently enabled ADC lines
- ADC_SET_MODE, ADC_GET_MODE: to set Sleep/Normal mode and get the status
- ADC_SET_STARTUP, ADC_GET_STARTUP: to set/get the startup time
- ADC_SET_SHTIM, ADC_GET_SHTIM: to set/get the Sample and Hold time
- ADC_SET_PRESCAL, ADC_GET_PRESCAL: to set/get the prescaler value.
- ADC_RESET: to reset the ADC

The third argument is a *void ** and depends on the ioctl request code used: see the examples below.

Returns:

0 if successful, -1 otherwise

Examples:

- ADC_SET_RESOLUTION, ADC_GET_RESOLUTION

The ADC_SET_RESOLUTION command sets the resolution of the digital result of the analog to digital conversion. The minimum digital value available after the conversion is always 0, and corresponds to an applied analog voltage of 0V. The maximum digital value available is 255 if you set the 8-bit resolution or 1023 if you set the 10-bit resolution. The maximum value is obtained with an applied analog voltage of 3.1V. If you want to use the 8-bit mode, type:

```
if (ioctl(fd, ADC_SET_RESOLUTION, ADC_8_BIT) < 0) {
    fprintf(stderr, "Error during IOCTL\n");
}
```

The default value for ADC_SET_RESOLUTION is the 10-bit mode, so if you want to use this mode you don't need to do anything. But, if you previously set the 8-bit mode, you may need to set the 10-bit mode. To do this, type:

```
if (ioctl(fd, ADC_SET_RESOLUTION, ADC_10_BIT) < 0) {
    fprintf(stderr, "Error during IOCTL\n");
}
```

The ADC_GET_RESOLUTION command returns the current ADC resolution mode in the status variable. An example of use is:

```
unsigned long status;
if (ioctl(fd, ADC_GET_RESOLUTION, &status) < 0) {
    fprintf(stderr, "Error during IOCTL\n");
}
```



```

}
else
{
    if(status== ADC_10_BIT)
    printf("Current status: 10 bit resolution\n");
    if(status== ADC_8_BIT)
    printf("Current status: 8 bit resolution\n");
}

```

- ADC_SET_TYPE, ADC_GET_TYPE

The ADC_SET_TYPE command allows the user to select the functionality he needs between the interrupt one and the trigger one. The interrupt functionality is useful if the user wants to perform only one analog to digital conversion when he makes a read call. The trigger functionality, instead, performs periodic and continuous conversions; so the user, with a read call, can receive the last converted values. By default, the driver is in interrupt mode.

IMPORTANT: the ioctl with the ADC_SET_TYPE command must always be called BEFORE the ioctl with the ADC_SET_CLOCK command.

If you want to set the interrupt mode, type:

```

if (ioctl(fd, ADC_SET_TYPE, ADC_INTERRUPT) < 0) {
    fprintf(stderr, "Error during IOCTL\n");
}

```

For the trigger mode, you need to use the external Timer Counter that will perform periodic trigger. So you must also load the Timer Counter driver and do its settings (see the Timer Counter chapter in this guide). Each trigger will start a new conversion. You can choose one among three Timer Counter devices (through the arguments ADC_TRIGGER_TC0, ADC_TRIGGER_TC1 and ADC_TRIGGER_TC2). For example, to set the trigger mode and use the Timer Counter 1 to generate the periodic triggers, type:

```

if (ioctl(fd, ADC_SET_TYPE, ADC_TRIGGER_TC1) < 0) {
    fprintf(stderr, "Error during IOCTL\n");
}

```

The ADC_GET_TYPE command returns the current ADC mode in the status variable. An example of use is:

```

unsigned long status;
if (ioctl(fd, ADC_GET_TYPE, &status) < 0) {
    fprintf(stderr, "Error during IOCTL\n");
}

```



```

}
else
{
    if(status== ADC_INTERRUPT)
    printf("ADC is in Interrupt Mode\n");
    else
    printf("ADC is in Trigger Mode \n");
}

```

- ADC_SET_MAX_WAIT, ADC_GET_MAX_WAIT

The ADC_SET_MAX_WAIT command allows the user to set the maximum wait for an interrupt in milliseconds, when the ADC is in Interrupt Mode. In fact, when a read call is performed in Interrupt Mode, the driver starts the analog to digital conversion and then waits for an interrupt from the ADC that indicates the end of the conversion. By default, the wait value is ADC_MAX_TIMEOUT that performs a “wait for ever”. To set a different value, use an ioctl call like the following, where we set a maximum wait time of five seconds:

```

if (ioctl(fd, ADC_SET_MAX_WAIT, 5000) < 0) {
    fprintf(stderr, "Error during IOCTL\n");
}

```

The ADC_GET_MAX_WAIT command returns the current maximum wait value. An example of use is:

```

unsigned long status;
if (ioctl(fd, ADC_GET_MAX_WAIT, &status) < 0) {
    fprintf(stderr, "Error during IOCTL\n");
}
else
{
    if(status == ADC_MAX_TIMEOUT)
    printf("The driver will wait for ever if it doesn't receive
an interrupt\n");
    else
    printf("The max wait value is %d \n", status);
}

```

- ADC_ECHLS



The ADC_ECHLS command allows the user to enable only the ADC lines he needs. You can choose among enabling ADC_CH0, ADC_CH1, ADC_CH2 or ADC_CH3. For example, to enable the line one (ADC_CH1) and the line three (ADC_CH3), type:

```
if (ioctl(fd, ADC_ECHLS, ADC_CH1 | ADC_CH3) < 0) {
    fprintf(stderr, "Error during IOCTL\n");
}
```

- ADC_DCHLS

The ADC_DCHLS command allows the user to disable only the ADC lines he needs. By default, all the lines are disabled. For example, if you have previously enabled the line 2, you can disable it typing:

```
if (ioctl(fd, ADC_DCHLS, ADC_CH2) < 0) {
    fprintf(stderr, "Error during IOCTL\n");
}
```

Obviously, if you try to disable a line already disabled you won't have any effect.

- ADC_SCHLS

The ADC_SCHLS command returns in the status variable the currently enabled ADC lines. To see if the channel x is enabled, you have to AND the status variable with the ADC_CHx macro. An example of use is:

```
unsigned long status;
if (ioctl(fd, ADC_SCHLS, &status) < 0) {
    fprintf(stderr, "Error during IOCTL\n");
}
else
{
    if(status & ADC_CH2)
        printf("The channel 2 is enabled\n");
    else
        printf("The channel 2 is disabled\n");
}
```

- ADC_SET_MODE, ADC_GET_MODE



The ADC_SET_MODE command allows the user to set the Sleep Mode in the ADC. This mode is useful to maximize power saving. By default, the ADC is in Normal Mode. To enable the Sleep Mode, type:

```
if (ioctl(fd, ADC_SET_MODE, SLEEP) < 0) {  
    fprintf(stderr, "Error during IOCTL\n");  
}
```

To disable the Sleep Mode, type

```
if (ioctl(fd, ADC_SET_MODE, NORMAL) < 0) {  
    fprintf(stderr, "Error during IOCTL\n");  
}
```

To know the current status, type

```
unsigned long status;  
if (ioctl(fd, ADC_GET_MODE, &status) < 0) {  
    fprintf(stderr, "Error during IOCTL\n");  
}  
else  
{  
    if(status == NORMAL)  
        printf("Current status: Normal mode\n");  
    if(status == SLEEP)  
        printf("Current status: Sleep mode\n");  
}
```

- ADC_SET_STARTUP, ADC_GET_STARTUP

The ADC_SET_STARTUP command allows the user to set the value of the startup time of the ADC. The startup time is the time between the reset of the ADC and the moment it is enabled to be used. Its range is between 0x0 and 0x1F, but you must consider that it's necessary to set the STARTUP value at least to 0x4 to obtain a correct conversion; lower values generate conversion errors. This is an example of setting the startup value:

```
if (ioctl(fd, ADC_SET_STARTUP, 0x7) < 0) {  
    fprintf(stderr, "Error during IOCTL\n");  
}
```

To know the current status, type

```
unsigned long status;
```



```
if (ioctl(fd, ADC_GET_STARTUP, &status) < 0) {
    fprintf(stderr, "Error during IOCTL\n");
}
else
{
    printf("The startup value is %X\n", status);
}
```

- ADC_SET_SHTIM, ADC_GET_SHTIM

The ADC_SET_SHTIM command allows the user to set the value of the Sample and Hold Time of the ADC. The Sample and Hold Time is the time needed to receive the signal clock, receive the analog signal to convert, hold this signal and give it to the conversion process. Its range is between 0x0 and 0xF, but you must consider that it's necessary to set the SHTIM value at least to 0x2 to obtain a correct conversion; lower values generate conversion errors. This is an example of setting the Sample and Hold Time value:

```
if (ioctl(fd, ADC_SET_SHTIM, 0x3) < 0) {
    fprintf(stderr, "Error during IOCTL\n");
}
```

To know the current status, type

```
unsigned long status;
if (ioctl(fd, ADC_GET_SHTIM, &status) < 0) {
    fprintf(stderr, "Error during IOCTL\n");
}
else
{
    printf("The Sample and Hold Time value is %X\n", status);
}
```

- ADC_SET_PRESCAL, ADC_GET_PRESCAL

The ADC_SET_PRESCAL command allows the user to set the prescal value of the ADC. This value is a prescaler that modifies the frequency of the signal clock to adapt the master clock to the frequency needed by the ADC conversion process. The range of the prescal value is between 0x0 and 0x3F, but you must consider that it's necessary to set the value at least to 0x3 to obtain a correct conversion; lower values generate conversion errors. This is an example of setting this value:

```
if (ioctl(fd, ADC_SET_PRESCAL, 0x9) < 0) {
    fprintf(stderr, "Error during IOCTL\n");
}
```



To know the current status, type

```
unsigned long status;
if (ioctl(fd, ADC_GET_PRESCAL, &status) < 0) {
    fprintf(stderr, "Error during IOCTL\n");
}
else
{
    printf("The prescal value is %X\n", status);
}
```

- ADC_RESET

The ADC_RESET command allows the user to reset the ADC. You can choose to completely reset the ADC using the 0 argument, or perform a reset plus a default initialization (PRESCAL=0x9, STARTUP=0x7, SHTIM=0x3) using the 1 argument. After a reset the ADC will be automatically set in INTERRUPT mode.

Examples:

```
if (ioctl(fd, ADC_RESET, 0) < 0) {
    fprintf(stderr, "Error during IOCTL\n");
}
if (ioctl(fd, ADC_RESET, 1) < 0) {
    fprintf(stderr, "Error during IOCTL\n");
}
```

4.6.4. read()

The *read()* function shall attempt to read *nbyte* bytes from the file associated with the open file descriptor, *fildev*, into the buffer pointed to by *buf*. It will return in the *buf* argument the four digital values, one for each line, obtained after the conversion. Each value use 2 bytes of *buf*, so the dimension of *buf* must be set to RESULT_SIZE, that is equal to 8. If some line is disabled the correspondent value in *buf* will be equal to -1.

Header file:

unistd.h

Prototype:

```
ssize_t read(int fildev, void *buf, size_t nbyte);
```



Parameters:

*fil*des – file descriptor
buf – destination buffer pointer
nbyte – number of bytes that `read()` attempts to read. It must be set to `RESULT_SIZE`.

Returns:

The number of bytes actually read if successful, -1 otherwise.

Example:

```
char buf[RESULT_SIZE];

if(read(fd,buf,RESULT_SIZE)<0){
    fprintf(stderr, "Error during read\n");
}
```

To convert the eight values stored in *buf* into four `short int` values, you can use one of the facilities of the `adc.h` header file, as you can see in the following examples:

Example 1:

```
struct result *r;
r=(struct result*)buf;
printf("The values are chn0=%d,chn1=%d,chn2=%d,chn3=%d\n",r->chn0,r->chn1,r->chn2,r->chn3);
```

Example 2:

```
printf("The values are chn0=%d, chn1=%d, chn2=%d, chn3=%d\n",
CONVERT(buf,0), CONVERT(buf,1), CONVERT(buf,2), CONVERT(buf,3));
```

4.6.5. `close()`

The `close()` function shall deallocate the file descriptor indicated by *fil*des. To deallocate means to make the file descriptor available for return by subsequent calls to `open()` or other functions that allocate file descriptors.

