wireless solutions

Low Level Driver APIs Application Note

GE863-PRO^3

80000nt10016a Rev. 1 – 11/09/08

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 application note.

All rights reserved.

© 2008 Telit Communications S.p.A.

Applicable Products

Product	Part Number
GE863-PRO[3]	3990250691

Contents

1 Introduction

1.1 Scope

The scope of this document is to describe the additional software provided to simplify the development of the Telit GE863-PRO^3 module.

This document is not intended to provide an overall description of all software issues and products that may be designed. The document serves as a guide line or starting point for software development of a product with the Telit GE863-PRO^3 module.

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

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 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.

1.5 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, Library Setup” describes briefly how to import the Low Level Driver APIs within a project.
- “Chapter 3, Low level driver APIs” provides a description of the Low Level Driver APIs.
- “Chapter 4, Appendix” provides a Low Level Driver flowchart, macro definitions in the library and C types redefinitions.

How to Use
If you are new to this product, it is highly recommended to start by reading through TelitGE863PRO3 Development Environment User Guide [2] and Atmel AT91SAM 9260 Summary Datasheet [7] and this document in their entirety in order to understand the concepts and specific features provided by the built in software of the GE863-PRO^3.

1.6 Text Conventions

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

Format	**Content**
Italic Arial	Prototypes, Parameters, Returned values, Types definition

1.7 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 U-Boot Software User Guide 1VV0300777

[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

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

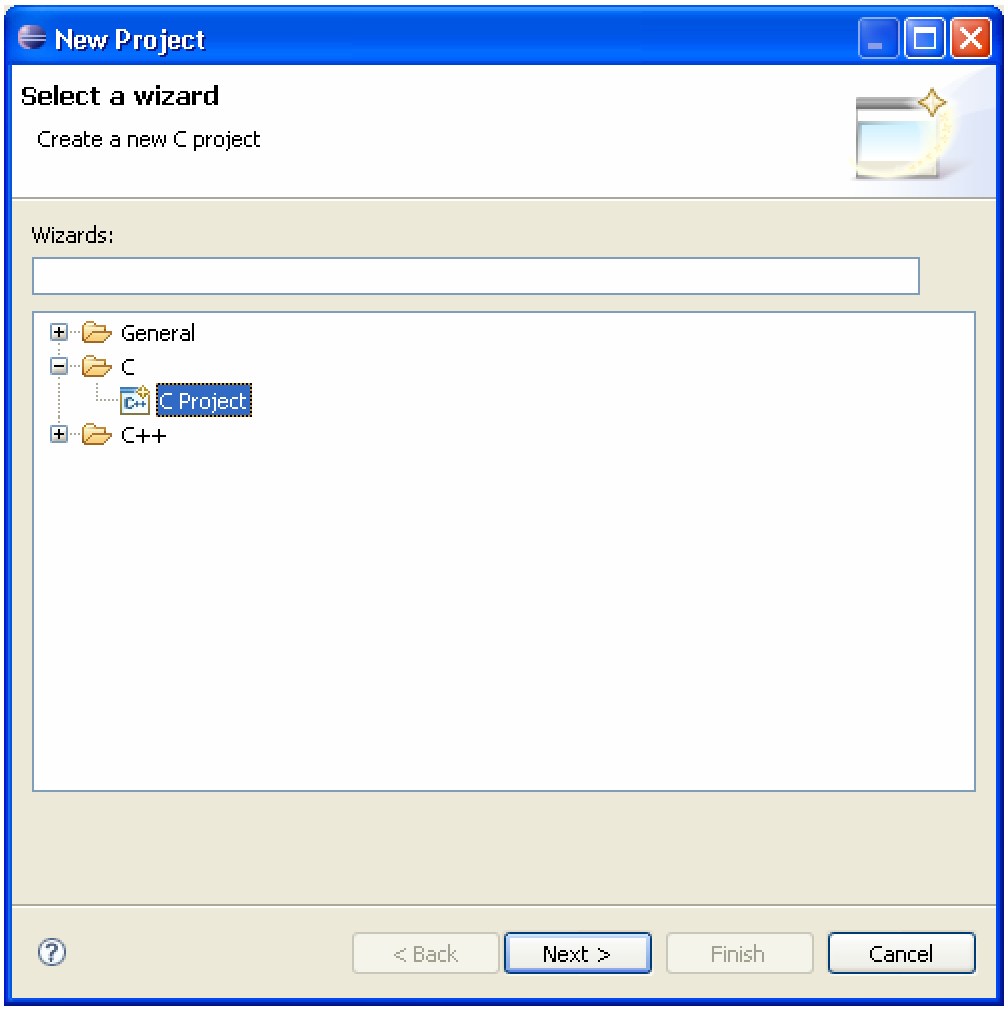
1.8 Document History

Revision	**Date**	**Changes**
ISSUE #0	18/07/08	First Release
ISSUE #1	11/09/08	Added Library Setup paragraph; added new STDIO APIs description paragraph 3.9; added STRING APIs description paragraph 3.10

2 Library setup

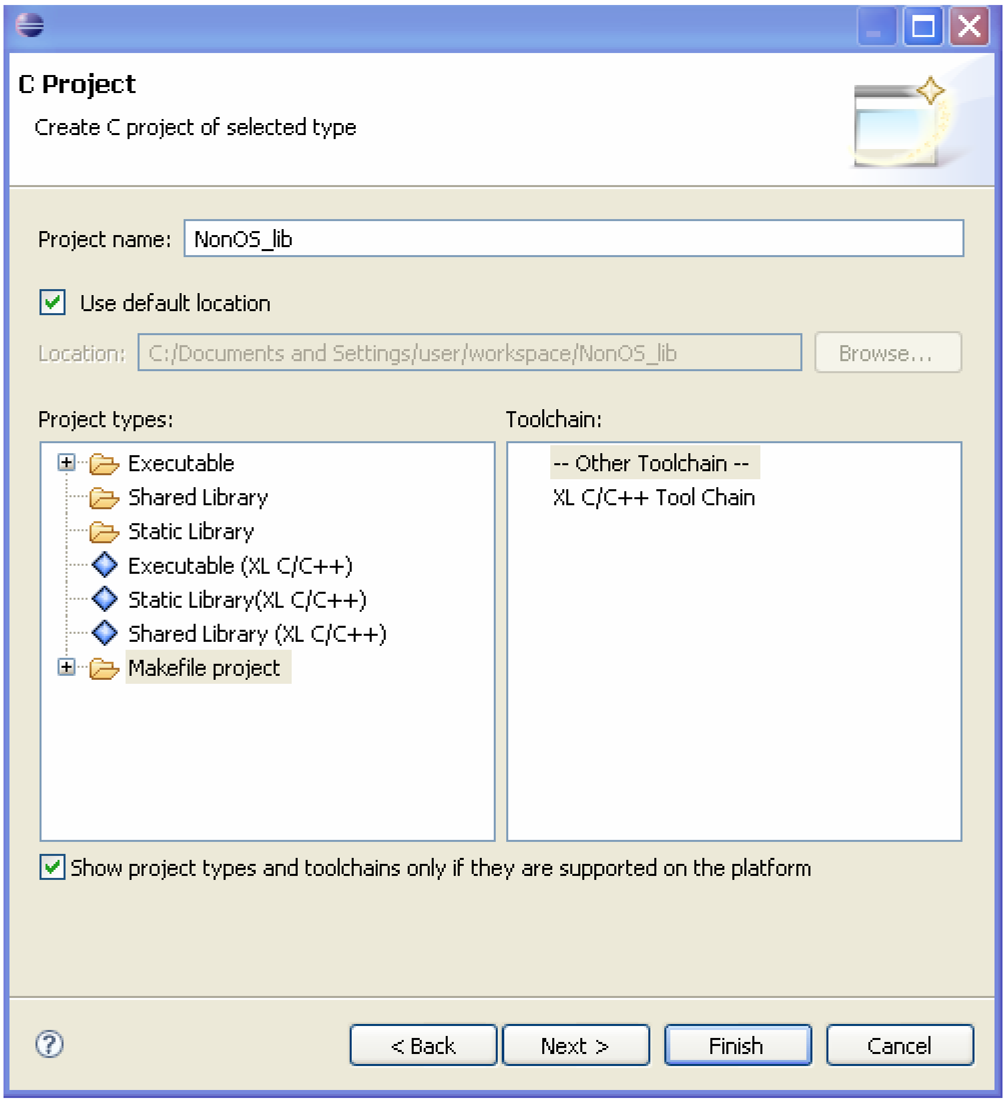
To Build a project for the GE863-PRO^3 and to import the Low Level Driver APIs follow the steps below (see also the document TelitGE863PRO3 Development Environment User Guide on the Telit official website$^)$.

- Open the IDE (**Start menu** → **All Programs** → **YAGARTO IDE** → **Eclipse Platform**).
- Open the menu **File** → **New** → **Project**.
- Open the **C** folder and choose the **C Project** entry, then click **Next**.



- Fill in the **Project Name** text field; in the **Project Types** area select the **Makefile Project** folder.

- In the **Toolchain** area choose the **Other Toolchain** entry and then click **Finish**.



