ADENEO

WINDOWS CE 5.0 PORT ON AT91SAM9263EK

TECHNICAL DESIGN DOCUMENT

Ref.: AT91SAM9263-TDD_CE5_100

HISTORY

BSP Version	Revision	Date	Update History
V 0.1.0	A	06/12/2006	Creation
V 0.9.0	A	28/06/2007	First Release
V 1.0.0	A	19/07/2007	Official Release

TABLE OF CONTENTS

1. Document presentation 6
1.1 Abbreviations used 6
1.2 Document purpose 6
2. BSP global architecture 7
2.1 Content 7
2.2 Files location and description 7
2.2.1 Code Location 7
2.2.2 The AT91SAM926X Directory 8
2.2.3 The AT91SAM9263 Directory 8
2.2.4 The AT91SAM9263EK Directory 8
2.3 Multi platform management 9
2.4 Components included in the BSP 9
2.4.1 Overview 9
2.4.2 List of the components 10
2.4.3 Dependencies management 10
2.5 Environment Variables 10
2.6 Automatic addition to the SDK 11
2.7 Memory mapping 12
2.7.1 Virtual Memory mapping 12
2.7.2 SDRAM mapping 13
2.8 Configuration files 15
2.8.1 Platform.reg 15
2.8.2 Platform.bib and config.bib 15
2.8.3 Platform.db and platform.dat 15
2.9 OEM Abstraction Layer (OAL) 15
2.9.1 Interrupts 15
2.9.2 PIOs management 18
2.9.3 Tick Management and High Resolution Timer 18
2.9.4 RTC Management 18
2.9.5 Reboot modes 19
2.9.6 Watchdog 19
2.9.7 Time Bomb 20
2.9.8 Suspend 20
2.9.9 Shutdown 20
2.9.10 Debug Serial 21
2.9.11 PLL management and core Frequency setting 21
2.9.12 KITL (Kernel Independent Transport Layer) 21
2.9.13 The EthDbg Library 22
2.10 Bootloader 24
2.10.1 Purpose 24
2.10.2 Description of the startup of the system 24
2.10.3 Update of the bootloader 24
2.10.4 The Bootloader Menu 24
2.11 How to store an image in flash 25
3. BSP's drivers 27
3.1 The USB Host driver 27
3.1.1 General description of a Windows CE USB host driver 27

3.1.2 Code Location 27
3.1.3 Description of the implementation 27
3.1.4 Registry Settings 27
3.2 The USB device driver 27
3.2.1 General description of a Windows CE USB device driver 28
3.2.2 Code Location 28
3.2.3 Description of the implementation 28
3.2.4 Registry Settings 29
3.2.5 USB Function clients : 29
3.3 The Display driver 29
3.3.1 General description of a Windows CE display driver 29
3.3.2 Implementation 30
3.3.3 Potential evolutions 31
3.3.4 Registry parameters 31
3.4 The ADS7843 Touch Screen driver 32
3.4.1 General description of a Windows CE touch screen driver 32
3.4.2 General description of the driver 33
3.4.3 Calibration application 33
3.4.4 Registry Settings 33
3.5 The Nand Flash driver 33
3.5.1 Description 33
3.5.2 Generic FMD library 34
3.5.3 Registry settings 35
3.6 The PWMC driver (PWMC) 35
3.6.1 General description of a Windows CE PWMC driver 35
3.6.2 Code Location 36
3.6.3 Description of the implementation 36
3.6.4 Registry Settings 36
3.7 The I2C (TWI) driver 37
3.7.1 General description of a Windows CE I2C (TWI) driver 37
3.7.2 Code Location 37
3.7.3 Description of the implementation 38
3.7.4 Registry Settings 38
3.8 The SPI driver 38
3.8.1 General description of a Windows CE SPI driver 38
3.8.2 Code Location 38
3.8.3 Description of the implementation 38
3.8.4 Limitations and known issues 39
3.8.5 Registry Settings 39
3.9 The SD Memory Card driver 40
3.9.1 General description of a Windows CE SD Memory Card driver 40
3.9.2 Code Location 40
3.9.3 Description of the implementation 41
3.9.4 Registry Settings 41
3.10 The AtapiEbi driver (EBI) 41
3.10.1 General description 41
3.10.2 Code Location 42
3.10.3 Description of the implementation 42
3.10.4 Registry Settings 43
3.11 EEPROM diver 44
3.11.1 Description 44
3.11.2 Code Location 44

3.11.3 Description of the implementation 44
3.11.4 Registry Settings 44
3.12 The Serial driver 45
3.12.1 General description of a Windows CE serial driver 45
3.12.2 Code Location 46
3.12.3 Implementation 46
3.12.4 Registry Settings 47
3.13 The Audio driver 48
3.13.1 General description of a Windows CE Audio driver 48
3.13.2 Code Location 48
3.13.3 Implementation 48
3.13.4 Additional and custom features 49
3.13.5 Registry Settings 50
3.14 The Ethernet driver 50
3.14.1 General description of the driver 50
3.14.2 Code Location 50
3.14.3 Implementation 51
3.14.4 Registry Settings 51

1. Document presentation

1.1 Abbreviations used

BSP	Board Support Package
OAL	OEM Adaptation Layer
PB	Platform Builder
WinCE	Windows CE
MDD	Model Device Driver – platform independent layer of a device driver
PDD	Platform Dependent Driver – platform dependent layer of a device driver

1.2 Document purpose

This document aims to give a description of the BSP for Windows CE 5.0 supporting the AT91SAM9263 processor and the AT91SAM9263EK board.

It focuses on the following points:

- global architecture of the BSP,
- for each component, a detailed description including:
  - purpose of the component,
  - behavior of the component,
  - parameters and configuration of the component,

This document will not give a detailed description of the implementation. This information is available in the documentation built with dOxygen.

This document addresses to developers that might perform maintenance or evolution on the BSP.

2. BSP global architecture

2.1 Content

The AT91SAM9263EK Board is built around the AT91SAM9263 processor.

The AT91SAM9263EK BSP contains the following components:

- the specific Windows CE bootloader,
- the OAL for AT91SAM9263EK board,
- drivers for devices available on the AT91SAM9263EK board.

In order to reduce the time required to port a BSP from one board to another, the code that's specific to the processor has been separated from the board specific code.

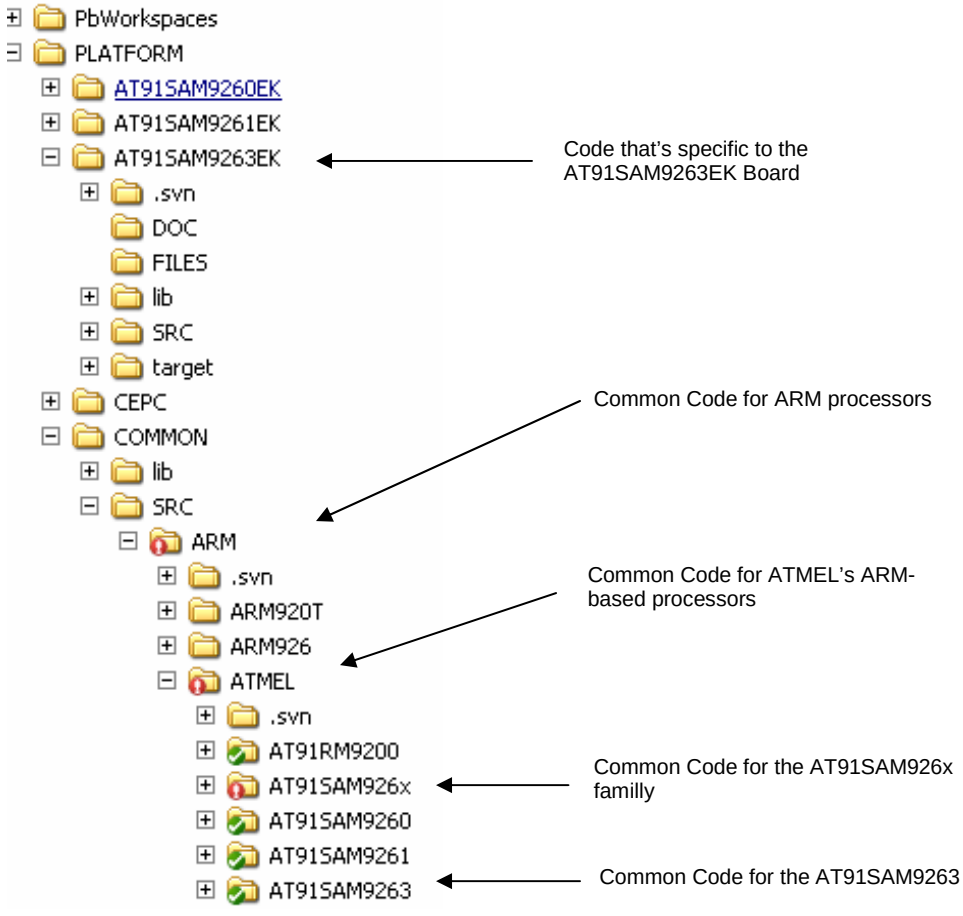
2.2 Files location and description

2.2.1 Code Location

The AT91SAM9263EK BSP is located under %WINCEROOT%\PLATFORM\AT91SAM9263EK (where %WINCEROOT% typically is C:\WINCE500). This BSP makes use of some code that's not specific to the board but to the processor, and thus that's been separated in another directory. The code that's specific to the AT91SAM9263 processor is located in:

- %WINCEROOT%\PLATFORM\COMMON\SRC\ARM\ATMEL\AT91SAM9263,
- %WINCEROOT%\PLATFORM\COMMON\SRC\ARM\ATMEL\AT91SAM926x.

This last directory contains code that's shared by the processors of the AT91SAM926x family (AT91SAM9260, AT91SAM9261, AT91SAM9263), whereas the first directory contains the part of the code that's used only for the AT91SAM9263.



Those directories can't be located elsewhere. This is a strong requirement from Platform Builder. However with the NTFS file system it's possible to create "junctions" and "hard links" that allow having multiple access location for the same data on the disk. For example we can have the BSP code located in c:\SourceControl\User\BSPs\AT91SAM9263EK and create a "Junction" to D:\WINCE500\PLATFORM\AT91SAM9263EK. This would allow us to have the code located in c:\SourceControl\User\BSPs\AT91SAM9263EK and still be able to compile it with platform builder. "Junctions" and "Hardlinks" are not available from the standard shell of Windows 2000/XP, but there are some free tools you can use (for example: http://forge.novell.com/modules/xfmod/project/?ntfslink).

2.2.2 The AT91SAM926X Directory

In this directory is located the code that is common for the AT91SAM926x processor family.

The AT91SAM926x directory contains the following file:

- *Dirs*: lists all subdirectories containing source files to be built when building a Windows CE image based on the BSP. In this case DRIVERS and KERNELS.

The AT91SAM926x directory contains the following subdirectories:

- *KERNEL:* This directory contains the source code for the various libraries that will help to build the kernel in the BSP. Those libraries will help to manage the DBGU, the interrupts, the PLL, the Power control, the RTC, the Timers, the Watchdog, and some other minor features. Those libraries may not be used in the BSP if the board needs a specific management.
- *DRIVERS:* This directory contains the source code for the various libraries that will help to create the driver in the BSP. Those libraries will help to manage the USB Host and device controllers, the serial ports, the SPI, the TWI, etc.
- BOOTLOADER: this directory contains the source file of the generic bootloader common to all BSPs. See paragraph on the bootloader for more details on its content.
- *INC:* This directory contains the include files that'll be included by the BSP and the common code to have the interfaces for the libraries and for the processor. Usually the files located in this directory, includes the header files of a specific processor. This allows us to use the exact same code (even the file inclusion) even if the declarations are different. This will be described in details in the Environment Variables section.

