GE863-PRO3 U-BOOT Software User Guide

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

U-boot SW Version
21.00.0000
211.00.0000

Contents

1. Introduction

1.1. Scope

This user guide serves the following purpose:
- Introduces the basic U-BOOT function of the GE863-PRO^3 module
- Details the system architecture of the U-BOOT loader and environment
- Describes how software developers can use the functions of the U-BOOT loader to manipulate the application processor's memories and to load and start an application or operating system.

1.2. Audience

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

1.3. Contact Information, Support

For general contact, technical support, to report documentation errors and to order manuals, contact Telit 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 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.   GNU General Public License

The U-boot code embedded into the module is licensed with the GNU General Public License as follows:

GNU GENERAL PUBLIC LICENSE
        Version 2, June 1991
Copyright (C) 1989, 1991 Free Software Foundation, Inc.
59 Temple Place, Suite 330, Boston, MA 02111-1307, 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://git.denx.de/?p=u-boot.git;a=blob;f=COPYING;h=f616ab96cc773d1719761d511a8649c9aa6eb473;hb=f4eb54529bb3664c3a562e488b460fe075f79d67

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-PRO^3 ARM Software Architecture" describes the general software architecture of the module's ARM CPU.
- "Chapter 3, GE863-PRO3 Memory Map" contains information on the memory structure and allocation of the module.
- "Chapter 4, Telit Customized U-Boot" provides a reference to the commands used to set and execute the functions of the U-boot, including examples, where relevant.
- "Chapter 5, Examples of Using the U-boot" provides scenarios and examples for using various product functionality, including setup and configuration.
- "Chapter 6, Flashing GE863-PRO^3 with XFP Tool" gives detailed instructions on how to program a binary in the Flash Memory of the **GE863-PRO^3** with XFP tool provided by Telit

- "Chapter 7, How to get U-Boot source code" provides information on how to get the source code to build a customized version of U-Boot
- "Chapter 8, Acronyms and Abbreviations" provides definition for all the acronyms and abbreviations used in this guide

How to Use
If you mainly use this document as reference, the main chapters of interest are Chapter 3, GE863-PRO3 Memory Map and Chapter 4, Telit Customized U-Boot.
If you are new to this product, it is recommended to start by reading through this document in its entirety in order to understand the concepts and specific features provided by the built in software of the GE863-PRO[3].

1.7. Text Conventions

The following attention signs are used to signal important notices, classified as follows

Danger – This information MUST be followed or catastrophic equipment failure or bodily injury may occur.

Caution or Warning – Alerts the user to important points about integrating the module, if these points are not followed, the module and end user equipment may fail or malfunction.

Tip or Information – Provides advice and suggestions that may be useful when integrating the module.

This table lists the paragraph and font styles used for the various types of information presented in this user guide. Moreover the table itself is an example of common table layout used in this document.

Format	Content
Courier	Example commands and responses from U-Boot.

1.8. Related Documents

The following documents are related to this user guide:
[1]   TelitGE863PRO3 Bootloader Recovery Application Note 80000nt10012a
[2]   TelitGE863PRO3 Development Environment User Guide 1vv0300775a
[3]   TelitGE863PRO3 EVK User Guide 1vv0300776
[4]   TelitGE863PRO3 Hardware User Guide 1vv0300773a
[5]   TelitGE863PRO3 Software User Guide 1vv0300784
[6]   TelitGE863PRO3 Product Description 80285ST10036a
[7]   Atmel AT91SAM 9260 Summary Datasheet: 6221s.pdf on web page
        link: http://www.atmel.com/dyn/products/datasheets.asp?family_id=605
[8]   TelitGE863PRO3 Linux Software User Guide 1vv0300781
[9]   TelitGE863PRO3 Linux Development Environment User Guide 1vv0300780
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	17/04/08	First Release
ISSUE #1	04/06/08	Introduced chapter 6 with instructions on how to use XFP tool for programming of GE863-PRO^3
ISSUE #2	04/08/08	Added new paragraph: 3.2 External flashes Update chapter 5 Examples of using U-boot Added new commands: autoram, ethinit, tftpboot, ping and wdt
ISSUE #3	21/05/09	4.2.1: Updated command guide and parameters 4.2.2.3: Modified example with full command usage 6 – Updated XFP Tool section 7- Added How to get U-Boot source code section

ISSUE #4	5/08/09	Applicable Products, Introduction modified to reflect the new HW configuration
		Updated:
		    3.1 GE863-PRO^3 (4/64 version)
		    4.2 U-Boot Commands
		    5.2 Example - Load Linux kernel (4/64 version)
		Added:
		    3.2 GE863-PRO^3 (128/64 version)
		    4.2.6.6 autoenv (128/64 version only)
		    4.3 U-Boot USB Console Support (128/64 version only)
		    5.3 Example – Load Linux kernel (128/64 version)
		Renamed
		    3.2 →3.3

ISSUE #5	16/11/09	Updated: 3.1.1 Flash Memory Map 4.2.2.1 Flinfo 4.2.2.6 mtest 4.2.3.2 protect 4.2.6.2 saveenv 4.2.6.2 saveenv 4.2.7.6 version Added: 4.2.3.3 df_lock (4/64 version only) Applied new layout
ISSUE #6	25/01/10	Updated: 3.2.1 Nand Flash 3.2.2 Nand Flash Memory Map

		4.2.1.1 flinfo 4.2.3.2 version Added: 4.2.6.9 usbser (128/64 version only) 4.2.7.7 ptser (128/64 version only) 5.2 Load an application (128/64 version)

2.   GE863-PRO^3 ARM Software Architecture

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. However, there are some differences related to memory management and addressing that will be explained in this User Guide.
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.

The high level GE863-PRO³ software architecture is based on the following components, located in flash memory:
- Telit Bootloader.
- Telit customized U-Boot.
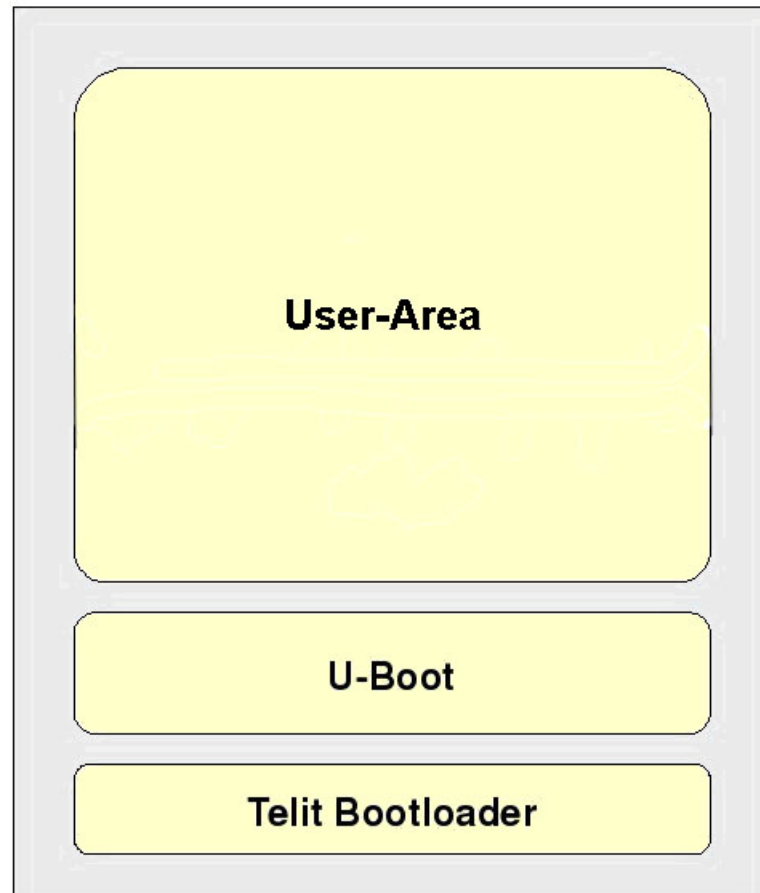
Figure 1, High level ARM software architecture

In addition, the ARM processor itself contains the Atmel "romboot" in mask ROM, please refer to [7] for further details on" romboot".
Both the Atmel "romboot" and Telit bootloader starts automatically upon power up and reset and cannot be changed or bypassed by user code.

2.1. Telit Bootloader

The Telit Bootloader is a small block of binary code that is used for low level hardware-related management and low-level initialization, with the main purpose to prepare the hardware and execute the Telit U-Boot boot loader.

2.2. Telit Customized U-boot

U-Boot is an Open Source "universal" cross-platform boot loader supporting hundreds of embedded boards and a wide variety of microcontrollers and applications.

The U-boot is a boot loader program generally residing in flash memory on an embedded system. U-Boot is capable of loading files from a variety of peripherals such as a serial connection or flash memories.

Normally, U-Boot is the initial program that gets executed at system reset that automatically loads up another application or operating system (e.g. a Linux kernel or a standalone application).

U-Boot can parse different types of file systems on different types of storage devices.

The main features of U-boot are:

- Initializing the hardware, especially the memory controller.
- Providing boot parameters for any execution program or operating system.
- Starting a specific application or an operating system.

It also provides other "convenient" features that may be of use during development:

- Reading and writing arbitrary memory locations.
- Uploading new binary images to the board's RAM via a serial line
- Copying binary images from RAM to FLASH memory.

The GE863-PRO³ uses a customized version of U-Boot, which is adapted specifically to this platform.

This user guide provides information on the capabilities of the custom Telit U-boot.

3.   GE863-PRO³ Memory Map

3.1.   GE863-PRO³ (4/64 version)

3.1.1.   Flash Memory Map

The GE863-PRO³ flash memory is divided into four main areas as shown in the picture below:
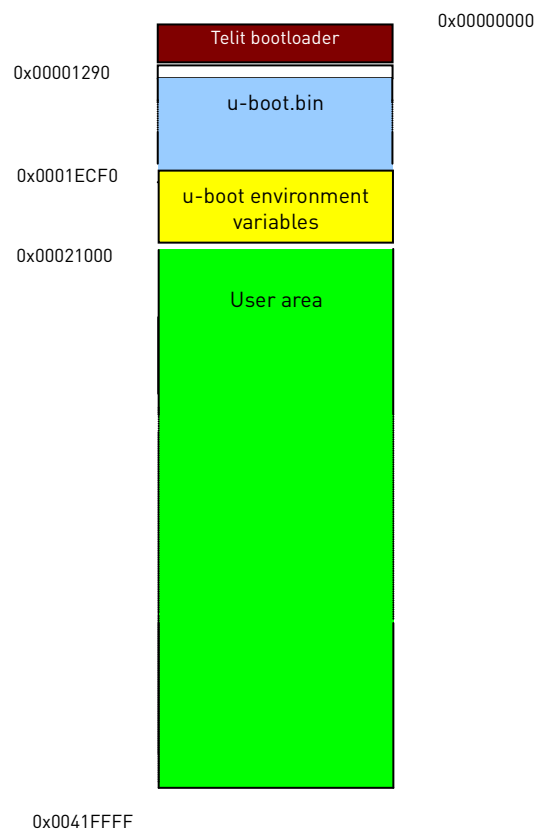
Note: All addresses are expressed in hexadecimal.
1. The range **0x00000000** to **0x0000128F** contains the Telit Bootloader.
2. The range **0x00001290** to **0x0001ECEF** holds the U-Boot image.
3. The range **0x0001ECF0** to **0x00020FFF** stores all environment variables for U-boot, such as command lines, boot parameters.

4. The range **0x00021000** to **0x0041FFFF** is the **user area** that can be allocated to any type of data, application or other purposes.

The user area can be customized to fit the needs of the target application.
The on-board embedded flash memory is organized by areas, blocks, sectors and pages:
- 1 page = 528 bytes
- 1 block = 8 pages
- 1 sector = 256 pages (32 blocks)

The entire embedded flash size is 0x420000 (4325376) bytes and has 8192 pages of 528 bytes each and the smallest erasable unit of the embedded flash is a page.

3.1.2. RAM Memory Map

The GE863-PRO³ does not execute code directly in flash memory; it must first copy the code to be executed into RAM memory.