C Project

Create C project of selected type

Project name: NonOS_lib

✓ Use default location

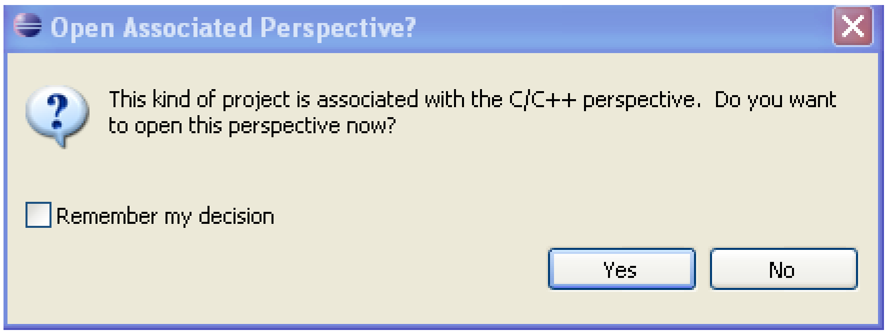
Location: C:/Documents and Settings/user/workspace/NonOS_lib

Project types:
- Executable
- Shared Library
- Static Library
- Executable (XL C/C++)
- Static Library(XL C/C++)
- Shared Library (XL C/C++)
- Makefile project

Toolchain:
- -- Other Toolchain --
- XL C/C++ Tool Chain

✓ Show project types and toolchains only if they are supported on the platform

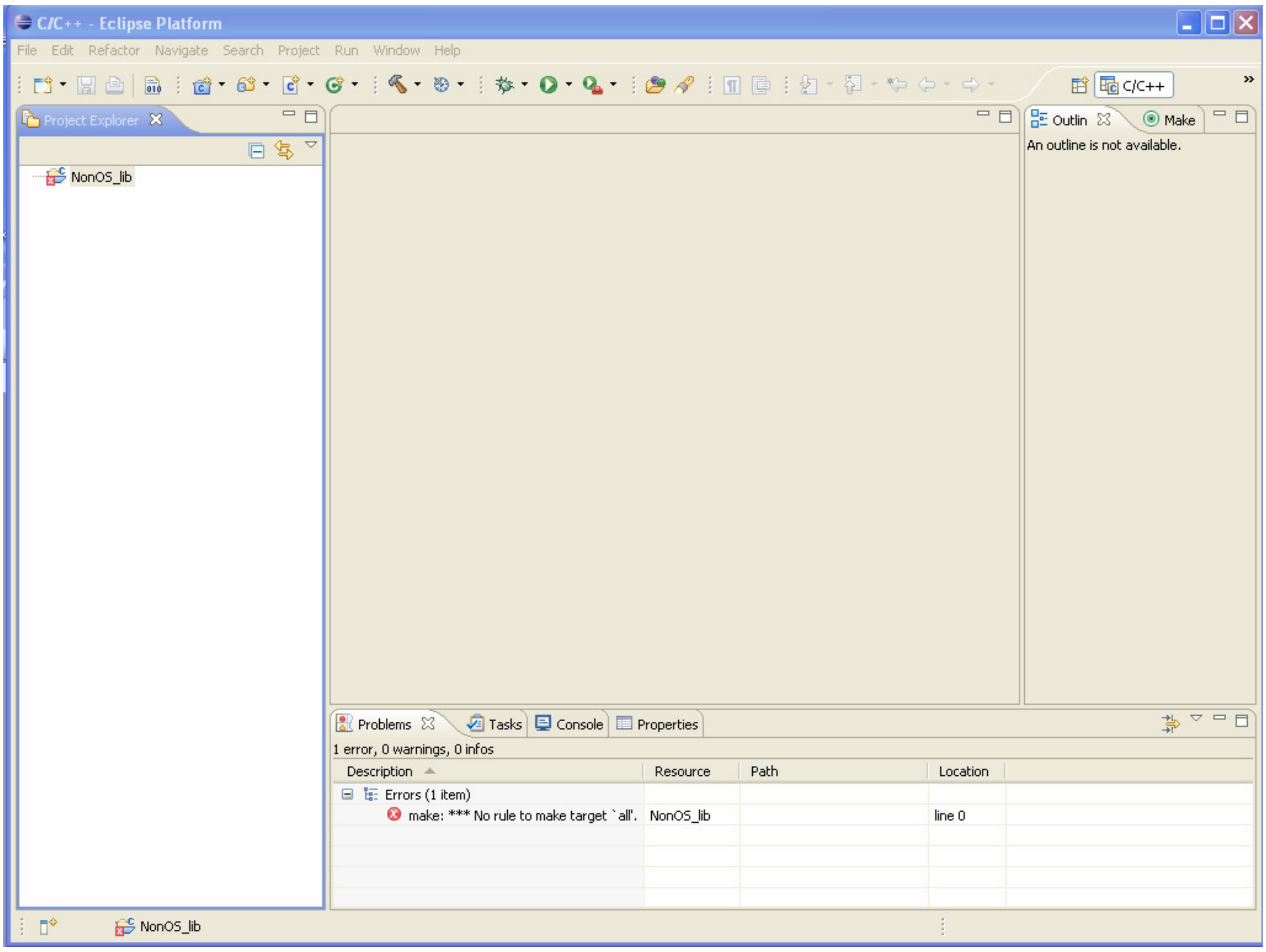
- If the window entitled **Open Associated Perspective** comes up answer **Yes**. This will open the IDE's C/C++ perspective.


Open Associated Perspective?

This kind of project is associated with the C/C++ perspective. Do you want to open this perspective now?

Remember my decision

- Now you should see the workspace with the newly created project: it will be empty because currently there are no files. The message in the panel **Problems:** *"make: *** No rule to make target `all'."* appears because at the moment in the project there are no files and it has not been yet configured the build directory.



An outline is not available.

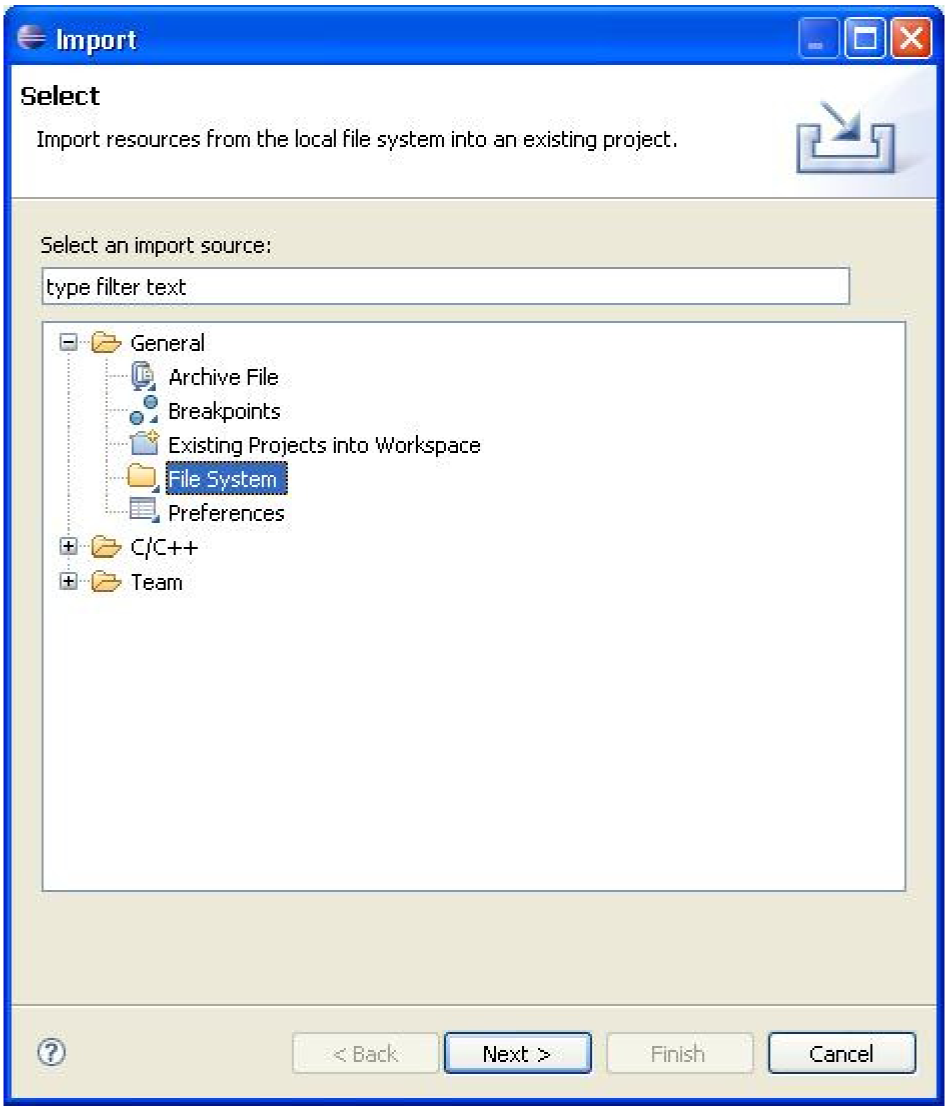
1 error, 0 warnings, 0 infos

Description	Resource	Path	Location
Errors (1 item)			
make: *** No rule to make target `all'.	NonOS_lib		line 0

- To start developing an application for the PRO^3 you need to download the files contained in the directory **GE863PRO3_nonOS_Low_Level_Drivers_Library_Example** provided by Telit.
  To get them following next steps:
  - go to http://www.telit.com/ → Download Zone,
  - LOGIN with user and password provided by Telit.
  - Click on **Software Tools_GSM/GPRS,**
  - Click on **GE863-PRO^3**
  - Click on the **ZIP_DOWNLOAD** of the **Telit_GE863-PRO^3_Low_Level_Drivers_Library_Example.zip**
  - **Save** it in a path like C:\Documents and Settings\user\Desktop.
  - UnZip **Telit_GE863-PRO^3_Low_Level_Drivers_Library_Example.zip** (the unzip creates the directory **GE863PRO3_nonOS_Low_Level_Drivers_Library_Example**)

- To start developing an application for the PRO^3 you need to import the files contained in the directory **GE863PRO3_nonOS_Low_Level_Drivers_Library_Example** provided by Telit. These files supply the drivers and configurations necessary to make the system work. Moreover in the main.c there is an application which can show you how to use several peripherals. To import the directory open the menu **File** → **Import**.

- Open the folder **General** and choose the entry **Filesystem** and then click on the **Next**.