2.2.3 The AT91SAM9263 Directory

The AT91SAM9263 directory contains the following file:

- *Dirs*: lists all subdirectories containing source files to be built when building a Windows CE image based on the BSP. In this case DRIVERS and KERNELS.

The AT91SAM9263 directory contains the following subdirectories:

- *KERNEL:* This directory contains the source code for the various libraries that will help to build the kernel in the BSP.
- *DRIVERS:* This directory contains the source code for the various libraries that will help to create the driver in the BSP.
- *INC:* This directory contains the include files that'll be included by the common header files

2.2.4 The AT91SAM9263EK Directory

The AT91SAM9263EK directory contains the following files:

- AT91SAM9263EK.CEC: this file is the catalog file for the BSP: it contains the description of all components of the BSP for Platform Builder Catalog.
- AT91SAM9263EK.BAT: this file is launched each time a project based on the BSP is opened under Platform Builder. It allows setting various environment variables to preconfigure all Windows CE project based on the BSP.
- Dirs: lists all subdirectories containing source files to be built when building a Windows CE image based on the BSP.
- Source.cmn: contains environment variable to set when building source files from the BSP.

The AT91SAM9263EK directory contains the following subdirectories:

- CESYSGEN: this directory contains the default makefile for the build of the components of the BSP.
- SRC: this directory contains the sources files of the BSP which are separated in the various subdirectories:
  - DRIVERS: this directory contains all source files for the various drivers of the BSP. See paragraphs on the drivers for more details on its content.
  - BOOTLOADER: this directory contains the source file of the bootloader for the BSP. See paragraph on the bootloader for more details on its content.
  - INC: this directory contains include files for the platform, shared between the different components of the platform (Bootloader, OAL, drivers).
  - KERNEL: this directory contains the OAL of the BSP. See paragraph on the OAL for details on its content.
- FILES: this directory contains configuration files for the generation of the Windows CE image. See chapter on platform integration for details on its content.
- LIB: this directory contains lib files generated when building the BSP. See chapter on platform integration for details on its purpose.
- TARGET: this directory contains binary files generated when building the BSP. See chapter on platform integration for details on its purpose.

2.3 Multi platform management

The AT91SAM9263EK BSP has been developed for the AT91SAM9263EK board only.

To support other boards, you should duplicate the BSP and make whatever change you have to do in the clone BSP. The duplication of code won't be a problem here because most of the common code is located outside of the BSP (in COMMON\SRC\ARM\ATMEL).

2.4 Components included in the BSP

2.4.1 Overview

The AT91SAM9263EK BSP components are defined in AT91SAM9263EK.CEC (BSP's catalog file). The role of this file has changed quite a bit since WinCE 4.2. It used to control directly the build system. Now it only can play around with environment variables.

Basically what this file does is:

- providing a description for each component,

- if required, associating each component with an environment variable for managing the dependencies: when the variable is set, then the component is automatically included in the image,
- if required, associating a component with its source code.

2.4.2 List of the components

The following components are included in the BSP:

- Ethernet BootLoader (eboot). This component is required. It is associated to the Windows CE BootLoader.
- AT91SAM9263 Support: This component is required. It tells that the selected processor is the AT91SAM9263.
- LCDC: This component is available only for display based devices. It implements the graphic display driver.
- TouchScreen: This component implements the driver that manages the touchscreen controller.
- USB Host: This component is required for CE images which are using USB host.
- USB Device: This component is required for CE images which are using USB device.
- PWM: This component implements the driver that manages the generation of PWM signals.
- SPI: This component implements the driver that manages the AT91SAM9263 SPI bus, to connect Touch screen controller.
- NANDFLASH: This component allows storing data in the NandFlash as if it were a hard driver.
- SDMemory Card Driver: This component allows storing data in the SDCard as if it were a hard driver.
- Saved Registry : This component specifies if the hive registry is saved in NAND Flash
- AtapiEbi: This component implements the driver that manages an ATAPI IDE hard-disk.
- Ethernet Driver: This component implements the driver that manages an Ethernet communication.
- Serial Driver: This component implements the driver that manages a serial communication.
- I2C: This component implements the drive that manages the TWI bus protocol.
- EEPROM: This component implements the driver that manages an EEPROM memory through I2C communication.
- [This list is subject to change]

2.4.3 Dependencies management

Some drivers and components can't be used simultaneously, because they use the same multiplexed pins or resources. The BSP doesn't implement any kind of dependency management. It's the responsibility of the developer to take care of it.

2.5 Environment Variables

An environment variable is a variable of the windows/dos shell. You can set/access from either a batch file or an executable. That's how platform builder exchange parameters between its different parts.

Platform Builder sets some variables that it will use during the compilation. The most useful for the developer are:

- %_FLATRELEASEDIR%: This is the directory where the output files are stored before the makeimg (last step of the image creation) is executed.
- %_TARGETPLATROOT%: This is the directory of the BSP.
- %_PLATFORMROOT%: This is the PLATFORM directory.
- %_PUBLICROOT%: This is the PUBLIC directory.
- %_WINCEROOT%: This is the WINCE500 directory.
- %WINCEDEBUG%: This variable tells whether to build a debug or a release object.

Some other variables are set when you add components into your OS-Design (an OS-Design is also sometimes referred to as a Project). For example, if you add the “USB Mass Storage support” you’ll see that the following variable SYSGEN_USB_STORAGE is set to 1.

Variables beginning with SYSGEN_ are usually variables that are used to automatically add a component to an OS-Design: setting it means that the component will be selected. Other variables are set the other way around: the variable is set when the component is selected. It allows changing some build parameters very easily by adding/removing a component. For example, when you select the AT91SAM9263EK SPI Driver, it sets the variable BSP_AT91SAM9263EK_SPI to 1. You can then use this variables in various files: *.reg,*.bib;*.dat; *.dab, *.bat, makefile, sources, and sources.cmn.

Here is an abstract of the platform.reg file, it tells to include the SPI registry settings if the component is selected:

```
IF BSP_AT91SAM9263EK_SPI
 #include "$(_TARGETPLATROOT)\SRC\DRIVERS\SPI\SPI.reg"
ENDIF BSP_AT91SAM9263EK_SPI
```

It is very important to note that you can’t use those variables in the source code. If you want to play with #define by selecting/removing components, you’ll have to add it in the makefile or the sources file or in the sources.cmn.

Here is an abstract of the sources.cmn file. It creates a new #define for C source code if a particular environment variable is set. Please note that in this case the environment variable and the #define have the same name, but it’s not mandatory at all.

```
!IF ("$(OAL_NO_WATCHDOG)"== "1")
CDEFINES=$(CDEFINES) -DOAL_NO_WATCHDOG
!ENDIF
```

2.6 Automatic addition to the SDK

Platform builder can generate a SDK that allows developers to create application for their board. However if you want to provide libraries or .h file that belongs only to your BSP, you must include them in the SDK. The usual way to do that is to manually customize the SDK with Platform Builder SDK Wizard. Or you can copy your files in the following directories:

$(_PROJECTSDKROOT)\Inc

$(_PROJECTSDKROOT)\Lib\$(_CPUINDPATH)

This BSP provides a way to do that automatically.

Files can be copied into those directories during the build process, by including the following “xcopy” command in the makefile.inc of the module (if makefile.inc doesn’t exist, just create a new one).

```
CopyFilesToSDK:
 if not EXIST $(_PROJECTSDKROOT)\Inc mkdir $(_PROJECTSDKROOT)\Inc
```

xcopy /I /D /Q <header file(s)> $(_PROJECTSDKROOT)\Inc > nul

if not EXIST $(_PROJECTSDKROOT)\Lib mkdir $(_PROJECTSDKROOT)\Lib\$(_CPUINDPATH)

xcopy /I /D /Q <lib file(s)> $(_PROJECTSDKROOT)\Lib\$(_CPUINDPATH)\ > nul

To enable the previous instruction you have to modify your SOURCES file:

WINCETARGETFILES=CopyFilesToSDK

The SDK can be generated from the command line, but you have to configure it using the PlatForm Builder SDK wizard first.

buildsdk <path to pbxml>

2.7 Memory mapping

The memory mapping is platform-dependent. It is not located in the common code.

2.7.1 Virtual Memory mapping

This paragraph describes the memory mapping implemented for the system. This mapping is linked to the usage of the MMU by Windows CE. It associates physical memory areas to Virtual memory areas used by the system. Windows CE memory architecture requires the following organization of address map:

- 0x0000.0000 - 0x7FFF.FFFF: reserved for virtual mapping performed by MMU for applicative memory space.
- 0x8000.0000 – 0x9FFF.FFFF: used to map physical memory areas in cached and buffered mode. This mapping is performed by the kernel when image is launched. It's accessible in kernel-mode only.
- 0xA000.0000 – 0xBFFF.FFFF: map the same areas as previous, but in non-cached and non-buffered mode. It's accessible in kernel-mode only.
- 0xC000.0000 – 0xC3FF.FFFF: Reserved for internal kernel usage, depending on OS versions and microprocessor architecture. It's accessible in kernel-mode only.
- 0xC400.0000 – 0xDFFF.FFFF: used internally by kernel to map dynamically portions of physical memory in non-cached and non-buffered mode from applications, drivers or OAL. It's accessible in kernel-mode only.
- 0xE000.0000 – 0xFFFF.FFFF: Reserved for internal kernel usage, depending on OS versions and microprocessor architecture. It's accessible in kernel-mode only.

All these memory areas are directly managed and used by Windows CE kernel, except the mapping of physical memory into virtual address space (areas 0x8000.0000-0x9FFF.FFFF and 0xA000.0000-0xBFFF.FFFF) which is defined by the OEM.

Those areas are mapped during the very early stage of the kernel initialization, by parsing a global array "g_oalAddressTable". This table is transmitted to kernel at the start of the Windows CE image.

This array looks like the following:

AT91SAM9263EK_VA_BASE_REG	EQU	0x9FF00000
AT91SAM9263EK_VA_BASE_NAND	EQU	0x9EA00000
AT91SAM9263EK_VA_BASE_SRAM	EQU	0x9d000000
AT91SAM9263EK_VA_BASE_SDRAM	EQU	0x80000000
AT91SAM9263EK_VA_BASE_EXT_SRAM	EQU	0x9D700000

```
AT91SAM9263EK_BASE_SRAM          EQU          0x00300000
AT91SAM9263EK_BASE_SDRAM         EQU          0x20000000
AT91SAM9263EK_BASE_NAND          EQU          0x40000000
AT91SAM9263EK_BASE_REG           EQU          0xFFF00000
AT91SAM9263EK_BASE_EXT_SRAM      EQU          0x70000000

g_oalAddressTable

    DCD     AT91SAM9263EK_VA_BASE_REG,     AT91SAM9263EK_BASE_REG,     1     ; Internal registers.
    DCD     AT91SAM9263EK_VA_BASE_SRAM,    AT91SAM9263EK_BASE_SRAM,    1     ; INTERNAL SRAM (160KB).
    DCD     AT91SAM9263EK_VA_BASE_SDRAM,   AT91SAM9263EK_BASE_SDRAM,   64    ; AT91SAM9263EK SDRAM (64MB).
    DCD     AT91SAM9263EK_VA_BASE_NAND,    AT91SAM9263EK_BASE_NAND,    5     ; NAND Flash memory (1MB).
    DCD   AT91SAM9263EK_VA_BASE_EXT_SRAM,  AT91SAM9263EK_BASE_EXT_SRAM, 4
    DCD   0x00000000, 0x00000000,  0     ; end of table
```

This mapping is given in %_TARGETPLATROOT%\SRC\INC\cfg.inc. This mapping is used when MMU is activated at the startup of the Windows CE image.

