GE863-PRO^3 Linux Software User Guide

1vv0300781 Rev. 0 - 04/06/08

Making machines talk.

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.

Applicable Products

Product	Part Number
GE863-PRO^3 with Linux OS	3990250698

Linux SW Version
01.0001

Contents

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

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

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

TS-EMEA@telit.com or http://www.telit.com/en/products/technical-support-center/contact.php

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-PRO^3 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-PRO^3 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-PRO^3 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-PRO^3 hardware resources and peripherals.
- "Chapter 5, CMUX", describes GSM 7.10 multiplexing protocol implemented by GE863-PRO³ and its use.

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 1VV0300776, TelitGE863PRO3_Linux_Development 1VV0300780 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-PRO^3.

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 1vv0300773a
[2] TelitGE863PRO3 U-BOOT Software User Guide 1vv0300777
[3] TelitGE863PRO3 EVK User Guide 1VV0300776
[4] TelitGE863PRO3 Linux Development Environment 1VV0300780
[5] TelitGE863PRO3 Linux GSM Library User Guide 1vv0300782
[6] TelitGE863PRO3 Product Description 80285ST10036a

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

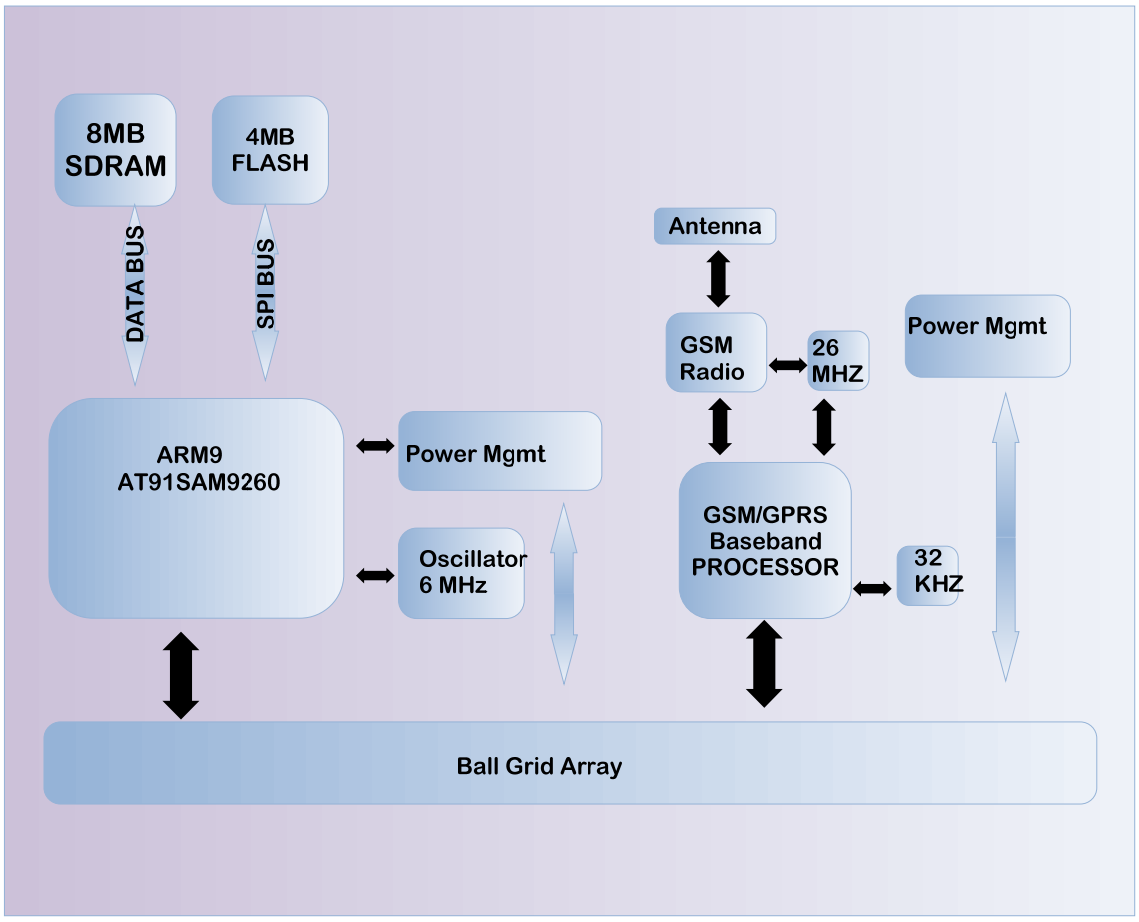
1.9 Document History

Revision	Date	Changes
ISSUE #0	04/06/08	First Release

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



- Atmel AT91SAM9260 microcontroller (200 Mips at 180Mhz, MMU, EmbeddedICE Debug Communication Channel Support).
- 8MB SDRAM Mobile Micron.
- 4MB Serial Flash Atmel via SPI interface.
- Quad-band EGSM 850 / 900 / 1800 / 1900 Mhz.
- 4 A/D with A/D trigger and 6 PWM D/A converters.
- 1 USB device and 2 USB host.
- 1 IIC interface.
- 2 SPI bus interface
- 1 SSC digital audio interface.
- 1 image sensor interface
- 6 USARTs and 1 UART
- 1 MMC and SD cards interface.

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 applications. With its extended programming capabilities in C/C++ and Python, the Telit GE863-PRO³ is the ideal hardware platform for complete and compact customer solutions.

Application development is accomplished easily given Telit's continued commitment to open systems. With the use of LINUX, developers have access to an extensive library of drivers for different peripherals and to complete development environments.

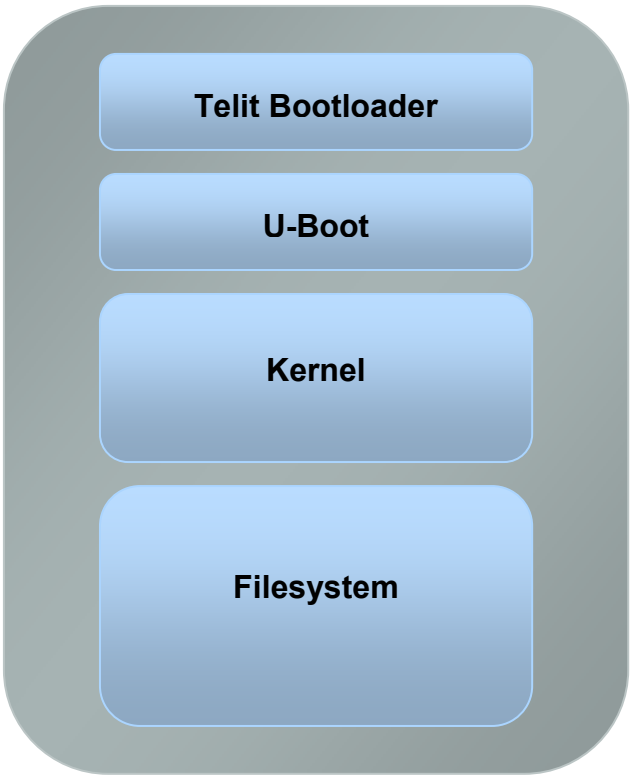
The GE863-PRO³ was designed to simplify connectivity through the availability of interfaces such as SPI, IIC, SD/MMC and USB (Host/Device). Telit offers a vast collection of reference designs enabling use of the PRO³ with external peripherals such as camera, keyboard, display, Wi-Fi®, Bluetooth®, SmartCard, SD Card, Ethernet, ZigBee® and GPS.