The general steps during the boot procedure are:
- The Telit bootloader loads the U-Boot image from flash into RAM at address 0x20000000, and then executes U-boot:
  o The U-Boot.bin image relocates itself at address 0x20700000 and starts execution, freeing the RAM interval between 0x20000000 and (0x20000000 + U-Boot.bin size).

The following picture outlines the RAM memory map:

0x20000000

U-Boot
(loaded by Telitbld.bin)

(0x20000000 + U-Boot.bin size)

Empty RAM

0x20700000

U-Boot
(relocated by U-Boot itself)

0x20700000 +
U-Boot size

Empty RAM

End of RAM

3.2.   GE863-PRO³ (128/64 version)

3.2.1.   Nand Flash

The GE863-PRO³ internal memory flash is a Parallel Nand. This on-board embedded flash is organized by areas, blocks and pages:
- 1 page = 2048 bytes
- 1 block = 64 pages

The entire embedded flash size is 0x8000000 (134217728) bytes and has 1024 blocks of 128 KB bytes each and the smallest erasable unit is a block.

The NAND flash page devices may contain bad blocks, that is blocks may contain one or more invalid bits whose reliability is not guaranteed. Additional bad blocks may develop during the lifetime of the device. The blocks already bad prior to shipping are called **factory bad**, whereas the blocks developed as bad during U-Boot use are called **worn-out bad**. During flashing with XFP tool all the **worn-out blocks** are changed into **faulty blocks** for logical partitions management. The **factory bad** and **faulty blocks** are managed in the same way unlike the **worn-out blocks**.

These bad blocks need to be managed using bad blocks management and error correction codes (ECC): there are 4 ECC bytes every 512 data bytes, able to correct one data error bit and detect two data error bit. These 4 bytes are calculated by ECC hardware Controller.

⚠️   *Warning – U-Boot is not able to read pages written using other ECC strategies.*

3.2.2.   Nand Flash Memory Map

The GE863-PRO³ flash memory is divided into five main areas, each of which has fixed size (not considering factory bad and faulty blocks) and offsets that can change according to factory bad blocks positions (the first block of every area must not be factory bad or faulty). If the flash is factory bad blocks free, it will have the map shown below:

Telit bootloader   0x00000000

0x003C0000   u-boot.bin

                                    0x00440000
                u-boot environment
                    variables

0x004C0000

                    User Area 1

0x007C0000

                    User Area 2

0x08000000
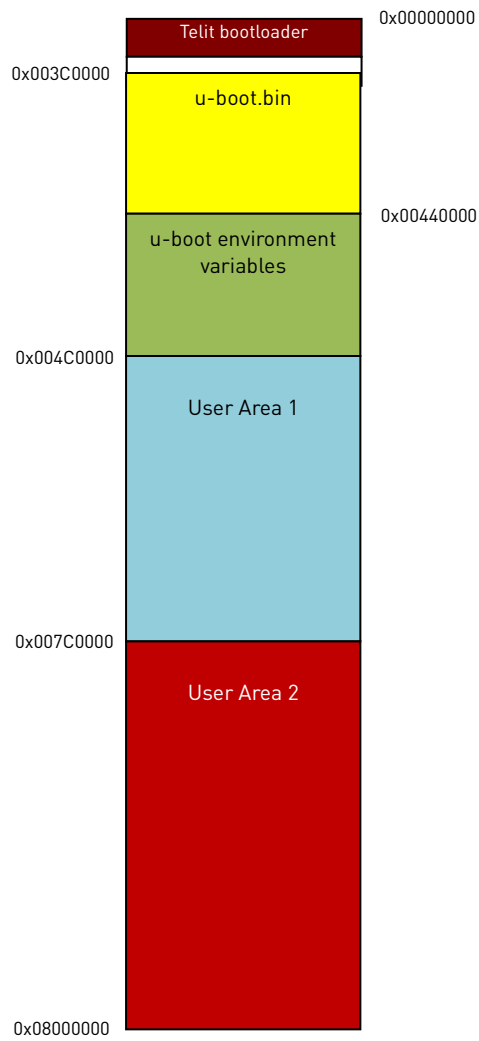
Note: All addresses are expressed in hexadecimal.
1. The range **0x00000000** to **0x003BFFFF** contains the Telit Bootloader and it should not be modified by users.
2. The range **0x003C0000** to **0x0043FFFF** holds the U-Boot image.
3. The range **0x00440000** to **0x004BFFFF** stores all environment variables for U-boot, such as command lines, boot parameters.
4. The range **0x004C0000** to **0x007BFFFF** is the **first user area** that can be allocated to any type of data, application or other purposes.
5. The range **0x007C0000** to **0x07FFFFFF** is the **second user area** that can be allocated to any type of data, application or other purposes.
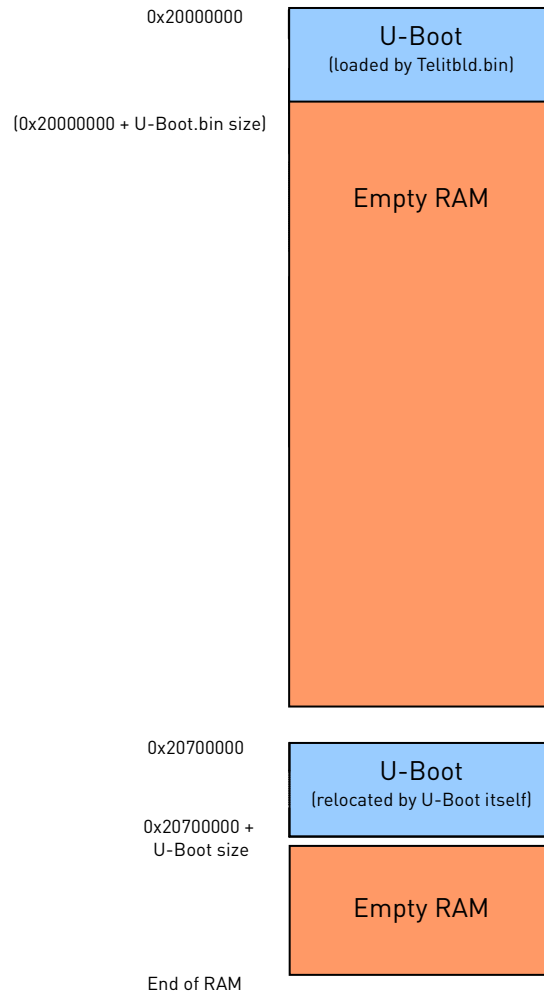
3.2.3. RAM Memory Map

The GE863-PRO³ does not execute code directly in flash memory; it must first copy the code to be executed into RAM memory.

The general steps during the boot procedure are:
- The Telit bootloader loads the U-Boot image from flash into RAM at address 0x20000000, and then executes U-boot:
  o The U-Boot.bin image relocates itself at address 0x23e00000 and starts execution, freeing the RAM interval between 0x20000000 and (0x20000000 + U-Boot.bin size).

The following picture outlines the RAM memory map:

```
0x20000000                  ┌─────────────────────┐
                            │      U-Boot          │
                            │  (loaded by Telitbld.bin) │
(0x20000000 + U-Boot.bin size) ├─────────────────────┤
                            │                     │
                            │                     │
                            │     Empty RAM       │
                            │                     │
                            │                     │
                            │                     │
                            │                     │
                            │                     │
                            │                     │
0x23e000000                 ├─────────────────────┤
                            │      U-Boot          │
                            │ (relocated by U-Boot itself) │
End of RAM                  └─────────────────────┘
```
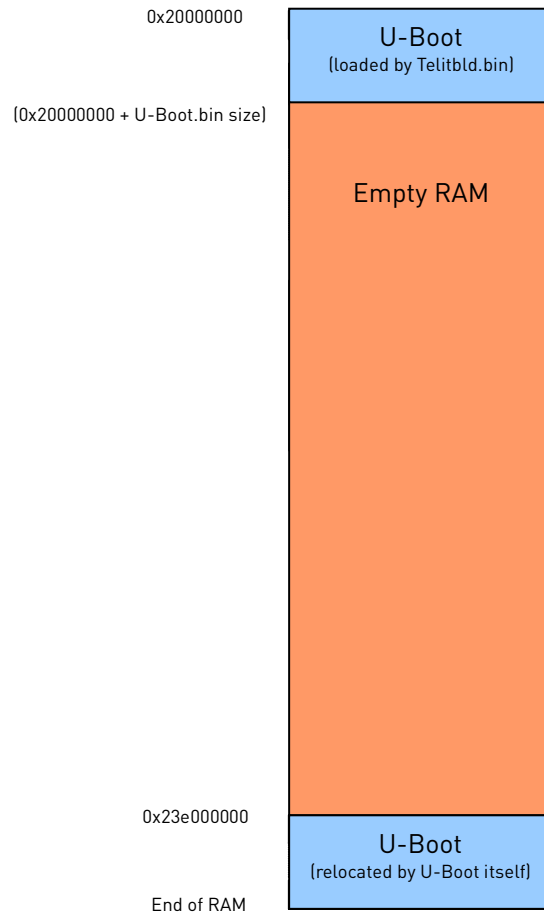
3.3.   External Flashes

U-boot can also support external flash memories (STMicroelectronics M25P32, M25P64, M25P128, M25PX32) connected to GE863-PRO³ by SPI0 bus.
Every external flash is organized in areas which can be customized to fit the needs of the target application.

- **M25P32**:
  - o The smallest erasable unit is a sector
  - o 1 page is 256 bytes
  - o 1 sector is 256 pages
  - o The entire flash size is 0x400000 (4194304) bytes and has 64 sectors of 65536 bytes
- **M25P64**:
  - o The smallest erasable unit is a sector
  - o 1 page is 256 bytes
  - o 1 sector is 256 pages
  - o The entire flash size is 0x800000 (8388608) bytes and has 128 sectors of 65536 bytes
- **M25P128**:
  - o The smallest erasable unit is a sector
  - o 1 page is 256 bytes
  - o 1 sector is 1024 pages
  - o The entire flash size is 0x1000000 (16777216) bytes and has 64 sectors of 262144 bytes
- **M25PX32**:
  - o The smallest erasable unit is a subsector
  - o 1 page is 256 bytes
  - o 1 subsector is 16 pages
  - o 1 sector is 16 subsectors
  - o The entire flash size is 0x400000 (4194304) bytes and has 1024 subsectors of 4096 bytes

U-boot can access different flashes by the following logical addresses:

- **0xD0000000** for the flash memory connected by chip select 1 (the GE863-PRO³ internal flash memory) (4/64 version)
- **0x40000000** for the Nand flash memory (the GE863-PRO³ internal flash memory) (128/64 version)
- **0xE0000000** for the flash memory connected by chip select 0
- **0xF0000000** for the flash memory connected by chip select 2

- **0xC0000000** for the flash memory connected by chip select 3

4.   Telit Customized U-Boot

As outlined in section 2.2 above, the Telit U-boot is adapted to the specific ARM CPU and environment used in the GE863-PRO[3].
This section gives additional information and details on the U-boot itself and its use in the GE863-PRO[3].

4.1.   U-Boot Environment Variables

The U-Boot environment is a block of memory that is kept in persistent storage and copied to RAM when U-Boot starts.

The environment variables that are used to configure the system are stored here and are protected by a CRC32 checksum.

Listed below are some of the most important environment variables, some of which have a special meaning to U-Boot.

These variables can be used to configure the behavior of U-Boot:

- **baudrate**: A decimal number that selects the console baud rate (in bps). Only a predefined list of baud rate settings is available. When you change the baud rate (using the "*setenv baudrate...*" command), U-Boot will switch the baud rate of the console terminal and wait for a newline which must be entered with the new speed setting.
  This is necessary to make sure you can actually type at the new speed.
  If this fails, you'll have to reset the board (which will operate at the old speed since you were not able to *saveenv* the new settings.)
  If no "*baudrate*" variable is defined, the default baud rate of 115200 is used.

- **bootargs**: The contents of this variable are passed to an Operating System or application (E.g. a Linux kernel) as boot arguments.

- **bootcmd**: This variable defines a command string that is automatically executed when the initial countdown is not interrupted.
  This command is only executed when the variable *bootdelay* is defined at the same time.

- **bootdelay**: After reset, U-Boot will wait this number of seconds before it executes the contents of the *bootcmd* variable.