Here is a more descriptive version of this table:

Desc.	Physical Base Address	Virtual Base Address	Size
SDRAM	0x20000000	0x80000000	32 MB
SRAM	0x00300000	0x9D000000	1 MB
Processor's register	0xFFF00000	0x9FF00000	1 MB
External NAND Flash	0x40000000	0x9EA00000	1 MB
External SRAM	0x70000000	0x9D700000	4MB

In this BSP the bootloader also turns MMU On and uses this mapping. The reason is that on ARM, data cache and MMU are very tightly linked, and we want to take advantage of the cache in the bootloader. It also allows using the same binary objects (we don't have to recompile the common code twice: once with a mapping for the bootloader, once with a mapping for the kernel).

2.7.2 SDRAM mapping

The SDRAM is used to store the binary image of the CE system, to host the data and the heap of the image and also to host the object store. The object store is a storage area that hosts the registry, the database and the RAM-disk of Windows CE.

A small portion of SDRAM is also used to host a structure that may be used to exchange data between the bootloader and the kernel, or between the kernel ISRs (Interrupt Service Routines) and the User-Level threads. This area is called the "drivers' global" or DRVGLOB. Part of this area is zeroed during the startup of the OS. It's defined in drv_glob.h.

The SDRAM mapping for the Bootloader is defined in %_TARGETPLATROOT%\SRC\BOOTLOADER\EBOOT\eboot.bib. It looks like :

```
MEMORY
;   Name      Start     Size      Type
```

```
;   -------  --------  --------  ----
    EBOOT    80000000  00030000  RAMIMAGE
    RAM      80030000  00020000  RAM
    PSHEAP   80050000  00008000  RESERVED
    EMACBUF  80058000  00012400  RESERVED
    DRVGLOB  8006b000  00001000  RESERVED
```

Description	Address	Size
EBOOT (code section for the booloader)	0x80000000	256 KB
RAM (data section for the bootloader)	0x80040000	64 KB
PSHEAP (a section reserved for a pseudo-heap management in the bootloader)	0x80050000	32 KB
EMACBUF (a section reserved for the Ethernet EmacB controller)	0x80058000	73kB
DRVGLOB (a section reserved for exchanging parameters between the kernel and the bootloader)	0x8006B000	4 KB

The SDRAM mapping for the Kernel is defined in %_TARGETPLATROOT%\FILES\config.bib. It looks like :

```
MEMORY
;   Name     Start     Size      Type
;   -------  --------  --------  ----
    BLDR     80000000  00058000  RESERVED
     EMACBUF  80058000  00013000  RESERVED
    DRVGLOB  8006b000  00001000  RESERVED
    NK       8006c000  00100000  RAMIMAGE
    RAM      8026c000  03B94000  RAM
    VIDEOMEM 83e00000  00200000  RESERVED
```

Description	Address	Size
BLDR (section reserved for the booloader)	0x80000000	352 KB
EMACBUF (a section reserved for the Ethernet EmacB controller)	0x80058000	76kB
DRVGLOB (a section reserved for exchanging parameters between the kernel and the bootloader)	0x80058000	4 KB
NK (Code section for the Kernel)	0x80059000	16 MB
RAM (data/heap/objectstore section for the kernel)	0x8026c000	59.5 MB
GDIRAM (a section reserved for GDI driver)	0x83E00000	2MB

Note: due to AUTOSIZE flag in Config.bib, if the CE image built is less (or more) than 32MB, then the build system automatically resizes the area reserved for the image to allow more (or less) space for RAM.

Besides at startup the OEMInit function will tell the kernel to get back the area reserved for eboot by using *OEMEnumExtensionDRAM.* The trouble is that in this case we have to make sure that during a warm boot, the bootloader doesn't get copied into the SDRAM and thus may not overwrite part of the object store. This is done by checking "Enable EBOOT space in memory" in the project's settings.

2.8 Configuration files

There are 4 types of configuration file:

- REG files: They describe the content of the default registry.
- BIB files: They describe the content of the image (files, modules, sections, …)
- DAT: They describe the file system organization of the device.
- DB files: They describe the Database of the device.

2.8.1 Platform.reg

Platform.reg is filled with the registry information for this BSP. It mostly contains the registry information to load the drivers.

2.8.2 Platform.bib and config.bib

Platform.bib is filled with the image content for this BSP. It gives the name of the different modules and files to be included in the image

Config.bib gives the memory layout for this BSP.

2.8.3 Platform.db and platform.dat

They are empty files.

2.9 OEM Abstraction Layer (OAL)

The OAL is the low-level layer of the Windows CE Kernel. In WinCE 5.0, the OAL has changed a lot because of the PQOAL (Production Quality OAL). Microsoft provides a wide range of libraries designed to help the OEM to implement its OAL. Those libraries implement most of the common part of the OAL. They may or may not be used depending on the wish of the OEM and the requirements of the platform. This common code is located in %_PLATFORMROOT%\COMMON\SRC.

2.9.1 Interrupts

2.9.1.1 Overview of the Interrupt system in WinCE 5.0

Here is an abstract of the Platform Builder documentation:

[…] Real-time applications use interrupts to respond to external events in a timely manner. The use of interrupts requires that an operating system (OS) balance performance against ease of use. Microsoft® Windows® CE balances these two factors by splitting interrupt processing into two steps: an interrupt service routine (ISR) and an interrupt service thread (IST).

Each interrupt request (IRQ) is associated with an ISR; an ISR can respond to multiple IRQ sources. When interrupts are enabled and an interrupt occurs, the kernel calls the registered ISR for that interrupt. Once finished, the ISR returns an interrupt identifier. The kernel examines the returned interrupt identifier and sets the associated event. When the kernel sets the event, the IST starts processing. […]

Basically this means that the interrupt management is split into two parts: one fast response with some limitation of use (ISR) and one is a delayed response that doesn't have any limitation (IST).

ISR means Interrupt Service Routine. The ISR is running within the Kernel space and is called by the global interrupt handler. The ISR cannot be blocking, it must return as soon as possible. The return value of the ISR indicates the IST that must run. An ISR can be statically embedded in the kernel (hard coded) or can be dynamically loaded by a driver. Usually an ISR only does this (returning a value), but it can also

be used to perform some time-critical treatment or to identify the source of the interrupt in case of shared interrupt (like PCI interrupt).

The IST is running in user mode (an IST can even be an application). IST means Interrupt Service Thread. It doesn't work the same way as a standard interrupt handler. Basically an IST is a thread that waits on an event that is bond with an interrupt. In other word, an IST is not a function registered with an interrupt, but a thread that waits on an event that is registered with an interrupt. The consequence is that an IST may actually be separated in many threads or even in many processes.

2.9.1.2 Processor dependent Interrupt management

The low-level interrupt management is split into two parts: one is processor-dependent and board independent and one is processor-independent but board-dependent.

The interrupt management functions dependent of the microprocessor are implemented in:

- %_PLATFORMROOT%\COMMON\SRC\ARM\ATMEL\AT91SAM96x\Kernel\INTR\intr.c.
  This file implements most of the interrupt engine.
- %_PLATFORMROOT%\COMMON\SRC\ARM\ATMEL\AT91SAM96x\Kernel\INTR\map.c.
  This file is an adaptation of map.c given by Microsoft. This file implements the mapping between SYSINTR and IRQ. The difference lies in the number Irq supported. Where Microsoft limits it to 64 Irq, we support any number of Irq. This number of IRQ is different for each processor of the AT91SAM926x family.
- %_PLATFORMROOT%\COMMON\SRC\ARM\ATMEL\AT91SAM96x\Kernel\INTR\pio_intr.c
  This file implements the PIO interrupt engine. It allows handling interrupts on PIOs as if they were standard interrupts.
- %_PLATFORMROOT%\COMMON\SRC\ARM\ATMEL\AT91SAM9263\Kernel\INTR\intr.c.

  This file initializes the variables used by the generic AT91SAM926x interrupt engine, so that it fits the targeted processor: it defines the addresses and the number of pin for each PIO banks and the maximum number of IRQs supported by the IRQ/SYSINTR mapping.

2.9.1.3 Board-dependent Interrupt management

The Board-dependent interrupt management is designed in such a way that it allows handling external interrupt controller seamlessly. It is located in:

%_TARGETPLATROOT%\SRC\KERNEL\OAL\intr.c

The BSP must provide the following functions so that the Kernel can perform BSP-specific management:

Interface	**Description**
BOOL BSPIntrInit();	This function initializes platform specific interrupt mapping and the platform's hardware
BOOL BSPIntrRequestIrqs(DEVICE_LOCATION *pDevLoc, UINT32 *pCount, UINT32 *pIrqs);	This function returns IRQ for the on-board devices based on their physical address.
UINT32 BSPIntrEnableIrq(UINT32 irq);	This function is called from OALIntrEnableIrq to enable interrupt on secondary interrupt controller

UINT32 BSPIntrDisableIrq(UINT32 irq);	This function is called from OALIntrDisableIrq to disable interrupt on secondary interrupt controller
UINT32 BSPIntrDoneIrq(UINT32 irq);	This function is called from OALIntrDoneIrq to finish interrupt on secondary interrupt controller
UINT32 BSPIntrActiveIrq(UINT32 irq);	This function is called from interrupt handler to give BSP chance to translate IRQ in case of secondary interrupt controller.

There is little specific interrupt management required in this BSP. Most of BSP-specific handlers are consequently empty, except for the initialization and the request of the interrupt which take care of the DM9000 IRQ setup.

2.9.1.4 Interrupts HOW-TO

- How to set up a standard interrupt?
  - If you don't have the IRQ number nor the SYSINTR, you have to get the IRQ number from the OAL by calling KernelIoControl (IOCTL_HAL_REQUEST_IRQ…). This will give the IRQ based on the device's location. This has been originally designed for PCI dynamic IRQ assignation, but can be implemented by the OAL for any other kind of bus.
  - If you have the IRQ number but not the SYSINTR, you have to request a SYSINTR for the specified IRQ. This is done by calling KernelIoControl (IOCTL_HAL_REQUEST_SYSINTR…).
  - Then you have to create an event and initialize the interrupt by calling InterruptInitialize. The interrupt is designated by its SYSINTR. The function binds the occurrence of the interrupt with the activation of an event.