Import

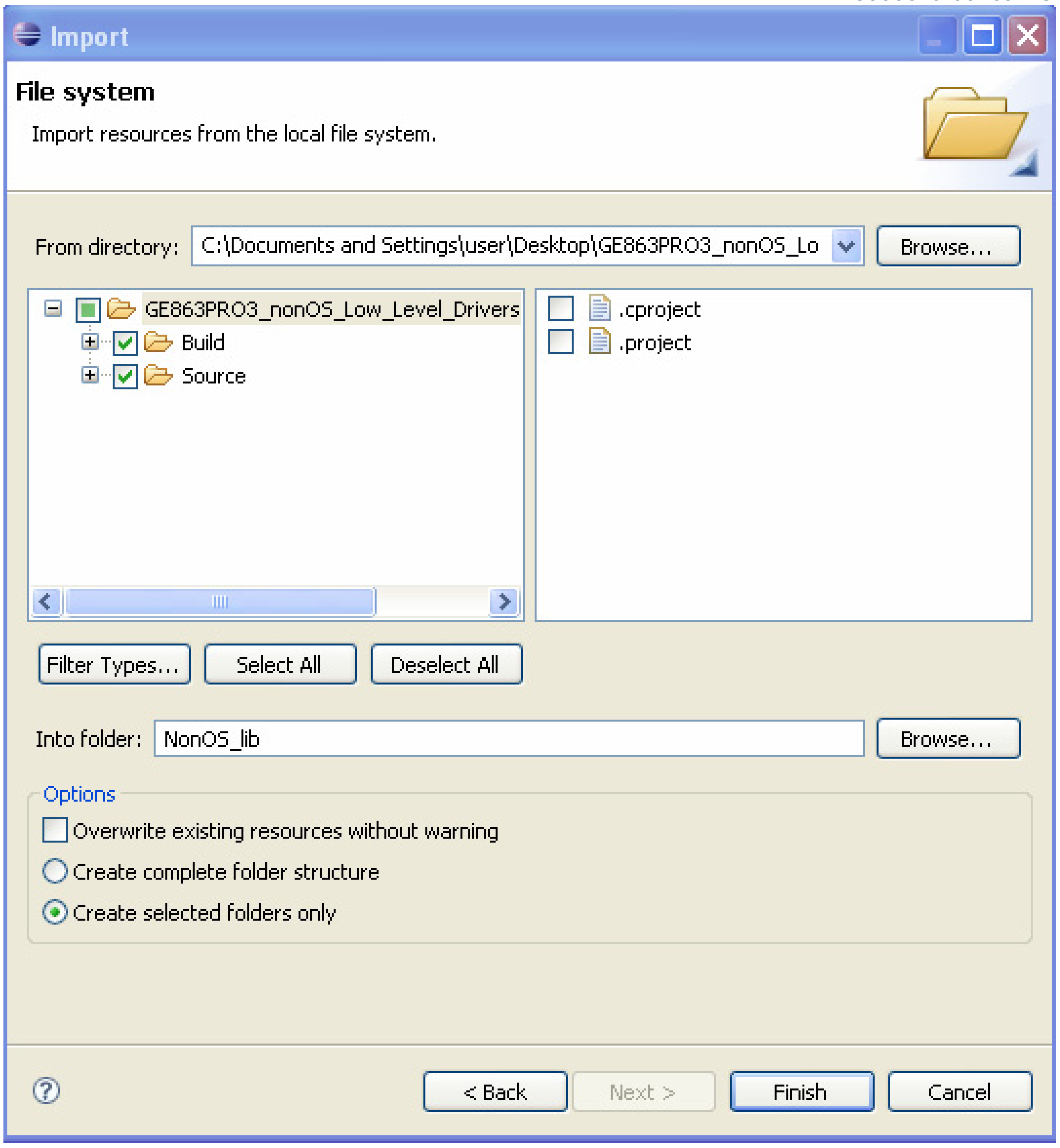
Select

Import resources from the local file system into an existing project.

Select an import source:

type filter text

- Browse to the directory containing the files to be imported and check on the folder **GE863PRO3_nonOS_Low_Level_Drivers_Library_Example**, then click **Finish**.



From directory: C:\Documents and Settings\user\Desktop\GE863PRO3_nonOS_Lo

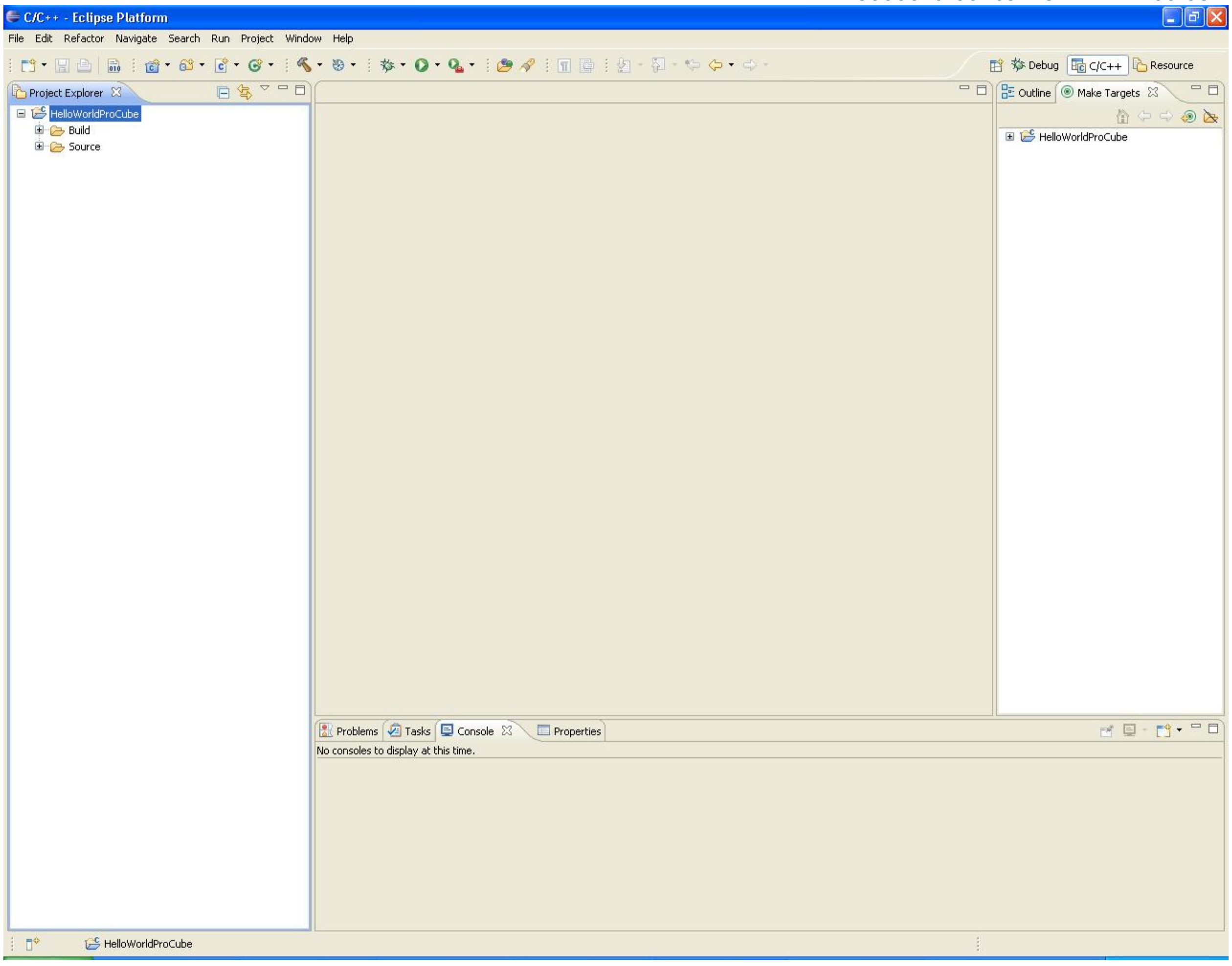
- GE863PRO3_nonOS_Low_Level_Drivers
  - ✓ Build
  - ✓ Source
- ☐ .cproject
- ☐ .project

Into folder: NonOS_lib

Options
- ☐ Overwrite existing resources without warning
- ○ Create complete folder structure
- ● Create selected folders only