Header file:

`unistd.h`

Prototype:

`int close(int fildes);`

Parameters:

*fil*des – file descriptor

Returns:



0 if successful, -1 otherwise

Example:

Close the ADC device.

```
if(close(fd) < 0)
{
    /* Error Management Routine */
} else {
    /* File Closed */
}
```

4.7. SSC

The Atmel Synchronous Serial Controller (SSC) provides a synchronous communication link with external devices.

It supports many serial synchronous communication protocols generally used in audio and telecom applications such as I2S, Short Frame Sync, Long Frame Sync, etc. The SSC is a six wire serial link used in audio application.

The SSC contains an independent receiver and transmitter and a common clock divider. Each of the receiver and transmitter interface use three signals: the TD/RD signal for data, the TK/RK signal for the clock and the TF/RF signal for the Frame Sync.

The transfers can be programmed to start automatically or on different events detected on the Frame Sync signal.

The SSC's high-level of programmability and its two dedicated PDC channels of up to 32 bits permit a continuous high bit rate data transfer without processor intervention.

4.7.1. Loading the SSC module

The SSC driver is released to customer under the shape of a loadable module.

To install the module, from the directory where the module is stored, the user shall type:

```
# modprobe atmel-ssc
# insmod ge863pro3_ssc.ko
```

When loaded, the SSC driver installs, under the /dev/ directory, a new device, named *ssc_ge863pro3*.

Once installed the SSC device can be used as a normal character device and can be accessed by any application running in user space.



4.7.2. open()

The *open()* function shall establish the connection between a file and a file descriptor. The file descriptor is used by other I/O functions to refer to the opened file.

Header file:

fcntl.h

Prototype:

```
int open(const char *pathname, int flags)
```

Parameters:

pathname – file name with its own path

flags – is an *int* specifying file opening mode: is one of O_RDONLY, O_WRONLY or O_RDWR which request opening the file read-only, write-only or read/write, respectively

Returns:

The new file descriptor *filde*s if operation is completed successfully, otherwise it is -1.

Example:

Open the /dev/ssc_ge863pro3.

```
int fd; // file descriptor for the /dev/ssc_ge863pro3 entry

if((fd = open("/dev/ssc_ge863pro3", O_RDWR) < 0)
{
    /* Error Management Routine */
} else {
    /* SSC Device Opened */
}
```

4.7.3. ioctl()

The *ioctl()* function manipulates the underlying device parameters. In particular, many operating characteristics can be controlled with *ioctl()* requests.

Header file:

sys/ioctl.h
linux/ssc/ge863pro3_ssc_ioctl.h

Prototype:

```
int ioctl(int filde, int request, ...)
```



Parameters:

fildev – file descriptor

request – device-dependent request code.

The following ioctls request codes can be used for SSC:

- SSC_IOCTL_CMR_SET and SSC_IOCTL_CMR_GET to set/get the Clock Mode Register
- SSC_IOCTL_RCMR_SET and SSC_IOCTL_RCMR_GET to set/get the Receive Clock Mode Register
- SSC_IOCTL_RFMR_SET and SSC_IOCTL_RFMR_GET to set/get the Receive Frame Mode Register
- SSC_IOCTL_TCMR_SET and SSC_IOCTL_TCMR_GET to set/get the Transmit Clock Mode Register
- SSC_IOCTL_TFMR_SET and SSC_IOCTL_TFMR_GET to set/get the Transmit Frame Mode Register
- SSC_IOCTL_CR_SET to set the Control Register

The third argument is a *void ** and depends on the ioctl request code used: see the examples below.

Returns:

0 if operation is completed successfully, otherwise it is -1.

Examples:

- SSC_IOCTL_CMR_SET and SSC_IOCTL_CMR_GET

The CMR register is used to configure the Master Clock divider. The Master Clock divider is determined by the 12-bit field DIV counter and comparator (so its maximal value is 4095) in the Clock Mode Register SSC_CMR, allowing a Master Clock division by up to 8190. The Divided Clock is provided to both the Receiver and Transmitter. Pass a pointer to a unsigned int which will return or assign the CMR configuration. Example:

```
div = (CLK_RATE/(44100*2*16))/2; /* Beware! this is just an
example.
Check your device before setting these fields*/
if(ioctl(fd, SSC_IOCTL_CMR_SET, &div))
{
    printf("ERROR IOCTL CMR\n");
}
```

- SSC_IOCTL_RCMR_SET and SSC_IOCTL_RCMR_GET



The receiver clock is generated from the transmitter clock or the divider clock or an external clock scanned on the RK I/O pad. The Receive Clock is selected by the CKS field in SSC_RCMR (Receive Clock Mode Register).

Pass a pointer to an unsigned int which will return or assign the RCMR configuration.

Example:

```
rcmr = 0x00010121;
if(ioctl(fd, SSC_IOCTL_RCMR_SET, &rcmr))
{
    printf("ERROR IOCTL RCMR\n");
}
```

- SSC_IOCTL_RFMR_SET, SSC_IOCTL_RFMR_GET, SSC_IOCTL_TFMR_SET and SSC_IOCTL_TFMR_GET

The Transmitter and Receiver Frame Sync pins, TF and RF, can be programmed to generate different kinds of frame synchronization signals. The Frame Sync Output Selection (FSOS) field in the Receive Frame Mode Register (SSC_RFMR) and in the Transmit Frame Mode Register (SSC_TFMR) are used to select the required waveform.

Programmable low or high levels during data transfer are supported.

Programmable high levels before the start of data transfers or toggling are also supported.

Pass a pointer to an unsigned int which will return or assign the RFMR/TFMR configuration.

Example:

```
tfmr = 0x001F018F;
rfmr = 0x0000018F;

if(ioctl(fd, SSC_IOCTL_TFMR_SET, &tfmr))
{
    printf("ERROR IOCTL TFMR\n");
}

if(ioctl(fd, SSC_IOCTL_RFMR_SET, &rfmr))
{
    printf("ERROR IOCTL RFMR\n");
}
```

- SSC_IOCTL_TCMR_SET and SSC_IOCTL_TCMR_GET

The transmitter clock is generated from the receiver clock or the divider clock or an external clock scanned on the TK I/O pad. The transmitter clock is selected by the CKS field in SSC_TCMR (Transmit Clock Mode Register).

Pass a pointer to a unsigned int which will return or assign the TCMR configuration.



Example:

```
tcmr = 0x0F010404;
if(ioctl(fd, SSC_IOCTL_TCMR_SET, &tcmr))
{
    printf("ERROR IOCTL RCMR\n");
}
```

- SSC_IOCTL_CR_SET

The SSC_CR is used to enable/disable transmit and receive data. The signal clock starts only after the enable of transmit or receive.

Pass a pointer to an unsigned int which will assign the CR configuration.

Example:

```
cr = (1<<8)|1; /*enable RX and TX*/
if(ioctl(fd, SSC_IOCTL_CR_SET, &cr))
{
    printf("ERROR IOCTL CR\n");
}
```

4.7.4. read()

The *read()* function reads *nbyte* bytes from the file associated with the open file descriptor, *fildev*, and copies them in the buffer that is pointed to by *buf*.

Header file:

unistd.h

Prototype:

```
ssize_t read(int fildev, void *buf, size_t nbyte);
```

Parameters:

fildev – file descriptor

buf – destination buffer pointer

nbyte – number of bytes that *read()* attempts to read

Returns:

The number of bytes actually read if operation is completed successfully, otherwise it is -1.

Example:

Read *sizeof(read_buff)* bytes from the file associated with *fd* and stores them in *read_buff*.



```
char read_buff[BUFF_LEN];

if(read(fd, read_buff, sizeof(read_buff)) < 0)
{
    /* Error Management Routine */
} else {
    /* Value Read */
}
```

4.7.5. write()

The *write()* function writes *nbyte* bytes from the buffer that are pointed by *buf* to the file associated with the open file descriptor, *fildes*.

Header file:

unistd.h

Prototype:

```
ssize_t write(int fildes, const void *buf, size_t nbyte);
```

Parameters:

fildes – file descriptor

buf – destination buffer pointer

nbyte – number of bytes that write() attempts to write

Returns:

The number of bytes actually written if operation is completed successfully, (this number shall never be greater than *nbyte*), otherwise it is -1

Example:

Write *strlen(value_to_be_written)* bytes from the buffer pointed by *value_to_be_written* to the file associated with the open file descriptor, *fd*.

```
char value_to_be_written[] = "dummy_write";

if (write(fd, value_to_be_written, strlen(value_to_be_written))
< 0)
{
    /* Error Management Routine */
} else {
    /* Value Written */
}
```



4.7.6. close()

The *close()* function deallocates the file descriptor indicated by *filides*. To deallocate means to make the file descriptor available for subsequent calls to *open()* or other functions that allocate file descriptors.

Header file:
unistd.h

Prototype:
int close(int filides);

Parameters:
filides – file descriptor

Returns:
0 if operation is completed successfully, otherwise it is -1

Example:
Close the SSC device.

```
if(close(fd) < 0)
{
    /* Error Management Routine */
} else {
    /* File Closed */
}
```

4.7.7. A Test Program

Below it is reported a simple piece of code that first opens the device and then writes some data.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <pthread.h>
#include <unistd.h>

#include <linux/ssc/ge863pro3_ssc_ioctl.h>

int main(int argc, char *argv[])
{
```




```

        return -1;
    }

    if(ioctl(fd, SSC_IOCTL_CR_SET, &cr))
    {
        printf("ERROR IOCTL CR\n");
        return -1;
    }

    memset(txt,'A', 20000);

    retW = write(fd,txt,20000);

    close(fd);

    return 0;
}

```

4.8. Watchdog

A Watchdog Timer (WDT) is a hardware circuit that can reset the computer system in case of a software fault.

Usually a userspace daemon will notify the kernel watchdog driver via the /dev/watchdog special device file that the userspace is still alive, at regular intervals. When such a notification occurs, the driver will usually tell the hardware watchdog that everything is in order, and that the watchdog should wait before resetting the system. If userspace fails (RAM error, kernel bug, whatever), the notifications cease to occur, and the hardware watchdog will reset the system (causing a reboot) after the timeout expires.

Userspace can interact with the kernel watchdog driver through the functions shown in the paragraphs below.

4.8.1. open()

The *open()* function establishes the connection between a file and a file descriptor. The file descriptor is used by other I/O functions to refer to the opened file.

Header file:
fcntl.h

Prototype:
int open(const char *pathname, int flags)

Parameters:



pathname – file name with its own path

flags – an *int* specifying file opening mode: that can be O_RDONLY, O_WRONLY or O_RDWR which request opening the file read-only, write-only or read/write, respectively

Returns:

The new file descriptor *fildes* if operation is completed successfully, otherwise it is -1

Example:

Open the /dev/watchdog.