During this time, a countdown will be printed to the console, and can be interrupted by pressing any key.
Set this variable to zero in order to boot without delay.
Note that depending on the contents of your `bootcmd` variable, the system can enter in a loop and prevent you from entering interactive commands.
In order to disable autoboot set this variable to -1.

- **mtdparts**: This variable (defined using the `mtdparts` command) allows the sharing of a common MTD partition scheme between U-Boot and is specifically used for the Linux kernel.

- **filesize**: Size (as hex number in bytes) of the file downloaded using the last `loady`, `loadb` or `tftpboot` command.

Any other user environment variable may be defined or modified and saved for any purpose; with environment variables commands (see section 4.2.6 for more details).

4.2.   U-Boot Commands

This section outlined the commands supported by U-boot

4.2.1.   Information Commands

4.2.1.1.   flinfo

This command prints flash memory information.
Syntax:
   flinfo [Flash Bank]
Parameters:
   Flash Bank <integer>:   Number of the flash bank.
Return: Flash Bank information, if bank parameter is not given all the flash status will be print.
For Nand Flash (128/64 version): factory bad, faulty and worn-out blocks are shown; partitions, shown by this commands, are logical, so they could be shifted according to factory bad or faulty blocks if found.

Note: short command **fli** or **f** can be used.

Example (4/64 version):
```
=> flinfo

Bank #1
Flash:AT45DB321
Nb min erasable unit:    8192
```

```
min erasable Size:     528
Size= 4325376 bytes
Logical address: 0xD0000000
Area 0: D0000000 to D000128F (RO)  Primary Bootstrap
Area 1: D0001290 to D001ECEF (RO)  U-Boot code
Area 2: D001ECF0 to D0020FFF (RO)  U-boot Environment
Area 3: D0021000 to D041FFFF        User Area


Bank #2
Flash:STM25P32
Nb min erasable unit:      64
min erasable Size:  65536
Size= 4194304 bytes
Logical address: 0xE0000000
Area 0: E0000000 to E01FFFFF        User Area 2
Area 1: E0200000 to E03FFFFF        User Area 3


Bank #3
Flash:STM25P64
Nb min erasable unit:     128
min erasable Size:  65536
Size= 8388608 bytes
Logical address: 0xF0000000
Area 0: F0000000 to F03FFFFF        User Area 4
Area 1: F0400000 to F07FFFFF        User Area 5


Bank #4
Flash:STM25P128
Nb min erasable unit:      64
min erasable Size: 262144
Size=16777216 bytes
Logical address: 0xC0000000
Area 0: C0000000 to C07FFFFF        User Area 6
Area 1: C0800000 to C0FFFFFF        User Area 7
=>
```

Example (128/64 version):
```
=> flinfo

Bank #1
Flash:NAND Numonyx NAND01GR3B2B 128MB 1.8V 8-bit
Hardware ECC (ECC data size = 512 bytes)
Nb min erasable unit:   1024
min erasable Size: 131072
```

```
        Size=134217728 bytes
        Logical address: 0x40000000
        Block 415 is factory bad (0x433e0000)
        Area 0: 40000000 to 4003FFFF (RO)  Primary Bootstrap
        Area 1: 403C0000 to 4043FFFF (RO)  U-Boot code
        Area 2: 40440000 to 404BFFFF (RO)  U-Boot Environment
        Area 3: 404C0000 to 407BFFFF        User Area 1
        Area 4: 407C0000 to 47FFFFFF        User Area 2


        Bank #2
        Flash:STM25P32
        Nb min erasable unit:     64
        min erasable Size:  65536
        Size= 4194304 bytes
        Logical address: 0xE0000000
        Area 0: E0000000 to E01FFFFF        User Area 3
        Area 1: E0200000 to E03FFFFF        User Area 4


        Bank #3
        Flash:STM25P64
        Nb min erasable unit:    128
        min erasable Size:  65536
        Size= 8388608 bytes
        Logical address: 0xF0000000
        Area 0: F0000000 to F03FFFFF        User Area 5
        Area 1: F0400000 to F07FFFFF        User Area 6


        Bank #4
        Flash:STM25P128
        Nb min erasable unit:     64
        min erasable Size: 262144
        Size=16777216 bytes
        Logical address: 0xC0000000
        Area 0: C0000000 to C07FFFFF        User Area 7
        Area 1: C0800000 to C0FFFFFF        User Area 8

        =>
```

4.2.2. Memory Commands

4.2.2.1. crc32

This command calculates a CRC32 checksum over a range of memory. Only RAM memory is supported.
Syntax:
  crc32 start_address count [address_to_store]
Parameters:
  start_address <integer>: start address to calculate checksum
  count <integer>: bytes count involved in calculation in hex format
  address_to_store <integer>: when present, checksum is stored at this address
Return: the checksum value.
Note: *count* is in hex format. The short command is **crc**.
Example:

```
=> crc 20201000 3fc
CRC32 for 20201000 ... 202013fb ==> d433b05b
=>
```

When used with 3 arguments, the command stores the calculated checksum at the given address:

```
=> crc 20201000 3fc 20401000
CRC32 for 20201000 ... 202013fb ==> d433b05b
=> md 20401000 4
20401000: d433b05b ec3827e4 3cb0bacf 00093cf5  .3.[.8'.<.....<.
=>
```

4.2.2.2. cmp

This command compares memory between addr1 and addr2 of count bytes. Only RAM memory is supported.

Syntax:
  cmp [.b, .w, .l] addr1 addr2 [count]
Parameters:
  addr1 <integer>: first source address to compare
  addr2 <integer>: second source address to compare
  count <integer>: bytes count involved in comparison
Return: result of test of the whole area as specified by the 3rd (count) argument or stop at the first difference if the count argument is not specified.

Note: Like most memory commands the cmp command accesses the memory in different sizes: 32 bit (long word), 16 bit (word) or 8 bit (byte) data.
If invoked just as cmp the default size (32 bit or long words) is used; the same can be selected explicitly by typing cmp.l instead.
To access memory as 16 bit (word data), use the variant cmp.w; to access memory as 8 bit (byte data) use cmp.b.
Please note that the *count* argument is in hex format and specifies the number of data items to process, i.e. the number of long words, words or bytes to compare.

Example:
The following example demonstrates comparing the memory ranges 0x20400000 - 0x2040002F to 0x20600000 -0x2060002F. The contents of the two memory ranges are shown below.

```
20400000:  27051956  50ff4342  6f6f7420  312e312e  '..VP.CBoot 1.1.
20400010:  3520284d  61722032  31203230  3032202d  5 (Mar 21 2002
-
20400020:  2031393a 35353a30 34290000 00000000 19:55:04)......
20600000:  27051956  50504342  6f6f7420  312e312e  '..VPPCBoot 1.1.
20600010:  3520284d  61722032  31203230  3032202d  5 (Mar 21 2002
-
20600020:  2031393a 35353a30 34290000 00000000 19:55:04)......

=> cmp 20400000 20600000 c
word  at  0x20400004  (0x50ff4342)  !=  word  at  0x20600004
(0x50504342)
Total of 1 word were the same
=>
```

4.2.2.3.   cp

This command copies data in memory, starting from "source" address to "target" address. The "count" field specifies the number of bytes, words or long words to be copied depending upon the extension field of the cp command.
If a ".b" extension is used, the count field specifies the number of bytes.
Likewise, if a ".w" or ".l" extension is used, the count field respectively specifies the number of words or long words.
Note: count is in hex format. Only RAM to flash and vice versa is supported. When the "target" address is in flash, before overwriting the flash contents, the cp command automatically erases the appropriate pages.
For Nand flash (128/64 version), this command jumps bad blocks.

Syntax:
        cp [.b, .w, .l] source_addr target_addr count
Parameters:
        source_addr <integer>:   source address to copy from
        target_addr <integer>:   target address to copy to
        count <integer>:   number of objects to be copied
(bytes(8bit) for .b, short(16bit) for .w, integer(32bit) for .l)
Return: the result of copy procedure.

Example (4/64 version):

```
=> cp 20400000 d0200000 10000
Copy to Flash... done
=>
=> cp d0200000 20400000 10000
Reading from Flash... done
```

Example (128/64 version):

```
=> cp 20400000 40200000 10000
Copy to Flash... done
=>
=> cp 40200000 20400000 10000
Reading from Flash... done
```

4.2.2.4.   md

This command displays memory contents starting at address parameter in both hexadecimal and ASCII data:.

Syntax:
        md [.b, .w, .l] address [# of objects]
Parameters:
        address <integer>:   address to start display memory contents
        # of objects <integer>:   number of objects to display (bytes for .b,
short for .w, integer for .l)
Return: the contents of memory selected.

Note: This command can be used with the type extensions .l, .w and .b.
# of objects is in hex format.
The last displayed memory address and the value of the count argument are

remembered, so when you enter md again without arguments it will automatically continue at the next address, and use the same count again.
For Nand Flash (128/64 version), this command does not jump bad blocks.

Example:
```
=> md 20400000 10
20400000: 48616c6c 6f202020 01234567 312e312e Hallo .#Eg1.1.
20400010: 3520284d 61722032 31203230 3032202d 5 (Mar 21 2002 -
20400020: 2031393a 35353a30 34290000 00000000 19:55:04)......
20400030: 00000000 00000000 00000000 00000000 ................
::::

=> md.w 20400000
20400000:  2705   1956   5050   4342   6f6f   7420   312e   312e '..VPPCBoot 1.1.
20400010: 3520 284d 6172 2032 3120 3230 3032 202d    5 (Mar 21 2002 -
20400020:  2031   393a   3535   3a30   3429   0000   0000   0000 19:55:04)......

=> md.b 20400000
20400000: 27 05 19 56 50 50 43 42 6f 6f 74 20 31 2e 31 2e '..VPPCBoot 1.1.
20400010: 35 20 28 4d 61 72 20 32 31 20 32 30 30 32 20 2d 5 (Mar 21 2002 -
20400020: 20 31 39 3a 35 35 3a 30 34 29 00 00 00 00 00 00 19:55:04)......
```


4.2.2.5.   mm

This method interactively modifies the memory contents. Only RAM memory is supported.
It will display the address and current contents and then prompt for user input.
If a legal hexadecimal number is entered, this new value will be written to the address and the next address will be prompted.
If no value is entered (e.g. by pressing Enter on the console), the contents of this address will remain unchanged.
If the "-" character is entered, no value will be written to the address and the previous address will be prompted.
The command stops as soon as you enter any data that is not a hex number (except for the "-" character).

Syntax:
  mm [.b, .w, .l] address
Parameters:
  address <integer>:   address to start display old memory contents and to modify it.
  Return: the old and new contents of memory selected.

Note: this command can be used with the type extensions ".l", ".w" and ".b".

Example:
```
=> mm 20400000
20400000: 27051956 ? 0
20400004: 50504342 ? AABBCCDD
20400008: 6f6f7420 ? 01234567
2040000c: 312e312e ? .
=> md 20400000 10
20400000: 00000000 aabbccdd 01234567 312e312e .........#Eg1.1.
20400010: 3520284d 61722032 31203230 3032202d 5 (Mar 21 2002 -
20400020: 2031393a 35353a30 34290000 00000000 19:55:04)......
20400030: 00000000 00000000 00000000 00000000 ................
=>

=> mm.w 20400000
20400000: 0000 ? 0101
20400002: 0000 ? 0202
20400004: aabb ? 4321
20400006: ccdd ? 8765
20400008: 0123 ? .
=> md 20400000 10
20400000: 01010202 43218765 01234567 312e312e ....C!.e.#Eg1.1.
20400010: 3520284d 61722032 31203230 3032202d 5 (Mar 21 2002 -
20400020: 2031393a 35353a30 34290000 00000000 19:55:04)......
20400030: 00000000 00000000 00000000 00000000 ................
=>

=> mm.b 20400000
20400000: 01 ? 48
20400001: 01 ? 61
20400002: 02 ? 6c
20400003: 02 ? 6c
20400004: 43 ? 6f
20400005: 21 ? 20
20400006: 87 ? 20
```

```
20400007: 65 ? 20
20400008: 01 ? .
```

4.2.2.6.   mtest

This command provides simple RAM memory test.