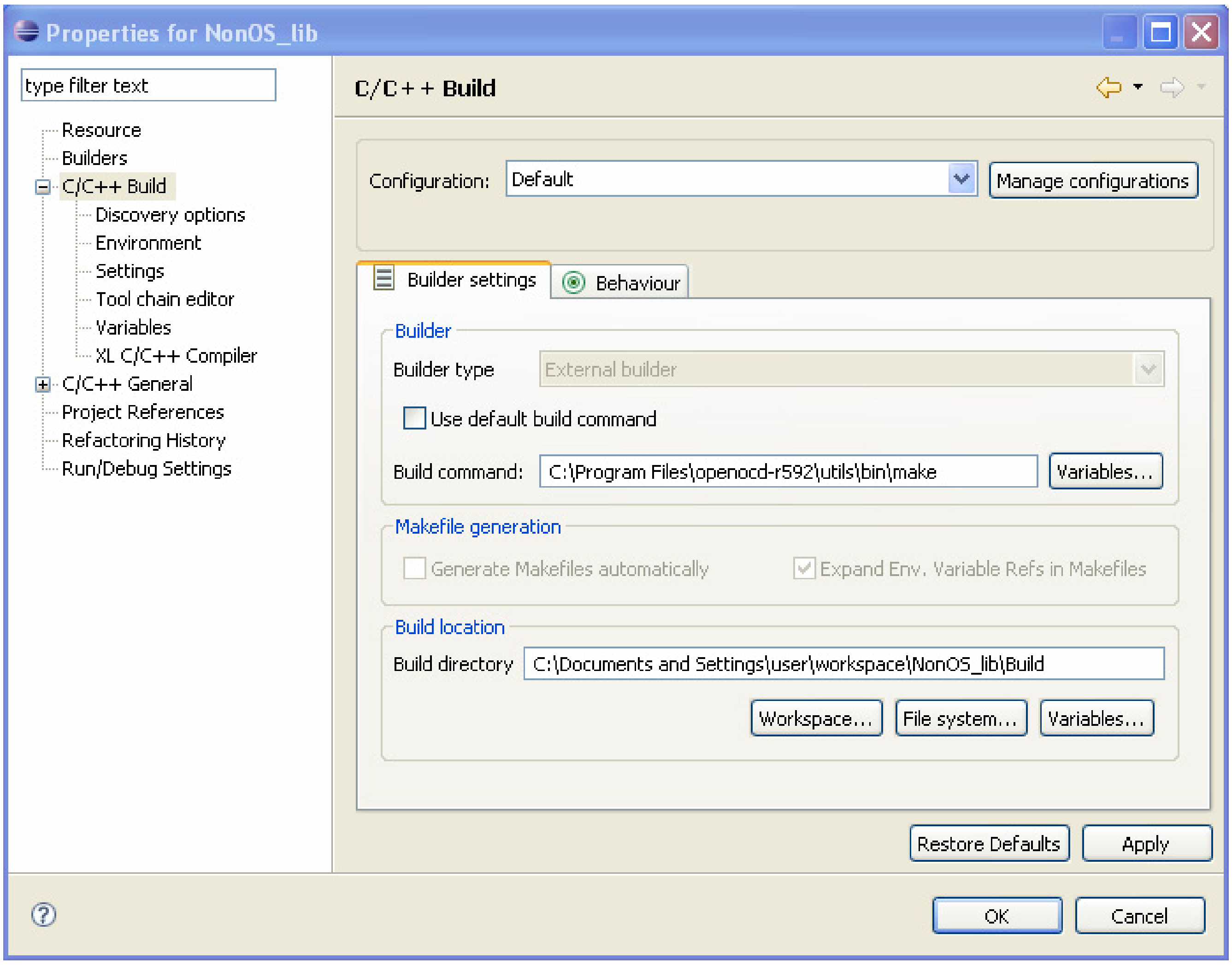
Click **Finish**

- Now you should see the workspace with the imported folders contained in the project.



No consoles to display at this time.

- Before building the project it’s necessary to configure the build directory. Select the project and open the menu **File → Properties**.
- In the left column choose the **C/C++ Build** entry and in the right part of the window select the tab **Builder Settings**. Fill in the **Build Directory** text field with the build directory of the NonOS_lib project, and then click **Ok**.


Properties for NonOS_lib

type filter text

- Resource
- Builders
- C/C++ Build
  - Discovery options
  - Environment
  - Settings
  - Tool chain editor
  - Variables
  - XL C/C++ Compiler
- C/C++ General
- Project References
- Refactoring History
- Run/Debug Settings

C/C++ Build

Configuration: Default

Builder settings | Behaviour

Builder

Builder type: External builder

☐ Use default build command

Build command: C:\Program Files\openocd-r592\utils\bin\make

Makefile generation

☐ Generate Makefiles automatically   ✓ Expand Env. Variable Refs in Makefiles

Build location

Build directory: C:\Documents and Settings\user\workspace\NonOS_lib\Build

- Go to **Project** → **Build All** and, if there are no errors, you will find the binaries in the directory previously inserted in the project preferences.

3 Low Level Driver APIs

3.1 Overview

This chapter describes the basic application and the low level driver provided for the GE863-PRO[3]. The general architecture of this environment is based on an abstraction layer between the hardware and high level code (Application).
The architecture is structured in the following way:

Testing routines
Application
Drivers (Low Level Driver)
HW (Microcontroller Peripherals)

Following chapters will focus on the low level driver APIs description.

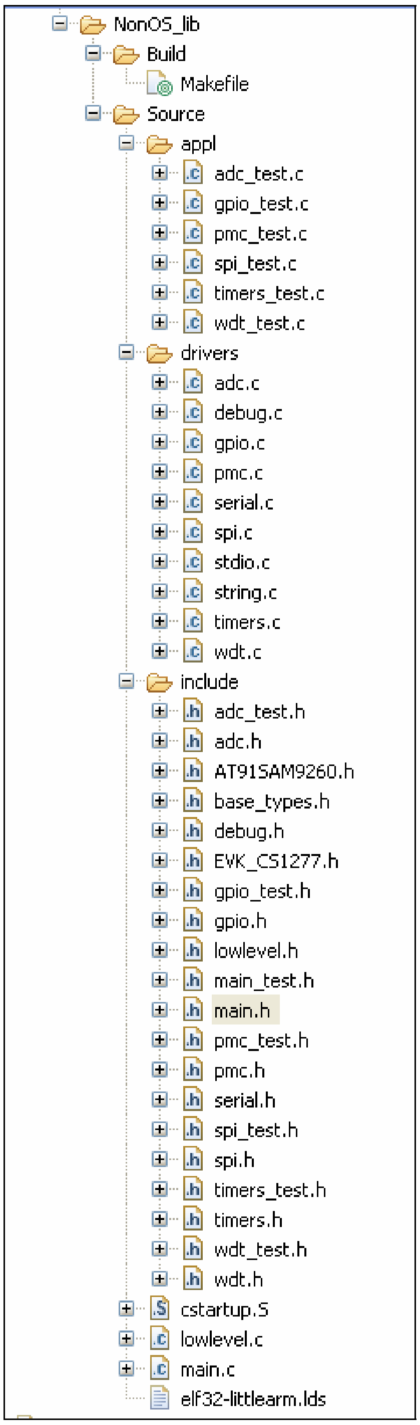
The drivers that will be described are the following:

- ADC
- DBGU (SERIAL) INTERFACE
- GPIO
- PMC
- SERIAL INTERFACES
- SPI
- STDIO
- STRING
- TIMERS
- WATCHDOG
- AIC (Advanced Interrupt Controller, see document Atmel AT91SAM 9260 Summary Datasheet [7]).

Drivers are explained in details through their APIs, with input and output parameters and mostly they are reported in a chronological order.

User can find in appendix a Low Level Driver flowchart, macro definitions in the library and C types redefinitions.

The baseline project is organised in a tree structure, as shown in the figure 1 below.



```
NonOS_lib
    Build
        Makefile
    Source
        appl
            adc_test.c
            gpio_test.c
            pmc_test.c
            spi_test.c
            timers_test.c
            wdt_test.c
        drivers
            adc.c
            debug.c
            gpio.c
            pmc.c
            serial.c
            spi.c
            stdio.c
            string.c
            timers.c
            wdt.c
        include
            adc_test.h
            adc.h
            AT91SAM9260.h
            base_types.h
            debug.h
            EVK_CS1277.h
            gpio_test.h
            gpio.h
            lowlevel.h
            main_test.h
            main.h
            pmc_test.h
            pmc.h
            serial.h
            spi_test.h
            spi.h
            timers_test.h
            timers.h
            wdt_test.h
            wdt.h
        cstartup.S
        lowlevel.c
        main.c
        elf32-littlearm.lds
```

Fig. 1

3.2 APIs summary

In the following table a summary of the APIs is shown.

Drivers	APIs	Description
ADC	*ADC_SoftReset ()*	*Performs a peripheral software reset*
	ADC_CfgPmc ()	*Enables Peripheral clock in PMC for ADC*
	ADC_CfgModeReg ()	*Configures the Mode Register of the ADC controller*
	ADC_GetModeReg ()	*Returns the Mode Register of the ADC controller value*
	ADC_CfgTimings ()	*ADC_CfgTimings: Configures the different necessary timings of the ADC controller.*
	ADC_DisableChannel ()	*Disables the desired channel*
	ADC_EnableChannel ()	*Enables the desired channel*
	ADC_EnableIt ()	*Enables ADC interrupt*
	ADC_Init()	*Initialize the ADC peripheral*
	ADC_StartConversion ()	*Performs a software request for an analog to digital conversion*
	ADC_IrqHandler ()	*Handler Routine for ADC peripheral*
	ADC_GetConvertedDataCh0 ()	*Returns the Channel 0 Converted Data*
	ADC_GetConvertedDataCh1 ()	*Returns the Channel 1 Converted Data*
	ADC_GetConvertedDataCh2 ()	*Returns the Channel 2 Converted Data*
	ADC_GetConvertedDataCh3 ()	*Returns the Channel 3 Converted Data*
	ADC_GetLastConvertedData ()	*Returns the Last Converted Data*
	ADC_GetStatus ()	*Returns ADC Interrupt Status*
DBGU	*DBGU_Configure ()*	*Configures DBGU peripheral and disable relative interrupts*
	DBGU_SetBaudrate()	*Changes Baudrate on Debug Serial Unit*
	DBGU_PrintAscii ()	*This function is used to send a string through the DBGU channel (Very low level debugging)*
	DBGU _PrintHex8 ()	*This function is used to print a 32-bit value in hexa*
GPIO	*GPIO_Setup ()*	*Configures PIO in peripheral mode according to the platform informations. It also disables interrupts*
	GPIO_SetAPeriph ()	*mux the pin to the "A" internal peripheral role.*
	GPIO _SetBPeriph ()	*mux the pin to the "B" internal peripheral role.*
	GPIO _SetDeglitch ()	*enable/disable the glitch filter; mostly used with IRQ handling*
	GPIO _SetGpioInput ()	*mux the pin to the gpio controller (instead of "A" or "B" peripheral), and configure it for an input*
	GPIO _SetGpioOutput ()	*mux the pin to the gpio controller (instead of "A" or "B" peripheral), and configure it for an output*
	GPIO _SetMultiDrive ()	*enable/disable the multi-driver; This is only valid for output and allows the output pin to run as an open collector output*
	GPIO _SetValue ()	*assuming the pin is muxed as a gpio output, set its value*
	GPIO_GetValue()	*reads the pin's value (works even if it's not muxed as a gpio)*
PMC	*PMC_CfgMck ()*	*Configures the main oscillator to the corresponding value*
	PMC _CfgPck ()	*Configures the Programmable Clock Register (Processor Clock - PCK) to the corresponding value*
	PMC _CfgPlla ()	*Configures the pll frequency to the corresponding value*
	PMC _CfgPllb ()	*Configures the pll frequency to the corresponding value*