For further information about hardware refer to document [1] .

2.2 Software

The GE863-PRO³ software architecture from a high level perspective is based on the following components:

- Telit Bootloader
- Telit customized U-Boot
- Customized Linux kernel based on 2.6.24
- Filesystem



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 document [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 document [2] .

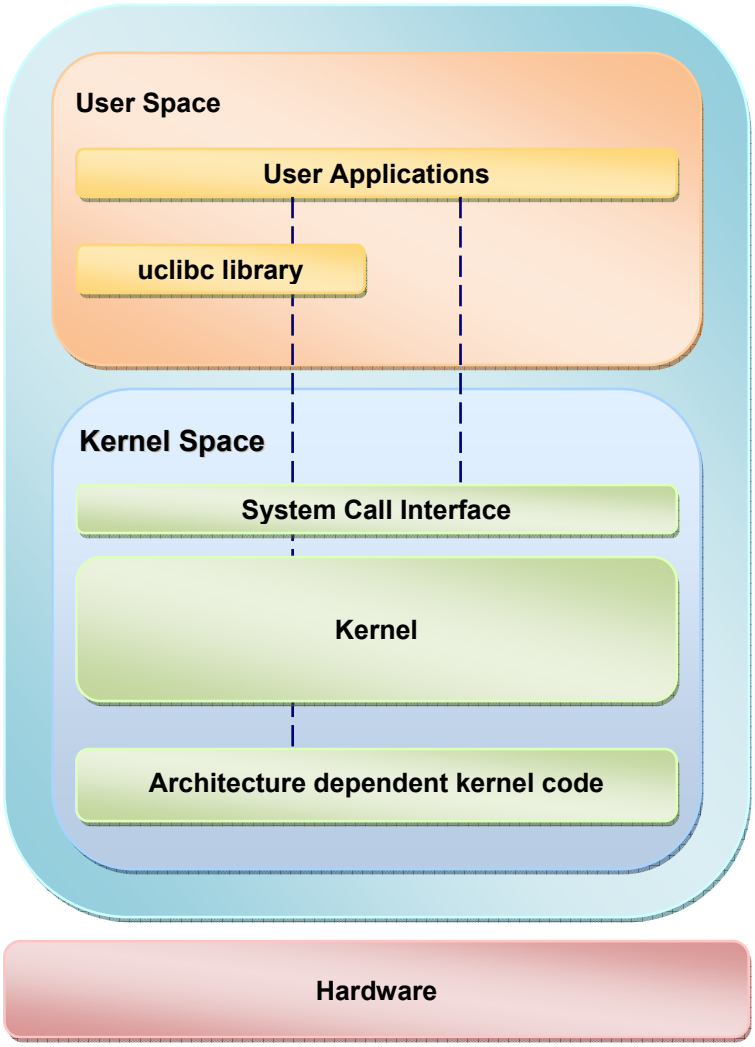
2.2.3 Linux kernel

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.

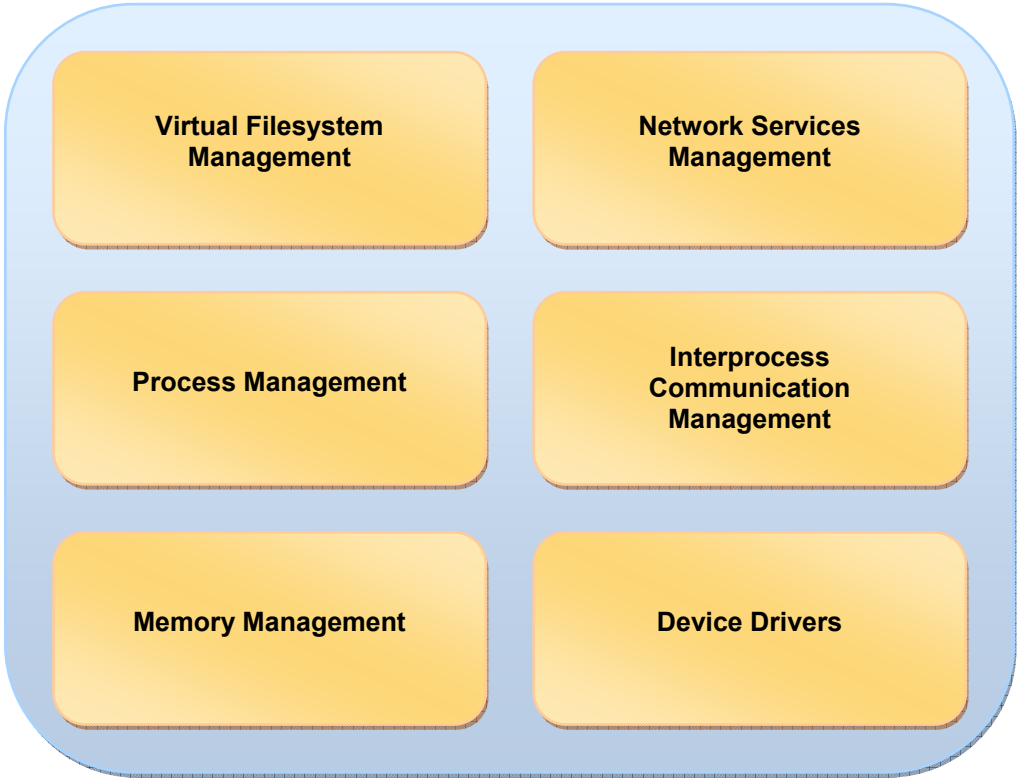
These two regions are separated and have different memory address spaces; there are several methods for user/kernel interaction:

- 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 is allows exploiting all hardware features of the GE863-PRO³.