Syntax:
    mtest [start [end [pattern [loops]]]]
Parameters:
    start <integer>:   address to start memory test.
    end <integer>:   last address to do memory test
    pattern <integer>:   pattern used to perform test
    loops <integer>:   number of write/read loops the test must perform
Return: test result.

Note: The command will fail when applied to ROM or flash memory.
This command writes to RAM memory, thus modifying the memory contents.
This command may crash the system when the tested memory range includes areas that are needed for the operation of the U-Boot firmware (like exception vector code, or U-Boot's internal program code, stack or heap memory areas).
The command will fail if the start address is not 4-byte aligned.

Example:
```
=> mtest 0x20200000 0x20500000 0xaabbccdd 50
Pattern 5544334A  Writing...  Reading... 50 loop succeed
MTest successful terminated!
=>
```

Without arguments:
```
=> mtest
Default Start address: 0x20000000
Default End   address: 0x207fffff
Pattern 00000000  Writing...  Reading... 1 loop succeed
MTest successful terminated!
```

4.2.2.7.   mw

The mw command represents a way to initialize (fill) memory with a specified value.
When called without a count argument, the value will be written only to the specified address.
When used with a count, the entire memory area specified will be initialized with this value.

Syntax:
    mw [.b, .w, .l] address value [# of objects]
Parameters:
    address <integer>:   address selected to write into.
    value <integer>:   value to write at address "address"
    # of objects <integer>:   number of objects to write with value "*value*" (bytes for .b, short for .w, integer for .l)
Return: error message if failed

Note: This command can be used with the type extensions ".l", ".w" and ".b". When the address is in flash, before overwriting the flash contents, the mw command automatically erases the appropriate pages. *# of objects* is in hex format.
For Nand Flash (128/64 version), this command does not write bad blocks.

Example:
```
=> md 20400000 10
20400000:  0000000f  00000010  00000011  00000012  ................
20400010:  00000013  00000014  00000015  00000016  ................
20400020:  00000017  00000018  00000019  0000001a  ................
20400030:  0000001b  0000001c  0000001d  0000001e  ................
=> mw 20400000 aabbccdd
=> md 20400000 10
20400000:  aabbccdd  00000010  00000011  00000012  ................
20400010:  00000013  00000014  00000015  00000016  ................
20400020:  00000017  00000018  00000019  0000001a  ................
20400030:  0000001b  0000001c  0000001d  0000001e  ................
=> mw 20400000 0 6
=> md 20400000 10
20400000:  00000000  00000000  00000000  00000000  ................
20400010:  00000000  00000000  00000015  00000016  ................
20400020:  00000017  00000018  00000019  0000001a  ................
20400030:  0000001b  0000001c  0000001d  0000001e  ................
=>
```

With object size postfix specification:
```
=> mw.w 20400004 1155 6
=> md 20400000 10
20400000:        00000000        11551155        11551155
11551155 .....U.U.U.U.U.U
20400010:  00000000  00000000  00000015  00000016  ................
20400020:  00000017  00000018  00000019  0000001a  ................
20400030:  0000001b  0000001c  0000001d  0000001e  ................
=> mw.b 20400007 ff 7
=> md 20400000 10
20400000:        00000000        115511ff        ffffffff
ffff1155 .....U.........U
```

```
20400010: 00000000 00000000 00000015 00000016 ................
```

4.2.2.8.   nm

This command modifies memory (non-incrementing memory) and interactively writes different data several times to the same memory address. Only RAM memory is supported.
This can be useful for accessing and modify device registers.

Syntax:
    nm [.b, .w, .l] address
Parameters:
    address <integer>:   address to start display old memory contents and to modify it.
Return: The old and new contents of address "*address*".

Note: The nm command too accepts the type extensions ".l", ".w" and ".b".

Example:
```
=> nm.b 20400000
20400000: 00 ? 48
20400000: 48 ? 61
20400000: 61 ? 6c
20400000: 6c ? 6c
20400000: 6c ? 6f
20400000: 6f ? .
=> md 20400000 8
20400000:      6f000000      115511ff      ffffffff      ffff1155    o....U.........U
20400010:      00000000      00000000      00000015      00000016    ................
=>
```

4.2.2.9.   loop

This command reads from a memory range using a tight infinite loop.
This is intended as a special form of a memory test, since this command aims to read the memory as fast as possible. Only RAM memory is supported.
This command will never terminate, there is no way to exit this command other than resetting the module!

Syntax:
    loop [.b, .w, .l] address #ofObjects
Parameters:
    address <integer>:   address selected to read.

# of objects <integer>:   number of objects to read (bytes for .b, short for .w, integer for .l)
Return: error message if failed

Example:
=> `loop 20400000 8`

4.2.3.   Flash Memory Commands

4.2.3.1.   erase

This command erases the contents of one or more erasable units of the flash memory.
It can be used by specifying start address and end address, start erasable unit and end erasable unit, for the flash memory.

Syntax:
        erase [start_addr] [end_addr] [bank N] [N:UF[-UL]] [all]
Parameters:
        start_addr <integer>: address to start erasing memory contents (must be an erasable unit start address)
        end_addr <integer>: address to end erasing memory contents (must be an erasable unit end address)
        "bank" N <string, integer>:   erase selected bank's areas which are not write protected
        N <integer>:   number of bank (numbered starting with 1)
        UF <integer>: First unit to erase
        UL <integer>: Last unit to erase
        "all" <string>:   erase all not write protected areas of each bank of flash
Return: address range erased, number and range of units erased.

Note: short command **era**.
Both the start and end addresses for this command must point *exactly* at the start and end addresses of flash erasable units, otherwise the command will not be executed.
A flash *erasable unit* is the smallest area that can be erased in one operation.
Flash erasable units count starts with 0.
For Nand Flash (128/64 version), this command does not erase bad blocks: an error is shown when a bad block is found.

Examples :

```
=> erase d0021000 d0042000
Erasing from d0021000 to d0042000

from erasable unit: 256 to erasable unit: 511

ERASED 256/256
=>

=> era 1:256-278
Erase Flash Units 256-278 in Bank #1

ERASED 23/23
=>

=> erase bank 3
Erase Flash Bank #3

Size: 8388608 erasable units: 128
ERASED 128/128
=>

=> erase all
Erase Flash Bank #1

Size: 4325376 erasable units: 8192
ERASED 7936/8192
NOT ERASED 256/8192 because protected

Erase Flash Bank #2

Size: 4194304 erasable units: 64
ERASED 64/64

Erase Flash Bank #3

Size: 8388608 erasable units: 128
ERASED 128/128

Erase Flash Bank #4

Size: 16777216 erasable units: 64
ERASED 64/64
=>
```

Examples (128/64 version):
```
=> erase 404c0000 407c0000
Erasing from 404c0000 to 407c0000
```

```
        from erasable unit: 38 to erasable unit: 61

        ERASED 24/24
        =>

        => era 1:256-278
        Erase Flash Units 256-278 in Bank #1

        ERASED 23/23
        =>

        => erase bank 3
        Erase Flash Bank #3

        Size: 8388608 erasable units: 128
        ERASED 128/128
        =>

        => erase all
        Erase Flash Bank #1

        Size: 134217728 erasable units: 1024

        ERASE 384/1024
        BAD BLOCK DETECTED BEFORE ERASING block = 415
        ERASED 985/1024
        NOT ERASED 39/1024 because 38 protected and 1 bad

        Erase Flash Bank #2

        Size: 4194304 erasable units: 64
        ERASED 64/64

        Erase Flash Bank #3

        Size: 8388608 erasable units: 128
        ERASED 128/128
        Erase Flash Bank #4

        Size: 16777216 erasable units: 64
        ERASED 64/64
        =protect
```
This command enables or disables flash write protection to certain parts of the
flash memory.
Flash memory that is "protected" (read-only) cannot be written (with the cp
command) or erased (with the erase command).

Protected areas are marked as (RO) (i.e. "read-only") in the output of the `flinfo` command (see 4.2.1.1 example).

Syntax:
     protect `on|off` [start_addr end_addr] [bank N] [N:AF[-AL]] [all]
Parameters:
     "on"/"off"   <string>: protected/unprotected flag string
     start_addr <integer>: start address of the area to be protected/unprotected
     end_addr <integer>: end address of the area to be protected/unprotected
     "bank" N <string, integer>:   protect/unprotect selected bank

     N <integer>:   number of bank (numbered starting with 1)
     AF <integer>:   First area to be protected/unprotected
     AL <integer>:   Last area to be protected/unprotected
     "all" <string>:   protect/unprotect all flash areas within each bank
Return: address range protected/unprotected, number and range of areas protected or unprotected.

Note: The actual level of protection depends on the flash device used, and ultimately on the implementation of the flash device driver for then application.
In most cases U-Boot provides just a simple software-protection, i.e. it prevents you from erasing or overwriting important data by accident (like the U-Boot code itself or the environment variables).

Examples (4/64 version):
```
=> protect on 2:1
Protect Flash Area 1 in Bank #2

=> flinfo

Bank #1
Flash:AT45DB321
Nb min erasable unit:   8192
min erasable Size:    528
Size= 4325376 bytes
Logical address: 0xD0000000
Area 0: D0000000 to D000128F (RO)  Primary Bootstrap
Area 1: D0001290 to D001ECEF (RO)  U-Boot code
Area 2: D001ECF0 to D0020FFF (RO)  U-boot Environment
Area 3: D0021000 to D041FFFF       User Area


Bank #2
Flash:STM25P32
```

```
Nb min erasable unit:      64
min erasable Size:  65536
Size= 4194304 bytes
Logical address: 0xE0000000
Area 0: E0000000 to E01FFFFF        User Area 2
Area 1: E0200000 to E03FFFFF (RO)  User Area 3
=>



=> protect on 0xd0021000 0xd041FFFF
Protect 1 Flash Areas in Bank #1

=> flinfo

Bank #1
Flash:AT45DB321
Nb min erasable unit:   8192
min erasable Size:     528
Size= 4325376 bytes
Logical address: 0xD0000000
Area 0: D0000000 to D000128F (RO)  Primary Bootstrap
Area 1: D0001290 to D001ECEF (RO)  U-Boot code
Area 2: D001ECF0 to D0020FFF (RO)  U-boot Environment
Area 3: D0021000 to D041FFFF (RO)  User Area

Bank #2
Flash:STM25P32
Nb min erasable unit:      64
min erasable Size:  65536
Size= 4194304 bytes
Logical address: 0xE0000000
Area 0: E0000000 to E01FFFFF        User Area 2
Area 1: E0200000 to E03FFFFF (RO)  User Area 3
=>



=> protect off all
Un-Protect Flash Bank #1
Un-Protect Flash Bank #2

=> flinfo

Bank #1
Flash:AT45DB321
Nb min erasable unit:   8192
min erasable Size:     528
Size= 4325376 bytes
```

```
Logical address: 0xD0000000
Area 0: D0000000 to D000128F     Primary Bootstrap
Area 1: D0001290 to D001ECEF     U-Boot code
Area 2: D001ECF0 to D0020FFF     U-boot Environment
Area 3: D0021000 to D041FFFF     User Area

Bank #2
Flash:STM25P32
Nb min erasable unit:      64
min erasable Size:  65536
Size= 4194304 bytes
Logical address: 0xE0000000
Area 0: E0000000 to E01FFFFF          User Area 2
Area 1: E0200000 to E03FFFFF          User Area 3
=>
```