```
int fd; // file descriptor for the /dev/watchdog entry

if((fd = open("/dev/watchdog", O_WRONLY) < 0)
{
    /* Error Management Routine */
} else {
    /* Watchdog Device Opened */
}
```

4.8.2. ioctl()

The ioctl() function manipulates the underlying device parameters. In particular, many operating characteristics can be controlled with ioctl() requests.

Header file:

sys/ioctl.h
linux/watchdog.h

Prototype:

int ioctl(int fildes, int request, ...)

Parameters:

fildes – file descriptor

request – device-dependent request code.

The following ioctls request codes can be used for watchdog device:

- WDIOC_KEEPLIVE: to notify the watchdog that userspace is still alive
- WDIOC_SETTIMEOUT: to set a timeout
- WDIOC_SETOPTIONS: to enable or disable the watchdog
- WDIOC_GETTIMEOUT: to query the timeout

The third argument is a *void ** and depends on the ioctl request code used: see the examples below.

Returns:



0 if operation is completed successfully, otherwise it is -1

Examples:

- WDIOC_KEEPALIVE

The AT91Sam9260 watchdog driver supports the WDIOC_KEEPALIVE ioctl. It notifies the watchdog that userspace is still alive. In this case the third argument in the ioctl is ignored. This function call resets the timer and leaves the system alive (is used to avoid reset when the watchdog is active). Example:

```
fd = open("/dev/watchdog", O_WRONLY);
int dummy;
for(;;){
    ioctl(fd, WDIOC_KEEPALIVE, &dummy);
    sleep(1)
}
```

- WDIOC_SETTIMEOUT

Setting Timeout is performed by the SETTIMEOUT ioctl. The third argument is an integer that represents the timeout in seconds (max timeout is 16 seconds in at91sam9260 architecture). The driver returns the real timeout used in the same variable, and this timeout can be different from the one that has been set due to limitation of the hardware.

The watchdog timeout can be written only once (at91sam9260 only permits one program operation).

Example:

```
int timeout=15;
ioctl(fd, WDIOC_SETTIMEOUT, &timeout);
```

Notice: If the watchdog start is enabled the user must set the timeout otherwise it will use the default value.

- WDIOC_SETOPTIONS

It enables or disables the watchdog. The third argument is an integer indicating the option to be set.

Example to disable the watchdog:

```
int options = WDIOS_DISABLECARD;
ioctl(fd, WDIOC_SETOPTIONS, &options);
```

Example to enable the watchdog:



```
int options = WDIOSEnableCard;
ioctl(fd, WDIOCSOptions, &options);
```

- WDIOCGTimeout

It is possible to query the timeout using the WDIOCGTimeout ioctl. The third argument is an integer representing the timeout in seconds.

Example:

```
ioctl(fd, WDIOCGTimeout, &timeout);
printf("The timeout is %d seconds\n", timeout);
```

4.8.3. close()

The `close()` function deallocates the file descriptor indicated by *fd*. To deallocate means to make the file descriptor available for subsequent calls to `open()` or other functions that allocate file descriptors.

Header file:

unistd.h

Prototype:

```
int close(int fd)
```

Parameters:

fd – file descriptor

Returns:

0 if operation is completed successfully, otherwise it is -1

Example:

Close the watchdog device.

```
if(close(fd) < 0)
{
    /* Error Management Routine */
} else {
    /* File Closed */
}
```



4.9. Power Management

The GE863-PRO³ makes available two power management modes:

- *standby*
- *suspend to RAM*

The *standby* mode puts RAM in self refreshing mode. This mode can be enabled by typing:

```
# echo standby > /sys/power/state
```

The *suspend to RAM* mode puts RAM in self refreshing mode and cpu in slow-clock mode. It can be enabled as follows:

```
# echo mem > /sys/power/state
```

In both modes GE863-PRO³ can be awakened by an interrupt from an awakeable source (e.g., by inserting a SD card or generating an event on `/dev/ttyS1`).

Each device listed in `/sys/devices/platform/` has a file named "wakeup" in its subdirectory `power/`. If a device can issue wakeup events, a read from its wakeup file gives the string "*enabled*\n" or "*disabled*\n", depending on whether its wakeup feature is enabled or not; if needed, the user can change a device's behaviour by writing on its wakeup file the string "*enabled*" or "*disabled*". If a device cannot issue wakeup events, a read from its wakeup file gives the string "\n", and it is not possible to write on the file.

In order for a device to be able to wake up the system, it is necessary to have a driver which controls the device and supports the suspend and resume mechanisms; so, for example, the Ethernet controller (whose directory is `/sys/devices/platform/macb/`) is not able to wake up the system unless its driver is loaded (see section 4.12). Also, some devices must be in use (for example, there must be a process which has their device file opened) to be able to wake up the system.

Some devices cannot wake up the system from *suspend to RAM* mode, even if their wakeup file shows that their wakeup feature is enabled: this is due to the fact that those devices in order to work properly need a high speed clock, and during *suspend to RAM* the high speed clocks are turned off. For example the serial ports can not receive characters if their clock is not correctly set, and thus it is not possible to wake up the system from *suspend to RAM* mode by sending characters to them; as an exception, the serial port `/dev/ttyS0` (whose directory is `/sys/devices/platform/atmel_usart.0/`) can wake up the system from any power save mode by sending characters to it, but in case of *suspend to RAM* some characters sent while the system is being woken up might get lost.



Power management can also be performed from source code as shown below. The functionalities provided by the Linux kernel Real Time Clock (RTC) (see paragraph 4.10) can be used to manage system awakenings.

4.9.1. open()

The *open()* function establishes the connection between a file and a file descriptor. The file descriptor is used by other I/O functions to refer to the opened file.

Header file:

fcntl.h

Prototype:

int open(const char *pathname, int flags)

Parameters:

pathname – file name with its own path

flags – an *int* specifying file opening mode: that can be O_RDONLY, O_WRONLY or O_RDWR which request opening the file read-only, write-only or read/write, respectively

Returns:

The new file descriptor *fildes* if operation is completed successfully, otherwise it is -1

Example:

Open the /sys/power/state file.

```
int fd; // file descriptor for the /sys/power/state entry

fd = open("/sys/power/state", O_RDWR);
if (fd == -1)
{
    /* Error Management Routine */
}
```

4.9.2. write()

The *write()* function writes *nbyte* bytes from the buffer that are pointed by *buf* to the file associated with the open file descriptor, *fildes*.

Header file:

unistd.h

Prototype:

ssize_t write(int fildes, const void *buf, size_t nbyte)




```
{
  /* Error Management Routine */
} else {
    /* File Closed */
}
```

4.10. Real Time Clock (RTC)

Most computers have a built-in hardware clock, usually called the real-time clock. This clock is normally battery powered so that it keeps the time even while the computer is switched off. It represents the current time as year, month, day of month, hour, minute, and second. The RTC should not be confused with the system time which is an independent, interrupt-driven software clock maintained by the kernel.

Below are described all the functions to be used to manage RTC and an example on how to perform power management operations.

4.10.1. open()

The *open()* function establishes the connection between a file and a file descriptor. The file descriptor is used by other I/O functions to refer to the opened file.

Header file:

fcntl.h

Prototype:

int open(const char *pathname, int flags)

Parameters:

pathname – file name with its own path

flags – an *int* specifying file opening mode: that can be O_RDONLY, O_WRONLY or O_RDWR which request opening the file read-only, write-only or read/write, respectively

Returns:

The new file descriptor *fdes* if operation is completed successfully, otherwise it is -1

Example:

Open the /dev/rtc0.

```
int fd; // file descriptor for the /dev/rtc0 entry

fd = open("/dev/rtc0", O_RDWR);
if (fd == -1)
{
  /* Error Management Routine */
}
```



4.10.2. ioctl()

The `ioctl()` function manipulates the underlying device parameters. In particular, many operating characteristics can be controlled with `ioctl()` requests.

Header file:

`sys/ioctl.h`
`linux/rtc.h`

Prototype:

`int ioctl(int fildes, int request, ...)`

Parameters:

`fildes` – file descriptor

`request` – device-dependent request code.

The following `ioctl`s request codes can be used for RTC:

- `RTC_RD_TIME`: to read time, returning the result as a Gregorian calendar date and 24 hour wall clock time
- `RTC_SET_TIME`: to set/update time and date
- `RTC_ALM_SET`: to set the alarm time to wake up the system up to 24 hours in the future
- `RTC_ALM_READ`: to read the alarm time set with the `RTC_ALM_SET` `ioctl`
- `RTC_WKALM_SET`: to issue alarms beyond the next 24 hours
- `RTC_WKALM_RD`: to read the alarm time set with the `RTC_WKALM_SET` `ioctl`
- `RTC_AIE_ON`: to enable the alarm
- `RTC_AIE_OFF`: to disable the alarm
- `RTC_UIE_ON`: to enable IRQs update whenever the "seconds" counter changes
- `RTC_UIE_OFF`: to disable IRQs update whenever the "seconds" counter changes

The third argument is a `void *` and depends on the `ioctl` request code used: see the examples below.

Returns:

0 if operation is completed successfully, otherwise it is -1

Examples:

- `RTC_RD_TIME`

It is used to read time, returning the result as a Gregorian calendar date and 24 hour wall clock time. The third argument is a pointer to a `struct rtc_time` used to store the read values:

```
struct rtc_time {
```



```
int tm_sec;
int tm_min;
int tm_hour;
int tm_mday;
int tm_mon;
int tm_year;
int tm_wday;
int tm_yday;
int tm_isdst;
};
```

Example:

```
int retval;
struct rtc_time rtc_tm;

/* Read the RTC time/date */
retval = ioctl(fd, RTC_RD_TIME, &rtc_tm);
if (retval == -1)
{
    perror("RTC_RD_TIME ioctl");
    exit(errno);
}
```

- RTC_SET_TIME

It is used to set/update time. The third argument is a pointer to a `struct rtc_time` that stores the value to be set.

Example:

```
int retval;
struct rtc_time rtc_tm;

/* Initialize the rtc_time struct to the value to be set later
*/
/* e.g. read the current RTC time/date and set it again */

/* Set the RTC time/date */
retval = ioctl(fd, RTC_SET_TIME, &rtc_tm);
if (retval == -1)
{
    perror("RTC_SET_TIME ioctl");
    exit(errno);
}
```

- RTC_ALM_SET



It is used to set the alarm time to wake up the system issuing an alarm IRQ up to 24 hours in the future. Only the `tm_sec`, `tm_min`, and `tm_hour` fields of struct `rtc_time` structure are used.

Example:

```
int retval;
struct rtc_time rtc_tm;
int time = 10; /* time seconds used to set alarm */

/* Initialize the rtc_time struct to the alarm value ([time]
seconds) to be set later and check for rollover */
rtc_tm.tm_sec += time;

if (rtc_tm.tm_sec >= 60)
{
    rtc_tm.tm_sec %= 60;
    rtc_tm.tm_min++;
}

if (rtc_tm.tm_min == 60)
{
    rtc_tm.tm_min = 0;
    rtc_tm.tm_hour++;
}

if (rtc_tm.tm_hour == 24)
    rtc_tm.tm_hour = 0;

/* Set the alarm to [time] seconds in the future */
retval = ioctl(fd, RTC_ALM_SET, &rtc_tm);
if (retval == -1)
{
    if (errno == ENOTTY)
    {
        fprintf(stderr, "\n...Alarm IRQs not supported.\n");
        close(fd);
    }
    perror("RTC_ALM_SET ioctl");
    exit(errno);
}
```

- `RTC_ALM_READ`

It reads the alarm time set with the `RTC_ALM_SET` `ioctl`. The third argument is a pointer to a struct `rtc_time` used to store the read values.

Example:



```
int retval;
struct rtc_time rtc_tm;

/* Read the RTC Alarm time set */
retval = ioctl(fd, RTC_ALM_READ, &rtc_tm);
if (retval == -1)
{
    perror("RTC_RD_TIME ioctl");
    exit(errno);
}
```

- RTC_WKALM_SET

Same as RTC_ALM_SET but to issue alarms beyond the next 24 hours.

- RTC_WKALM_RD

Same as RTC_ALM_READ but to read the alarm time set with the RTC_WKALM_SET ioctl.

- RTC_AIE_ON

It is used to enable the alarm set with RTC_ALM_SET. The third argument is ignored.
Example:

```
int retval;
struct rtc_time rtc_tm;

/* Enable alarm interrupts */
retval = ioctl(fd, RTC_AIE_ON, 0);
if (retval == -1)
{
    perror("RTC_AIE_ON ioctl");
    exit(errno);
}
```

- RTC_AIE_OFF

It is used to disable the alarm set with RTC_ALM_SET. The third argument is ignored.
Example:

```
int retval;
struct rtc_time rtc_tm;

/* Disable alarm interrupts */
retval = ioctl(fd, RTC_AIE_OFF, 0);
if (retval == -1)
```



```
{
    perror("RTC_AIE_OFF ioctl");
    exit(errno);
}
```

- **RTC_UIE_ON**

It is used to enable IRQs update whenever the "seconds" counter changes. The third argument is ignored.

Example:

```
int retval;

/* Turn on update interrupts (one per second) */
retval = ioctl(fd, RTC_UIE_ON, 0);
if (retval == -1)
{
    if (errno == ENOTTY)
    {
        fprintf(stderr, "\n...Update IRQs not supported.\n");
    }
    perror("RTC_UIE_ON ioctl");
    exit(errno);
}
```

- **RTC_UIE_OFF**

It is used to disable IRQs update whenever the "seconds" counter changes. The third argument is ignored.

Example:

```
int retval;

/* Turn off update interrupts */
retval = ioctl(fd, RTC_UIE_OFF, 0);
if (retval == -1)
{
    perror("RTC_UIE_OFF ioctl");
    exit(errno);
}
```

4.10.3. close()

The *close()* function deallocates the file descriptor indicated by *fd*. To deallocate means to make the file descriptor available for subsequent calls to *open()* or other functions that allocate file descriptors.

Header file:
unistd.h



Prototype:

int close(int fildes)

Parameters:

fildes – file descriptor

Returns:

0 if operation is completed successfully, otherwise it is -1

Example:

Close the watchdog device.