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

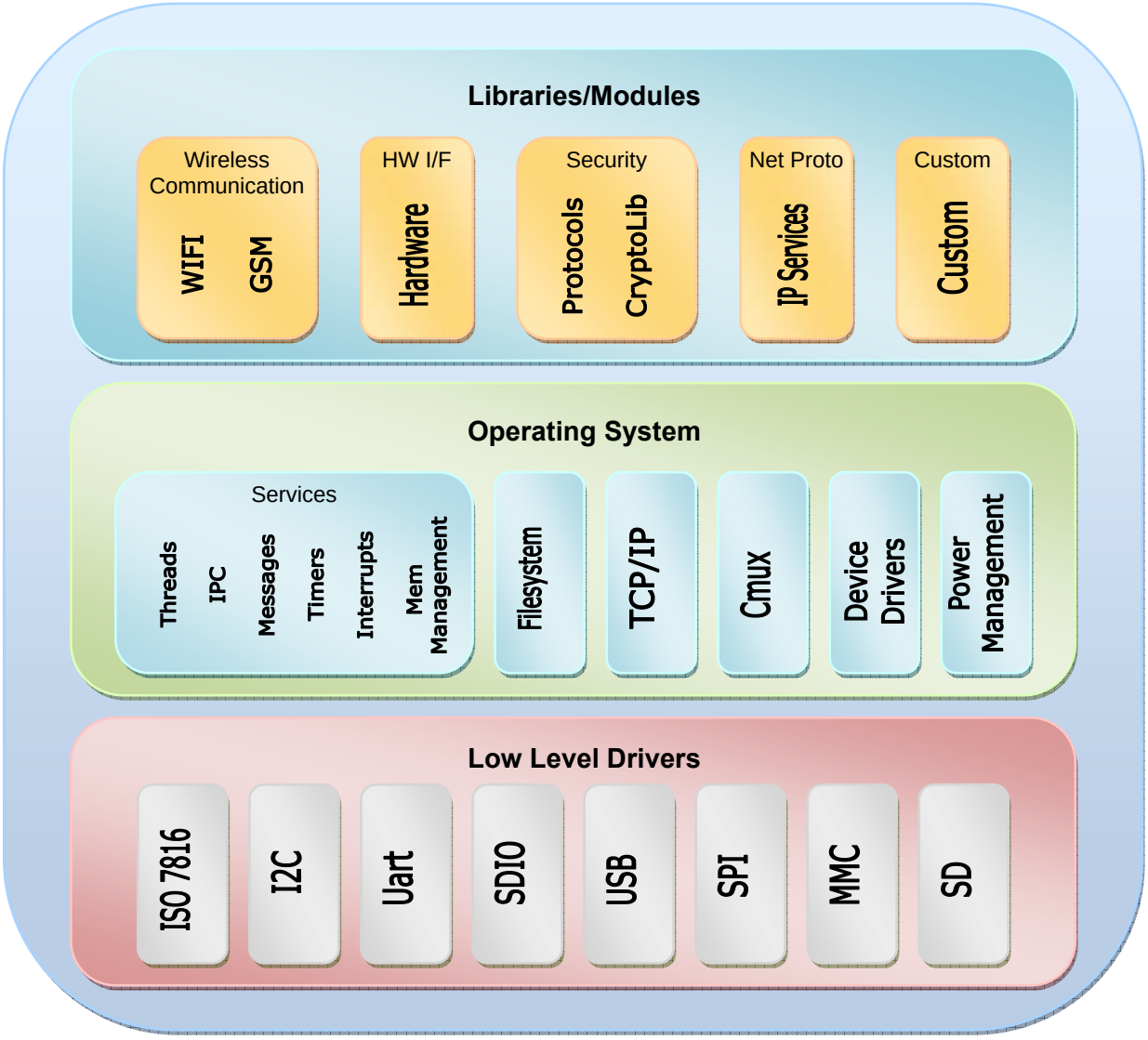


The Linux kernel is a monolithic one (i.e. all OS services run along with the main kernel thread, thus also residing in the same memory area), but it has the capability to dynamically load/unload some of its components called modules.

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
GPIO	N	N/A
MMC/SD	N	N/A
Ethernet	Y	macb
Usb Human Interface Device	Y	usbhid
Support for Host-side USB	Y	ohci-hcd
Support for USB Mass Storage	Y	usb-storage
USB Ethernet Gadget	Y	g-ether
I2C	Y	i2c-core i2c-algo-bit i2c-dev i2c-gpio
SPI	Y	Spidev

2.2.4 Filesystem

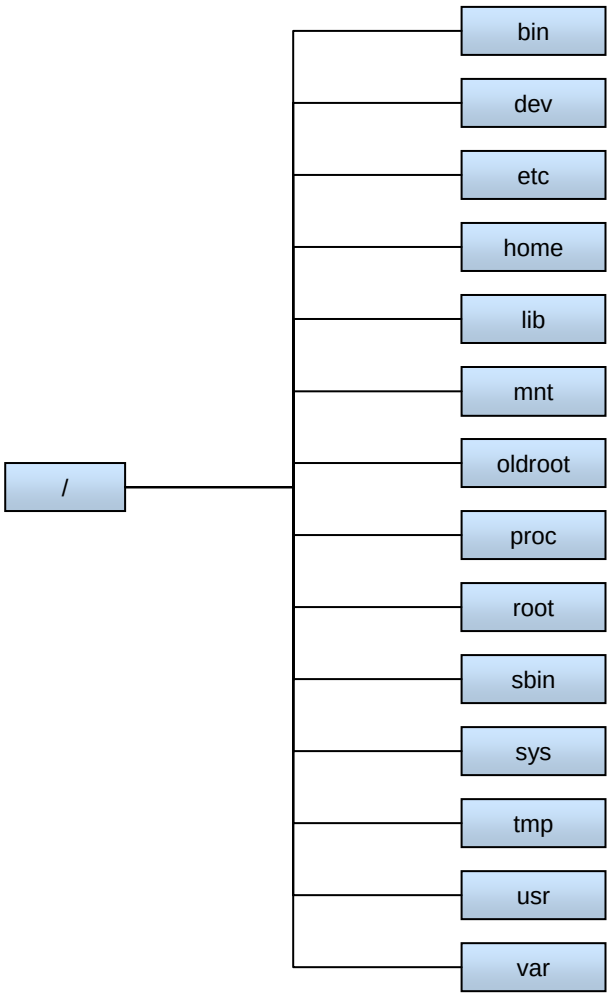
2.2.4.1 Overview

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`

This directory contains the home directories of the various users.

`/lib`

This directory contains shared libraries requiered by programs; for example, here there can be found the uclibc libraries. Moreover in its subdirectories there are the various loadable kernel modules (for example the modules for USB Mass Storage Devices): they can be easily recognized by the .ko extension.

`/mnt`

This directory contains the mount points[1] for the various devices.

`/oldroot`

By default this directory is empty. It is used by the system for the startup process.

`/proc`

This directory contains virtual folders and files which represent the current state of the kernel; for example here you can find info about the cpu (type `cat /proc/cpuinfo`).

`/root`

This directory is the home of the user root. By default it is empty.

`/sbin`

This directory contains executable files are mainly used by the root user for administration task. By default in the GE863-PRO³, most of these files are symbolic links that depend on Busybox (see paragraph 2.2.4.3).

`/sys`

This directory contains system files used for device configuration.

`/tmp`

This directory contains temporary files.