USART	*UART0_Select ()*	*Sets Global Pointer to USART0*
	UART0_Init ()	*Initializes Serial Interface and disable relative interrupts*
	UART0_SetBaudrate ()	*Changes Baudrate on current USART*
	RxByteFromSerial ()	*Waits until a character is received in the uart*
	UART0_PrinAascii ()	*This function is used to send a string through the USART channel (Very low level debugging)*
	UART0_Putc ()	*Transmits a Character*
SPI	*SPI_Reset ()*	*Resets the SPI controller*
	SPI_Configure ()	*Configures a SPI peripheral as specified*
	SPI_ConfigureNpcs ()	*Configures a chip select of a SPI peripheral*
	SPI_Enable ()	*Enables a SPI peripheral*
	SPI_Disable ()	*Disables a SPI peripheral*
	SPI_Close ()	*Closes SPI: disables IT, disables transfert, closes PDC*
	SPI_Write ()	*Sends data through a SPI peripheral. If the SPI is configured to use a fixed peripheral select, the npcs value is meaningless. Otherwise, it identifies the component which shall be addressed*
STDIO	*printf()*	*Outputs a formatted string on the DBGU stream, using a variable number of arguments*
	sprint()	*Writes a formatted string inside another string*
	snprintf()	*Stores the result of a formatted string into another string*
	puts()	*Outputs a string on stdout*
	PutChar()	*This function writes a character inside the given string*
STRING	*memcpy()*	*Copies data from a source buffer into a destination buffer*
	memset()	*Fills a memory region with the given value*
	strchr()	*Search a character in the given string*
	strcpy()	*Copy from source string to destination string*
	strlen()	*Returns the length of a given string*
	strncmp()	*Compare the first specified bytes of 2 given strings*
	strncpy()	*Copy the first number of bytes from source string to destination string*
	strrchr()	*Search a character backword from the end of given string*
Timers	*TC_Configure ()*	*Configures TC*
	PIT_Configure ()	*Configures PIT*
	TC_Handler ()	*Handler for TC interrupt*
	PIT_Handler ()	*Handler for PITC interrupt*
WDT	*WDT_Disable ()*	*Disables watchdog*
	WDT_GetPeriod ()	*Translates ms into Watchdog Compatible value.*
	WDT_SetMode ()	*Set Watchdog Mode Register*
	WDT_Restart ()	*Restart Watchdog*

3.3 Analog to Digital Converter Driver (ADC)

This section describes APIs exported for the ADC interface. The files involved in the driver are the following:

- *adc.c*: main source with the driver APIs
- *adc.h*: header of driver (definitions and prototypes)

3.3.1 ADC APIs

3.3.1.1 ADC_SoftReset

Prototype:
void ADC_SoftReset (AT91PS_ADC pADC)
This method performs a peripheral software reset, simulating a hardware reset. Should be called as the first one in a adc init sequence. It is useful at start up time.

Parameters:
pADC = AT91C_BASE_ADC (pointer to a ADC controller)

Returns:
None

3.3.1.2 ADC_CfgPmc

Prototype:
void ADC_CfgPmc (void)

This method enables the Peripheral Clock in the Power Management Controller (PMC) for ADC clocking, hence this should be called at the beginning of adc init.
AD Converter needs in fact to be clocked by Peripheral Clock (Prescaled Main Clock) in order to perform analog to digital conversions.

Parameters:
none

Returns:
none

3.3.1.3 ADC_CfgModeReg

Prototype:
void ADC_CfgModeReg (AT91PS_ADC pADC, UNIT32 mode)

This method configures the Mode Register of the ADC controller. In this case the source trigger, conversion resolution, and conversion mode (sleep for power saving or normal) should be programmed through a configuration mask. The right peripheral's register address must be pointed (see AT91SAM9260.h header also), and give in input a valid configuration mask. The mask can be a 4 bytes hexavalue (0x$B_7B_6B_5B_4B_3B_2B_1B_0$) or a most comprehensive *OR bitwise* form.

Parameters:
pADC = AT91C_BASE_ADC (pointer to a ADC controller)
mode = mode register with following values in a bitwise form could be given in the following way:

```
AT91C_ADC_TRGEN_DIS          | // Software triggering
AT91C_ADC_TRGSEL             | // With no effect
AT91C_ADC_LOWRES_10_BIT      | // 10-bit result output
AT91C_ADC_SLEEP_NORMAL_MODE;   // Normal Mode
```

Or

```
AT91C_ADC_TRGEN_DIS          | // Software triggering
AT91C_ADC_TRGSEL             | // With no effect
AT91C_ADC_LOWRES_8_BIT       | // 8-bit result output
AT91C_ADC_SLEEP_MODE;          // Sleep Mode
```

Returns:
none

3.3.1.4 ADC_GetModeReg

Prototype:
UINT32 ADC_GetModeReg (AT91PS_ADC pADC)

This method returns the Mode Register of the ADC controller value.

Parameters:
pADC = AT91C_BASE_ADC (pointer to a ADC controller).

Returns:
ADC_MR register value. (For details see datasheet [7] at page 740)

3.3.1.5 ADC_CfgTimings

Prototype:
void ADC_CfgTimings (AT91PS_ADC pADC,
UINT32 mck_clock,
UINT32 adc_clock,
UINT32 startup_time,
UINT32 sample_and_hold_time)

This method acts again on the mode register, but only configures the different necessary timings of the ADC controller. Giving in input adc clock and master clock in MHz, startup and sample&hold time in µs, remaining fields are programmed by relative formula (see Atmel datasheet [7] page 741).

Parameters:
pADC = AT91C_BASE_ADC (pointer to a ADC controller)
adc_clock = 5 (in MHz)
mck_clock = 48 (Master Clock in MHz)
*startup*_time = 21 (in us)
sample_and_hold_time = 800 (in µs)

Returns:
none

3.3.1.6 ADC_DisableChannel

Prototype:
void ADC_DisableChannel (AT91PS_ADC pADC, UINT32 channel)

This method disables the desired channel/s. The right peripheral's register address must be addressed (see AT91SAM9260.h header), and give in input a valid AD channel.
The micro has 4 AD channels, AT91C_ADC_EOC0… AT91C_ADC_EOC3 (see AT91SAM9260 Atmel library).

Parameters:
pADC = AT91C_BASE_ADC (pointer to a ADC controller)
channel = selected channel with following bitwise form syntax (if all channels are enabled):
```
AT91C_ADC_EOC0 | AT91C_ADC_EOC1 | AT91C_ADC_EOC2 | AT91C_ADC_EOC3
```

Returns:
None

3.3.1.7 ADC_EnableChannel

Prototype:
void ADC_EnableChannel (AT91PS_ADC pADC, UINT32 channel)

This method enables the desired channel/s. The right peripheral's register address must be addressed (see AT91SAM9260.h header), and give in input a valid AD channel.
The micro has 4 AD channels, AT91C_ADC_EOC0… AT91C_ADC_EOC3 (see AT91SAM9260 Atmel library).

Parameters:
pADC = AT91C_BASE_ADC (pointer to a ADC controller)
channel = selected channel with following bitwise form syntax (all channels enabled):
`AT91C_ADC_EOC0 | AT91C_ADC_EOC1 | AT91C_ADC_EOC2 | AT91C_ADC_EOC3`

Returns:
none

3.3.1.8 ADC_EnableIt

Prototype:
void ADC_EnableIt (AT91PS_ADC pADC, UINT32 flag)

This method enables ADC interrupts.

Parameters:
pADC = AT91C_BASE_ADC (pointer to a ADC controller)
flag = IT to be enabled (flag indicate the interrupt event/s the user wants to enable – see Atmel datasheet [7] page 745):
see previous paragraph

Returns:
none

3.3.1.9 ADC_Init

Prototype:
void ADC_Init(void)

Initializes the ADC peripheral calling adc low APIs driver. This method sequentially resets the converter, activates the peripheral clock, sets the mode register and activates interrupts for the conversion control. User must call this method before starting conversion (see next paragraph).

Parameters*:*
none

Returns*:*
none

3.3.1.10 ADC_StartConversion

Prototype:
void ADC_StartConversion (AT91PS_ADC pADC)

This method performs a software request for an analog to digital conversion. Result of conversion on previous selected channel/s should be obtained on interrupt handler routine (see ADC_irq_handler).

Parameters:
pADC = AT91C_BASE_ADC (pointer to a ADC controller)

Returns:
none

3.3.1.11 ADC_IrqHandler

Prototype:
void ADC_IrqHandler (void)

This is the handler Routine for ADC peripheral where the converted data must be stored (use following ADC_GetConvertedDataCHx methods).

Parameters:
none

Returns:
none

3.3.1.12 ADC_GetConvertedDataCh0

Prototype:
UINT32 ADC_GetConvertedDataCh0 (AT91PS_ADC pADC)

This method returns the Channel 0 Converted Data in high (10 bit) or low (8 bit) resolution previously programmed in *ADC_CfgModeReg*.

Parameters:
pADC = AT91C_BASE_ADC (pointer to a ADC controller)

Returns:
ADC_CDR0 = register value in low or high resolution. (For details see datasheet [7] at page 748)

3.3.1.13 ADC_GetConvertedDataCh1

Prototype:
UINT32 ADC_GetConvertedDataCh1 (AT91PS_ADC pADC)

This method returns the Channel 1 Converted Data in high (10 bit) or low (8 bit) resolution previously programmed in *ADC_CfgModeReg*.

Parameters:
pADC = AT91C_BASE_ADC (pointer to a ADC controller)

Returns:
ADC_CDR1 register value in low or high resolution. (For details see datasheet [7] at page 748)

3.3.1.14 ADC_GetConvertedDataCh2

Prototype:
UINT32 ADC_GetConvertedDataCh2 (AT91PS_ADC pADC)

This method returns the Channel 2 Converted Data in high (10 bit) or low (8 bit) resolution previously programmed in *ADC_CfgModeReg*.

Parameters:
pADC = AT91C_BASE_ADC (pointer to a ADC controller)

Returns:
ADC_CDR2 register value in low or high resolution. (For details see datasheet [7] at page 748)

3.3.1.15 ADC_GetConvertedDataCh3

Prototype:
UINT32 ADC_GetConvertedDataCh3 (AT91PS_ADC pADC)

This method returns the Channel 3 Converted Data in high (10 bit) or low (8 bit) resolution previously programmed in *ADC_CfgModeReg*.