Examples (128/64 version):
```
=> protect on 2:1
Protect Flash Area 1 in Bank #2

=> flinfo

Bank #1
Flash:NAND Numonyx NAND01GR3B2B 128MB 1.8V 8-bit
Hardware ECC (ECC data size = 512 bytes)
Nb min erasable unit:    1024
min erasable Size: 131072
Size=134217728 bytes
Logical address: 0x40000000
Block 415 is factory bad (0x433e0000)
Area 0: 40000000 to 403BFFFF (RO)  Primary Bootstrap
Area 1: 403C0000 to 4043FFFF (RO)  U-Boot code
Area 2: 40440000 to 404BFFFF (RO)  U-Boot Environment
Area 3: 404C0000 to 407BFFFF        User Area 1
Area 4: 407C0000 to 47FFFFFF        User Area 2

Bank #2
Flash:STM25P32
Nb min erasable unit:      64
min erasable Size:  65536
Size= 4194304 bytes
Logical address: 0xE0000000
Area 0: E0000000 to E01FFFFF        User Area 3
Area 1: E0200000 to E03FFFFF (RO)  User Area 4
=>

=> protect on 0x404C0000 0x407BFFFF
```

```
Protect 1 Flash Areas in Bank #1

=> f 1

Bank #1
Flash:NAND Numonyx NAND01GR3B2B 128MB 1.8V 8-bit
Hardware ECC (ECC data size = 512 bytes)
Nb min erasable unit:   1024
min erasable Size: 131072
Size=134217728 bytes
Logical address: 0x40000000
Block 415 is factory bad (0x433e0000)
Area 0: 40000000 to 403BFFFF (RO)  Primary Bootstrap
Area 1: 403C0000 to 4043FFFF (RO)  U-Boot code
Area 2: 40440000 to 404BFFFF (RO)  U-Boot Environment
Area 3: 404C0000 to 407BFFFF (RO)  User Area 1
Area 4: 407C0000 to 47FFFFFF        User Area 2

=>

=> protect off all
Un-Protect Flash Bank #1
Un-Protect Flash Bank #2

=> flinfo

Bank #1
Flash:NAND Numonyx NAND01GR3B2B 128MB 1.8V 8-bit
Hardware ECC (ECC data size = 512 bytes)
Nb min erasable unit:   1024
min erasable Size: 131072
Size=134217728 bytes
Logical address: 0x40000000
Block 415 is factory bad (0x433e0000)
Area 0: 40000000 to 403BFFFF             Primary Bootstrap
Area 1: 403C0000 to 4043FFFF             U-Boot code
Area 2: 40440000 to 404BFFFF             U-Boot Environment
Area 3: 404C0000 to 407BFFFF             User Area 1
Area 4: 407C0000 to 47FFFFFF             User Area 2

Bank #2
Flash:STM25P32
Nb min erasable unit:     64
min erasable Size:  65536
Size= 4194304 bytes
Logical address: 0xE0000000
Area 0: E0000000 to E01FFFFF             User Area 2
```

```
Area 1: E0200000 to E03FFFFF          User Area 3
=>
```

4.2.3.2. df_lock (4/64 version only)

This command locks **permanently** the sector 0a in dataflash (0xd0000000 - 0xd000107F), so that this sector becomes read only.

WARNING: once this sector is locked down, it can't be erased, programmed or unlocked anymore with any tool.

If sector 0a is locked down, area 0 (0xd0000000 . 0xd000128F) will be always read only under U-Boot.

Note: short command **d**.
Examples:
```
=> df_lock
AREA 0 WILL BE PERMANENTLY LOCKED DOWN

Do you want to continue? please press 'yes' or 'no' and ENTER
=>yes
Area 0 is locked down
```

4.2.4. Execution Control Commands

4.2.4.1. bootm

This command starts operating system images.
It retrieves information about the type of the operating system, the file compression method used (if any), the load and entry point addresses, etc from the image header
The command will then load the image into the specified RAM memory address, uncompressing it if necessary.
Depending on the operating system it will pass the required boot arguments and start the operating system at its entry point.
The first argument to bootm is the memory address (only RAM is supported) where the image is stored, followed by optional arguments that depend on the OS.
Taking Linux as an example, exactly one optional argument can be passed.
In this case the bootm command consists of three steps:
- The Linux kernel image is uncompressed and copied into RAM
- The ramdisk image is loaded to RAM

- Control is passed to the Linux kernel, passing information about the location and size of the ramdisk image.

Syntax:
    bootm [addr [arg ...]]
Parameters:
    addr <integer>:   start address of OS Image (RAM memory)
    arg <integer>:If it is present defines arguments to be passed to OS image. For linux OS for example it is interpreted as the start address of a initrd ramdisk image (in RAM, ROM or flash memory).
Return: start OS procedure messages.

Examples:
```
=> bootm ${kernel_addr}

=> bootm ${kernel_addr} ${ramdisk_addr}
```

4.2.4.2. go

This command starts a standalone application at address 'addr'.

Syntax:
    go addr [arg ...]
Parameters:
    addr <integer>:   start address of the application to be executed (RAM memory)
    arg <integer>:   optional arguments needed by the application passed without modification
Return: address of starting application and user application messages.

Examples:
```
=> go 0x20200000
## Starting application at 0x20200000
...
...
```

4.2.5. Download Commands

4.2.5.1. loadb

This command loads a binary file over serial communications link (using the "Kermit" protocol).

Syntax:
    loadb [addr] [baud]
Parameters:
    addr <integer>:   address where image will be loaded (RAM memory)
    baud <integer>:   baudrate of serial line connection
Return: address, size of binary image and upload statistics.
Note: Use of HyperTerminal program is recommended.

Example:
```
=> loadb 0x20200000
## Ready for binary (kermit) download to 0x20200000 at 115200
bps...
```

At this point, the file should be transferred to the module using a communications or terminal emulation program (e.g. HyperTerminal), with the "Kermit" transfer type for sending the data file.

4.2.5.2. loads

Load an S-Record file over the serial line:

Syntax:
    loads [addr]
Parameters:
    addr <integer>:   address where image will be loaded (RAM memory)
Return: address, size of binary image and upload statistics.

4.2.5.3. loady

Load binary file over the serial line (using the "ymodem" protocol)

Syntax:
    loady [addr] [baud]
Parameters:
    addr <integer>:   address where image will be loaded (RAM memory)
    baud <integer>:   baudrate of serial line connection
Return: address, size of binary image and upload statistics.

Examples:

```
=> loady 0x20200000
## Ready for binary (ymodem) download at 0x20200000 at 115200
bps...
```

At this point, the file should be transferred to the module using a communications or terminal emulation program (e.g. HyperTerminal), with the "ymodem" transfer type for sending the data file.

4.2.5.4.   tftpboot

Load binary file via network using tftp protocol.

Syntax:
    tftpboot ram_addr remote_file_name
Parameters:
    ram_addr <integer>:   address where image will be loaded (RAM memory)
    remote_file_name <string>: name of the file to be downloaded, it has to be remotely located in the tftp server root.
Return: address, size of binary image and download statistics.

Note: short command **tftp** or **t** can be used.
This command needs an environment variable to be setup (see setenv command for details on how to set a variable) before using it and ethernet initialization already done (see ethinit for details) otherwise it will fail.
It must be defined:
serverip: remote tftp server ip address

Examples:

```
=> tftpboot 0x20200000 binaryFile.bin
phy_id 181 found at 3
macb0: link up, 100Mbps full-duplex (lpa: 0x41e1)
Using macb0 device
TFTP from server 10.255.252.52; our IP address is 10.255.252.198
Filename 'binaryFile.bin'.
Load address: 0x20200000
Loading:
#################################################################
done
Bytes transferred = 990896 (f1eb0 hex)
=>
```

4.2.6.   Environment Variables Commands

4.2.6.1.   printenv

This command prints one, several or all variables from the U-Boot environment.

Syntax:
  printenv [variable...]
Parameters:
  variables <string>:   variables to display
Return: print values of all environment variables. When variable are given, these are interpreted as the names of environment variables which will be printed with their values.

Example:
```
=> printenv ipaddr hostname netmask
ipaddr=10.0.0.99
hostname=tqm
netmask=255.0.0.0
=>
```

```
=> printenv
bootdelay=3
baudrate=115200
bootargs=console=ttyS0,115200 mem=8M rw mtdparts=spi0.1-
AT45DB321x:1221k(ARMboot)ro,-@1221k(root);
bootcmd=cp.b 0xd0021000 0x20200000 0x110000; bootm 0x20200000
stdin=serial
stdout=serial
stderr=serial
filesize=0
ipaddr=10.0.0.99
hostname=tqm
netmask=255.0.0.0


Environment size: 278/32764 bytes
=>
```

4.2.6.2.   saveenv

This command saves environment variables to persistent storage.
All changes made to the U-Boot environment are made in RAM memory only.
They are lost as soon as the system is reset.

In order to make these changes permanent, the saveenv command will write a copy of the environment settings from RAM memory into persistent storage, from where they are automatically loaded during startup.

Syntax:
> **saveenv**

Parameters:
> *none*

Return: saving messages

Example (4/64 version):
```
=> saveenv
Saving Environment to flash...

Writing u-boot environment in d001ecf0 size: 8192
=>
```

Example (128/64 version):
```
=> saveenv
Saving Environment to flash...

Writing U-Boot environment in 40440000 size: 32768
=>
```

4.2.6.3.   setenv

This command sets the value of a specific environment variable.

Syntax:
> **setenv** name [value]

Parameters:
> *name* <string>:   name of the variable to be changed
> *value* <integer/string>:   value to be assigned to variable name

Return: none

Note: If a value is not present, `setenv` deletes the specified environment variable. New variables will automatically be created, while existing variables will be overwritten.
When called with additional arguments, the first is the name of the variable, and all following arguments will (concatenated by single space characters) form the value that gets stored for this variable.
Use the backslash (\) character to escape any special characters.

Always remember that name and value have to be separated by space and/or tab characters.

Note: some environment variables (i.e. stdin, stdout, stderr, ethaddr etc.) are needed by u-boot environment and they don't have to be deleted or changed.

Example:
```
=> printenv foo
foo=This is an example value.
=> setenv foo
=> printenv foo
## Error: "foo" not defined
=>

=> printenv bar
## Error: "bar" not defined

=> setenv bar This is a new example.

=> printenv bar
bar=This is a new example.
=>

=> setenv cons_opts console=tty0 console=ttyS0,\${baudrate}
=> printenv cons_opts
cons_opts=console=tty0 console=ttyS0,${baudrate}
=>
```

4.2.6.4. ethinit

This command initializes Ethernet GPIO and u-boot variables to setup Ethernet connection in order to activate all Ethernet based commands (ping, tftpboot).

Syntax:
    ethinit target_ipaddress
Parameters:
    target_ipaddress <string>:   ipaddress chosen for target.
Return: initialization messages.

4.2.6.5. autoram

This command sets "AutoRamSizeCfg" environment variable on/off in order to enable or disable automatic configuration of "bootargs" variable to the right Ram memory size detected at startup.

Syntax:
    autoram on|off
Parameters:
    "on"/"off" <string>:   enable/disable auto Ram size bootargs configuration.
Return: enable/disable messages.

4.2.6.6.   autoenv (128/64 version only)

This command sets "AutoLinuxEnvCfg" environment variable on/off in order to enable or disable automatic configuration of "bootargs" and "bootcmd" variables to the right nand flash addresses according to bad blocks positions.

Syntax:
    autoenv on|off
Parameters:
    "on"/"off" <string>:   enable/disable automatic configuration of "bootargs" and "bootcmd" variables to the right nand flash addresses.

Return: enable/disable messages.

4.2.6.7.   run
This command runs commands in an environment variable:

Syntax:
    run var [...]
Parameters:
    var <string>:   environment variable that contains the command to be ran
Return: execution messages.

Note: If a U-Boot *variable* contains several commands (separated by semicolon), and one of these commands fails when this variable is "run", the remaining commands will be executed regardless.
If several variables are executed with one call to `run`, any failing command will terminate the "run", i.e. the remaining variables are not executed.

Example:
```
=> setenv test echo This is a test\;printenv ipaddr\;echo Done.=>
printenv test
test=echo This is a test;printenv ipaddr;echo Done.
=> run test
This is a test ipaddr=10.0.0.99
Done.
=>

=> setenv test2 echo This is another Test\;printenv hello_string\;echo
Done.
=> printenv test test2
test=echo This is a test;printenv ipaddr;echo Done.
test2=echo This is another Test;printenv hello_string;echo Done.
=> run test test2
This is a test
ipaddr=10.0.0.99
Done.
This is another Test
hello_string=Hello World!
Done.
=>
```

4.2.6.8. bootd

This command executes the default boot command.

Syntax:
>   **bootd**
Parameters:
>   *none*
Return: execution messages.

Note: The `bootd` command (short: boot) is a synonym for the "`run bootcmd`" command .i.e. what happens when the initial countdown is uninterrupted.

4.2.6.9. usbser (128/64 version only)

This command sets the "usbser_timeout" environment variable.
The value indicates how many seconds the usbser driver waits for the disconnect-reconnect sequence.
The value 0 means that the usbser is disabled.
The value is in seconds.