```
if(close(fd) < 0)
{
    /* Error Management Routine */
} else {
    /* File Closed */
}
```

4.10.4. A Test Program

```
/*
 *      Power Management Test Program
 */

#include <stdio.h>
#include <linux/rtc.h>
#include <sys/ioctl.h>
#include <sys/time.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>

static const int default_time = 10;
static const char modes[2][8] = {"standby", "mem"};

int main(int argc, char **argv)
{
    int fd, retval;
    struct rtc_time rtc_tm;
    int time = default_time;
    const char *mode = modes[0];
```



```

    if (argc > 3) {
        fprintf(stderr, "usage:  pwrmgng_test [sleep_time]
[mode]\n"
                    "[sleep_time] is expressed in seconds and
must be between 1 and 60\n"
                    "[mode] can be \'standby\' or \'mem\'
(suspend-to-RAM)\n");
        return 1;
    }
    if (argc > 1)
        time = atoi(argv[1]);
    if (argc > 2)
        if (strcmp(argv[2], modes[1]) == 0)
            mode = modes[1];

    fprintf(stderr, "\n\t\t\tPower Management Test Program.\n\n"
                "Using %s for %d seconds\n\n",
            (mode == modes[0]) ? mode : "suspend-to-RAM",
time);
    fflush(stderr);
    sleep(1);

    while (1) {
        fd = open("/dev/rtc0", O_RDWR);
        if (fd == -1) {
            perror("/dev/rtc0");
            exit(errno);
        }

        /* Read the RTC time/date */
        retval = ioctl(fd, RTC_RD_TIME, &rtc_tm);
        if (retval == -1) {
            perror("RTC_RD_TIME ioctl");
            exit(errno);
        }

        /* Set the RTC time/date */
        retval = ioctl(fd, RTC_SET_TIME, &rtc_tm);
        if (retval == -1) {
            perror("RTC_SET_TIME ioctl");
            exit(errno);
        }

        /* Set the alarm to [time] seconds in the future, and
check for rollover */
        rtc_tm.tm_sec += time;
        if (rtc_tm.tm_sec >= 60) {

```



```

        rtc_tm.tm_sec %= 60;
        rtc_tm.tm_min++;
    }
    if (rtc_tm.tm_min == 60) {
        rtc_tm.tm_min = 0;
        rtc_tm.tm_hour++;
    }
    if (rtc_tm.tm_hour == 24)
        rtc_tm.tm_hour = 0;

    retval = ioctl(fd, RTC_ALM_SET, &rtc_tm);
    if (retval == -1) {
        if (errno == ENOTTY) {
            fprintf(stderr, "\n...Alarm IRQs not
supported.\n");
            close(fd);
        }
        perror("RTC_ALM_SET ioctl");
        exit(errno);
    }

    /* Enable alarm interrupts */
    retval = ioctl(fd, RTC_AIE_ON, 0);
    if (retval == -1) {
        perror("RTC_AIE_ON ioctl");
        exit(errno);
    }

    close(fd);

    fd = open("/sys/power/state", O_RDWR);
    if (fd == -1) {
        perror("/sys/power/state");
        exit(errno);
    }

    /* This blocks until the alarm ring causes an interrupt */
    retval = write(fd, (unsigned char *)mode, strlen(mode));
    if (retval == -1) {
        perror("write");
        exit(errno);
    }

    close(fd);
}

return 0;
}

```



4.11. SD/MMC

GE863-PRO³ support for Secure Digital (SD) and Multimedia Card (MMC) is built in the kernel.

First create a directory where the device will be mounted, for example:

```
# mkdir /mnt/sdcard
```

Then, connect the device and mount it:

```
# mount /dev/mmcblk0p1 /mnt/sdcard
```

Depending on the SD or MMC cards and how they are partitioned, different device names could be created on card insertion:

```
# mount /dev/mmcblk0 /mnt/sdcard
```

is also possible when `mmcblk0p1` is not created. In general would be correct to check mmc devices before trying to mount it typing:

```
ls /dev/mmc*
```

The device is now ready to be used.

Note that the card filesystem must be Fat/Fat32 in order to work correctly.

To un-mount the device type:

```
# umount /mnt/sdcard
```

4.12. Ethernet

The GE863-PRO³ Linux operating system is able to drive an Ethernet interface; the Ethernet support is built as a module. To load the module type in the terminal:

```
# modprobe macb
```

Once the module is loaded, you will find a new Ethernet device called `eth0`; to properly configure the device use the `ifconfig` command.



First of all you have to assign the MAC address:

```
# ifconfig eth0 hw ether <MAC address>
```

For example:

```
# ifconfig eth0 hw ether AA:BB:AA:BB:AA:BB
```

Then, you have to assign the IP address and netmask: you can assign it manually or you can automatically set it by making a request from a dhcp server connected to the same local network.

For manual configuration you have to type the following command:

```
# ifconfig eth0 <ip address> netmask <netmask>
```

For example, suppose you want to assign the IP address 192.168.1.12 to the device, with a netmask 255.255.255.0:

```
# ifconfig eth0 192.168.1.12 netmask 255.255.255.0
```

On the contrary, for an automatic assignment, you can call the built-in dhcp client application to request a IP address to dhcp server typing:

```
# udhcpc -nq
```

The n option is not mandatory, but for memory optimizations it kills udhcpc application once the IP address has been caught.

The q option is not mandatory as well, but it is very useful to avoid an infinite IP search: in fact it quits the udhcpc application after some retries if the IP address assignment has failed, due for example to a cable disconnection or to dhcp server powered down.

To test if everything works correctly on the device you can type:

```
# ping <ip address of another station in the network>
```

And you should see packets transmission.



4.13. USB

4.13.1. USB Mass Storage

Any type of USB Mass Storage Device can be connected to the GE863-PRO³. For that purpose some kernel modules have to be loaded. Type in the terminal:

```
# modprobe ohci-hcd
# modprobe usb-storage
```

Now create a directory where the device will be mounted, for example:

```
# mkdir /mnt/usbdev0
```

Connect the device then mount it:

```
# mount /dev/sda1 /mnt/usbdev0
```

Note that the GE863-PRO³ has two USB ports A-type, so if you connect two devices they will be called `/dev/sda1` and `/dev/sdb1`.

To un-mount the device type:

```
# umount /mnt/usbdev0
```

4.13.2. USB device (Ethernet Gadget)

The GE863-PRO³ has Ethernet on USB capabilities, so the USB link B-type (target side) / A-type (PC side) can behave like a normal Ethernet link. For that purpose some kernel modules need to be loaded. Type in the terminal:

```
# modprobe g_ether
```

Connect the USB cable and configure the interface:

```
# ifconfig usb0 <ip address> netmask <netmask>
```

For example:

```
# ifconfig usb0 192.168.121.3 netmask 255.255.255.0
```

To test the correct working of the device you can type:

```
# ping <ip address of another station in the network>
```



And you should see packets transmission.

Note that: on the host you need to load the suitable drivers and configure the new virtual Ethernet link for correct working. For further details refer to [4] .

To remove the link type:

```
# ifconfig usb0 down
```

4.14. Timer Counter

The GE863-PRO³ is provided with a Timer Counter (TC) with three identical 16-bit Timer Counter channels. Each channel can perform a wide range of functions, including frequency measurement, event counting, interval measurement, pulse generation, delay timing and pulse width modulation.

Each channel can be programmed to have a specific clock, chosen among three external clock inputs, five internal clock inputs or two multi-purpose input/output signals which can be configured by the user. Each channel can also generate processor interrupts or not, depending of the user configuration.

Finally, the three channels can be chained to connecting the output of a Timer Counter with the external clock input of another Timer Counter.

The Timer Counter can be set in one of the two modes available, the Capture Mode or the Waveform Mode, depending of the features you want to run.

Briefly, the main tasks you can do are counting something until the counter reaches a programmable value (using the Capture Mode) or generate waveforms (using the Waveform Mode).

An example can be the use of the TC with the Analog to Digital Converter (ADC): the ADC can receive the output waveform of the Timer Counter as an external trigger, and start a new analog to digital conversion after each trigger received. For more informations about ADC with TC see also the ADC chapter in this guide.

In this paragraph we'll see how to use the Telit Timer Counter Driver and the options it provides.

4.14.1. Loading the Timer Counter Module

To use the Timer Counter devices through the /dev interface of the Linux Operating System, you have to load the tc.ko module.

To do this, select the folder containing that file and type



```
insmod ./tc.ko
```

After the loading, in the `/dev` folder three devices will be created: they are called `tc0`, `tc1` and `tc2`.

4.14.2. open()

SD The `open()` function shall establish the connection between a file and a file descriptor. The file descriptor is used by other I/O functions to refer to that file.

Header file:

`fcntl.h`

Prototype:

```
int open(const char *pathname, int flags)
```

Parameters:

`pathname` – file name with its own path

`flags` – is an *int* specifying file opening mode: is one of `O_RDONLY`, `O_WRONLY` or `O_RDWR` which request opening the file read-only, write-only or read/write, respectively

Returns:

The new file descriptor *fdes* if successful, `-1` otherwise

Example:

Open the `/dev/tc0`.

```
int fd; // file descriptor for the /dev/tc0 entry

if((fd = open("/dev/tc0", O_RDONLY) < 0)
{
    /* Error Management Routine */
} else {
    /* TC0 Device Opened */
}
```



4.14.3. ioctl()

The `ioctl()` function manipulates the underlying device parameters of special files. In particular, many operating characteristics of character special files may be controlled with `ioctl()` requests.

Header file:

`sys/ioctl.h`
`linux/tc/tc.h`

Prototype:

`int ioctl(int fildes, int request, ...)`

Parameters:

`fildes` – file descriptor

`request` – device-dependent request code.

The following `ioctl`s request codes can be used for Timer Counters:

- `TC_SET_CAPTURE_MODE`: to set the TC capture mode
- `TC_SET_WAVEFORM_MODE`: to set the TC waveform mode
- `TC_GET_MODE`: to get the current mode (capture or waveform)
- `TC_GET_CAPTURE_SETTINGS`: to get the capture mode settings
- `TC_GET_WAVEFORM_SETTINGS`: to get the waveform mode settings
- `TC_SET_CLOCK`, `TC_GET_CLOCK`: to select/get the TC clock
- `TC_ENABLE`: to enable/disable the timer counter
- `TC_SOFTWARE_TRIGGER`: to generate a software trigger
- `TC_GET_CV`: to read the current value of the counter and how many wraparounds occurred.
- `TC_SET_RA`, `TC_GET_RA`: to set/get the register A value (`TC_SET_RA` works in waveform mode only)
- `TC_SET_RB`, `TC_GET_RB`: to set/get the register B value (`TC_SET_RB` works in waveform mode only)
- `TC_SET_RC`, `TC_GET_RC`: to set/get the register C value.

The third argument is a *void ** and depends on the `ioctl` request code used: see the examples below.

Returns:

0 if successful, -1 otherwise

Examples:



- TC_SET_CAPTURE_MODE

The TC_SET_CAPTURE_MODE command allows the user to enter in the TC Capture Operating Mode, useful to perform measurements such as pulse timing, frequency, period, duty cycle and phase on TIOA and TIOB signals, which are considered as inputs. The argument of the `ioctl` call is a pointer to a