- How to set up an interrupt on a PIO?
  - The IRQ number for a PIO IRQ is given by the following formula :

    Irq = LOGINTR_BASE_PIOy + pin number (where y is the name of the bank: A, B, C, etc.)
  - You have to request a SYSINTR for the specified IRQ. This is done by calling KernelIoControl(IOCTL_HAL_REQUEST_SYSINTR,….).
  - Then you have to create an event and initialize the interrupt by calling InterruptInitialize. The interrupt is designated by its SYSINTR. The function binds the occurrence of the interrupt with the activation of an event.
- How to modify the OAL to handle an external interrupt controller?
  - You have to modify the code in %_PLATFORMROOT%\AT91SAM9263EK\SRC\KERNEL\OAL\intr.c. Most of the functions are given an IRQ number as an input parameter and return an IRQ number as a result. What you have to do, if you have an external interrupt controller connected to the IRQ 6 of the AIC for example, is to test in every function if the IN parameter is 6 and in this case perform the necessary treatment in the external controller, and finally return another interrupt number. That's what is done for the PIO interrupts.

2.9.2 PIOs management

The PIO management is not implemented yet. The PIO management is board-specific because a single pin may have many functions, and a function may be output on many pins.

2.9.3 Tick Management and High Resolution Timer

The Periodic Interval Timer (PIT) provides the operating system's scheduler tick and the high resolution timer. It is designed to offer maximum accuracy and efficient management, even for systems with long response time. PIT is compounded of a 20bits counter, and a 12bits adder. PIT counter is incremented at MasterClock/16. When PIT reaches a count limit fixed by OS, an interrupt is fired, Adder is incremented, and Counter is reset to zero.

This BSP implements the variable tick:

During normal computing sessions, the count limit is set to generate a 1ms periodic IT, which provide WinCE scheduler interrupt. CurMSec increments on each IT and count number of milliseconds from boot.

During a Sleep session or a WaitForEvent the processor clock is stopped and CPU is waiting for an IT. WinCE tick interrupt slows down, to reach a maximum period of 300ms. Immediately after an event (IT, end of Sleep session, etc.), system clock is reset to 1ms.

Before CPU-clock goes idle, the sleeping time is computed, the OS can sleep from 1ms to 300ms. If required sleeping time is more than 300ms, scheduler divided sleep period in fragment of 300ms. This computation is done in OEMIdle.

This feature improves power-saving in special under loaded designs.

2.9.4 RTC Management

2.9.4.1 Implementation

The RTC of the system is based on the RTT (Real Time Timer) and the backup registers of the AT91SAM9263 processor. The RTT timer and the backup registers are running on VDDBU (VDD BackUp) and thus are not dependent of VDDCORE or VDDIO. In consequence we can stop the processor and its peripherals, and have the RTT keeping running if the VDDBU is provided.

In the OAL the RTT is running at the slow clock frequency divided by 33. This divider is configurable as a #define in AT91SAM926x\KERNEL\RTC\rtc.c.

The RTT counter is 32-bit wide and is incremented 32768/33 times per second. Consequently it can run 50 days without rolling over. This means that the date will be lost after 50 days of the device being powered-off. The duration can be easily increased by increasing the divider. The drawback would be a small loss in the RTC granularity.

Every time there's a call to a RTC function (OEMGetRealTime, OEMSetRealTime or KernelIoControl(IOCTL_HAL_INIT_RTC,…)), the current time is saved in the back-up registers and the RTT's counter is restarted. This allows us not to care about the possible RTT's counter overflow: we can safely assume that those functions are called at least once every 50 days.

At startup, if there has been a clean boot, the RTC is initialized with the default system time.

Note:

- The RTC Alarm is not implemented
- When using Platform Builder, if you want to keep the RTC, do uncheck "Clear Memory on Soft Reset" in Target->Connectivity Options->Core Service Settings.

- In the earlier version of the AT91SAM9263, the RTT controller doesn't work properly if there's a lot of transaction on the bus. In this case it's preferable to use the RTC implementation based on the System Timer by undefining RTC_IS_BASED_ON_RTT.

2.9.4.2 How to change the current date and time?

The user can get/update the current date and time by calling GetSystemTime/ SetSystemTime.

The parameter of those functions is a basic structure:

```
typedef struct _SYSTEMTIME {
  WORD wYear;
  WORD wMonth;
  WORD wDayOfWeek;
  WORD wDay;
  WORD wHour;
  WORD wMinute;
  WORD wSecond;
  WORD wMilliseconds;
} SYSTEMTIME;
```

In kernel mode, this can be done by calling OEMSetRealTime(…).

2.9.5 Reboot modes

3 Levels of reboot are supported:

- cold reboot : Asserts the external reset line. Resets the processor and the internal peripherals. Forces a clean boot (starting with a clean object-store).
- warm reboot : Resets the processor and the internal peripherals. The object-store is preserved if not located in the Eboot memory area.
- soft reboot : Jumps at the beginning of the Kernel (the StartUp entry point). The object-store is preserved.

Reboots can be triggered by calling KernelIoControl(IOCTL_HAL_REBOOT,…). The default reboot mode is the soft reboot (This may be changed in AT91SAM926x\KERNEL\IOCTL\reboot.c). The other boot mode selected with the second parameter of the function. This parameter is DWORD that may take the following values:

- 1 : Cold reboot
- 2 : Warm Reboot
- 3 : Soft Reboot

Notes:

- If you plan to use the warm reboot, make sure that you won't use the eboot memory space for the object store. This is done by checking "Enable EBOOT space in memory" in the project's settings. Otherwise Eboot may overwrite part of the object store, and corrupt it.
- The reset through KITL/Platform Builder uses the default reboot mode.

2.9.6 Watchdog

2.9.6.1 Description of the Watchdog feature in Windows CE 5.0

There are many levels of watchdog in Windows CE 5.0.

At the applicative level it's possible to create a watchdog that can either terminate the process or reboot the device (the type of reboot is a parameter given when creating the watchdog). This is a pure software watchdog.

At the kernel level there's also a possibility to manage a watchdog. This watchdog must be a hardware watchdog and will fire if the kernel doesn't refresh it.

2.9.6.2 Implementation

The BSP supports both types of watchdog.

Only the kernel watchdog is part of the OAL, the other is provided by Microsoft and relies on IOCTL_HAL_REBOOT.

The Kernel knows that there is a watchdog when dwOEMWatchDogPeriod is set to a non-null value. In this case the Kernel will call the function pointed by pfnOEMRefreshWatchDog every dwOEMWatchDogPeriod ms.

In this BSP, pfnOEMRefreshWatchDog points toward OEMRefreshWatchDog(). This function only refreshes the hardware watchdog. The watchdog is set-up by OEMInitWatchDogTimer() that is called in OEMInit.

The code is located in %WINCEROOT%\PLATFORM\COMMON\SRC\ARM\ATMEL\AT91SAM926x\KERNEL\WATCHDOG.

2.9.7 Time Bomb

This BSP is provided freely in a trial version. This version is binary only and is limited in time. After 60 minutes, it will reboot. This mechanism is called the time bomb. To enable this time bomb, it's necessary to set OAL_TIMEBOMB in the environment variables of the OSDesign and recompile both the part located in COMMON\SRC\ARM\ATMEL and the part located in _TARGETPLATROOT.

2.9.8 Suspend

Suspend mode is supported. It consists in stopping the PCK (processor clock). The system is woken up every time an interrupt occurs. The interrupts that may wake the system up are configured by callings the following Kernel IOCTLs:

- IOCTL_HAL_ENABLE_WAKE
- IOCTL_HAL_DISABLE_WAKE

When the system enters (respectively leaves) the suspend mode, the PowerDown (respectively PowerUp) of every mounted driver is called. This allows every driver to put its peripheral in a low power state. However PowerUp and PowerDown are not easy to implemented because they're called in a peculiar context and thus have some limitations (example: they may not perform any blocking system call). It's now preferred to use the Power Manager Component for putting the driver in a low power state.

2.9.9 Shutdown

The shutdown is implemented as a KernelIoControl. It makes sue of the Shutdown controller of the AT91SAM9263 Processor. Before shutting the system down, it refreshes the RTC.

The wake-up is available through the Wakeup pin only. The de-bouncing time is given by a #define, its default value is 243 ms (maximum value).

2.9.10 Debug Serial

The debug serial is the serial port that's used for outputting the debug information. It's also used by the boot loader as an input. The debug serial uses the DBGU port of the processor. Only two lines are used: Tx and Rx. Its configuration is fixed: 115200 8N1.

2.9.11 PLL management and core Frequency setting

There are four clock sources, MainClock, SlowClock @ 32.768 Hz, and two PLLs that takes either main or slow clock as an input.

The processor needs several clocks:

- Processor Clock (PCK). It can be generated from any clock sources.
- BusClock (or Master Clock) which is Processor Clock divided by a power of 2.
- USB clock which must be set at 48.000.000 Hz (For example MainClock / 240 * 625) and can only be generated by PLLB
- PixelClock witch is deduced from BusClock.

The PLL management implementation provides kernel-level routines:

- AT91SAM926x_SetPLLxFreq. You should pass a multiplier and a divider. The source is always MainClock. For PLLB, there is one more parameter to set USB divider, so you can divide PLLB Frequency before passing it to the USB controller.
- AT91SAM926x_SetProcessorAndMasterClocks. You should pass a target processor clock and a bus ratio. This function modifies the output frequency of PLL A with a tolerance set with a #define.
- A set of getter: AT91SAM926x_GetMasterClock (which return BusClock), GetPLLxFreq.
- AT91SAM926x_TurnProcessorClockOff. Useful for stopping processor clock during idle time.

And an IOCTL for user-level programming:

- IOCTL_HAL_MASTERCLOCK return the value of BusClock, like AT91SAM926x_GetMasterClock function.

As long as you are in kernel mode, you can change any of those frequencies, but if you change a BusClock you must reconfigure every device that relies on Master Clock (almost all of them do rely on master clock)

The core Frequency setting is given in the BSP_ARGS structure that is filled by the bootloader. If the settings are not valid, the kernel will start with default values (200MHz core, 50MHz bus).

2.9.12 KITL (Kernel Independent Transport Layer)

2.9.12.1 Description

The Kernel Independent Transport Layer provides a hardware-independent mechanism for communication between the development workstation and a target. This layer enables a debug communications channel between the target and the platform builder. KITL relies on a ETHDBG library just as the bootloader does.

Note: Windows CE provides a Virtual Ethernet Adapter, called VMINI, that relies on KITL. It behaves like a real Ethernet card and allows the use of network while the BSP is still in development.

2.9.12.2 Implementation

Depending of the build options, KITL may be excluded from the image. Four Kernel executables are built when building the BSP. Two of them embeds a real KITL implementation, others embeds only stubs.

Data must be passed from the bootloader to the kernel in order to keep the debug link initiated by the bootloader. This information is located in the `OAL_KITL_ARGS` structure :

-KITLArgs.flags = (OAL_KITL_FLAGS_ENABLED | OAL_KITL_FLAGS_DHCP | OAL_KITL_FLAGS_VMINI); // Enable KITL, DHCP and VMINI

- KITLArgs.devLoc.IfcType = Internal;
- KITLArgs.devLoc.BusNumber = 0;
- KITLArgs.devLoc.PhysicalLoc = (PVOID)(AT91C_BASE_MACB);
- KITLArgs.devLoc.LogicalLoc = (DWORD)KITLArgs.devLoc.PhysicalLoc;

Must of the KITL implementation is delivered by Microsoft. The OEM provides the EthDbg library that KITL relies on.