If the "usbser_timeout" environment variable is not set, a default value of 10 seconds is used.

Syntax:
        usbser [seconds]
Parameters:
        seconds <integer>: decimal number in seconds

Without parameter, it shows the actual value of "usbser_timeout" or the default value.

4.2.7.   Miscellaneous Commands

4.2.7.1.   echo

This commands sends echo arguments to the console:

Syntax:
        echo [args...]
Parameters:
        args <string>: argument to be printed
Return: string parameter

Example:
```
=> echo The book is on the table.
The book is on the table.
=>
```

4.2.7.2.   ping

It performs a network ping command, in order to detect if ethernet link is up, it checks also connection status to a given ip address specified as an input parameter.

Syntax:
        ping ip_address
Parameters:
        ip_address <string>: ip_address used to check ethernet connection of the target.

Return: result print if the remote machine with specified ip_address is connected with the target.

4.2.7.3.   reset
This command performs a reset of the CPU and effectively reboots the module.

4.2.7.4.   wdt
Disable watchdog reset.

Syntax:
   wdt d
Parameters:
   d <option> disable watchdog reset
Return: disable watchdog string message

4.2.7.5.   sleep
This command delays execution a number of seconds passed as an argument.

Syntax:
   sleep [N]
Parameters:
   N <integer>: decimal number in seconds
Return: none

Example:
```
=> sleep 5
=>
```


4.2.7.6.   version
This command prints version and build date of the Telit U-Boot image itself.

Syntax:
   version
Parameters:
   none
Return: version and build date of the U-Boot image.

Note: short command **ver** or **v** can be used.

Example (4/64 version):
```
=> version
U-Boot 1.2.0 (Jul 20 2009 - 15:34:47)- 21 Telit
```

Example (128/64 version):
```
=> version
U-Boot 1.2.0 (Oct 22 2009 - 17:09:36)- 211 Telit
```

4.2.7.7.   ptser (128/64 version only)

This command can be used to connect the serial console to another serial port.

Syntax:

ptser [serial_port]

Parameters:

serial_port <string>: keyword uart0, uart2 are supported

Without parameter, it shows which serial_ports are supported.

The only way to exit this command is to reset the arm processor.

Example:

```
=> ptser uart2
at+gmr
07.03.940
```

OK

4.3.   U-Boot USB Console Support (128/64 version only)

The U-Boot console can be used through the serial port and the usb device port. This section describes the access of the console through the USB device port.

 Please follow the steps below:
• Connect the USB cable to the device port.
• If the device is powered then power off.
• Power on the device.
• Disconnect and reconnect the USB cable.

- If this is the first time you use the USB Console Windows ask for a driver, When Windows ask for a driver:
  o First dialog select "No, not this time"
  o Second dialog "Install from a list or specific location (Advanced)
  o Third dialog: Click on Browse and indicate the right directory where is the "usbser.sys" and "`Telit_GE_863-PRO3.inf`" driver's files (For information about this files see § 4.3.1)
- Now the USB console is ready to be used. The driver has created a new virtual com port. The virtual com port number is incremental and does not have a fixed value, usually is just the highest port number. Use it with HyperTerminal (or a similar program) just like a standard serial port. To find the exact value check the list of the COM ports and get the value of "AT91 USB to Serial Converter (COMx)".

4.3.1.   Windows USBser Drivers

The Microsoft USB CDC ACM driver "usbser.sys", is a part of Microsoft Windows XP, so can be founded inside Windows installation directory or inside Microsoft Windows XP Service Pack 3.
So please make a recursive search inside the following positions (considering the default Windows installation "C:\WINDOWS"):
- C:\WINDOWS\ServicePackFiles\i386
- C:\WINDOWS\system32\drivers
If the driver is not present in the positions above, please search inside the Microsoft Windows XP Service Pack 3:
- If you do not have a copy of Microsoft Windows XP Service Pack 3, download it from http://www.microsoft.com (the English version is named: WindowsXP-KB936929-SP3-x86-ENU.exe)
- With a CAB file extractor tool like 7-Zip (http://www.7-zip.org) extract the "i386\usbser.sy_" file from the Service Pack
- Finally using the system command line utility "expand", which decompresses "usbser.sy" as "usbser.sys":

            expand -r usbser.sy_ .

The .inf file is downloadable from http://telit.com in Download Zone, section *Software > Cellular > Software Tools GSM/GPRS > GE863-PRO^3_without_OS* or it can be created by copying the following portion of text into a file.
Example for the "*Telit_GE863-PRO3_usbser.inf*":


```
[Version]                                               ; Version section
```

```
Signature="$Chicago$"                                    ; All Windows versions
Class=Ports                                      ; This is a serial port driver
ClassGuid={4D36E978-E325-11CE-BFC1-08002BE10318}         ; Associated GUID
Provider=%ATMEL%                                    ; Driver is provided by ATMEL
DriverVer=09/12/2006,1.1.1.5    ; Driver version 1.1.1.5 published on 09/12/2006

[DestinationDirs]              ; DestinationDirs section
DefaultDestDir=12               ; Default install directory is \drivers or
\IOSubSys

[Manufacturer]              ; Manufacturer section
%ATMEL%=AtmelMfg          ; Only one manufacturer (ATMEL), models section is named
                          ; AtmelMfg

[AtmelMfg]                    ; Models section corresponding to ATMEL
%USBtoSerialConverter%=USBtoSer.Install,USB\VID_03EB&PID_6119   ; Identifies a
device with ATMEL Vendor ID (03EBh) and
                ; Product ID equal to 6119h. Corresponding Install section
                ; is named USBtoSer.Install

[USBtoSer.Install]      ; Install section
include=mdmcpq.inf
CopyFiles=FakeModemCopyFileSection
AddReg=USBtoSer.AddReg          ; Registry keys to add are listed in
USBtoSer.AddReg

[USBtoSer.AddReg]              ; AddReg section
HKR,,DevLoader,,*ntkern                          ;
HKR,,NTMPDriver,,usbser.sys
HKR,,EnumPropPages32,,"MsPorts.dll,SerialPortPropPageProvider"

[USBtoSer.Install.Services]                      ; Services section
AddService=usbser,0x00000002,USBtoSer.AddService        ; Assign usbser as the
PnP driver for the device

[USBtoSer.AddService]      ; Service install section
DisplayName=%USBSer%          ; Name of the serial driver
ServiceType=1          ; Service kernel driver
StartType=3          ; Driver is started by the PnP manager
ErrorControl=1      ; Warn about errors
ServiceBinary=%12%\usbser.sys              ; Driver filename

[Strings]      ; Strings section
ATMEL="ATMEL Corp."          ; String value for the ATMEL symbol
USBtoSerialConverter="AT91 USB to Serial Converter"    ; String value for the
USBtoSerialConverter symbol
USBSer="USB Serial Driver"                  ; String value for the USBSer symbol
```

5. Examples of Using the U-boot

This section contains more complex examples on how to use certain U-boot commands, and how to perform common tasks using several U-boot commands. These examples for using U-boot outline typical operations, particularly how to prepare the U-boot environment and how to boot an Operating system (e.g. Linux) from a flash file system, or simply launch an application.

The very first step is to connect the EVK-PRO3 board to a PC via serial link (see the figure below), and open a communications/terminal emulation program like "HyperTerminal".
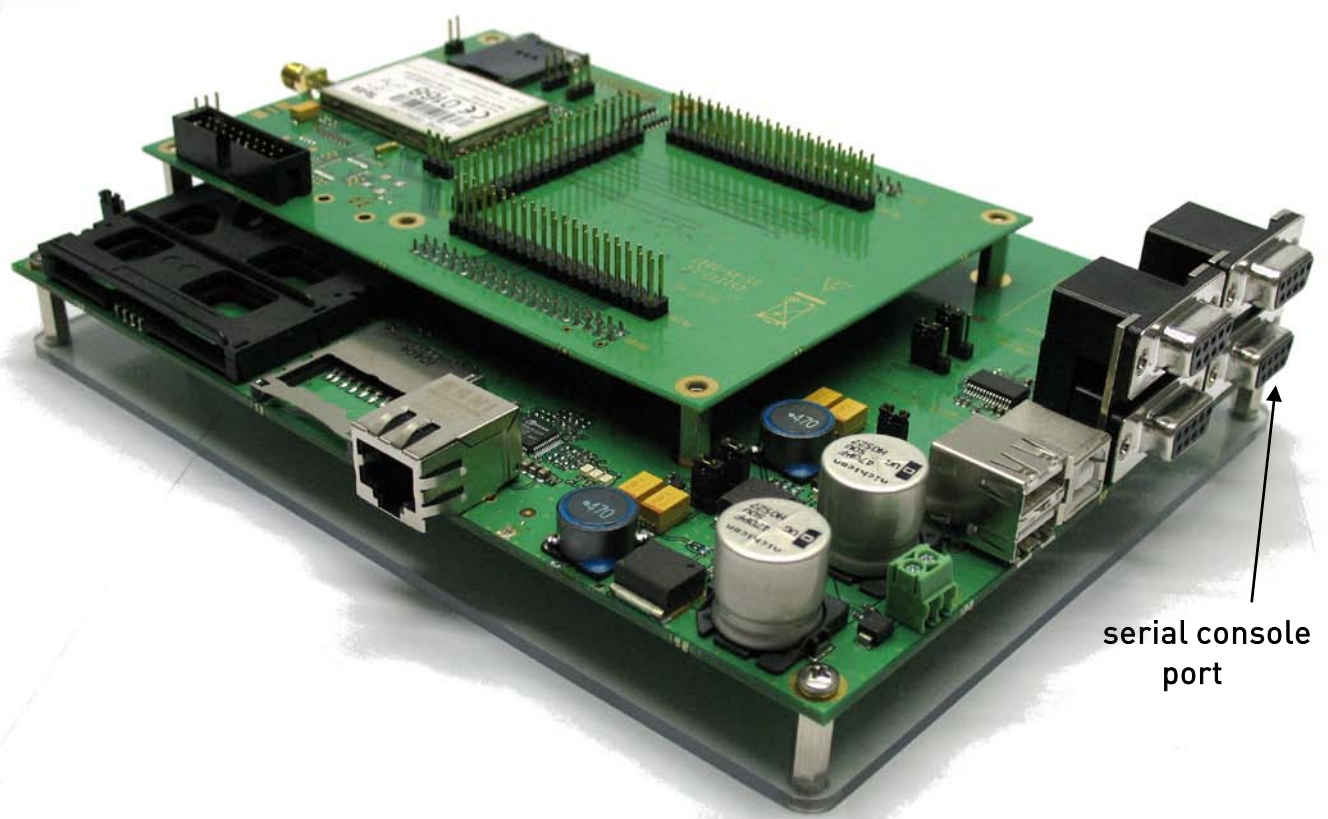For software customization and jumper settings see [3])

serial console port

Figure 2, EVK-PRO^3

Then, after switching on the EVK-board, the Telit Bootloader automatically starts and loads U-boot.
At this point, all of the U-boot features and customized commands are available to the user.

5.1. Example – Load an Application (4/64 version)

This section outlines how to start an application from the U-boot environment (see section 2.2.1 for details), and how to set it up in order to execute the application at system startup.
The binary file containing the application must be loaded from the attached PC to the target RAM-area using ymodem in this example:

```
=> loady 20012000
## Ready for binary (ymodem) download to 0x20012000 at 115200
bps...
CCCxyzModem  -  CRC  mode,  2(SOH)/2(STX)/0(CAN)  packets,  5
retries
## Total Size      = 0x000007d4 = 2004 Bytes

=> printenv
bootdelay=3
baudrate=115200
bootargs=console=ttyS0,115200 mem=8M rootfstype
stdin=serial
stdout=serial
stderr=serial
filesize=7D4

Environment size: 280/32764 bytes
```

Note: A temporary variable named "filesize" is created by the loady command in order to keep track of the size of the last file loaded.
It can be useful to use this variable ${filesize} as an argument to commands that need file sizes instead of a constant value.

```
=> md 0xd0131400
d0131400:         ffffffff        ffffffff        ffffffff
ffffffff    ................
d0131410:         ffffffff        ffffffff        ffffffff
ffffffff    ................
…
…
```
Application must be copied to memory flash:

```
=> cp.b 20012000 0xd0131400 0x7d4 (or ${filesize})
Copy to Flash... done
=> md 0xd0131400
d0131400:  e92d4030  e59f3048  e24dd008  e5933000        0@-
.H0....M..0..
```

```
d0131410:           e28d5008           e5253004           eb000151
e59f0034    .P...0%.Q...4...
…
…
```

The "bootcmd" variable is set in order to let the application automatically start at next system startup.

```
=>  setenv  bootcmd  cp.b  0xd0131400  0x20012000  0x7d4\;  go
0x20012000

=> printenv
bootdelay=3
baudrate=115200
bootargs=console=ttyS0,115200 mem=8M rootfstype
bootcmd=cp.b 0xd0131400 0x20012000 0x7d4; go 20012000
stdin=serial
stdout=serial
stderr=serial
filesize=7D4

Environment size: 280/32764 bytes
=> saveenv
```

The application can be executed by typing "run bootcmd" or simply by resetting the module.