```
struct tc_capture_mode{
    short ext_trg;
    short ext_trg_edge;
    short RC_trigger;
    short RA_load_edge;
    short RB_load_edge;
};
```

which is useful to customize the use of the Timer Counter in Capture Mode. In the `ext_trg` field you can choose one between TIOA and TIOB as external trigger. The effect of the trigger is to reset the counter and start the counter clock. To choose the external trigger you have to give one of these values to `ext_trg` field:

```
TC_TIOA_EXT_TRG
TC_TIOB_EXT_TRG
```

Then you have to select the external trigger edge you want to use. To do this, set the `ext_trg_edge` field with one of this option:

```
TC_NONE
TC_RISING_EDGE
TC_FALLING_EDGE
TC_EACH_EDGE
```

If you set `ext_trg_edge = TC_NONE`, the value set in `ext_trg` won't be considered.

If you set the `RC_trigger` field with 1, when the counter will reach the value stored in the register C it will be restarted from zero; if `RC_trigger` is set to 0, the register C value won't have any trigger effect on timer counter.

The `RA_load_edge` and `RB_load_edge` fields define during which edge of TIOA you want to load the current timer counter value to, respectively, register A or register B. To do this, set both `RA_load_edge` and `RB_load_edge` with one of this option:

```
TC_NONE
TC_RISING_EDGE
TC_FALLING_EDGE
TC_EACH_EDGE
```

In the following example for the Capture Mode we want to know the time elapsed between the rising and the falling edge of TIOA, using TIOA also as external trigger. To do so, we set `TC_TIOA_EXT_TRG` as external trigger on `TC_RISING_EDGE`: in this way,



during the rising edge the counter will be set to zero and restarted. Then we set RA_load_edge on TC_FALLING_EDGE: in this way the value of the counter will be stored in RA when TIOA edge is falling. Then we simply read RA to know the value we wanted.

```

struct tc_capture_mode cmd;
cmd.ext_trg = TC_TIOA_EXT_TRG;
cmd.ext_trg_edge = TC_RISING_EDGE;
cmd.RC_trigger = 0;
cmd.RA_load_edge = TC_FALLING_EDGE;
ret=ioctl(fd, TC_SET_CAPTURE_MODE, &cmd);
if (ret < 0)
    printf("An error occurred in ioctl");

```

- TC_SET_WAVEFORM_MODE

The TC_SET_WAVEFORM_MODE command allows the user to enter in the TC Waveform Operating Mode. In this mode the TC channel can generate 1 or 2 PWM signals with the same frequency and independently programmable duty cycles, or generate different types of one-shot or repetitive pulses. To do this, TIOA is configured as an output and TIOB is defined as an output if it is not used as an external event. The argument of the ioctl call is a pointer to a:

```

struct tc_waveform_mode{
    short ext_event;
    short ext_event_edge;
    short wave_sel;
    short ra_2_tioa;          /* RA Compare Effect on TIOA */
    short rc_2_tioa;          /* RC Compare Effect on TIOA */
    short extevnt_2_tioa;     /* External Event Effect on TIOA */
    short swtrg_2_tioa;      /* Software Trigger Effect on TIOA */
    /*
    short rb_2_tioab;         /* RB Compare Effect on TIOB */
    short rc_2_tioab;         /* RC Compare Effect on TIOB */
    short extevnt_2_tioab;    /* External Event Effect on TIOB */
    short swtrg_2_tioab;     /* Software Trigger Effect on TIOB */
    */
};

```

So you have to set these parameters to customize the waveform mode. In the ext_event field you can choose the external event source using