2.9.13 The EthDbg Library

This library provides a set of functions that are used by the bootloader and/or the KITL to interface handle the Ethernet controller.

2.9.13.1 Driver Interface

The following functions are implemented in the EMACB EthDbg driver; they respect the standard Windows CE API.

Initialization

*BOOL EMACInit(UINT8 *pAddress, UINT32 offset, UINT16 mac[3])*

Purpose: This function initializes and configures the EMAC and external PHY and check if the Ethernet link is valid.

Parameters:

pAddress [in]:base address of the EMAC Ethernet controller, if the MMU is activated in the bootloader, be sure the address passed in parameters is virtual, else the physical address must be used.

Mac[3] [out]: at the exit of this function, this three bytes will contain the MAC of the controller read in its registers.

Sending frames

*UINT16 EMACSendFrame(UINT8 *pBuffer, UINT32 length)*

Purpose: This function sends data over the debug Ethernet adapter.

The Ethernet frames are stored in TX SRAM, then the length of the frame is set in the appropriate register and the TXREQ (Transmission request) bit is setting.

Parameters:

pData [in]: Frame data buffer to send. The buffer must be DWORD aligned.

dwLength [in]: Number of bytes in *pData*.

Return Values:

If this function succeeds, it returns TRUE.

If this function fails, it returns FALSE.

Receiving frames

*UINT16 EMACGetFrame(UINT8 *pBuffer, UINT16 *pLength)*

Purpose: This function receives data from the debug Ethernet adapter.

The received data is stored in RX SRAM, if the packet status and length is valid; the data of the packet is received and stored in the buffer.

Parameters:

pData [out]: Buffer to hold received frame bytes. The buffer must be DWORD aligned.

pwLength [in]: Number of bytes in the receiving buffer when the function is entered.

Return Values

If the frame is present, the function returns TRUE and fills the *pData* and *pwLength* buffers.

If no frame is present, the function returns FALSE and *pwLength contains the value zero*.

Multicast management

void EMACCurrentPacketFilter(UINT32 filter)

Nothing to do in this function.

*BOOL EMACMulticastList(UINT8 *pAddresses, UINT32 count)*

Nothing to do in this function.

MAC address settings

Since the MAC address is not stored in an internal EEPROM, you must call this function before the DM9161 initialization function to store a valid MAC address.

*void EMACSetAddressMac(UINT8 *pAddress, UINT16 mac[3])*

Purpose: This function sets the MAC address “mac” in the EMAC registers.

Parameters:

pAddress [in]: base address of the EMAC Ethernet controller, if the MMU is activated in the bootloader, be sure the address passed in parameters is virtual, else the physical address must be used..

Mac[3] [out]: at the exit of this function, this three bytes will contain the MAC of the controller read in its registers.

All these functions must be respectively associated to the following functions pointers:

pfnEDbgEnableInts = EMACEnableInts;

pfnEDbgDisableInts = EMACDisableInts;

pfnEDbgGetFrame = EMACGetFrame;

pfnEDbgSendFrame = EMACSendFrame;

pfnCurrentPacketFilter= EMACCurrentPacketFilter;

pfnMulticastList = EMACMulticastList;

Then, these functions pointers will be used by the system in the following functions:

- OEMEthGetFrame
- OEMEthSendFrame

2.9.13.2 Limitations

The multicast is not implemented.

2.10 Bootloader

2.10.1 Purpose

This component allows booting a Windows CE image.

The bootloader performs the following treatments:

- Board minimal initialization to allow the use of the devices needed for the bootloader,
- Serial debug port (DBGU) initialization and management to allow:
  - Debug traces,
  - Bootloader configuration.
- Ethernet management to allow Windows CE image downloading and advanced debugging initialization,
- Flash management to allow the storage a Windows CE image,
- Startup of a Windows CE image downloaded from Flash or from Ethernet.

2.10.2 Description of the startup of the system

The AT91SAM9263 can boot either from its internal boot ROM or from an external linear flash.

The AT91SAM9263EK is designed so that the processor boots from its internal boot ROM.

The boot ROM's software looks for an executable portion of code store at the beginning of the Serial Data Flash (located on the SPI bus). If it finds it, it loads it into the internal SRAM and then jumps to it.

As the Bootloader may be too big to be located entirely in the SRAM, it is split into two parts. One small part will be loaded and executed in the SRAM, and the other part (the bigger one) will be loaded into SDRAM by the first part. From now on, we'll call the first small part the *FirstBoot* (or first level boot-loader) and the second part will be called *EBOOT* (or second-level bootloader).

The FirstBoot performs the basic initialization: Debug port, SDRAM and the Storage Device (the Serial Dataflash in this case), then copy EBOOT from its storage location (Serial Data Flash) to the SDRAM (at address 0x20000000) and finally jump to 0x20000000.

2.10.3 Update of the bootloader

The FirstBoot is stored into the SPI dataflash. The update of the FirstBoot is done through SAMBA by using the appropriated script:

- "Init & Select AT45 Dataflash on CS0"
- "Send BOOT File (Only for Chip Select 0)". The file to be sent is Firstboot.nb0
- "Send File" with the correct eboot.nb0 file and the address 0x5000

2.10.4 The Bootloader Menu

2.10.4.1 FirstBoot's Menu

> *RomBOOT*
>
> *>*
>
> *INFO : Low Level Init : OK*
>
> *Init Data flash*
>
> *Starting eboot ...*

kMaster Clock is 49921225 Hz

Debug serial initialized .....OK

2.10.4.2 Eboot's Menu

Microsoft Windows CE Ethernet Bootloader Common Library Version 1.1 Built Jul 4 2007 18:17:29

Microsoft Windows CE 5.0 Ethernet Bootloader for the AT91SAM926xEK board

Adaptation performed by ADENEO (c) 2007

Master Clock is 49921225 Hz

Master Clock is 49921225 Hz

WARNING : LoadEBootCFG: No valid Eboot configuration found.

INFO : Loading default bootloader settings

*Press [ENTER] to download now or **[SPACE]** to cancel.*

Initiating image download in 5 seconds

Ethernet Boot Loader Configuration :

0) Mac address ..... (00:12:72:72:20:20)

1) Ip address ..... (192.168.111.115)

2) Subnet Mask address .. (255.255.255.0)

3) DHCP ..... (Enabled)

4) Boot delay (seconds).. (5)

5) Frequency settings ... (core at 200, bus divider 4)

6) Download image to SDRAM

7) Download new image at startup

l) Launch flash resident image now

d) Download from ethernet now

s) Save configuration now

r) Restore default configuration and save now

n) Image flash menu

>System ready!

This menu allows you:

- to choose the network settings
- to select whether the image will be downloaded from the Ethernet or launched from a storage device (Flash)
- to select the frequency settings used by the image

2.11 How to store an image in flash

The boot loader permits to select whether the image will be stored in RAM or in Flash.

In the eboot menu, select option 6) to see “Download image to Flash”. So now, when you download your image, it will be written in NandFlash (it is not possible to change the flash destination used to store the image).

Now you can launch your image, using eboot menu option L) or set the option 7) with “Launch existing flash resident image at startup”.

3. BSP's drivers

3.1 The USB Host driver

3.1.1 General description of a Windows CE USB host driver

USB host controllers mostly conform to one of the 2 standard specifications: OHCI and UHCI. WinCE provides support for this kind of controllers. All the processing is done by the MDD. The PDD have little to initialize since the OHCI and UHCI specs define the register's list (and behavior and mapping as well).

The driver works with USB keyboards and mousse and Mass Storage Devices, but some AT91SAM9263 revisions don't support full-speed devices and Mass Storage Devices.

3.1.2 Code Location

The driver is composed of two parts. One part is processor specific and is located in COMMON\SRC\ARM\ATMEL\COMMON\DRIVERS\USBHCD. The other one is located in %_TARGETPLATROOT%\SRC\DRIVERS\USBHCD.

3.1.3 Description of the implementation

The AT91SAM9263 embeds an OHCI controller.

The PDD only turns on the OHCI controller and initializes some variables (address of the controller, sysintr number and some objects specific to the OHCI MDD). The driver assumes that the 48 MHz clock is initialized by the OAL.

3.1.4 Registry Settings

```
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\OHCI]
   "Prefix"="HCD"
   "Dll"="AT91SAM9263_ohci.dll"
   "Index"=dword:1
   "Order"=dword:1
   "TotalAvailablePhysMem"=dword:10000   ;64K
   "HighPrioPhysMem"=dword:4000          ;16K
```

3.2 The USB device driver

This driver is also known as a USB Function driver. Its primary purpose is providing connectivity between a Windows CE device and a desktop PC via USB.

The idea is that a Windows CE device that contains suitable USB Function controller hardware will be used as a serial (COM:) port, a mass storage device or a network device (RNDIS) to connect to a desktop PC. Essentially, there are 2 drivers that will be used in this design:

a) **USB Function driver** that runs on the Windows CE platform and exposes the device as the wanted device to the desktop,

b) **Host-side USB driver** that runs on the desktop PC and talks to the Windows CE via USB transfers. The host driver also exposes a serial interface to the application layer so that a client application (such as ActiveSync) running on the desktop can use it for any serial applications.

3.2.1 General description of a Windows CE USB device driver

Interface

The USB Function driver is very similar to the sample USB Function driver in Windows CE, hence it uses the same MDD and thus exposes the same DDSI as a standard USB Function driver.

USB specific requirements for USB Function Controller

Communication over the USB is done by sending packet of data on the same physical line.

However USB (as a protocol) provides multiple logical lines. Those lines are called endpoints. For more information about USB, please have a look at http://www.beyondlogic.org/usbnutshell/usb1.htm

There are different ways for configuring those endpoints.

The sample USB Function driver supports two settings. We must expose one of the following modes in the USB function controller that is embedded in their Windows CE device.

1. This is the first setting, which is the minimum requirement - 1 control, 2 bulk endpoints. In this setting/configuration, 1 bulk endpoint is used for READ and the other bulk endpoint is used for WRITE. We send data back and forth without using a USB interrupt endpoint. For example:

   - Endpoint 0: Control endpoint (in, out)
   - Endpoint 1, IN: Bulk endpoint
   - Endpoint 2, OUT: Bulk endpoint

The above setting is a very simple interface and is provided for those who can't provide an interrupt endpoint on their USB function chip. Note that the above mode is potentially intrusive on the host system bus, depending on bus controller implementation. Degradation of host performance may occur while the Windows CE device is attached to the bus, regardless of ActiveSync activity to the device.

2. This one is the one to expose in the USB function hardware - 1 control, 2 bulk, and 1 interrupt endpoint. This is similar to the first configuration, but the function side can use USB interrupts to notify the host that there is data available for read (serial control lines emulation). (Note that the host-side USB driver is written to support both interrupted and non-interrupt devices). The tradeoff with this configuration is that less bus activity is generated on the host, but since read polling is replaced by slower interrupt endpoint polling, small delays will be introduced on data transmitted to host from the device. For example:

   - Endpoint 0: Control endpoint (in, out)
   - Endpoint 1, IN: Bulk endpoint
   - Endpoint 2, OUT: Bulk endpoint
   - Endpoint 3, IN: Interrupt endpoint (Status)

3.2.2 Code Location

%_TARGETPLATROOT%\SRC\DRIVERS\USBFN\AT91SAM9263EK_usbfn.cpp : contains the AT91SAM9263EK card specific code.