Parameters:
pADC = AT91C_BASE_ADC (pointer to a ADC controller)

Returns:
ADC_CDR3 register value in low or high resolution. (For details see datasheet [7] at page 748)

3.3.1.16 ADC_GetLastConvertedData

Prototype:

UINT32 ADC_GetLastConvertedData (AT91PS_ADC pADC)

This method returns the Last Converted Data in high (10 bit) or low (8 bit) resolution previously programmed in *ADC_CfgModeReg*.

Parameters:
pADC = AT91C_BASE_ADC (pointer to a ADC controller)

Returns:
ADC_LCDR register value. (For details see datasheet [7] at page 745)

3.3.1.17 ADC_GetStatus

Prototype:
UINT32 ADC_GetStatus (AT91PS_ADC pADC)

This method returns ADC Status register’s content. Its flags indicate whether a conversion has been completed, an error occurred etc. (For details see datasheet [7] at page 744).

Parameters:
pADC = AT91C_BASE_ADC (pointer to a ADC controller)

Returns:
ADC_SR register value. (For details see datasheet [7] at page 744)

3.4 Debug Unit Driver (DBGU)

This section describes in details the main APIs for the DBGU interface management. The driver is implemented through the following source files:

- *debug.c*: main source with the driver APIs
- *debug.h*: header of driver (definitions and prototypes)

3.4.1 DBGU APIs

3.4.1.1 DBGU_Configure

Prototype:
void DBGU_Configure (UINT32 baudrate)

This method configures the serial debug unit.
User should take care to do following steps:

- call UART*0_select(USART_DBGU)* before. This assures the right peripheral addressing (see chapter 3.7 for serial interface driver).
- pass in input a valid baudrate (normally a standard value: 115200, 9600 etc.) through the BAUDRATE conversion macro (see appendix 6.2). Following operation are performed: reset of receiver, interrupts disabling, storing of baudrate converted value, setting of functional mode (normal), re-enabling of receiver and transmitter unit.

Parameters:
baudrate = must be already converted from bits/sec to register content through the BAUDRATE conversion macro. (normally passing a standard value: 115200, 9600 etc.). See appendix 6.2.

Returns:
none

3.4.1.2 DBGU_SetBaudrate

Prototype:
void DBGU_setBaudrate (UINT32 baudrate)

This method changes Baudrate on Debug Serial Unit.

Parameters:

baudrate = must be already converted from bits/sec to register content through the BAUDRATE conversion macro. (normally passing a standard value: 115200, 9600 etc.). See appendix 6.2.

Returns:
none

3.4.1.3 DBGU_PrintAscii

Prototype:
*void DBGU_PrintAscii (const INT8 *buffer)*

This function is used to send a string through the DBGU channel. Hyper terminal, correctly configured, will be able to display debug messages sent with this API.

Parameters
**buffer* = pointer to a string item (for example *buffer=”This is a DEBUG Message!”)

Returns:
none

3.4.1.4 DBGU_PrintHex8

Prototype:
void DBGU_PrintHex8 (UINT32 value)

This function is used to print a 32-bit value in hexadecimal format.

Parameters:
value = unsigned 32 bit value to be printed in hexadecimal format.

Returns:
none

3.5 GPIO Controller Driver

This section describes in detail the main APIs for the Parallel I/O Controller (GPIO) management.
User could implement a table containing all information related to the set of pins needed for his specific application (platform).
See par. 3.5.1.1 for the table implementation.
With a single call of *pio_setup()* API (see par. 3.5.1.1) user may configure all pins described in the mentioned table, otherwise he will configure single pins with the apposite APIs.
Table information consists of: pin name (literal constant describing the pin functionality), pin number (a number within the interval 0…95, which is the limit of the micro GPIO pins), default value (in case of output configuration), attribute for pullup/opendrain/glitch filter configuration and eventually the main GPIO destination (input, output or peripheral A/B control).

The driver is implemented through the following source files:

- *gpio.c*: main source with the driver APIs
- *gpio.h*: header of driver (definitions and prototypes)

3.5.1 GPIO APIs

3.5.1.1 GPIO_Setup

Prototype:
*INT32 GPIO_Setup (const struct pio_desc *pio_desc, UINT16 table_size)*

This is the main driver feature which could be used once at startup time (i.e. inside hw_init() routine).
If the user correctly points his platform description table, the setup routine iteratively configures the entire set of pins.

Parameters:
** pio_desc* = pointer to the platform description table:

```
pio_desc {
        const INT8  *pin_name;
        UINT16 pin_num;
        UINT16 dft_value;
        UINT8 attribute;
        enum pio_type type;
      };
```

table_size = limit to avoid scan exceeding of GPIO bank

Data Fields detailed description:
pin_name = literal constant describing the pin functionality (i.e. “PA01”,”PB02” etc.)
pin_num = Pin number 0....95
dft_value = Default value for outputs (1 / 0)

attribute =
```
PIO_PULLUP    I/O attribute - enable pullup on selected pin
PIO_DEFAULT   I/O attribute - reset attribute
PIO_DEGLITCH  I/O attribute - enable glitch filter on selected pin
NO_DEGLITCH   I/O attribute - disable glitch filter on selected pin
PIO_OPENDRAIN I/O attribute - enable opendrain on selected pin
NO_OPENDRAIN        I/O attribute - disable opendrain on selected pin
```

type =
PIO_PERIPH_A (Peripheral A Muxing)
PIO_PERIPH_B (Peripheral B Muxing)
PIO_INPUT GPIO (Input Configuration)
PIO_OUTPUT GPIO (Output Configuration)

Returns:
pin > 0: Number of configured pins (Peripheral correctly configured)
pin = EINVAL (-1): Peripheral not correctly configured

3.5.1.2 GPIO_SetAPeriph

Prototype:
INT32 GPIO_SetAPeriph (UINT32 pin, INT32 use_pullup)

This method mux the pin given in input to the "A" internal peripheral role. Control of pin is assigned to the connected peripheral.

Parameters:
pin = id number of pin (from 0 to 95)
use_pullup = flag for pin pullup enabling/disabling
(PIO_PULLUP / PIO_DEFAULT)
Returns:
0 = method correctly executed
EINVAL = Error code returned when the PIN is unknown (-1)

3.5.1.3 GPIO_SetBPeriph

Prototype:
INT32 GPIO_SetBPeriph (UINT32 pin, INT32 use_pullup)

This method mux the pin given in input to the "B" internal peripheral role. Control of pin is assigned to the connected peripheral.

Parameters:
pin = id number of pin (from 0 to 95)
use_pullup = flag for pin pullup enabling/disabling
(PIO_PULLUP / PIO_DEFAULT)
Returns:

0 = method correctly executed
EINVAL = Error code returned when the PIN is unknown (-1)

3.5.1.4 GPIO_SetDeglitch

Prototype:
INT32 GPIO_Set_Deglitch (UINT32 pin, INT32 is_on)

This method enables/disables the glitch filter; mostly used with IRQ handling.

Parameters:
pin = id of pin (from 0 to 95)
is_on = flag for enabling/disabling the filter
PIO_DEGLITCH / PIO_DEFAULT

Returns:
0 = method correctly executed
EINVAL = Error code returned when the PIN is unknown (-1)

3.5.1.5 GPIO_SetGpioInput

Prototype:
INT32 GPIO_SetGpioInput (UINT32 pin, INT32 use_pullup)

This method mux the pin to the GPIO controller (instead of "A" or "B" peripheral), and configure it for an input.

Parameters:
pin = id of pin (from 0 to 95)
use_pullup = flag for pin pullup enabling/disabling (PIO_PULLUP / PIO_DEFAULT)

Returns:
0 = method correctly executed
EINVAL = Error code returned when the PIN is unknown (-1)

3.5.1.6 GPIO_SetGpioOutput

Prototype:
INT32 GPIO_SetGpioOutput (UINT32 pin,
INT32 use_pullup,
INT32 value)

This method mux the pin to the GPIO controller (instead of "A" or "B" peripheral), and configure it for an output.

Parameters:
pin = id of pin (from 0 to 95)
use_pullup = flag for pin pullup enabling/disabling (PIO_PULLUP / PIO_DEFAULT)
value = level to set on output pin (0 / 1)

Returns:
0 = method correctly executed
EINVAL = Error code returned when the PIN is unknown (-1)

3.5.1.7 GPIO_SetMultiDrive

Prototype:
INT32 GPIO_SetMultiDrive (UINT32 pin, INT32 is_on)

This method enables/disables the multi-driver. This is only valid for output and allows the output pin to run as an open collector output.

Parameters:
pin = id of pin (from 0 to 95)
is_on = flag for enabling/disabling the opendrain configuration (PIO_OPENDRAIN / PIO_DEFAULT)

Returns:
0 = method correctly executed
EINVAL = Error code returned when the PIN is unknown (-1)

3.5.1.8 GPIO_SetValue

Prototype:
INT32 GPIO_SetValue (UINT32 pin, INT32 value)

Assuming the pin is muxed as a GPIO output, sets its value.

Parameters:
pin = id of pin (from 0 to 95)
value = level to set on output pin (0 / 1)

Returns:
0 = method correctly executed
EINVAL = Error code returned when the PIN is unknown (-1)

3.5.1.9 GPIO_GetValue

Prototype:
INT32 GPIO_GetValue (UINT32 pin)

reads the pin's value (works even if it's not muxed as a gpio).

Parameters:
pin = id of pin (from 0 to 95)

Returns:
ret_val = pin's value
EINVAL = Error code returned when the PIN is unknown (-1)

Reproduction forbidden without Telit Communications S.p.A. written authorization - All Rights Reserved

3.6 Power Management Controller Driver (PMC)

This section describes in details the main APIs for the Power Management Controller (PMC) management.
The driver is implemented through the following source files:

- *pmc.c*: main source with the driver APIs
- *pmc.h*: header of driver (definitions and prototypes)

3.6.1 PMC APIs

3.6.1.1 PMC_CfgMck

Prototype:
INT32 pmc_CfgMck (UINT32 pmc_mckr,
UINT32 timeout)

This method configures the main oscillator to the corresponding value.