```

TC_TIOB_EXT_EVENT          /* TIOB generate external events */
TC_XC0_EXT_EVENT           /* XC0 generate external events */
TC_XC1_EXT_EVENT           /* XC1 generate external events */
TC_XC2_EXT_EVENT           /* XC2 generate external events */

```

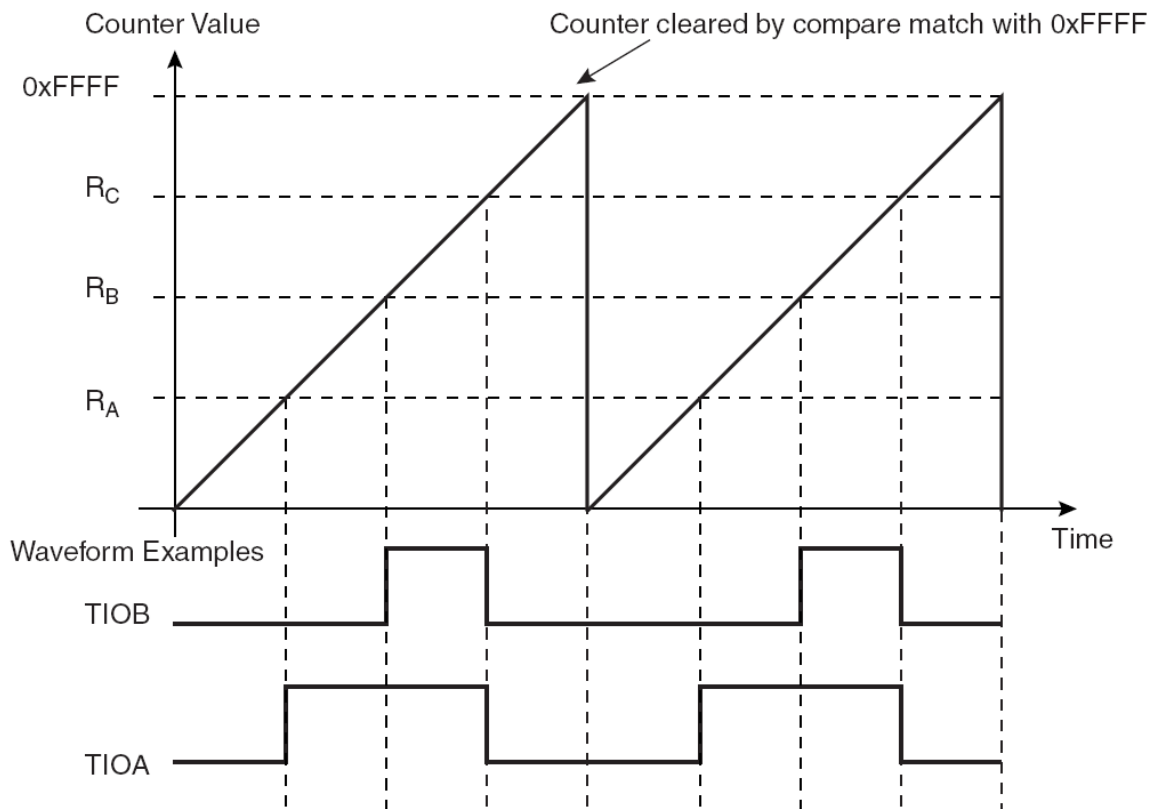


If TIOB is chosen as the external event signal, it is configured as an input and no longer generates waveforms and IRQs.

Then you have to select the external event edge you want to use. To do this, set the `ext_event_edge` field with one of this option:

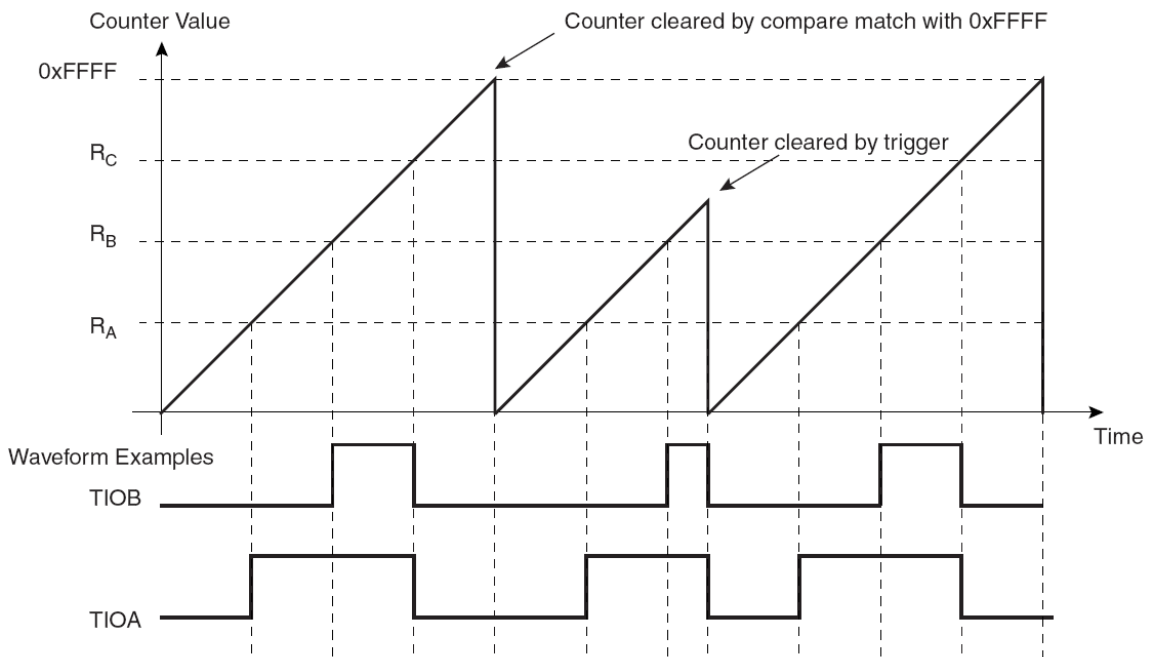
- TC_NONE
- TC_RISING_EDGE
- TC_FALLING_EDGE
- TC_EACH_EDGE

The `wave_sel` field selects the behavior of the 16-bit counter values and consequently the behavior of the TIOA and TIOB outputs, depending on the values of the registers A, B and C. When `wave_sel = TC_UP_MODE` the counter value is incremented from 0 to 0xFFFF, and then it'll be automatically reset. Once the counter value has been reset, it is then incremented and so on. See the following figure:



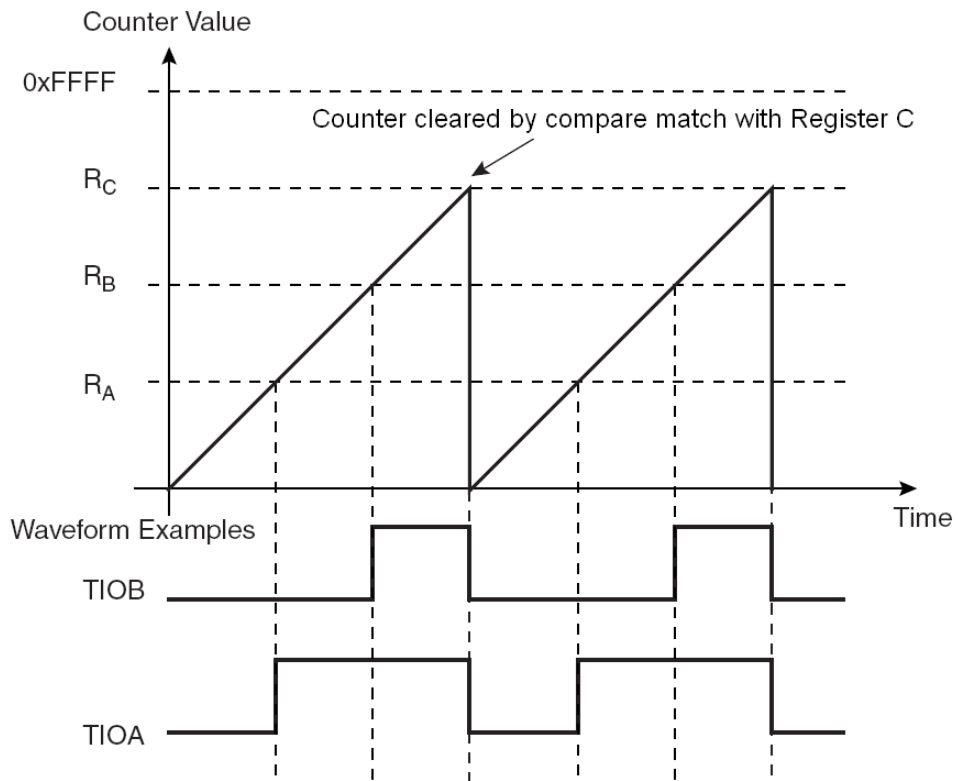
If you use an external trigger, the counter value will be reset and restarted not only when 0xFFFF is reached, but also when a trigger occurs:





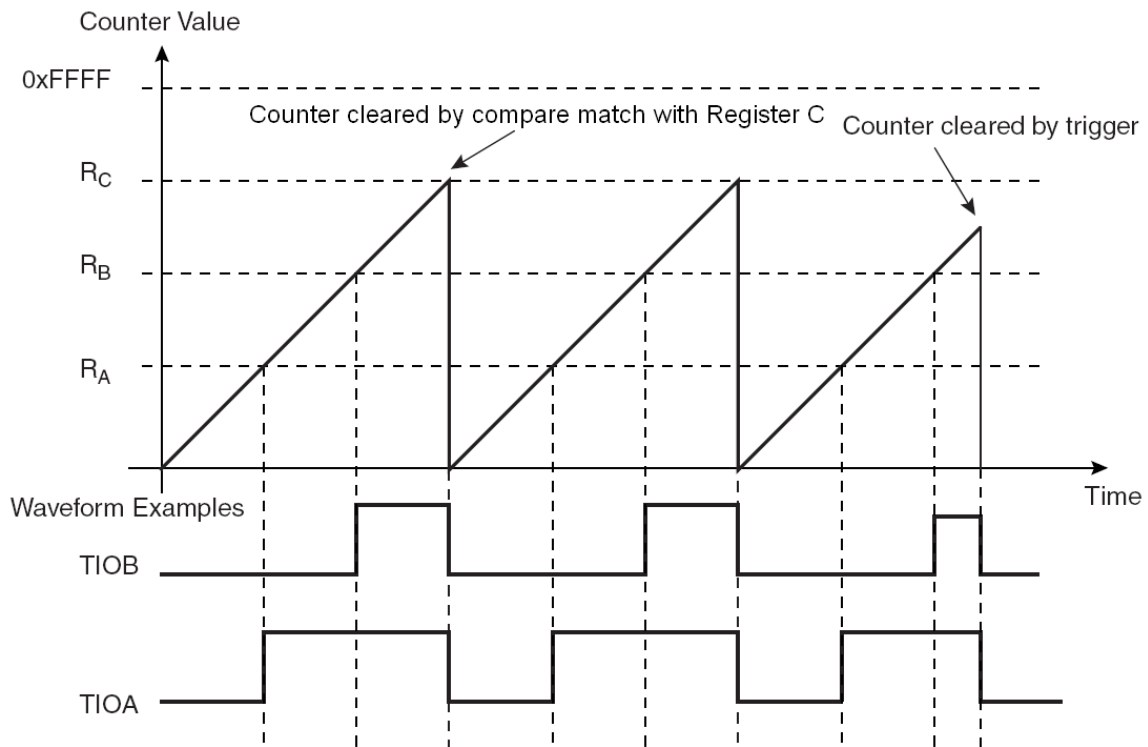
When `wave_sel = TC_UP_MODE_AUTO` the counter value is incremented from 0 to the value of register C, and then it'll be automatically reset. Once the counter value has been reset, it is then incremented and so on. See the following figure:



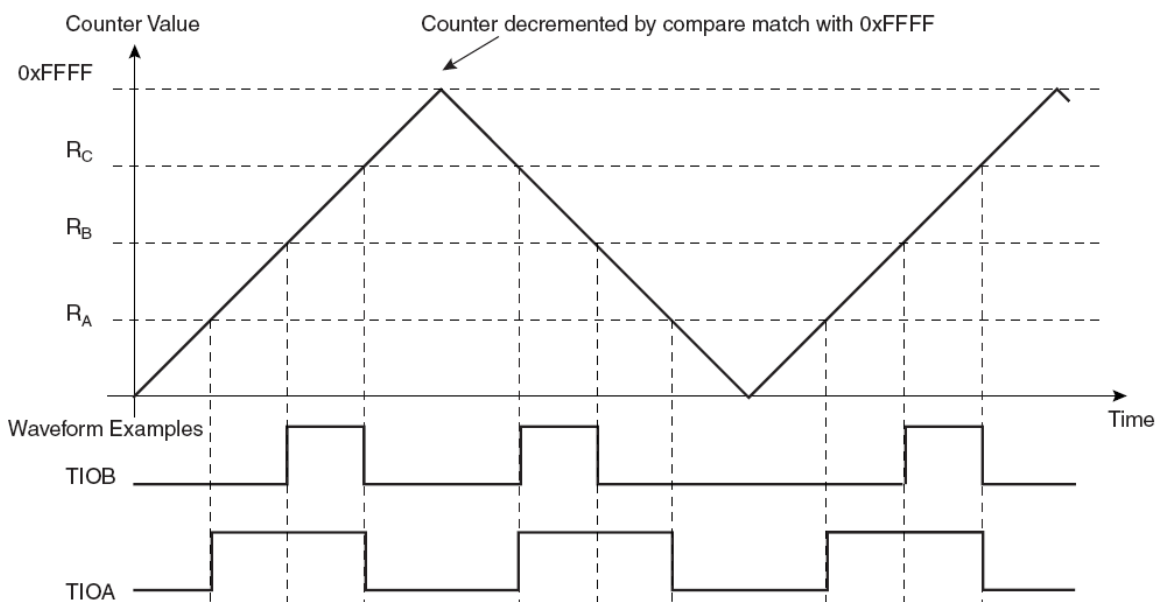


If you use an external trigger, the counter value will be reset and restarted not only when Register C value is reached, but also when a trigger occurs:

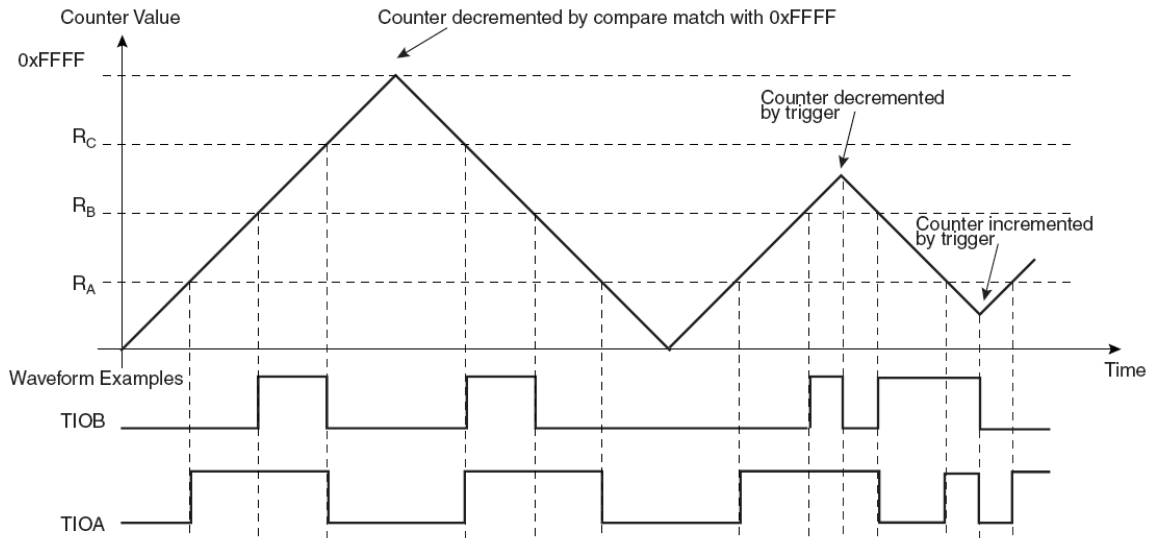




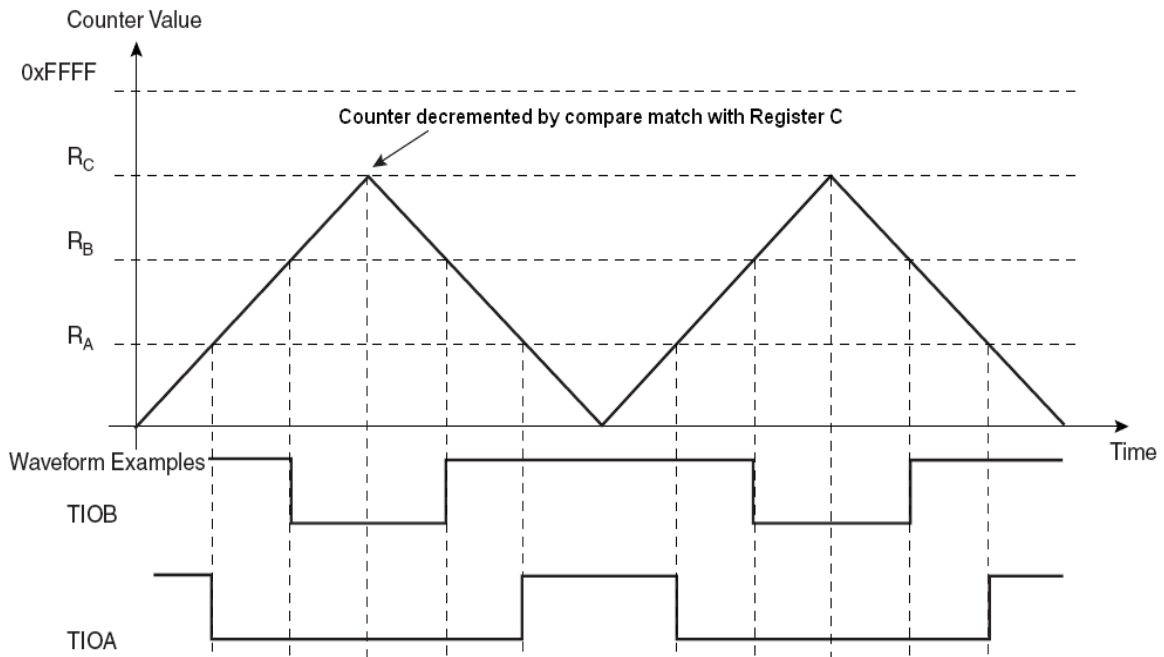
When `wave_sel = TC_UPDOWN_MODE` the counter value is incremented from 0 to 0xFFFF. Once that value is reached, the counter value is decremented to 0, then re-incremented to 0xFFFF and so on:



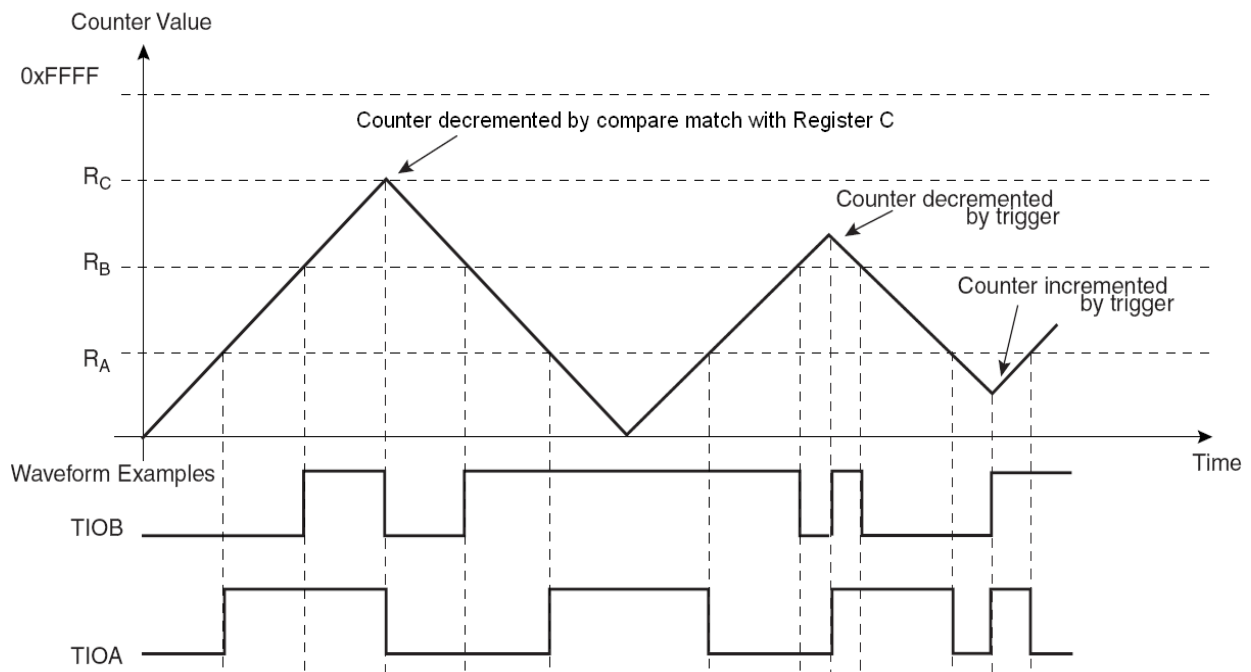
If you use an external trigger, the counter value will be decremented not only when 0xFFFF is reached, but also when a trigger occurs. Furthermore, if the counter value is decrementing and a trigger occurs, the counter value will be incremented:



When `wave_sel = TC_UPDOWN_MODE_AUTO` the counter value is incremented from 0 to the value of register C. Once that value is reached, the counter value is decremented to 0, then re-incremented to Register C value and so on:



If you use an external trigger, the counter value will be decremented not only when Register C value is reached, but also when a trigger occurs. Furthermore, if the counter value is decrementing and a trigger occurs, the counter value will be incremented:



The other fields have to be set to specify the effect of the available events on the TIOA and TIOB outputs. The possible values for these fields are:

```

TC_NONE
TC_SET      /*set the output to 1 */
TC_CLEAR    /*set the output to 0 */
TC_TOGGLE   /*set the output to 0 if it's 1, otherwise to 1*/

```

For example, if we want TIOA signal to become high when the timer counter reaches the value stored in Register C, we'll have to put `rc_2_tioa = TC_SET`; if we want TIOB signal to be low when a software trigger occurs, we'll have to put `swttrg_2_tioa = TC_CLEAR`; if we want TIOA to change its state from high to low and from low to high when an external event occurs, we'll put `extevnt_2_tioa = TC_TOGGLE`; if the other events don't care, we'll put `ra_2_tioa = TC_NONE`, `swttrg_2_tioa = TC_NONE`, etc...

In this example of use for the Waveform Mode we want to generate a regular output signal on the TIOA line, so we disable the external events, set the up mode and set the TIOA output to 1 when the counter reaches the Register A value and to 0 when it reaches the Register C value:

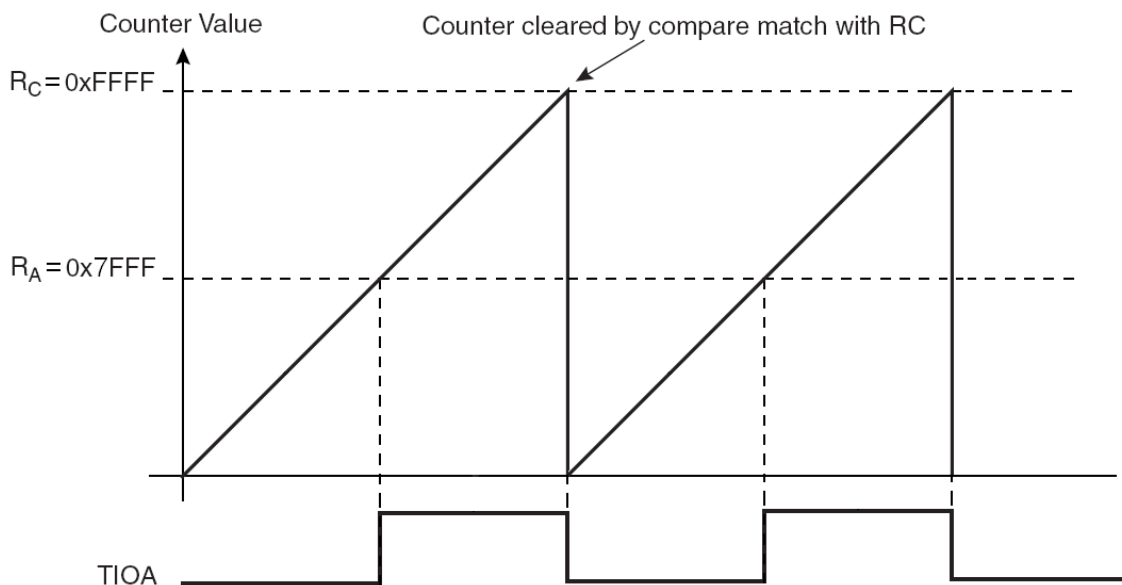


```

struct tc_waveform_mode cmd;
cmd.ext_event_edge = TC_NONE;
cmd.wave_sel = TC_UP_MODE;
cmd.ra_2_tioa = TC_SET;
cmd.rc_2_tioa = TC_CLEAR;
ret=ioctl(fd, TC_SET_WAVEFORM_MODE, &cmd);
if (ret < 0)
    printf("An error occurred in ioctl");

```

Besides we have to set Register C to 0xFFFF and Register A to 0x7FFF. The following figure shows the waveform generated with the above code:



- TC_GET_MODE

The TC_GET_MODE command allows the user to know the current mode (capture or waveform).

```

int cmd;
ret=ioctl(fd, TC_GET_MODE, &cmd);
if (ret < 0)
{
    printf("An error occurred in ioctl");
}
else{
    if(cmd == TC_CAPTURE_MODE)
    {

```



```

        printf("Capture Mode");
    }
    if(cmd == TC_WAVEFORM_MODE)
    {
        printf("Waveform Mode");
    }
}

```

- TC_GET_CAPTURE_SETTINGS

The TC_GET_CAPTURE_SETTINGS command allows the user to know the current TC settings (only if the current mode is Capture Mode).

```

int cmd;
struct tc_capture_mode tccm;
ret=ioctl(fd, TC_GET_MODE, &cmd);
if (ret < 0)
{
    printf("An error occurred in ioctl");
}
else{
    if(cmd == TC_CAPTURE_MODE)
    {
        ret=ioctl(fd, TC_GET_CAPTURE_SETTINGS, &tccm);
    }
}

```

- TC_GET_WAVEFORM_SETTINGS

The TC_GET_WAVEFORM_SETTINGS command allows the user to know the current TC settings (only if the current mode is Waveform Mode).

```

int cmd;
struct tc_waveform_mode tcwm;
ret=ioctl(fd, TC_GET_MODE, &cmd);
if (ret < 0)
{
    printf("An error occurred in ioctl");
}
else{
    if(cmd == TC_WAVEFORM_MODE)
    {
        ret=ioctl(fd, TC_GET_WAVEFORM_SETTINGS, &tcwm);
    }
}

```



- TC_SET_CLOCK, TC_GET_CLOCK

The timer counter needs a clock pulse to work, so you have to choose the clock you want through the TC_SET_CLOCK command, which also allows the user to set other options. To do this, you have to use the `ioctl` call with this argument

```
struct tc_clock{
    short sel_clk;
    short clk_invert;
    short burst;
    short stop_on_load_RB;
    short disable_on_load_RB;
    short stop_on_RC_compare;
    short disable_on_RC_compare;
};
```

The `sel_clk` field must be set with the selected clock, chosen among an internal clock

```
TC_TIMER_CLOCK1      /* master clock / 2 */
TC_TIMER_CLOCK2      /* master clock / 8 */
TC_TIMER_CLOCK3      /* master clock / 32 */
TC_TIMER_CLOCK4      /* master clock / 128 */
TC_TIMER_CLOCK5      /* slow clock */
```

or an external clock

```
TC_TCLK0             /* external clock 0 */
TC_TCLK1             /* external clock 1 */
TC_TCLK2             /* external clock 2 */
TC_TIOA0
TC_TIOA1
TC_TIOA2
```

In the case you choose an external clock, you must consider that for the Timer Counter `i` you can use only the `TC_TCLKi` and you can't use the `TC_TIOAi`. For example, the external clocks available for Timer Counter 1 are `TC_TCLK1`, `TC_TIOA0` and `TC_TIOA2`. The external clock chosen will be called `TC_XCi`. By default, `TC_XC0 = TC_TCLK0`, `TC_XC1 = TC_TCLK1`, `TC_XC2 = TC_TCLK2`.

With the `clk_invert` field you can decide to increment the counter on rising edge of the clock (if `clk_invert = 0`), or on the falling edge (if `clk_invert = 1`).

The `burst` field can be set to AND the chosen clock with an external signal (`XC0`, `XC1` or `XC2`). Use the following macros:



```

    TC_NONE          /* Clock isn't gated by an external signal
*/
    TC_XC0           /* the clock is ANDed with external clock 0
*/
    TC_XC1           /* the clock is ANDed with external clock 1
*/
    TC_XC2           /* the clock is ANDed with external clock 2
*/

```

The `stop_on_load_RB` and `disable_on_load_RB` fields work in Capture Mode only. If one of these fields is set to 1, whenever the counter value is loaded on register B the counter will be respectively stopped or disabled. The difference is that if the counter is stopped it can be restarted through a (hardware or software) trigger, while if the counter is disabled it can be re-enabled only with an `ioctl` call with the `TC_ENABLE` option.

The `stop_on_RC_compare` and `disable_on_RC_compare` fields are similar to the previous and work in Waveform Mode only. If one of these fields is set to 1, whenever the counter value reaches the value stored in register C the counter will be respectively stopped or disabled.

If you want to know the current configuration of the clock, type the command:

```

struct tc_clock tcc;
ret=ioctl(fd, TC_GET_CLOCK, &tcc);
if (ret < 0)
{
    printf("An error occurred in ioctl");
}
else
{
    /* your code */
}

```

- `TC_ENABLE`

After choosing the Timer Counter mode and the clock, you have to enable the Timer if you want to start working. To do so, use the command:

```

ret=ioctl(fd, TC_ENABLE, 1);
if (ret < 0)
{
    printf("An error occurred in ioctl");
}

```



To disable the clock, use the command

```
ret=ioctl(fd, TC_ENABLE, 0);
if (ret < 0)
{
    printf("An error occurred in ioctl");
}
```

When the Timer Counter is disabled it can't be started or stopped.

- TC_SOFTWARE_TRIGGER

A software trigger has the effect of resetting the counter and restarting it from zero. To give a software trigger to the timer Counter, type the command:

```
ret=ioctl(fd, TC_SOFTWARE_TRIGGER, 0);
if (ret < 0)
{
    printf("An error occurred in ioctl");
}
```

A software (or hardware) trigger doesn't have any effect if the timer is not enabled.

- TC_GET_CV

An `ioctl` with this option returns the current value of the Timer Counter and gets the number of times the Counter reached the maximum value and restarted (since the last software trigger or enable condition). The used structure is:

```
struct tc_value{
    int value;
    unsigned long wraparounds;
};
```

The `value` field can reach the maximum value (0xFFFF) if no trigger on Register C compare was previously enabled (see `TC_SET_CAPTURE_MODE` and `TC_SET_WAVEFORM_MODE` `ioctl` options in this guide). Otherwise, the maximum reachable value is equal to the Register C value.

The `wraparounds` field contains the times the maximum value was reached since the last software trigger or "enable" condition.

This is an example of use where the maximum reachable value is 0xFFFF:



```
struct tc_value cmd;
ret=ioctl(fd, TC_GET_CV, &cmd);
if (ret < 0)
{
    printf("An error occurred in ioctl");
}
else{
printf("The Counter Value is %d",cmd.wraparounds * 0xFFFF +
cmd.value);
}
```

- TC_SET_RA, TC_GET_RA, TC_SET_RB, TC_GET_RB, TC_SET_RC, TC_GET_RC

When the Timer Counter is in Waveform Mode you need to use Register A, Register B and Register C to perform comparison with the Counter Value. You can set their value in a range between 0 and 0xFFFF. This is an example of use:

```
ret=ioctl(fd, TC_SET_RA, 0x7FFF);
if (ret < 0)
{
    printf("An error occurred in ioctl");
}
```

Register A and Register B can be set only in Waveform Mode.

When the Timer Counter is in Capture Mode the counter value can be stored in Register A or Register B when a programmable event occurs. Follow this example to read a register value:

```
int value;
ret=ioctl(fd, TC_GET_RA, &value);
if (ret < 0)
{
    printf("An error occurred in ioctl");
}
else{
    printf("The Register A Value is %d",value);
}
```

Register A and Register B can be read both in Waveform Mode and in Capture Mode. Register C can be set and read in both Waveform Mode and in Capture Mode.



4.14.4. close()

The *close()* function shall deallocate the file descriptor indicated by *fil-des*. To deallocate means to make the file descriptor available for return by subsequent calls to *open()* or other functions that allocate file descriptors.

Header file:

unistd.h

Prototype:

int close(int fil-des);

Parameters:

fil-des – file descriptor

Returns:

0 if successfull, -1 otherwise

Example:

Close the TC device.