`/usr`

This directory contains files and directories related to user tools and applications. By default in the GE863-PRO³ some of these files are symbolic links that depend on Busybox (see paragraph 2.2.4.3).

`/var`

This directory contains variable data files.

[1] The mount point is the location in the operating system's directory structure where a mounted file system appears

2.2.4.3 Busybox

Inside the filesystem in the directory `/bin` there is the Busybox. Busybox gives the user a set of commands useful to interact with the system; with it the user can carry out normal tasks such as copying files (`cp` command), listing directories (`ls` command), deleting files (`rm` command) etc. etc.

BusyBox combines tiny versions of many common UNIX utilities into a single small executable. It provides replacements for most of the utilities usually found in a GNU/Linux system. Note that the utilities in BusyBox generally have fewer options than their full-featured GNU counterparts; however, the options that are included usually satisfy the user's needs and provide a fairly complete POSIX environment.

Busybox is composed of:

- A single binary placed in `/bin`, which is the real executor of the various utilities.
- A number of symbolic links through all the filesystem, having the name of the various utilities (`cp, ls, kill` etc.) which refer to the Busybox binary.

To discover the currently defined commands for the GE863-PRO³, open a terminal (refer to 3.1 for further details) and type:

```
# cd /bin
# ./busybox
```

To see help content for any command type in the terminal:

```
# <command name> --help
```

3 System Startup

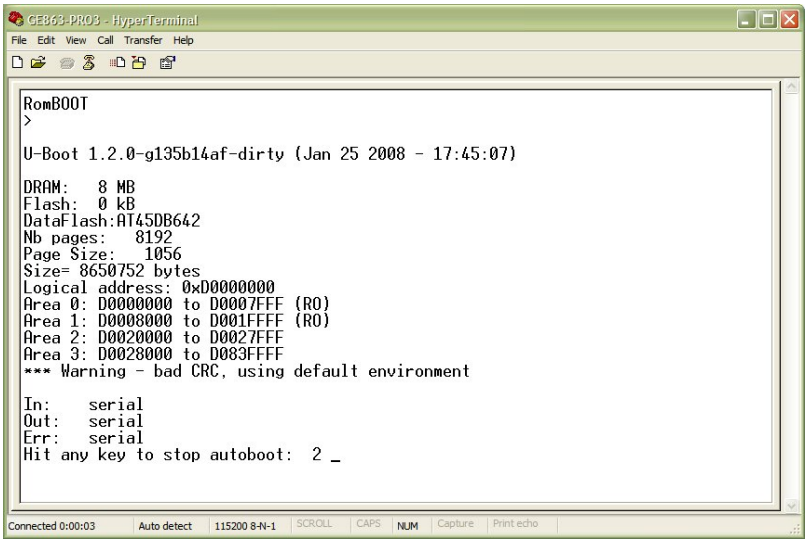
3.1 Startup process

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:

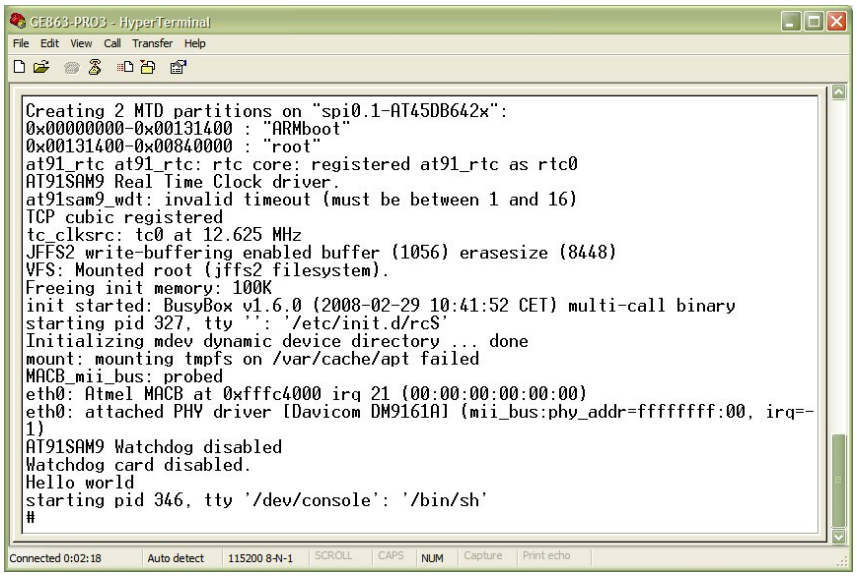
Bits per second: 115200
Data bits: 8
Parity: None
Stop bits: 1
Flow Control: None

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.

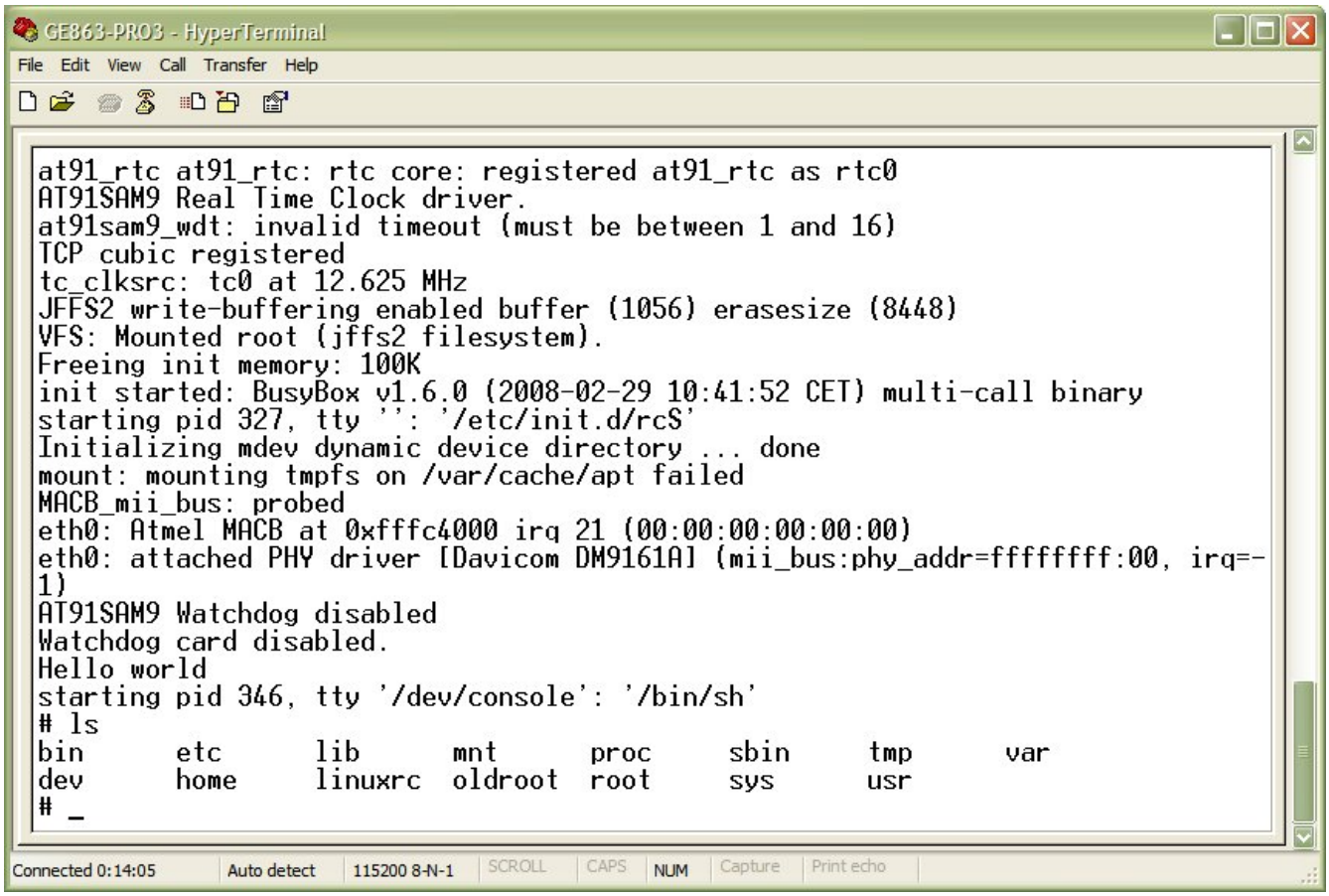
When the target has finished booting, in the terminal the shell prompt will appear:

```
#
```