Parameters:
pmc_mckr = Master Clock Settings. Default value is:
`(AT91C_PMC_CSS_PLLA_CLK | AT91C_PMC_PRES_CLK | AT91C_PMC_MDIV_2)  (Switch Master Clock MCK on PLLA output; PCK = PLLA = 2 * MCK)`
For SlowClock Mode user should change first item `AT91C_PMC_CSS_PLLA_CLK` in: `AT91C_PMC_CSS_SLOW_CLK`

timeout = ticks for Master Clock timeout configuration. Default value is:
`PLL_LOCK_TIMEOUT    (Timeout for PLL activation)`

Returns:
0 - method correctly executed
-1 - method not correctly executed

3.6.1.2 PMC_CfgPck

Prototype:
INT32 PMC_CfgPck (UINT8 x,
UINT32 clk_sel,
UINT32 prescaler)

This method configures the Programmable Clock Register (Processor Clock - PCK) to the corresponding value.

Parameters:
x = offset for Programmable Clock Output Enable (0/1)

clk_sel = clock source selection (Slow, Main, PLL-A, PLL-B)
prescaler = Programmable Clock Prescaler (0x0...0x7)

Returns:
0 - method correctly executed

3.6.1.3 PMC_CfgPlla

Prototype:
INT32 PMC_CfgPlla (UINT32 pmc_pllar, UINT32 timeout)

This method configures the pll frequency to the corresponding value.

Parameters:
pmc_pllar = `PLLA_SETTINGS` (PLLA Settings Default value)
timeout = ticks for PLL timeout configuration. Default value is:
```
PLL_LOCK_TIMEOUT    (Timeout for PLL activation)
```

Returns:
 0 - method correctly executed
-1 - method not correctly executed

3.6.1.4 PMC_CfgPllb

Prototype:
INT21 PMC_CfgPllb (UINT32 pmc_pllbr, UINT32 timeout)

This method configures the pll frequency to the corresponding value.

Parameters:
pmc_pllbr = = `PLLB_SETTINGS` (PLLB Settings Default value)
timeout = ticks for PLL timeout configuration. Default value is:
```
PLL_LOCK_TIMEOUT    (Timeout for PLL activation)
```

Returns:
 0 - method correctly executed
-1 - method not correctly executed

3.7 USART - Serial Driver

This section describes in details the driver APIs for the Serial Peripheral management.
The driver is implemented through the following source files:

- *serial.c*: main source with the driver APIs
- *serial.h*: header driver (definitions and prototypes)

3.7.1 USART APIs

3.7.1.1 UART0_Select

Prototype:
void UART0_Select (USART currentUSART)

This method set a Global Pointer to USART0, and should be called as first in a serial init process.

Parameters:
currentUSART = current pointer to a USART item. Possible values are:

USART_DBGU	*(to select the DBGU Unit)*
USART_US0	*(to select USART 0)*
USART_US1	*(to select USART 1)*
USART_US2	*(to select USART 2)*
USART_US3	*(to select USART 3)*
USART_US4	*(to select USART 4)*
USART_US5	*(to select USART 5)*

Returns:
none

3.7.1.2 UART0_Init

Prototype:
void UART0_Init (UINT32 baudrate)

This method initializes Serial Interface and disables relative interrupts.

Parameters:
baudrate = must be already converted from bits/sec to register content through the BAUDRATE convertion macro. (normally passing a standard value: 115200, 9600 etc.). See appendix 6.2.

Returns:
none

3.7.1.3 UART0_SetBaudrate

Prototype:
void UART0_setBaudrate (UINT32 baudrate)

This method changes Baudrate on current USART.

Parameters:
baudrate = must be already converted from bits/sec to register content through the BAUDRATE convertion macro. (normally passing a standard value: 115200, 9600 etc.). See appendix 6.2.

Returns:
none

3.7.1.4 RxByteFromSerial

Prototype:
INT32 RxByteFromSerial (void)

This method waits until a character is received by the usart.

Parameters:
none

Returns:
US_RHR register value.(for details see datasheet [7] pag. 486)

3.7.1.5 UART0_print_ascii

Prototype:
*void UART0_print_ascii (const INT8 *buffer)*

This function is used to send a string through the USART channel (Very low level debugging).

Parameters:
**buffer* = pointer to a string item (for example *buffer=”This is a DEBUG Message!”)

Returns:
none

3.7.1.6 UART0_putc

Prototype:
INT32 UART0_putc (INT32 ch)
Transmits a Character.

Parameters:
ch = character to print on terminal

Returns:
US_THR register value.(for details see datasheet [7] pag. 487)

3.8 SPI Driver

This section describes in details the main APIs for the Serial Peripheral Interface management.

The driver is implemented through the following source files:

- *spi.c*: main source with the driver APIs
- *spi.h*: library driver (definitions and prototypes)

3.8.1 SPI APIs

3.8.1.1 SPI_Reset

Prototype:
void SPI_Reset (AT91PS_SPI pSPI)

Resets the SPI controller. This is the method that user should call for first in an API application.

Parameters:
pSPI = AT91C_BASE_ SPI (pointer to an SPI instance).

Returns:
none

3.8.1.2 SPI_Configure

Prototype:
void SPI_Configure (AT91PS_SPI pSPI, UINT32 configuration)

This method configures a SPI peripheral. SPI peripheral must be correctly addressed with the input pointer, and pass in input a valid configuration mask. The mask can be a 4 bytes hexadecimal value (0x$B_7B_6B_5B_4B_3B_2B_1B_0$) or a most comprehensive *OR bitwise* form.
The configuration can be computed using several macros (see "SPI configuration macros") and the constants defined in spi.h and spi_test.h libs.
Clock on the peripheral is also enabled inside this method.

Parameters:
pSPI = AT91C_BASE_ SPI0 or AT91C_BASE_ SPI 1 (pointer to an SPI instance)

configuration = mask value for the SPI configuration register. Possible values are:
`AT91C_SPI_MSTR | AT91C_SPI_PS_FIXED | AT91C_SPI_MODFDIS | AT91C_SPI_PCS`
The following initialisations can be preformed:

- Master Mode
- Fault detection disabled

- Fixed peripheral select
- No CS selected

When the chipselect must be selected user must substitute `AT91C_SPI_PCS` with `NPCS0…NPCS3`.

Returns:
none

3.8.1.3 SPI_ConfigureNpcs

Prototype:

void SPI_ConfigureNpcs (AT91PS_SPI pSPI,
UINT32 npcs,
UINT32 configuration)

Configures a chip select of a SPI peripheral.
The mask can be a 4 bytes hexa value (0x$B_7B_6B_5B_4B_3B_2B_1B_0$) or a most comprehensive *OR bitwise* form.

Parameters:
pSPI = AT91C_BASE_ SPI0 or AT91C_BASE_ SPI 1 (pointer to an SPI instance)
npcs = Chip select to configure (0, 1, 2 or 3).
configuration = Desired chip select configuration (for details see datasheet [7] pag. 367).

Returns:
none

3.8.1.4 SPI_Enable

Prototype:

void SPI_Enable (AT91PS_SPI pSPI)

Enables SPI peripheral.

Parameters:
pSPI = AT91C_BASE_ SPI0 or AT91C_BASE_ SPI 1 (pointer to an SPI instance)

Returns:
none

3.8.1.5 SPI_Disable

Prototype:

void SPI_Disable (AT91PS_SPI pSPI)

Disables SPI peripheral.

Parameters:
pSPI = AT91C_BASE_ SPI0 or AT91C_BASE_ SPI 1 (pointer to an SPI instance)

Returns:
None

3.8.1.6 SPI_Close

Prototype:
void SPI_Close (AT91PS_SPI pSPI)

Disables interrupts, disables transfer, closes PDC (DMA).

Parameters:
pSPI = AT91C_BASE_ SPI0 or AT91C_BASE_ SPI 1 (pointer to an SPI instance)

Returns:
none

3.8.1.7 SPI_Write

Prototype:
```
void SPI_Write (AT91PS_SPI pSPI,
                UINT32 npcs,
                UINT16 data)
```

Sends data through SPI peripheral. If the SPI is configured to use a fixed peripheral (already programmed in par. 7.1.2), the *npcs* value is meaningless. Otherwise, it identifies the component which shall be addressed.

Parameters:
pSPI = AT91C_BASE_ SPI0 or AT91C_BASE_ SPI 1 (pointer to an SPI instance)
npcs = Chip select of the component to address (0, 1, 2 or 3).
data = Word of data to send.

Returns:
none

3.9 STDIO

This section describes in details the APIs for manage input and output operations.

The driver is implemented through the following source files:

- *stdio.c*: main source with the APIs

3.9.1 STDIO APIs

3.9.1.1 printf

Prototype:
*INT32 printf(const INT8 *pFormat, ...)*

This method outputs a formatted string on the DBGU stream, using a variable number of arguments.

Parameters:
pFormat = String that contains the text to be written
... = Depending on the format string, the function may expect a sequence of additional arguments.

Returns:
On success, the total number of characters written.
On failure, a negative number.

3.9.1.2 sprintf

Prototype:
*INT32 sprintf(INT8 *pStr, const INT8 *pFormat, ...)*

This method writes a formatted string inside another string.

Parameters:
pStr = storage string
pFormat = string that contains the text to be written

Returns:
On success, the total number of characters written.
On failure, a negative number.

3.9.1.3 snprintf

Prototype:
*INT32 snprintf(INT8 *pString, size_t length, const INT8 *pFormat, ...)*

This method stores the result of a formatted string into another string.

Parameters:
pString = destination string
pFormat = string that contains the text to be written
length = length of destination string;

Returns:
On success, the total number of characters written.
On failure, a negative number.

3.9.1.4 puts

Prototype:
*INT32 puts(const INT8 *pStr)*

This method outputs a string on stdout.

Parameters:
pStr = string to output.

Returns:
On success, a non-negative value is returned.
On error, the function returns EOF.