%_TARGETPLATROOT%\SRC\DRIVERS\USBFN\AT91SAM9263EK_usbfn.reg : contains the registry keys needed for the USB Function driver to work.

%_TARGETPLATROOT%\SRC\DRIVERS\USBFN\AT91SAM9263EK_usbfn.def

%_PLATFORMROOT%\COMMON\SRC\ARM\ATMEL\AT91SAM926x\DRIVERS\USBFN: contains the code specific to the AT91SAM926x family.

3.2.3 Description of the implementation

The library ufnmdd.lib (that is made from the code located in public directory) is linked to the AT91SAM9263EK_usbfn.dll. It contains the code for the mmd interface. This library is generic and manages the software part of the device. Then, it calls functions in AT91SAM9263EK_usbfn.cpp and

AT91SAM926x_usbfn.cpp. These functions are platform-specific and manage the hardware part of the device.

3.2.4 Registry Settings

[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\USBFN]

"Prefix"="UFN"

"Dll"="AT91SAM9263EK_usbfn.dll"

"Index"=dword:1

"Order"=dword:1

"InterfaceType"=dword:0 ; Internal

"Priority256"=dword:64

"BusIoctl"=dword:2a0048 ; to call post_init

"IClass"=multi_sz:"{E2BDC372-598F-4619-BC50-54B3F7848D35}=%b","{6F40791D-300E-44E4-BC38-E0E63CA8375C}"

3.2.5 USB Function clients :

A Windows CE device can be used as a serial (COM:) port, a mass storage device or a network device (RNDIS) to connect to a desktop PC. There is one client for each functionality of the USB Device controller. Most of these clients are generic, and located in %WINCEROOT%\PUBLIC\COMMON\OAK\DRIVERS\USBFN\CLASS.

Two clients are specific to the BSP :

- USBFNSERIAL is a client which create a serial COM port,
- USBSerial_ActiveSync, which is an uncurbed client using ActiveSync connection.

The client must be selected in the Windows CE catalog, and then, in the registry file AT91SAM9263EK_usbfn.reg. The following registry key allow to selected the default client, xxxxxx is the registry name of the client.

[HKEY_LOCAL_MACHINE\Drivers\USB\FunctionDrivers]

3.3 The Display driver

3.3.1 General description of a Windows CE display driver

This paragraph describes the architecture, the way of working and the required entry points of a standard Windows CE display driver.

Windows CE display driver is built on a layered architecture. Applications call *core.dll*, this one calls *GWES* for displaying demands.

The display driver is loaded by the *Graphics, Windowing and Events Subsystem* (*GWES*) at Windows CE boot and each time it receives information to display. *GWES* calls *DDI.dll* that is the default DLL for the display driver. This DLL exports a single function called *DrvEnableDriver* that returns to the caller a pointer to an array of 27 function pointers. Those functions are called as soon as GWES needs to display something on the device.

To make a DDI driver development easy, Windows CE provides a class called GPE (the Graphics Primitive Engine class) that contains most of the base code required for any display driver. Consequently, we just need to write a class that extends this GPE class to implement initializations and other specific behaviors for the driver (accelerated video, line drawing…). As it is a pure virtual class, a set of 11

functions must be implemented, others have default behavior (implement them to include DirectDraw support for example). Functions to be implemented are:

- *NumModes:* Returns the number of display modes supported by the display driver.
- *GetModeInfo:* Returns information about a specific display mode, such as display width and height in pixels and number of bits per pixel. This function should handle the number of display modes returned by the *NumModes()* function. The first mode entry in the list of supported modes is always the one selected for configuration when *SetMode()* is later called.
- *SetMode:* Sets the display mode. This can be the most time consuming function to write in the entire driver, especially for VGA controllers, since configuration of the display device can be a fairly involved process.
- *AllocSurface:* Allocates a surface, which is just a block of system or video memory to store pixel data. The GPESurf class can be used to represent surfaces in system memory. To represent surfaces in video memory, the GPESurf class must be derived from.
- *SetPointerShape:* Sets the cursor bitmap and cursor hotspot.
- *MovePointer*: Moves the cursor.
- *BltPrepare:* Called before a BLIT operation is performed. It allows the driver to setup the BLIT hardware for performing the operation, if it is supported. It must return the actual function to be called to perform the BLIT operation, which can be the default BLIT function provided in the GPE class.
- *BltComplete:* Called after a BLIT operation has been performed. It allows the driver to do any cleanup required after the BLIT operation, if necessary.
- *Line:* Called before and after a line drawing operation. When it is called before the line drawing is done, the function can setup the line drawing hardware for performing the operation, if it is supported. It must return the actual function to be called to perform the line drawing operation, which can be the default line drawing function in the GPE class. When it is called after the line drawing is done, the function can do any cleanup required after the line drawing operation, if necessary.
- *SetPalette:* Sets the palette. This only applies for modes that support a palette, which is typically 8 bits per pixel or less.
- *InVBlank:* Indicates if the display update is in the vertical blanking period. This is useful for reducing an animation problem known as tearing, where the display memory update is not in sync with the display refresh on the monitor.

Another important class is GPESurf, which is used to represent surfaces located in system memory. If the driver supports the creation of surfaces in video memory, then the GPESurf class will need to be derived. However, these changes are fairly minor.

3.3.2 Implementation

- General structure of the driver

  It uses GPE to provide a fast and reliable interface with HW, implements DirectDraw and provide a D3DMobile support. The driver is designed to use a rectangle video memory space, placed either in general RAM or in specific VideoRAM. The driver permits to select which operation you want to do with HW, and which you want to do in SW. All HW calls are asynchronous and push to a FIFO. The driver supports all display modes supported by the GPE.

- Video Memory

Due to technical limitation, video memory must be viewed as one rectangle of pixel having one pixel format (ex: 16bpp). You cannot put in video memory a B/W 1bpp mask, or any other type of surface.

AT91SAM9263EK LCDC cannot display surfaces not contiguous (last pixel of a line is followed in memory by the first pixel of next line) so this constrains video memory width to the screen width (ex: 240pxl @ 16bpp = 480 Bytes). This means you cannot put in video memory surfaces wider than 240 pxl.

3.3.3 Potential evolutions

- Video memory cannot extend to all the system memory, so a system to virtualize VidMem could improve performance on application that create many surfaces. A surface placed in VidMem should be backed-up in SysMem and reloaded on demand into VidMem.
- swFifo & hwFifo should be compared to a pipeline, and a SW call breaks the pipeline. The driver must flush it before rendering SW command. To avoid pipeline flush, we should maintain a list of concerned surfaces, the ones that will probably be modified by the HW pipeline. And if a SW command doesn't concern any of the "dirty" surfaces, it could be done without flushing the pipeline. In the same way, we could define "dirty" rectangle inside a surface (ex : Don't redraw the cursor if the frame buffer isn't modified in the place the cursor is.). This trick is very powerful in some cases, FrameBuffer dirty rectangle should be the first test, it will decrease the resource needed for cursor management.

3.3.4 Registry parameters

Display driver configuration is made via registry, in the key:

```
"Width"=dword:F0
"Height"=dword:140
"Bpp"=dword:10
;       "forceRGB"=dword:1
;       "Cached"=dword:1
"VRAMWidthInPixel"=dword:F0
"VRAMHeightInPixel"=dword:156
; When the frame buffer of the screen is  located in SRAM, PCK can't be shut down in idle mode
; because it causes the screen to oscillate. If you locate the frame buffer in SDRAM
; in the display file, you can enable PCK shut down in common\arm\atmel\at91sam926x\kernel\power\
waitforinterrupt.s

"VRAMaddress"=dword:23f00000        ; allocate the video memory in SDRAM
;       "VRAMaddress"=dword:300000        ; allocate the video memory in SRAM
"UpperMargin"=dword:21
"LowerMargin"=dword:13
"LeftMargin"=dword:44
"RightMargin"=dword:0A
"Vsync"=dword:1
"Hsync"=dword:1
"PixelClock"=dword:5B8D80 ; 6000000 Hz
```

The settings allow to set the width of the screen, the height of the screen and the pixel depth.

If you don't specify any value or if it doesn't match a supported mode, the driver takes default mode, currently set to 320x240x16.

The pixel depths available are 8bpp, 16bpp and 24bpp.

A key concerning the screen rotation is added here :

[HKEY_LOCAL_MACHINE\System\GDI\ROTATION]

"Angle"=dword:00

3.4 The ADS7843 Touch Screen driver

3.4.1 General description of a Windows CE touch screen driver

The touchpad driver is a layered native driver:

- Layered means that it is composed of two layers:
  - The MDD layer (upper one) which implements the interface with the system and is independent from the platform. It is the functional part of the driver.
  - The PDD layer (lower one) which implements the interface with the hardware and is dependant from the platform.
- Native means that this driver has a specific interface dedicated to touch screen drivers.

The MDD layer is the standard touch screen MDD layer. It is located in %_WINCEROOT%\PUBLIC\COMMON\OAK\DRIVERS\TOUCH directory. There is no need to perform any modification on it.

Only the PDD layer has to be modified. The interface between MDD and PDD is standardized and consists in 8 functions. It is known as the DDSI touch interface:

- *DdsiTouchPanelGetDeviceCaps*: this function is called by the MDD layer to obtain capabilities of the touch screen device, in terms of sampling rate, number of calibration points required and coordinates of each calibration point when performing calibration.
- *DdsiTouchPanelSetMode:* this function allows the MDD to modify the sampling rate between two states (low or high), if the touch screen device can manage them.
- *DdsiTouchPanelEnable*: this function is used to activate the touch screen. It should apply power to it.
- *DdsiTouchPanelDisable:* this function is used to deactivate the touch screen. It should remove power from it.
- *DdsiTouchPanelAttach*: this function is called when the MDD layer is started. It allows the PDD to perform internal initialization independent from applying power to the touch screen.
- *DdsiTouchPanelDetach:* this function is called when the MDD layer is stopped. It allows the PDD to perform internal deinitialization independent from removing power from the touch screen.
- *DdsiTouchPanelGetPoint:* this function is called by the MDD to retrieve the last point acquired by the touch screen.
- *DdsiTouchPanelPowerHandler*: this function is called to indicate the driver that the system is entering or leaving suspend mode. If the device as power management possibilities, then the PDD should use it.

The touchpad driver is launched by GWES process which manages all HMI drivers. It is build as a DLL that is known by GWES thanks to the following registry keys located in [HKEY_LOCAL_MACHINE\HARDWARE\DEVICEMAP\TOUCH]:

- DriverName: specifies the DLL name that implements the driver,
- CalibrationData: specifies calibration points coordinates for the device.

3.4.2 General description of the driver

Source Files list and purpose for each

- ADS7843Touch.cpp: the PDD functions implementation
- ADS7843Touch.h: the definition of PDD functions
- ADS7843SPI.cpp: the management of the accesses to the SPI bus driver
- ADS7843SPI.h : the definition of SPI accesses
- ADS7843Regs.h: ADS7843 registers values definitions
- TouchScreen.reg: registry entries needed to load the driver at startup

The ADS7843 driver uses the SPI driver for the communication, and uses specific IOControl to perform transaction between the chip and the driver.

A timer forces input detection to perform the sampling for the detection of stylus movement on the touchscreen.

3.4.3 Calibration application

At startup of the OS, a calibration application is launched to allow users to calibrate the touchscreen if no information about this calibration is found in the registry.