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.



3.3 Loading a module

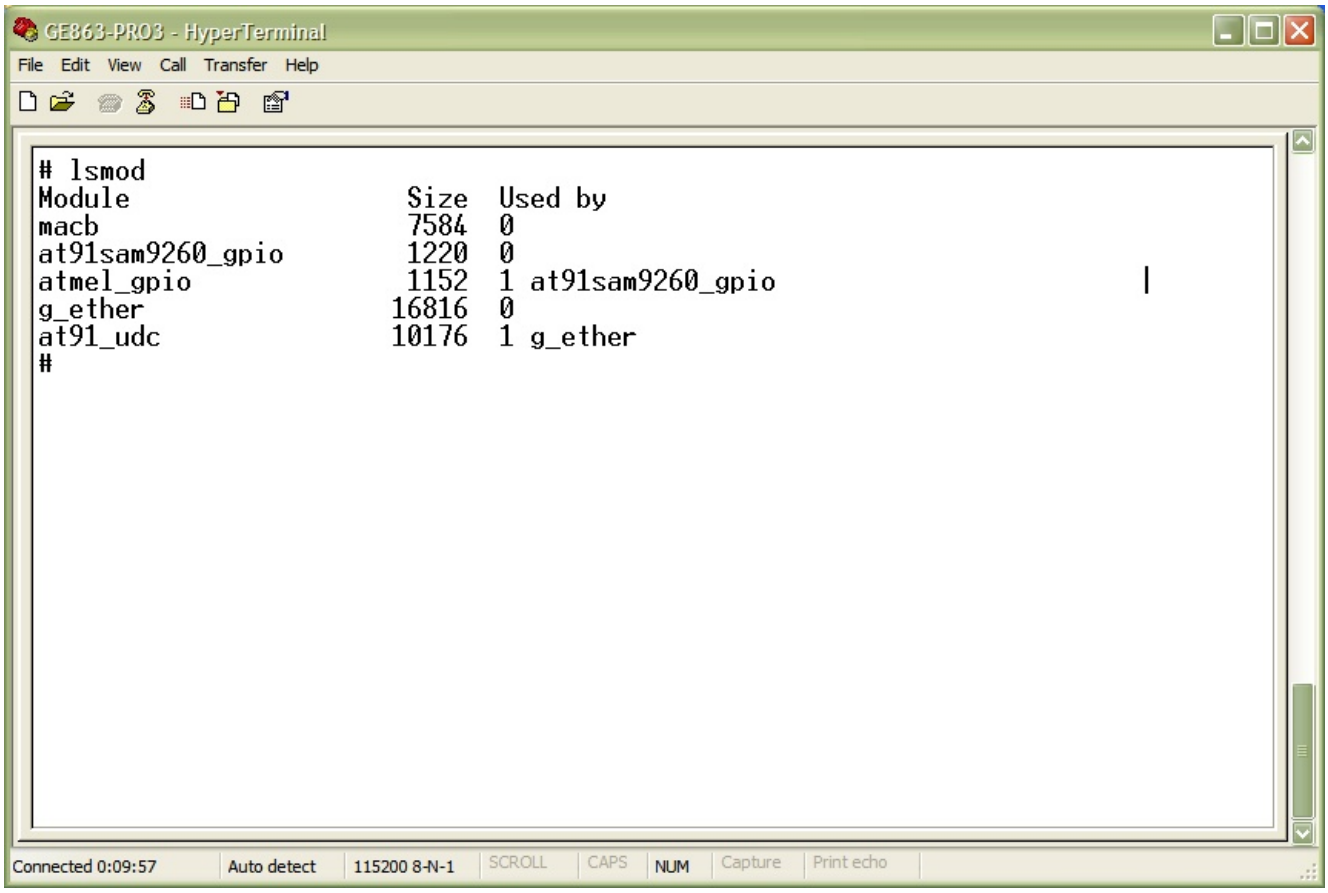
To load a module type in the terminal:

```
# modprobe <module name>
```

refer to 2.2.3.2 for modules' name.

By default `modprobe` tries to load the module from the directory (and subdirectories) where modules are usually stored. Some of the modules are dependent form others and require loading of other modules first. `modprobe` commits to solve these dependencies before loading the specified module.

To see which modules are currently loaded in the kernel use the `lsmod` command as shown below:



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 Downloading a file into GE863-PRO3