```
if(close(fd) < 0)
{
    /* Error Management Routine */
} else {
    /* File Closed */
}
```



ensure it is operating with the right type of card. After a successful reset, it is possible to:

- format the card
- write data into the card
- read data written into the card
- unformat the card

Optionally, once smartcard services are not needed anymore, the `iso7816_cleanup()` function may be called.

5.1.1. Defines

16-bit identifier of on-board file:

```
#define ISO7816_FILE_ID 0x1234
```

Predefined size of the on-board file:

```
#define ISO7816_FILE_SZ 1024
```

5.1.2. Types

The basic type used by all of the ISO7816 library functions is defined as follows:

```
typedef struct _iso7816_slot iso7816_slot
```

where struct `_iso7816_slot` is:

```
struct _iso7816_slot
{
    int fd;                /**< File descriptor **/
    int reset;            /**< If the card was reset **/
    unsigned int pos;    /**< Position of the next read/write
                        operation **/
};
```

An `iso7816_slot` variable must be always initialized through a call to `iso7816_init()` before calling any other function. Internal fields of the structure must be never accessed directly but only through an API call.

5.1.3. Enums

The enum `iso7816_rv` is the ISO7816 Library error code:

```
/** ISO7816 Library error code **/
typedef enum iso7816_rv
{
    /** Operation successfully completed **/
```



```

ISO7816_OK = 0,
/** Operation failed because there is no card,
** or the card is of the wrong type **/
ISO7816_E_NOCARD      = -1,
/** Operation failed because parameters are not valid **/
ISO7816_E_INVALID    = -2,
/** Operation failed because of general I/O issues **/
ISO7816_E_IO         = -3,
/** Operation failed because inconsistent with the
** current card or library status **/
ISO7816_E_STATUS     = -4,
/** Operation failed because of a lack of authorization **/
ISO7816_E_UNAUTHORIZED = -5,
/** Not enough on-board EEPROM **/
ISO7816_E_EEPROM_MEMORY = -6
} iso7816_rv;

```

5.1.4. Functions

5.1.4.1. iso7816_strerror()

The iso7816_strerror() function turns into strings ISO7816 Library error codes.

Header file:

iso7816.h

Prototype:

const char * iso7816_strerror(int rv)

Parameters:

rv – return value of another library function

Returns:

one of the string contained into the following:

```

static char *iso7816_errors[] = {
    "Operation successfully completed",
    "No card detected, or wrong card type",
    "Invalid parameters",
    "General I/O error",
    "Requested operation inconsistent with status",
    "Unauthorized",
    "No EEPROM space"
};

```

or "Unknown error" string for an unknown error



Example:

```
int rv;
iso7816_slot slot;
rv = iso7816_cleanup(&slot);
if (rv != ISO7816_OK)
    fprintf(stderr, "Unrecoverable error: %s\n",
iso7816_strerror(rv));
```

5.1.4.2. iso7816_init ()

The iso7816_init() function initializes the serial communication with the ISO7816 slot.

Header file:

iso7816.h

Prototype:

```
iso7816_rv iso7816_init(iso7816_slot *p_slot, const char *device_name)
```

Parameters:

p_slot – pointer to the iso7816_slot structure to be initialized

device_name – path of the device connected to the smartcard reader (default: /dev/ttyS1)

Returns:

An iso7816_rv error code

Example:

```
int rv;
iso7816_slot slot;

printf("Initializing...\n");
rv = iso7816_init(&slot, "/dev/ttyS1");
if (rv != ISO7816_OK)
    /* Error Management */
```

5.1.4.3. iso7816_reset ()

The iso7816_reset() function resets and identifies the smart-card device type. In order to detect correctly card insertion, iso7816_reset() shall be called after iso7816_cleanup() and iso7816_init().

Header file:

iso7816.h

Prototype:

```
iso7816_rv iso7816_reset(iso7816_slot *p_slot)
```



Parameters:

p_slot – pointer to the iso7816_slot structure

Returns:

An iso7816_rv error code

Example:

```
int rv;
iso7816_slot slot;

printf("Resetting...\n");
rv = iso7816_reset(&slot);
while (rv != ISO7816_OK)
{
    printf("Waiting for card insertion...\n");
    sleep(1);
    iso7816_cleanup(&slot);
    iso7816_init(&slot, "/dev/ttyS1");
    rv = iso7816_reset(&slot);
};
```

5.1.4.4. iso7816_format ()

The iso7816_format() formats the smart-card device.

Header file:

iso7816.h

Prototype:

iso7816_rv iso7816_format(iso7816_slot *p_slot);

Parameters:

p_slot – pointer to the iso7816_slot structure

Returns:

An iso7816_rv error code

Example:

```
int rv;
iso7816_slot slot;

printf("Formatting device...\n");
rv = iso7816_format(&slot);
if (rv != ISO7816_OK)
    /* Error Management */
```



Header file:

iso7816.h

Prototype:

iso7816_rv iso7816_seek(iso7816_slot *p_slot, uint16_t pos)

Parameters:

p_slot – pointer to the iso7816_slot structure
pos – position offset

Returns:

An iso7816_rv error code

Example:

```
int rv, pos;
iso7816_slot slot;

pos = 512;

rv = iso7816_seek(&slot, pos);
if (rv != ISO7816_OK)
    /* Error Management */
```

5.1.4.9. iso7816_cleanup()

The iso7816_cleanup() cleans-up the serial communication with the ISO7816 slot.

Header file:

iso7816.h

Prototype:

iso7816_rv iso7816_cleanup(iso7816_slot *p_slot)

Parameters:

p_slot – pointer to the iso7816_slot structure

Returns:

An iso7816_rv error code

Example:

```
int rv;
iso7816_slot slot;

printf("Cleaning up...\n");
rv = iso7816_cleanup(&slot);
```



```
if (rv != ISO7816_OK)
    /* Error Management */

printf("Exiting...\n");

return 0;
```



6. CMUX

GE863-PRO³ implements the GSM 7.10 multiplexing protocol that enables one serial interface to transmit data to different customer applications. Using the multiplexer features, GE863-PRO³ can perform e.g. a fax/data/GPRS call while using the SMS service.

Cmux protocol can be enabled by starting `cmuxt` daemon once the modem has been turned on, with the options shown below:

Synopsis

`cmuxt` [parameters]

Parameters

-p <serport> : Serial port device to connect to [default value: /dev/ttyS1]
-b <baudrate> : MUX mode baudrate [0,9600,19200,...115200]
-t <delay> : Escape Prompt Delay expressed in fiftieth of a second [20,...255]
-e <char> : Escape Character [0,...255]
-u <sleep time> : Sleep time after send AT command (msec)
-c : (crtcts) Hw control OFF
-d : Daemonize: create a daemon
-l : Don't make pts symLinks. Links needs root privileges
-v : Show cmuxt version
-h -? : Show this help message

Please note that if **-t** and/or **-e** options are not used to specify different values for Escape Prompt Delay and Escape Character respectively, the following default values will be used (please see [7]):

- 50 (1 second) for Escape Prompt Delay
- 43 ("+") for Escape Character

In any case, at `cmuxt` termination, the startup values (the ones set into modem before `cmux` running) for the Escape Prompt Delay and Escape Character will be restored.

Since GE863-PRO³ use /dev/ttyS3 device to access to modem, `cmuxt` is called typing:

```
# cmuxt -p /dev/ttyS3 -b 115200 -d
```

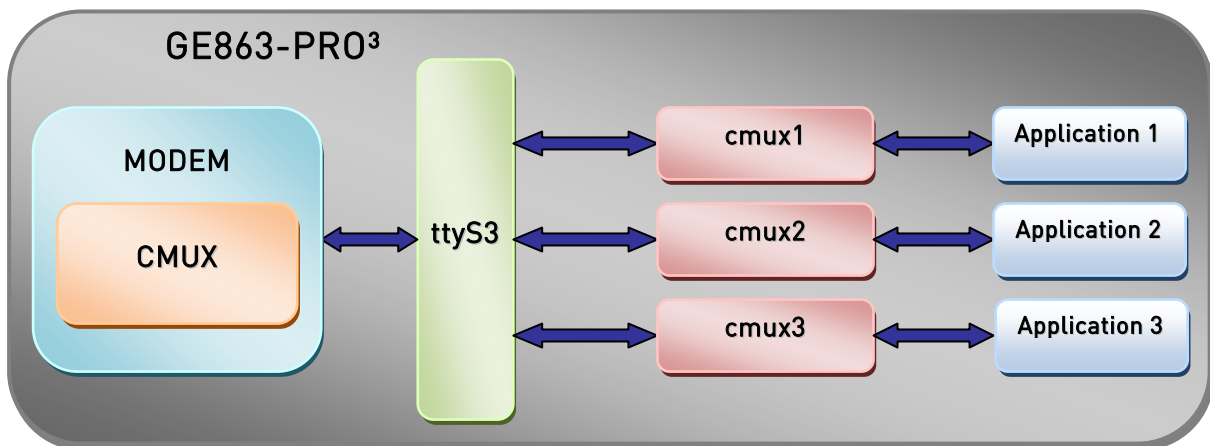


Once cmuxt has been started the following four devices are created into /dev directory:

```
/dev/cmux1
/dev/cmux2
/dev/cmux3
/dev/cmux4
```

The devices above are the serial virtual channels that have to be accessed by applications in order to perform multiple operations at the same time (see figure below).

Please note: virtual port cmux4 is reserved and cannot be used.



6.1. Code example

Cmux virtual channels can be simply managed as a serial device as shown in the example below:

- Open the virtual channel port (for example /dev/cmux1):

```
int fdPts;
cmux_port[] = "/dev/cmux1";
struct termios serCfg;

if((fdPts = open(cmux_port, O_RDWR)) < 0)
    return -1;
else if(tcgetattr(fdPts, &serCfg)!= 0)
    return -1;
```



7.2. Erasing the flash memory

This step shall be performed when the flash is used for the first time or whenever it shall be erased⁴.

`flash_eraseall` command performs erasing on every flash partition.

`flash_eraseall` command is already present in the delivered Linux file system and takes the parameter `mtDX` (number associated to the flash partition to erase).

The association between the number `X` and the flash partition can be shown typing `cat /proc/mtd` on shell:

```
# cat /proc/mtd
dev:   size  erasesize  name
mtd0: 00131400 00000210 "ARMboot"
mtd1: 002eec00 00000210 "root"
mtd2: 00400000 00010000 "spi0.0"
mtd3: 00800000 00010000 "spi0.2"
#
```

In this example, `mtd2` is associated to the flash connected to chip select 0 on spi bus 0 (M25P32) and `mtd3` is associated to the flash connected to chip select 2 on spi bus 0 (M25P64).

So to use `flash_eraseall` command just type:

```
# flash_eraseall /dev/mtdX
```

where `X` shall be replaced with the number, retrieved from the previous command, associated to the flash partition to erase.

After a few seconds all the flash will be erased and will be ready to be used.

7.3. Mounting a JFFS2 file system

Once the flash has been erased, it is possible to mount and use the new file system located over the external flash.

Type the command:

⁴ Please note that once the erased procedure is completed all the data contained in the flash will be lost.