5.2. Example – Load an Application(128/64 version)

This section outlines how to start an application from the U-boot environment (see section 2.2.1 for details), and how to set it up in order to execute the application at system startup.
The binary file containing the application must be loaded from the attached PC to the target RAM-area using ymodem in this example:

```
=> loady 20012000
## Ready for binary (ymodem) download to 0x20012000 at 115200
bps...
CCCxyzModem  -  CRC  mode,  2(SOH)/2(STX)/0(CAN)  packets,  5
retries
## Total Size      = 0x000007d4 = 2004 Bytes

=> printenv
bootdelay=3
baudrate=115200
bootargs=console=ttyS0,115200 mem=64M rootfstype
stdin=serial
stdout=serial
stderr=serial
filesize=7D4

Environment size: 280/32764 bytes
```

Note: A temporary variable named "filesize" is created by the loady command in order to keep track of the size of the last file loaded.
It can be useful to use this variable ${filesize} as an argument to commands that need file sizes instead of a constant value.

Application must be copied to memory flash:

```
=> cp.b 20012000 0x404C0000 0x7d4 (or ${filesize})
Copy to Flash... done
=> md 0x404C0000
404C0000:  e92d4030  e59f3048  e24dd008  e5933000       0@-
.H0....M..0..
404C0010:        e28d5008         e5253004        eb000151
e59f0034    .P...0%.Q...4...
…
…
```
The "bootcmd" variable is set in order to let the application automatically start at next system startup.

```
=>  setenv  bootcmd  cp.b  0x404C0000  0x20012000  0x7d4\;  go
0x20012000

=> printenv
bootdelay=3
baudrate=115200
bootargs=console=ttyS0,115200 mem=8M rootfstype
bootcmd=cp.b 0x404C0000 0x20012000 0x7d4; go 20012000
stdin=serial
stdout=serial
stderr=serial
filesize=7D4

Environment size: 280/32764 bytes
=> saveenv
```

The application can be executed by typing "run bootcmd" or simply by resetting the module.

5.3.   Example – Load Linux Kernel (4/64 version)

This example shows how the Linux operating system can be loaded onto the GE863-PRO[3].
The following steps outline the procedure:

1. For this purpose the *U-boot environment* must be setup to activate the boot procedure (see section 4.1 for details). At the U-boot prompt, the `setenv` command has to be used to set up the environment:

```
=>    setenv    bootargs    console=ttyS0,115200    mem=64M
rootfstype=jffs2   root=/dev/mtdblock1   rw   mtdparts=spi0.1-
AT45DB321x:1221k(ARMboot)ro,-@1221k(root)
```

The bootargs environment variable is set in order to define the arguments that will be used by the boot sequence, like console serial device (ttyS0) and related baud rate, max ram memory (64Mb), the type of file system that will be mounted (jffs2) and in which mtdblock partition to use, and also information about the flash hardware and driver the system will make use of.

```
=> setenv bootcmd cp.b 0xd0021000 0x20200000 0x110000\; bootm
0x20200000
```

The bootcmd environment variable is set to define a sequence of commands to be executed for booting the kernel and file system.

With the `saveenv` command this environment is saved to flash memory so at the next reboot, the U-boot will automatically run the `bootcmd` command.

```
=> saveenv
```

1. *Binary images of kernel and file system* must now be loaded into RAM memory temporarily in order to permanently copy them into flash memory.

For the kernel:
Load the kernel image in "uImage" binary format in RAM memory using ymodem:

```
=> loady 20200000
## Ready for binary (ymodem) download to 0x20200000 at 115200
bps...
CCCCCxyzModem - CRC mode, 2(SOH)/925(STX)/0(CAN) packets, 7
retries
## Total Size      = 0x000e703c = 946236 Bytes
```

Now copy the binary image data from RAM memory (0x20200000) into the flash starting at the specified location (start of user area, see Flash Memory Map in 3.1.1) 0xd0021000.

```
=> cp.b 20200000 d0021000 0xe703c
Copy to Flash....... done
```

Note that the U-boot remaps the flash memory's physical address into a virtual address with offset 0xd0000000.
Thus, typing 0xd0021000 into a U-boot command means that it is going to write in 0x00021000 physical flash memory.
This is due to the fact that the GE863-PRO³ uses a serial flash memory device that is virtualized into the processor memory space for ease of operation.

For file system:
Load the file system binary image into RAM memory using ymodem:

```
=> loady 20200000
## Ready for binary (ymodem) download to 0x20200000 at 115200
bps...
CCCCCxyzModem - CRC mode, 2(SOH)/925(STX)/0(CAN) packets, 3
retries
## Total Size      = 0x00147F00 = 1343232 Bytes
```

Copy the binary image data from RAM memory (0x20200000) into the flash starting at the specified location 0xd0131400.

```
=> cp.b 20200000 d0131400 0x147F00
Copy to Flash....... done
```

1. Now, the module can be reset and will perform the boot procedure automatically (by boot default command) or the boot command could be typed manually in order to start boot procedure at this point:

```
=> run bootcmd
```

or

```
=> bootd
```

Note that when U-boot starts, after a customizable countdown, it automatically executes the `bootd` command (boot default command) which is the same as "`run bootcmd`" (see section 4.2.6.7 for details).

5.3.1. Example – Load Linux Kernel using Ethernet connection and tftp protocol

Linux operating system can be loaded onto the GE863-PRO[3] also making use of tftpboot command and an Ethernet connection; that should be faster for development purposes.

As described above we have to set first of all bootargs environment variable which is the same as the one written above.
```
=>     setenv     bootargs     console=ttyS0,115200     mem=64M
rootfstype=jffs2   root=/dev/mtdblock1   rw   mtdparts=spi0.1-
AT45DB321x:1221k(ARMboot)ro,-@1221k(root) <enter>
```
The command *tftpboot* makes use of an ethernet connection, then, before using it, it must be set up (only once) by calling a command initialization and by specifying an environment variable:
Ethernet initialization command:

```
=> ethinit 10.255.252.198 <enter>
```

Once a tftp remote server, from where files will be downloaded, has been set up and initialized in the remote host machine, its ip address must be set in the target:
```
=> setenv  serverip  10.255.252.52<enter>
```

Save all variables modifications
=> saveenv <enter>

To flash kernel and filesystem binary files making use of tftpboot command:

Kernel:
```
        => tftp 0x20200000 remote_kernel_file <enter>
phy_id 181 found at 3
macb0: link up, 100Mbps full-duplex (lpa: 0x41e1)
Using macb0 device
TFTP from server 10.255.252.52; our IP address is
10.255.252.198
Filename 'remote_kernel_file'.
Load address: 0x20200000
Loading:
#############################################################
done
Bytes transferred = 990896 (f1eb0 hex)


        => cp.b 0x20200000 0xd0021000 0xf1eb0 <enter>
```

Filesystem:
```
        => tftp 0x20200000 remote_filesystem_file <enter>
phy_id 181 found at 3
macb0: link up, 100Mbps full-duplex (lpa: 0x41e1)
Using macb0 device
TFTP from server 10.255.252.52; our IP address is
10.255.252.198
Filename 'remote_filesystem_file'.
Load address: 0x20200000
Loading:
###########################################################
        done
        Bytes transferred = 1047552 (ffc00 hex)


        => cp.b 0x20200000 0xd0131400 0xffc00 <enter>
```

Once all images are written in flash memory the bootcmd variable must be configured. There are 2 configurations according to boot method. Because of filesystem type (jffs2), its image must be started from flash by kernel using bootargs obtained by u-boot; contrariwise kernel image is executed from RAM memory, so it could be downloaded on the fly during the bootstrap phase before being executed, this could be done by tftp command, since it performs a very fast download.

- Setup for booting kernel from flash (regular way):

```
=> setenv bootcmd cp.b 0xd0021000 0x20200000 0x110400\;
bootm 0x20200000      <enter>
=> saveenv <enter>
```

- Setup for booting kernel directly from ethernet/tftpserver :

```
=> setenv bootcmd tftp 0x20200000 remote_kernel_file \;
bootm 0x20200000 <enter>
=> saveenv <enter>
```

Note: In this last case tftpserver must be always connected to target at bootstrap time. This feature could be used for development purposes, because *remote_kernel_file* may be changed very often in the remote tftpserver directory and then it could be executed via tftp protocol without writing it into flash every time before bootstrap.

5.4. Example - Load Linux Kernel (128/64 version)

This example shows how the Linux operating system can be loaded onto the GE863-PRO[3].
The following steps outline the procedure:
1. For this purpose the *U-boot environment* must be setup to activate the boot procedure (see § 4.1 for details).
"bootargs" and "bootcmd" variables can be automatically set if "AutoLinuxEnvCfg" variable is equal to "ON", otherwise at the U-boot prompt, the `setenv` command has to be used to set up the environment:
```
=> setenv bootargs console=ttyS0,115200 mem=64M ver=4384k
icofat=257k rootfstype=jffs2 root=/dev/mtdblock1 rw
mtdparts=at91_nand:7936k(ARMboot)ro,-@7936k(root)
=> setenv bootcmd cp.b 0x404c0000 0x20200000 0x110000\; bootm
0x20200000
```
The bootargs environment variable is set in order to define the arguments that will be used by the boot sequence, like console serial device (ttyS0) and related baud rate, max ram memory (64MB), the locations where U-Boot version and ICOFAT are stored, the type of file system that will be mounted (jffs2) and in which mtdblock partition to use, and also information about the flash hardware and driver the system will make use of. The root partition offset must take any bad blocks into account.
The bootcmd environment variable is set to define a sequence of commands to be executed for booting the kernel and file system.

With the `saveenv` command this environment is saved to flash memory so at the next reboot, the U-boot will automatically run the `bootcmd` command.

```
=> saveenv
```

1. *Binary images of kernel and file system* must now be loaded into RAM memory temporarily in order to permanently copy them into flash memory.

For the kernel:
Load the kernel image in "uImage" binary format in RAM memory using ymodem:

```
=> loady 20200000
## Ready for binary (ymodem) download to 0x20200000 at 115200
bps...
CCCCCxyzModem - CRC mode, 2(SOH)/925(STX)/0(CAN) packets, 7
retries
## Total Size      = 0x000e703c = 946236 Bytes
```

Now copy the binary image data from RAM memory (0x20200000) into the flash starting at the specified location (start of user area, see Flash Memory Map in 3.1.1) 0x404C0000.

```
=> cp.b 20200000 404c0000 0xfb148
Copy to Flash....... done
```

Note that the U-boot remaps the flash memory's physical address into a virtual address with offset 0x40000000.
Thus, typing 0x404C0000 into a U-boot command means that it is going to write in 0x004C0000 physical flash memory.
This is due to the fact that the GE863-PRO[3] uses a serial flash memory device that is virtualized into the processor memory space for ease of operation.

For file system:
Load the file system binary image into RAM memory using ymodem:

```
=> loady 20200000
## Ready for binary (ymodem) download to 0x20200000 at 115200
bps...
CCCCCxyzModem - CRC mode, 2(SOH)/925(STX)/0(CAN) packets, 3
retries
## Total Size      = 0x00147F00 = 1343232 Bytes
```

Copy the binary image data from RAM memory (0x20200000) into the flash starting at the specified location 0x407C0000.

```
=> cp.b 20200000 407C0000 0x11be34
Copy to Flash....... done
```

1. Now, the module can be reset and will perform the boot procedure automatically (by boot default command) or the boot command could be typed manually in order to start boot procedure at this point:

```
=> run bootcmd
```

or

```
=> bootd
```

Note that when U-boot starts, after a customizable countdown, it automatically executes the `bootd` command (boot default command) which is the same as "`run bootcmd`" (see section 4.2.6.7 for details).