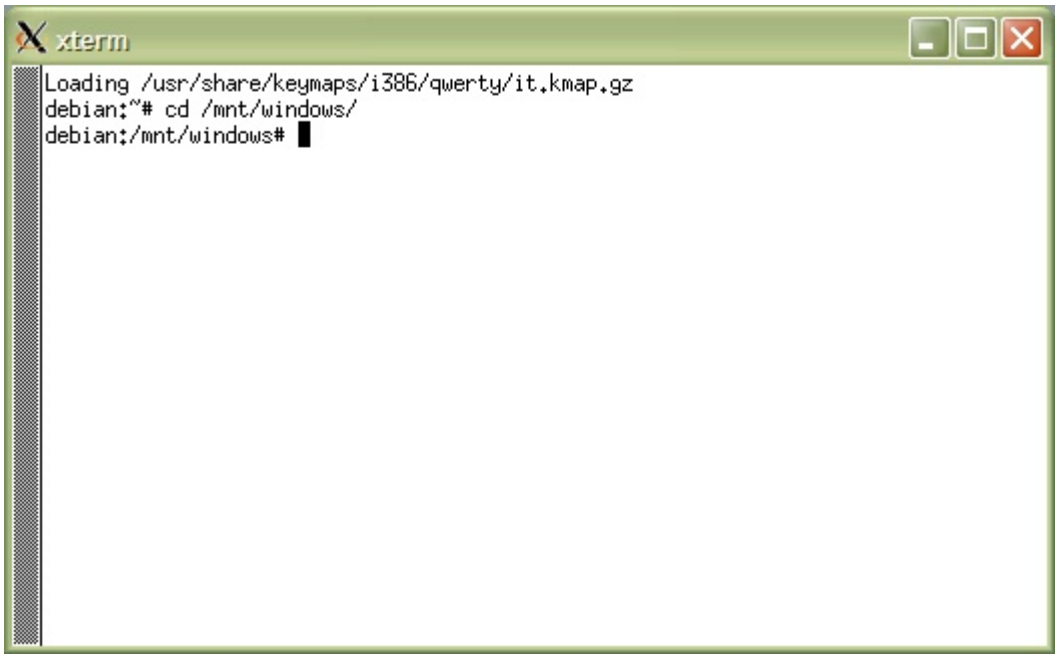
There are several ways to download a file in the target. In the following paragraphs you can find explained two of these methods; for further details refer to document [4] .

3.5.1 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/
```

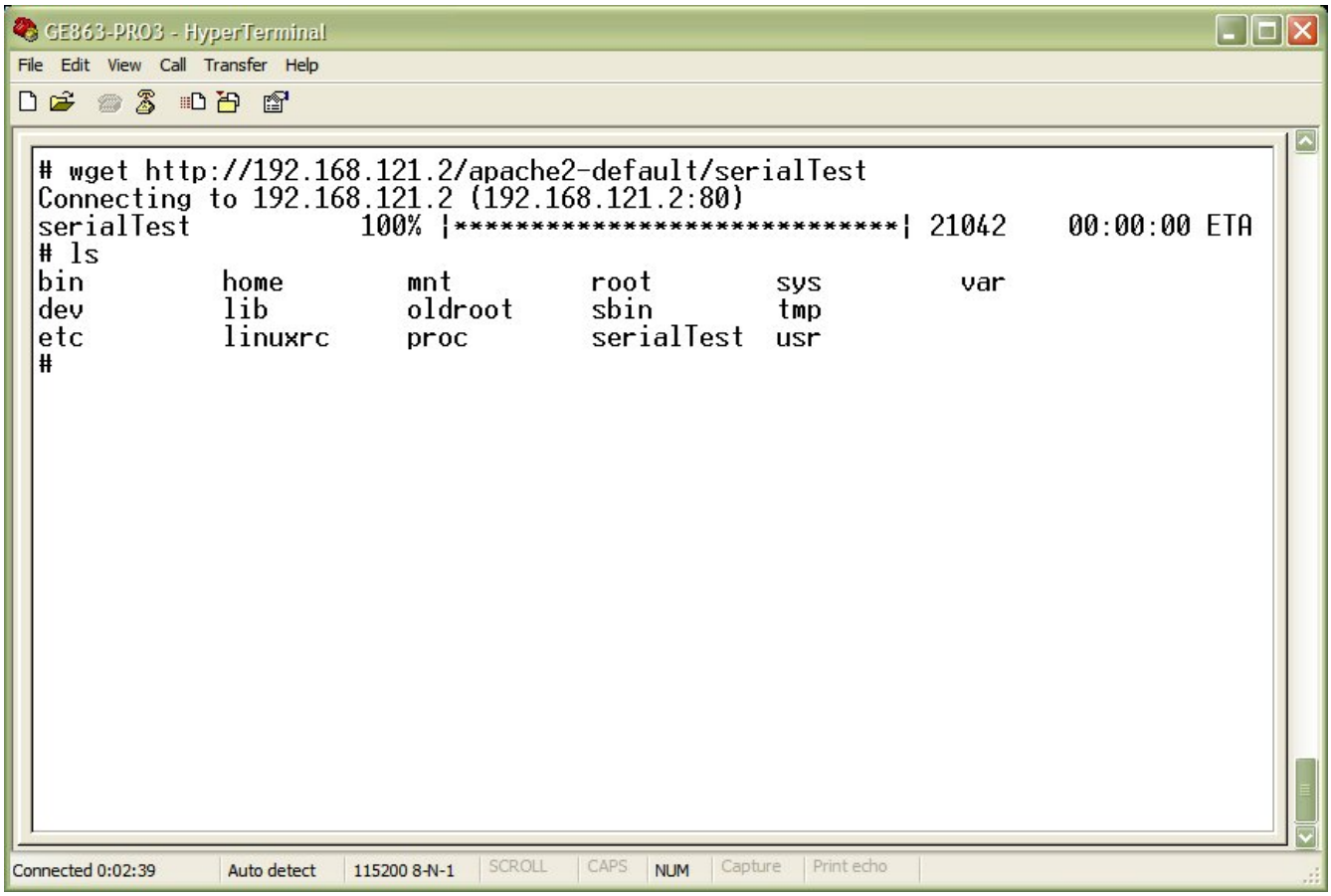
For example:

```
# 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>
```

3.5.2 Downloading a file using an USB mass storage device

Follow the procedure described in 4.8.1 for mounting the device (for example an USB memory key). Then, type in the terminal:

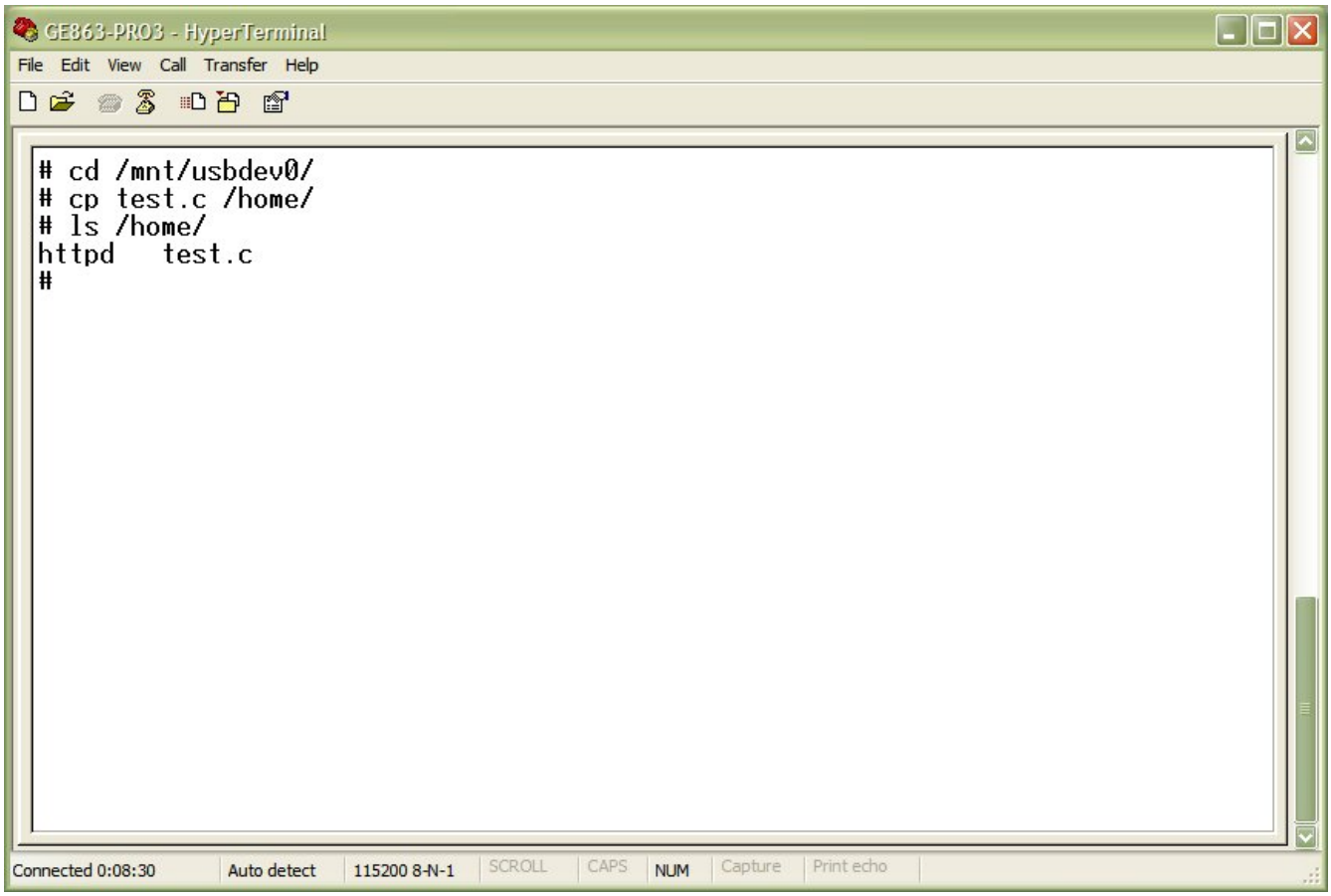
```
# cd /mnt/usbdev0
```

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.

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:

ttyS0 used by the shell
ttyS1 available (e.g. used for debugging)
ttyS2 available
ttyS3 available (e.g. for accessing the GSM modem)

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.

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 descriptor[2] 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

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

[2] Identifies ID of the file

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 */
}
```

4.1.3 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 *sizeof(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, sizeof(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.

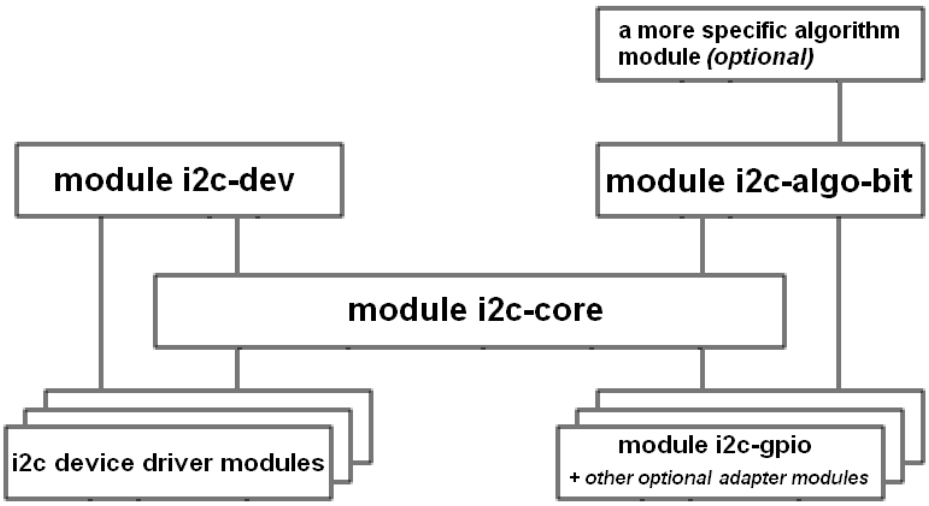
```
if(close(fd) < 0)
{
      /* Error Management Routine */
} else {
            /* File Closed */
}
```

4.2 I²C

The GE863-PRO³ is provided with the Atmel Two-wire Interface (TWI). It is useful to connect several components, like EEPROM and I²C compatible device, using a unique two-wire bus.

In this chapter is described how to load I²C modules on the Linux Operating System and how to do a little test with a C program that opens the I²C bus interface (i.e., /dev/i2c-0) and writes some data on the device.

Although there are many I²C devices and chips, we can use a unique I²C kernel interface composed by a core module used by both master and slave device modules. In the following picture we can see the whole module structure:



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

```
# modprobe i2c-algo-bit
```

3. The i2c-dev module implements a useful tool that allows the user to access all devices on an adapter from userspace, through the `/dev` interface. So we'll have one item in the `/dev` directory for each bus adapter in our system. Each registered i2c adapter gets a number, counting from 0. I^2C device files are character device files with major device number 89 and a minor device number corresponding to the number assigned as explained above. So, for example, we'll have in `/dev` the following files: i2c-0, i2c-1, ..., i2c-10, ... All 256 minor device numbers are reserved for i2c.

To load the i2c-dev module type the command

```
# modprobe i2c-dev
```

4. The i2c-gpio module is an adapter driver or, in other words, a bus controller for the GPIO bus. If we don't load adapter driver we won't be able to find any item in the `/dev` directory. So we have to load one adapter driver for each I^2C bus in our system. In this case, we'll have only one bus and its correspondent driver.

To load the i2c-gpio module type the command

```
# modprobe i2c-gpio
```

5. Of course you'll need to use the I^2C Interface to connect I^2C chips or devices to your system. So, after loading the previous modules, you may also need to load the driver module(s) of your specific device(s). In order to do this, you can use the modprobe command followed by your module's name.