3.9.1.5 PutChar

Prototype:
*INT32 PutChar(INT8 *pStr, INT8 c)*

This method writes a character inside the given string.

Parameters:
pStr = Storage string
c = Character to write.

Returns:
return 1.

3.10 STRING

This section describes in details the APIs for strings and memory handling functions.

The driver is implemented through the following source files:

- *string.c*: main source with the APIs

3.10.1 STRING APIs

3.10.1.1 memcpy

Prototype:
*void * memcpy(void *pDestination, const void *pSource, size_t num)*

This method copies data from a source buffer into a destination buffer. The two buffers must NOT overlap.

Parameters:
pDestination = destination buffer
pSource = dource buffer
num = number of bytes to copy.

Returns:
return the destination buffer.

3.10.1.2 memset

Prototype:
*void * memset(void *pBuffer, INT32 value, size_t num)*

This method fills a memory region with the given value.

Parameters:
pBuffer = pointer to the start of the memory region to fill
value = value to fill the region with
num = number of bytes to copy, size to fill in bytes.

Returns:
return pointer to the memory region.

3.10.1.3 strchr

Prototype:
*INT8 * strchr(const INT8 *pString, INT32 character)*

This method Search a character in the given string.

Parameters:
Param = pString   pointer to the start of the string to search
character = the character to find.

Returns:
return pointer to the character location.

3.10.1.4 strcpy

Prototype:
*INT8 * strcpy(INT8 *pDestination, const INT8 *pSource)*

This method copies from source string to destination string.

Parameters:
pDestination = pointer to the destination string
pSource = pointer to the source string.

Returns:
return pointer to the destination string.

3.10.1.5 strlen

Prototype:
*size_t strlen(const INT8 *pString)*

This method returns the length of a given string.

Parameters:
param = pString Pointer to the start of the string

Returns:
return the length of a given string.

3.10.1.6 strncmp

Prototype:
*INT32 strncmp(const INT8 *pString1, const INT8 *pString2, size_t count)*

This method compares the first specified bytes of 2 given strings.

Parameters:
String1 = Pointer to the start of the 1st string
pString2 = Pointer to the start of the 2nd string

count = Number of bytes that should be compared

Returns:
return 0 if equals; > 0 if 1st string > 2nd string; < 0 if 1st string < 2nd string.

3.10.1.7 strncpy

Prototype:
*INT8 * strncpy(INT8 *pDestination, const INT8 *pSource, size_t count)*

This method copies the first number of bytes from source string to destination string.

Parameters:
pDestination = Pointer to the start of destination string
pSource = Pointer to the start of the source string
count = Number of bytes that should be copied.

Returns:
pointer to the destination string.

3.10.1.8 strrchr

Prototype:
*INT8 * strrchr(const INT8 *pString, INT32 character)*

Search a character backward from the end of given string.

Parameters:
param = pString   Pointer to the start of the string to search
character = The character to find.

Returns:
a pointer to the character location.

3.11 Timers Driver

This section describes in details the APIs for the Timer Units management (PIT/TC).

The driver is implemented through the following source files:

- *timers.c*: main source with the driver APIs
- *timers.h*: header driver (definitions and prototypes)

3.11.1 Timers APIs

3.11.1.1 TC_Configure

Prototype:
void TC_Configure (void)

This method directly configures and starts the Timer Counter Unit – TC.
Interrupts are also enabled inside the method.
It generates interrupts with half of SlowClock frequency.

Parameters:
none

Returns:
none

3.11.1.2 PIT_Configure

Prototype:
void PIT_Configure (void)

This method directly configures and starts the Periodic Interval Timer (PIT). It provides the operating system's scheduler interrupt. A resolution of 1 msec. interrupt routine is generated by this method (see pitc_handler).

Parameters:
none

Returns:
None

3.11.1.3 TC_Handler

Prototype:
void TC_Handler (void)

Handler routine for TC interrupt called with SlowClock/2 frequency.

Parameters:
none

Returns:
none

3.11.1.4 PIT_Handler

Prototype:
void PIT_Handler (void)

Handler routine for PITC (periodic) interrupt

Parameters:
none

Returns:
none

3.12Watchdog Controller Driver (WDT)

This section describes in details the APIs for the Watchdog management (WDTC).

The driver is implemented through the following source files:

- *wdt.c*: main source with the driver APIs
- *wdt.h*: header driver (definitions and prototypes)

3.12.1 WDT APIs

3.12.1.1 WDT_Disable

Prototype:
void WDT_Disable (AT91PS_WDTC pWDTC)

This method disables the watchdog.

Parameters:
pWDTC = AT91C_BASE_WDTC (Pointer to the WDT instance).

Returns:
none

3.12.1.2 WDT_GetPeriod

Prototype:
UINT16 WDT_GetPeriod (UINT32 ms)

This method translates *ms* into a watchdog compatible value the user can directly store in the WDV field of the Watchdog Timer Mode register WDT_MR. This value represents the maximum watchdog restart period.

Parameters:
ms = period in ms to be converted in WDV (Watchdog Counter Value).

Returns:
WDV = 12-bit value for Watchdog Counter Value.

3.12.1.3 WDT_SetMode

Prototype:
void WDT_SetMode (AT91PS_WDTC pWDTC, UINT32 Mode)

This method set the Watchdog Mode Register.

Parameters:
pWDTC = AT91C_BASE_WDTC (Pointer to the WDT instance).
Mode = Configuration Mode for watchdog controller.
AT91C_WDTC_WDRSTEN | *AT91C_WDTC_WDD*
Watchdog Reset Enable with default delta value (16 sec).

Returns:
none

3.12.1.4 WDT_Restart

Prototype:
void WDT_Restart (AT91PS_WDTC pWDTC)

This method restarts the Watchdog. It is very important to assure the watchdog refresh where the controller must remain enabled. The consequence of out of time restarting (underflow) or error watchdog is the microcontroller reset.

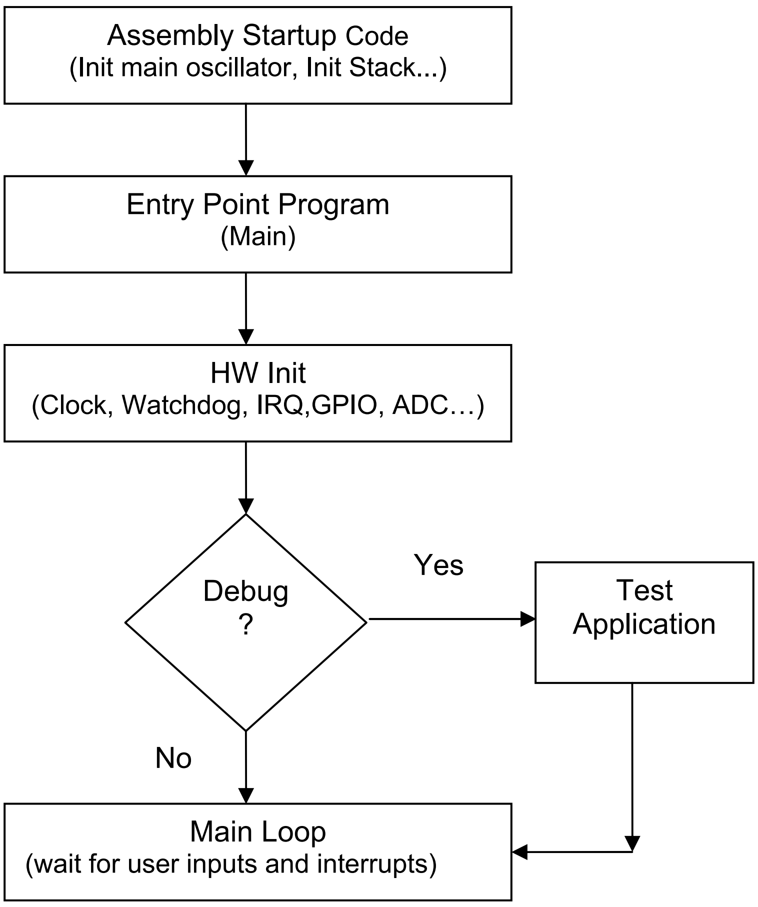
Parameters:
pWDTC = AT91C_BASE_WDTC (Pointer to the WDT instance).

Returns:
none

4 Appendix

4.1 Low Level Driver APIs Flowchart

This is the software entry point, defined in the *main.c* source file.
As for each embedded software environment the hardware initialization routine is performed at startup. A test application is launched on a terminal emulator if the *debug mode* is activated at compile time in the project. Eventually an infinite loop is entered waiting for user inputs or any interrupt events coming from peripherals. The procedure can be summarised with a flow diagram:


Assembly Startup Code
(Init main oscillator, Init Stack...)
Entry Point Program
(Main)
HW Init
(Clock, Watchdog, IRQ,GPIO, ADC…)
Debug
?
Yes
Test
Application
No
Main Loop
(wait for user inputs and interrupts)

4.2 Macro functions

This sections describes all macro functions useful for the developer:

Prototype:
BAUDRATE(mck,baud)
This is the macro used to configure the *debug serial* baudrate.
It is defined inside debug.h

Parameters:
mck = this is the clock value. Available values: MASTER_CLOCK / SLOW_CLOCK
baud = baudrate value expressed in bit / sec (use standard values as 115200, 9600, 300 etc.)

Prototype:
BAUDRATE0(mck,baud)
This is the macro used to configure the *serial interface* baudrate.
It is defined inside serial.h

Parameters:
mck = this is the clock value. Available values: MASTER_CLOCK / SLOW_CLOCK
baud = baudrate value expressed in bit / sec (use standard values as 115200, 9600, 300 etc.)

4.3 Basic C types redefinitions

Following C types introduced in our low level library correspond to the basic standard C types:

INT8 = signed char (8 bit)

UINT8 = unsigned char (8 bit)

INT16 = signed short int (16 bit)

UINT16 = unsigned short int (16 bit)

INT32 = signed long int (32 bit)

UNIT32 = unsigned long int (32 bit)

Previous definitions are collected in the *base_types.h* header file.