5.4.1.   Example – Load Linux Kernel using Ethernet connection and tftp protocol

Linux operating system can be loaded onto the GE863-PRO^3 also making use of tftpboot command and an Ethernet connection; that should be faster for development purposes.

As described above we have to set first of all bootargs environment variable which is the same as the one written above.
```
=> setenv bootargs=console=ttyS0,115200 mem=64M ver=4384k
icofat=257k rootfstype=jffs2 root=/dev/mtdblock1 rw
mtdparts=at91_nand:7936k(ARMboot)ro,-@7936k(root)
```
The command _tftpboot_ makes use of an ethernet connection, then, before using it, it must be set up (only once) by calling a command initialization and by specifying an environment variable:
Ethernet initialization command:

```
=> ethinit 10.255.252.198 <enter>
```

A tftp remote server, from where files will be downloaded, shall be set up and initialized in the remote host machine. Its ip address must be provided as environment variable in the target. In this example, the tftp server is hosted on 10.255.252.52:

=> setenv serverip 10.255.252.52<enter>

Save all variables modifications
=> saveenv <enter>

To flash kernel and filesystem binary files making use of tftpboot command:

Kernel:
```
        => tftp 0x20200000 remote_kernel_file <enter>
phy_id 181 found at 3
macb0: link up, 100Mbps full-duplex (lpa: 0x41e1)
Using macb0 device
TFTP from server 10.255.252.52; our IP address is
10.255.252.198
Filename 'remote_kernel_file'.
Load address: 0x20200000
Loading:
#########################################################
done
Bytes transferred = 990896 (f1eb0 hex)
```

```
        => cp.b 0x20200000 0x404c0000 0xf1eb0 <enter>
```

Filesystem:
```
        => tftp 0x20200000 remote_filesystem_file <enter>
phy_id 181 found at 3
macb0: link up, 100Mbps full-duplex (lpa: 0x41e1)
Using macb0 device
TFTP from server 10.255.252.52; our IP address is
10.255.252.198
Filename 'remote_filesystem_file'.
Load address: 0x20200000
Loading:
#########################################################
        done
        Bytes transferred = 1047552 (ffc00 hex)
```

```
        => cp.b 0x20200000 0x407c0000 0xffc00 <enter>
```

Once all images are written in flash memory the bootcmd variable must be configured. There are 2 configurations according to boot method. Because of filesystem type (jffs2), its image must be started from flash by kernel using bootargs obtained by u-boot; conversely, kernel image is executed from RAM memory, so it could be downloaded on the fly during the bootstrap phase before

being executed, this could be done by tftp command, since it performs a very fast download.

- **Setup for booting kernel from flash (regular way):**

    => setenv bootcmd cp.b 0x404c0000 0x20200000 0x110400\; bootm 0x20200000   <enter>
    => saveenv <enter>

- **Setup for booting kernel directly from ethernet/tftpserver :**

    => setenv bootcmd tftp 0x20200000 remote_kernel_file \; bootm 0x20200000 <enter>
    => saveenv <enter>

Note: In this last case tftpserver must be always be connected to target at bootstrap time. This feature can be used for development purposes, because *remote_kernel_file* may be changed very often in the remote tftpserver directory and then it could be executed via tftp protocol without writing it into flash every time before bootstrap.

6. Flashing GE863-PRO^3 with XFP Tool

In the following paragraphs you can find detailed instructions on how to program a binary in the Flash Memory of the **GE863-PRO³** with XFP tool provided by Telit. It runs on Windows based PCs.

WARNING: Please note that Telit highly recommends using software binaries provided by Telit. Telit disclaims any and all responsibility for the application of custom software included those which may contain modification of software originally provided by Telit.

6.1. How to program the stream

XFP Tool takes a stream file as input: it can be a package, the Operating System, the complete firmware or just an application.

Telit provides stream files with the following filename convention:

packageXXTelit.stream → a Package (XX is the variable part name)

LinuxYYYYYYTelit.stream → the Telit Linux OS (YYYYYY contains information on the version)

FirmwareXX_YYYYYYTelit.stream → the complete firmware (XX for the package, YYYYYY for the OS version)

Whatever the stream actually contains, the target device programming procedure is always the same, as described in the following subsection.

## 6.2. Step by step programming instructions

Follow the steps below in order to program the **Stream** provided by Telit:

- Connect a serial port of the host system to the serial port of the target called AP DEBUG:

Port#4
PC<->AP/AP->GSM
or GSM TRACE

Port#3
PC<->GSM/GSM->AP
or GSM TRACE

Port#2
AP UART0

Port#1
AP DEBUG

CS1241
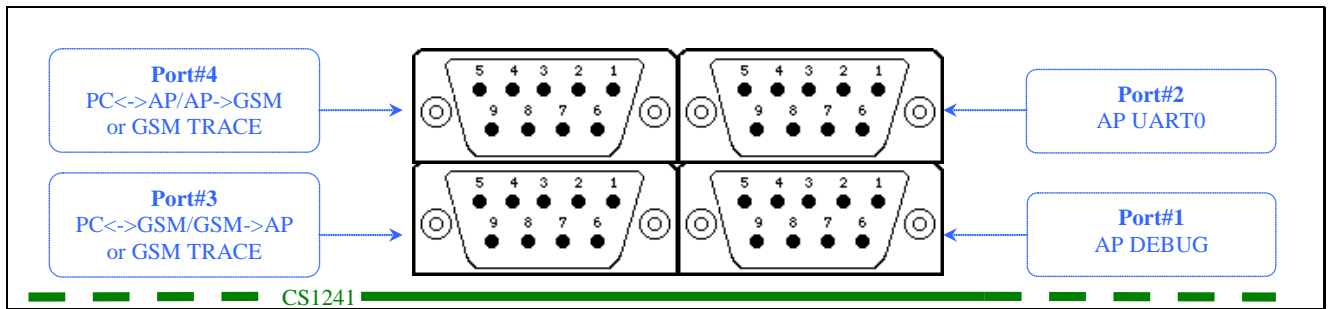
- Copy the Stream **example.stream** in the directory where you want to load it
- Launch the program **xfp.exe** (latest version is recommended): the following windows appear:
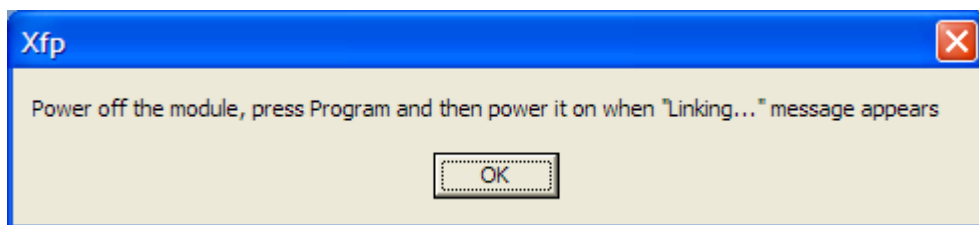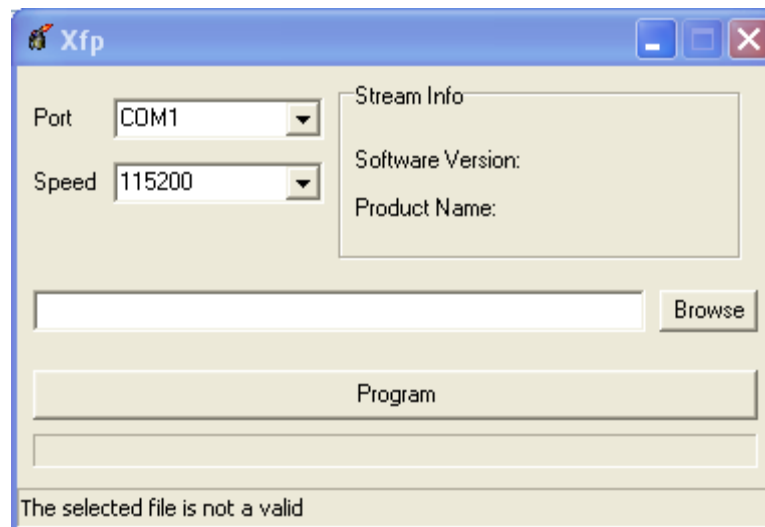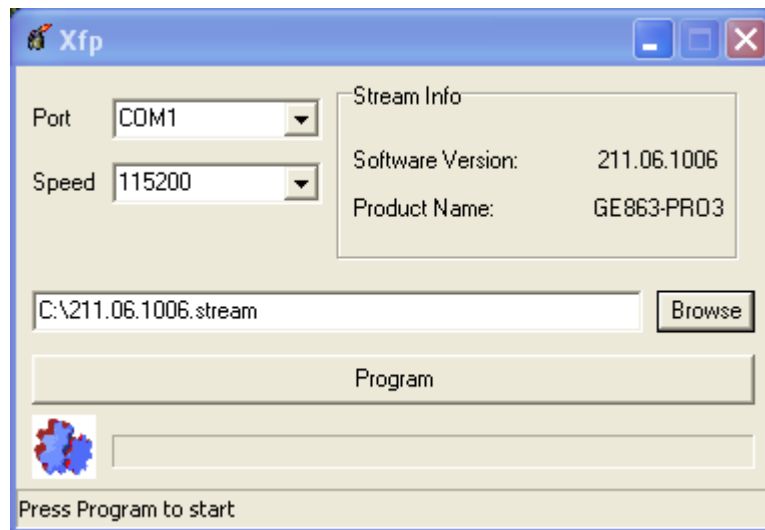

Click **Ok.**

- In the **Xfp** window:

  o Select the **Port** of your host system (if your PC has only a serial port, it should be COM1)
 o Select the **Speed**: 115200 bits per second

Page 72 of 76

o   Click on **Browse,** a file open dialog window will open. Select the file **example.stream**.

The following window illustrates an example of selected stream: the "Stream Info" box will now show information about Software Version and Product Name of the **stream**.

•   Power Off the module, Press **Program** and power it on when "Linking" message appears.
At the end of the programming procedure, the following dialog window will open
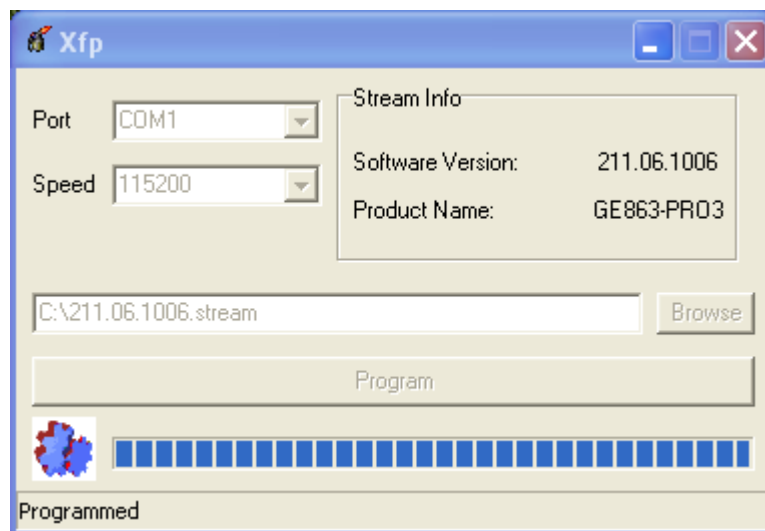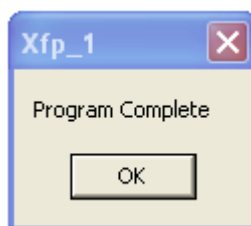
Click **Ok**

- Close the program xfp.exe
- The **GE863-PRO**[3] package provided by Telit has now been programmed.

8.   Acronyms and Abbreviations

Term	Definition
JTAG	Joint Test Action Group(ANSI/ICEEE Std. 1149.1-1990)
OS	Operating System
UART	Universal Asynchronous Receiver Transmitter
USB	Universal Serial Bus