4.2.2 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 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 requests 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/i2c-0.

```
int fd;   // file descriptor for the /dev/watchdog 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 fildes, int request, ...)

Parameters:
fildes – file descriptor
request – device-dependent request code.
The following ioctls request codes can be used for I^2C:

- 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_SMBUS: to use SMBus functionalities
- I2C_RETRIES: to set the number of retries when not acknowledged
- I2C_TIMEOUT: to set the timeout value.

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:

- I2C_SLAVE, I2C_SLAVE_FORCE

Both set or change the slave address for the device opened with `file_descriptor`. The address is passed in the 7 lower bits of the `addr` argument (except for 10 bit addresses, passed in the 10 lower bits). With the I2C_SLAVE_FORCE request, the address passed is used as slave address even in case it's already in use by another program or driver.
Example:

```
long addr = 0x5;
ret = ioctl(file_descriptor,I2C_SLAVE, addr);
```


```
if (ret < 0)
          printf("An error occurred in ioctl");
```

- I2C_TENBIT

Selects ten bit addresses if the “select” argument does not equal 0, selects normal 7 bit addresses if the “select” argument equals 0. The default value is 0. This request is only valid if the adapter has I2C_FUNC_10BIT_ADDR.
Example:

```
long select = 0;
ret = ioctl(file_descriptor,I2C_TENBIT, select);
```

- I2C_PEC

Selects SMBus PEC (Packet Error Checking) generation and verification if “select” argument does not equal 0, disables if select equals 0. The default value is 0. It is used only for SMBus transactions. This request only has an effect if the adapter has I2C_FUNC_SMBUS_PEC (this might not be available and will have no effect).
Example:

```
long select = 0;
ret = ioctl(file_descriptor,I2C_PEC, select);
```

- I2C_FUNCS

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 11111111111110000000000001111, 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_SMBUS

Set the system to do SMBus transfers. You also have to use 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
    }
};
```

The read_write variable can be configured with I2C_SMBUS_READ or with I2C_SMBUS_WRITE. The command values are always valid.
The size value can be I2C_SMBUS_BLOCK_PROC_CALL, I2C_SMBUS_QUICK, I2C_SMBUS_PROC_CALL, I2C_SMBUS_BYTE, I2C_SMBUS_BYTE_DATA, I2C_SMBUS_WORD_DATA, I2C_SMBUS_BLOCK_DATA, I2C_SMBUS_I2C_BLOCK
Example:

```
struct i2c_smbus_ioctl_data mycmd;
mycmd.read_write=0;
mycmd.command=0;
mycmd.size=I2C_SMBUS_QUICK;  // this means "probe if this address
replies"
mycmd.data=NULL;
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 *sizeof(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[] = “dumy_write”;

if (write(fd, value_to_be_written, sizeof(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 I^2C 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 I^2C 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 installs a new device named `spidev1.1`. under the `/dev/` directory. The "`1.1`" name extension represents the number of the selected SPI bus and the selected chipselect.

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 *fildes* if operation is completed successfully, otherwise it is -1.

Example:
Open the /dev/spidev1.1.

```
int fd;      // file descriptor for the /dev/watchdog 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 fildes, int request, ...)

Parameters:
fildes – file descriptor
request – device-dependent request code.
The following ioctls 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*/
                .speed_hz = 0,/* Must be set to 0 */
                .bits_per_word = 0,/* Must be set to 0*/
        };
        for (i = 0; i < 100 ; i++)
        {
                ret = ioctl(fd, SPI_IOC_MESSAGE(1), &tr);
                if (ret == -1)
                        printf("can't send spi message");
        }
```

4.3.4 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 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 */
}
```

4.3.5 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 *sizeof(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, sizeof(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 A bank, the second 32 pins refer to the B bank and the others refer to the C bank.
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:

- O for output enable with pull up
- o for output enable without pull up
- I 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 O1 > /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 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 *fildes* 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 */
}
```

4.4.2 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:
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 */
}
```

4.4.3 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 greater than *nbyte*), otherwise it is -1.

Example:
Write *sizeof(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[] = “O1”;

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

4.4.4 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 GPIO device.

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

4.5 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.

4.5.1 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 */
}
```

4.5.2 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_KEEPALIVE: 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 = WDIOS_ENABLECARD;
ioctl(fd, WDIOC_SETOPTIONS, &options);
```

- WDIOC_GETTIMEOUT

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

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

4.5.3 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 watchdog device.

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

4.6 SDIO

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
```

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
```

4.7 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:

```
# 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
```

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.

4.8 USB

4.8.1 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
```

4.8.2 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 document [4] .

To remove the link type:

```
# ifconfig usb0 down
```

5 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]
-u <sleep time> : Sleep time after send AT command (msec)
-c : (crtscts) Hw control OFF
-d : Daemonize: create a daemon
-l : Don't make pts symLinks. Links needs root privileges
-h -? : Show this help message

Since GE863-PRO³ use `/dev/ttyS3` device to access to modem, `cmuxt` is called typing:

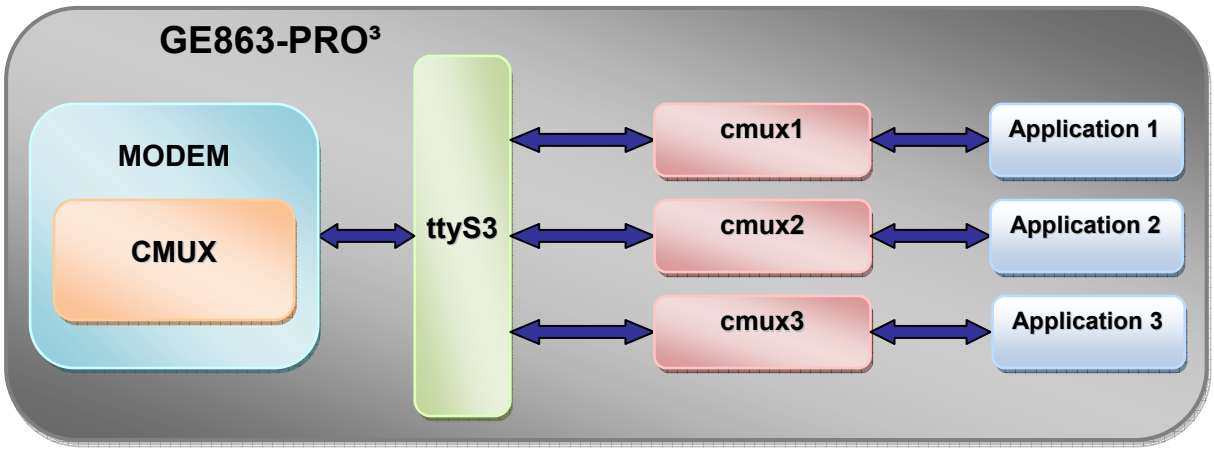
```
# cmuxt –p /dev/ttyS3 –b 115200 –d
```

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.



5.1 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;

cfmakeraw(&serCfg);

if(tcsetattr(fdPts, TCSANOW, &serCfg) != 0)
	return -1;
```

- Commit the desired operation:

```
   /* Write example */
   char AT_cmd1[] = "AT&V?\r";

if write(fdPts, AT_cmd1, strlen(AT_cmd1)) < 0)
{
        /* ERROR MANAGEMENT ROUTINE */
} else {
```


```
        /* VALUE WRITTEN */
}
/* Read example */
char read_buff[BUFF_LEN];

read(fdPts, read_buff, sizeof(read_buff));
```

- Close the serial port

```
close(fdPts);
```