If Hive based registry is used for the device, the calibration is perform only one time at the first device boot.

3.4.4 Registry Settings

```
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\Touch]
    "Priority256"=dword:64 ;hex value
    "Order" = dword:1
    "HighPriority256"=dword:64 ;hex value

[HKEY_LOCAL_MACHINE\HARDWARE\DEVICEMAP\TOUCH]
    "DriverName"="AT91SAM9263EK_touchscreen.dll"
    "MaxCalError"=dword:10

[HKEY_LOCAL_MACHINE\init]
  "Launch70"="touchcalibrate1.exe"
  "Depend70"=hex:14,00, 1e,00
```

3.5 The Nand Flash driver

3.5.1 Description

The NAND flash driver is a block driver that interfaces with a NAND flash. It is separated in two layers:

- the FAL (Flash Abstraction Layer): this layer is oriented towards the system. It exposes the Block driver interface. It relies on the FMD to access the hardware. The FAL also manages the wear leveling. This layer is provided by Microsoft.

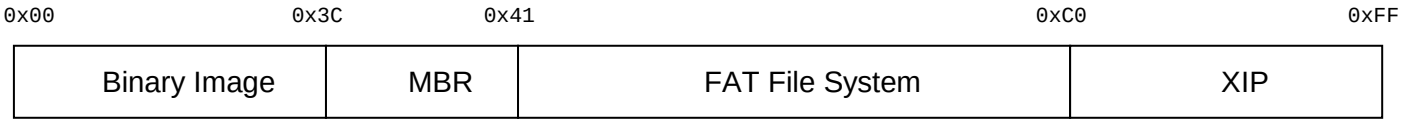
- The FMD: this layer handles the hardware transactions. It is provided by the OEM. It basically consists in a set of functions to read, erase or write the sectors. On the AT91SAM9263EK board, the FMD has been enhanced to add another level of translation and protection. This allows us among other things to set the MBR at any physical address instead of sector #0. This FMD is called the Generic FMD.

3.5.2 Generic FMD library

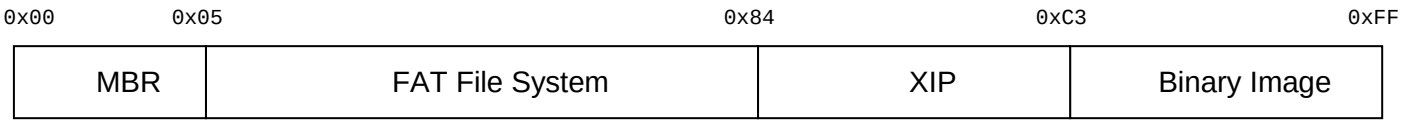
This library Driver is divided into two parts, a generic top-layer for sector mapping and write protection, and a device-specific bottom-layer. There are two features in the generic part:

- BTL, Block Translation Layer provides a block translation map (BTM) which remaps logical block addresses (LBA) used by the OS into physical block addresses (PBA) used by the Low Level Driver. This enables all flash accesses either in kernel-mode or in user-mode to use the same driver. Assuming an application knows the BTM, it could easily rewrite a Binary Image placed at PBA 0x00 by writing at LBA 0xC3.

Physical Device



Logical Device after mapping



- WPS, Write Protection System allows or denies access to each block separately, depending on driver parameters. The Write Protection Table could be generated according to BTM (i.e.: you can protect last logical blocks against write or erase) or you can save the table onto a persistent medium. If you need to change implementation, please refer to `BOOL LoadBlocksInfo()` function, in `GenericFMD.c`. For example you can prevent binary image from being erased by setting blocks from 0xC3 to 0xFF to protected mode, so that an application can't erase binary image.

The device-specific bottom-layer is called the LLD (Low Level Driver). It must be designed for each couple device/board, but most of the time it is possible to use already implemented generic functions. It deals with all chip accesses. To setup specific functions, define an Init function with Nand IDs.

```
typedef struct _LLD_Interface
{
      BOOL (*EraseBlock)(NandChip *, DWORD);
      BOOL (*ReadSector)(NandChip *, DWORD, DWORD, PBYTE);
      BOOL (*WriteSector)(NandChip *, DWORD, DWORD, PBYTE);
} LLD_Interface;
typedef struct _NF_Interface {
      void (*Enable)(NandChip *pChip);
      void (*Disable)(NandChip *pChip);
      void (*WriteBuf)(NandChip *pChip, BYTE *pInBuffer, DWORD len);
      void (*ReadBuf)(NandChip *pChip, BYTE *pOutBuffer, DWORD len);
      WORD (*ReadWord)(NandChip *pChip);
```

```
      BYTE (*ReadByte)(NandChip *pChip);
      void (*WriteAddr)(NandChip *pChip, UCHAR addr);
      void (*WriteCmd)(NandChip *pChip, UCHAR cmd);
      BOOL (*IsReady)(NandChip *pChip);
} NF_Interface;
```

Notes :

- All addresses are physical.
- The fZone parameter indicates whether the function reads (or writes) the sector, the spare part or the sector and the spare part.

3.5.3 Registry settings

[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\NandFlash]

"Profile"="NandFlash"

"IClass"=multi_sz:"{A4E7EDDA-E575-4252-9D6B-4195D48BB865}"

"Order"=dword:0

"FriendlyName"="NAND Flash Driver"

"Dll"="NANDFLASH.dll"

"Prefix"="DSK"

[HKEY_LOCAL_MACHINE\System\StorageManager\Profiles\NandFlash]

"AutoMount"=dword:1

"AutoPart"=dword:0

"AutoFormat"=dword:0

"PartitionDriver"="mspart.dll"

"Name"="NANDFLASH"

"Folder"="NandFlash"

"DefaultFileSystem"="FATFS"

3.6 The PWMC driver (PWMC)

3.6.1 General description of a Windows CE PWMC driver

The PWMC driver is a stream driver, meaning that this driver is accessed through a Win32 file IO interface.

This driver exports the standard stream driver functions:

PWC_Init, PWC_Deinit

PWC_Open, PWC_Close

PWC_IOControl

PWC_Read, PWC_Write, PWC_Seek *as dummy functions*

PWC_PowerDown, PWC_PowerUp *as dummy functions*

3.6.2 Code Location

As the PIO configuration is board specific, a part of the driver is located in “PLATFORM\AT91SAM9263EK\SRC\DRIVERS\Pwmc”. The other part is common to the processor, so it is located in “PLATFORM\COMMON\SRC\ARM\ATMEL\AT91SAM9263\DRIVERS\Pwmc”.

3.6.3 Description of the implementation

This driver implements PWMC. It uses a new hardware block, the PWMC. *PWMC is configured in WaveForm mode. All the 4 channels are supported.* Each PWMC channel is viewed as a device : Channel 0 is accessed through “PWC0:”, channel 1 through “PWM1:”, etc.

- PWC_Init

This function is called by the Windows CE Device Manager. Memory is allocated to create the device context : T_PWMINIT_STRUCTURE. The device context is initialized with the virtual addresses of the physical registers. Furthermore, the PWM controller is configured *in waveform mode* which allows to generate PWM signal.

It returns a pointer to the device context.

- PWC_Deinit

This function frees memory allocated in the function PWC_Init.

- PWC_Open

This function is called when an application makes a “CreateFile”. Memory is allocated to create an open context : T_PWMOPEN_STRUCTURE. The open context and the PIOs (specific to the board) are initialized.

It returns a pointer to the open context.

- PWC_Close

This function is called when an application makes a “CloseHandle” on the open context. Memory allocated for the open context is freed.

PWC_IOControl

This function implements 3 IOControls :

IOCTL_PWC_CONFIG : This IOControl is called to configure the PWM signal. The parameters are the frequency in Hz and the duty cycle in %. If this function is never called, the IOControl *“IOCTL_PWC_START”* configure the PWM with default parameters from registry.

IOCTL_PWC_START : This IOControl starts the PWM output. If the PWM signal is not configured (*“IOCTL_PWC_CONFIG”* not called), the default parameters stored in the registry are used to do the configuration.

IOCTL_PWC_STOP : This IOControl stops the PWM output.

3.6.4 Registry Settings

```
;=============== PWM Controller Channel 0 (PWM0 pin) ===============
;===============       Used by Beeper       ===============
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\Pwm0]
    "Index"=dword:0
    "Dll"="AT91SAM9263EK_Pwmc.dll"
```

```
    "Prefix"="PWC"
    "Order"=dword:1
    "DefaultFrequency"=dword:3E8  ;1000 Hz  Beeper tone
    "DefaultDutyCycle"=dword:32   ;50%      not to be changed

;=============== PWM Controller Channel 1 (PWM1 pin) ===============
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\Pwm1]
    "Index"=dword:1
    "Dll"="AT91SAM9263EK_Pwmc.dll"
    "Prefix"="PWC"
    "Order"=dword:1
    "DefaultFrequency"=dword:3E8  ;1000 Hz  can be changed
    "DefaultDutyCycle"=dword:32   ;50%            can be changed

;=============== PWM Controller Channel 2 (PWM2 pin) ===============
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\Pwm2]
    "Index"=dword:2
    "Dll"="AT91SAM9263EK_Pwmc.dll"
    "Prefix"="PWC"
    "Order"=dword:1
    "DefaultFrequency"=dword:3E8  ;1000 Hz  can be changed
    "DefaultDutyCycle"=dword:32   ;50%            can be changed

;=============== PWM Controller Channel 3 (PWM3 pin) ===============
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\Pwm3]
    "Index"=dword:3
    "Dll"="AT91SAM9263EK_Pwmc.dll"
    "Prefix"="PWC"
    "Order"=dword:1
    "DefaultFrequency"=dword:3E8  ;1000 Hz  can be changed
```

3.7 The I2C (TWI) driver

3.7.1 General description of a Windows CE I2C (TWI) driver

The I2C driver is a stream driver, meaning that this driver is accessed through a Win32 file IO interface.

3.7.2 Code Location

The driver is divided into three parts :

- %_TARGETPLATROOT%\SRC\DRIVERS\I2C : this directory contains the code specific to the AT91SAM9263EK board,
- %_PLATFORMROOT%\COMMON\SRC\ARM\ATMEL\AT91SAM9263\DRIVERS\I2C : this directory contains the code specific to the AT91SAM9263 family.

- %_PLATFORMROOT%\COMMON\SRC\ARM\ATMEL\AT91SAM926x\DRIVERS\I2C : this directory contains the code specific to the AT91SAM926x family.

3.7.3 Description of the implementation

This driver implements I2C (TWI) peripheral of the AT91SAM9263. The I2C is viewed as a device and accessed through “I2C1:”

Read, write access and set the transfer rate actions are done by calling the DeviceIoControl function. So I2C_IOControl function implements 3 IOControls :

IOCTL_I2C_READ : this IOControl is called to read data from an I2C device in the specified internal device address. Use the T_I2CIOCTL_READ struct when calling this IOControl.

IOCTL_I2C_WRITE : this IOControl starts is called to write a buffer of data to an I2C device at the specified internal device address. Use the T_I2CIOCTL_WRITE struct when calling this IOControl.

IOCTL_I2C_TRANSFERT_RATE : this IOControl sets the I2C bus transfer rate. If this function is never called, the driver is configured with default parameter from registry (*TransfertRate*). The transfer rate value unit is *bits per second*

3.7.4 Registry Settings

```
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\I2C]
    "Prefix"="I2C"
    "Dll"="AT91SAM9263EK_i2c.dll"
    "Index"=dword:1
    "Order"=dword:1
```

3.8 The SPI driver

3.8.1 General description of a Windows CE SPI driver

There is no standard interface for a SPI driver. This SPI driver is implemented as a stream driver, meaning that this driver is accessed through a Win32 file IO interface.

3.8.2 Code Location

The driver is divided into two parts :

- %_TARGETPLATROOT%\SRC\DRIVERS\SPI : this directory contains the code specific to the AT91SAM9263EK board,
- %_PLATFORMROOT%\COMMON\SRC\ARM\ATMEL\AT91SAM926x\DRIVERS\SPI : this directory contains the code specific to the AT91SAM926x family.

3.8.3 Description of the implementation

This driver implements the transactions with IOControls only (read and write functions won’t work).

There is one instance of the driver for each ChipSelect. SPI0: is for ChipSelect 0, SPI1: is for ChipSelect 1, … The settings for a given ChipSelect (clock polarity, baudrate, …) are given in the registry and cannot be changed dynamically.

There's only one supported IOControl :

IOCTL_SPI_TRANSACTION : this IOControl is called to perform an SPI transaction. For this SPI driver, an SPI transaction is a set of unitary transfers (a transfer is a read, write or read/write operation) that are performed without the ChipSelect being deasserted.

The parameter used by this IOControl is a pointer to an array of operation descriptors. An operation descriptor is constituted of a buffer for the reception and a buffer for the emission and the size of the unitary transfer.

```
typedef struct {
    PVOID pRxBuffer;  /*! < \brief location of the destination data */
    PVOID pTxBuffer;  /*! < \brief location of the source data */
    DWORD dwSize;           /*! < \brief size of the buffer*/
} T_SPI_TRANSACTION_ELEMENT_PARAM;
```

3.8.4 Limitations and known issues

Slave Mode is not supported.

It is possible to connect the 4 CS lines to a decoder in order to manage up to 16 Chip Select. This mode hasn't been tested.

The maximum size for a unitary transfer may not be greater than the maximum buffer sizes defined in the registry.

The baudrate for a given ChipSelect is given in the registry as a divider and doesn't take into account the actual value of the Master Clock.

3.8.5 Registry Settings

There are two kinds of registry settings :

- for setting up a the SPI controller

```
[HKEY_LOCAL_MACHINE\Drivers\SPIControler0]
    "ControllerIndex"=dword:0     ; Index of the SPI controller (in case there are many of them, which is not the case on AT91SAM9263 but may be on AT91SAM9262)
    "ChipSelectDecode"=dword:0    ; Chip Selects are connected directly to a peripheral device
;   "ChipSelectDecode"=dword:1    ; The four chip selects are connected to a 4to16 bits decoder !! Not tested !!
    "ClockSelection"=dword:0      ; The SPI operate at MCK
;   "ClockSelection"=dword:1      ; The SPI operate at MCK/32
;   "ModeFaultDetection"=dword:0 ; Mode Fault detectionis enabled
    "ModeFaultDetection"=dword:1 ; Mode Fault detectionis disabled
    "LoopBackEnable"=dword:0      ; Local Loopback disable
;   "LoopBackEnable"=dword:1      ; Local Loopback enable
    "DelayBetweenChipSelects"=dword:0  ; if ClockSelection (Div32) is 0 : Delay = DelayBetweenChipSelects/MCK
                                  ; if ClockSelection (Div32) is 1 : Delay = (DelayBetweenChipSelects * 32)/MCK
    "TxBufferSize"=dword:400      ;max transmit buffer size is 1Kbytes
    "RxBufferSize"=dword:400      ;max receive buffer size is 1Kbytes
```

- for setting up every individual ChipSelect driven by a controller

```
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\SPICS0]
    "SPIController"=dword:0; Index of the SPI controller
    "Index"=dword:0        ; Index of the Device (in this case SPI0:)
   "SPICS"=dword:0         ; Chip Select line
    "Dll" = "AT91SAM9263EK_spi.Dll"
    "Prefix" = "SPI"
    "Order" = dword:0
    "FriendlyName" = "SPIDriver Driver CS0 (on SPI0)"
;    "UsePIOCS"="A3" ; Use PIOA 3 as the chip select (this may be useful in case you want to use another line than NPCSx however it doesn't seem to work properly with the AT91SAM9261 (because of a glitch on the SPI clock at the very beginning of the transfer))
    "CSSetup"=dword:17180509     ;See AT91 SPI_CSR Register (value in hexa)
                            ; CPOL = 1
                            ; CPHA = 0
                            ; CSAAT = 1 -> NPCS rise right when LASTXFER is set instead of after a PDC transfer complted
                            ; BITS = 8 Bits
                            ; SCBR = 5 -> Baud Rate = 6.9 MHz if MCK is 69 MHz
                            ; DLYBS = 0x18 -> Delay = 0x18 x 1/MCK (=350 ns if MCK is 69 MHz)
                            ; DLYBCT = 0x17 -> Delay = 0x17 x 1/MCK (=333 ns if MCK is 69 MHz)
```

3.9 The SD Memory Card driver

3.9.1 General description of a Windows CE SD Memory Card driver

The SD Memory Card driver is a block driver: it implements a block per block access device driver. This driver type is dedicated to storage drivers (Flash, HDD, …).

It implements a standard stream interface with DSK prefix, plus a specific interface to manage the disk device. The specific interface contains specific Disk IoControl codes for disk accesses, implemented in DSK_IoControl. These IOControls manage Read/Write accesses, plus setting or getting information controls.

3.9.2 Code Location

The driver is divided into two parts :

- %_TARGETPLATROOT%\SRC\DRIVERS\SDMEMORY : this directory contains the code specific to the AT91SAM9263EK board,
- %_PLATFORMROOT%\COMMON\SRC\ARM\ATMEL\AT91SAM9263\DRIVERS\SDMemory : this directory contains the code specific to the AT91SAM9263 processor.
- %_PLATFORMROOT%\COMMON\SRC\ARM\ATMEL\AT91SAM926x\DRIVERS\SDMemory : this directory contains the code specific to the AT91SAM926x family.

3.9.3 Description of the implementation

The driver is divided into 3 levels:

The first level is implemented by the file mci_device.c. This is the low level layer which manages initialization of the MCI (Multimedia Card Interface) or the configuration for the Read/Write operations.

The second level is implemented by the file diskIo.c. This is the layer between the low level (hardware) and the top level (system). This file implements the specific SD Memory Card functional treatments of the driver as format or read/write blocks. These functions are called by the upper layer.

The third level is implemented by the file system.c. This is the top level layer which implements the stream interface of the driver.

3.9.4 Registry Settings

```
[HKEY_LOCAL_MACHINE\System\StorageManager\Profiles\SDMemory]
    "AutoMount"=dword:1
    "AutoPart"=dword:0
    "AutoFormat"=dword:0
    "PartitionDriver"="mspart.dll"
    "MountFlags"=dword:0
    "Name"="SDMemory"
    "Folder"="SD Memory Card"
    "DefaultFileSystem"="FATFS"

[HKEY_LOCAL_MACHINE\Drivers\sdmem]
    "Profile"="SDMemory"
    "IClass"=multi_sz:"{A4E7EDDA-E575-4252-9D6B-4195D48BB865}"
    "Order"=dword:4
    "SlotID"=dword:0
    "FriendlyName"="SDMemory"
    "Dll"="AT91SAM9263EK_sdmemory.dll"
    "Prefix"="DSK"
```

3.10 The AtapiEbi driver (EBI)

3.10.1 General description

The AtapiEbi driver is a block driver: it implements a block per block access device driver. This driver is dedicated to storage on hard-disk (HDD), using standard ATA-1 (IDE) transmission, on 16 bits data bus, in PIO mode 0.

The driver is based on the generic PCI ATAPI driver for PCMCIA. It is located in PUBLIC\COMMON\OAK\DRIVERS\BLOCK\ATAPI directory. This driver works with the components “FAT File system”, “Partition Driver” and “ATAPI PCI/IDE Storage Block Driver”. The FATFS driver uses the IDE driver (atapi_common_lib.lib). The IDE driver uses the AtapiEbi driver (DSK driver).

The AtapiEbi driver exports the functions ATARead(), ATAWrite()...

Important note about the BOOT configuration: when using the extension boards, set the peripheral clock at 50 MHz to avoid problems due to the inter-board connector.

3.10.2 Code Location

The driver is divided into two parts:

- %_TARGETPLATROOT%\SRC\DRIVERS\ATAPIEBI : this directory contains the code specific to the AT91SAM9263EK board,
- %_PLATFORMROOT%\COMMON\SRC\ARM\ATMEL\AT91SAM926x\DRIVERS\ATAPIEBI: this directory contains the code specific to the AT91SAM926x family.

3.10.3 Description of the implementation

It uses the EBI0 block, with the chip select NCS4.

HW blocks and timings are configured by AT91_IDEConfigure():

SMC configuration

Data bus is on 16 bits. The access mode is “byte select”. The R/W timings are configured for NWE and NRD signals.

PIO controller configuration

PC2 -> NWAIT, PC4 -> CFCS0, PC6 -> CFCE1, PC7 -> CFCE2

EBI configuration

EBI is configured for Compact Flash support, in “True IDE mode”.

EBI_CSA register is programmed with CS4A = 1 (CFCS0 mapped on NCS4/CFCS0 pin).

CFCE1 = 0 = /CS0, CFCE2 = 1 = /CS1

3.10.3.1 PIO and other board specific settings

IDE connector		on AT91	Blocks accessed	CF	SMC timing	IDE standard	
Pin	Name	Name	Name	Name	Name	Name	Description
1	CFRST, AB23	A23	EBI PIO A PA30/periph B	-	-	/RESET	must not be grounded
23	CFIOW	CFIOW, NBS3, NWR3	EBI CF	CFIOW	NWE	/IOWR	write
25	CFIOR	CFIOR, NBS1, NWR1	EBI CF	CFIOR	NRD	/IORD	read
31	CFIRQ, SPARE1, E19	PA21	-	-	-	INTRQ	DRQ interrupt. can be not used
37	CFCE1	CFCE1, PC6	PIO – EBI CF	CFCE1	NCS1	/CS0	access to registers
38	CFCE2	CFCE2, PC7	PIO – EBI CF	CFCE2	NCS2	/CS1	access to registers
...	AB[0-2]	A[0-2]	EBI	A0 A1 A2	NBS0 A1 A2	A[0-2]	access to registers
...	DB[0-15]	D[0-15]	EBI	D[0-15]		D[0-15]	data
27	NWAIT	NWAIT, PC2	PIO – EBI	NWAIT		IORDY	can be not used

3.10.4 Registry Settings

```
   [HKEY_LOCAL_MACHINE\Drivers\BuiltIn\ATAPI]
      "IClass"=multi_sz:"{CDDC3621-3512-4b3f-BB6F-B4DD5E061795}"
      "Prefix"="IDE"
      "Dll"="AT91SAM9263EK_atapiebi.dll"
      "Order"=dword:2
      "Class"=dword:01
      "SubClass"=dword:01
      "ConfigEntry"="GenericConfig"   ; PCI configuration entry point
      "Legacy"=dword:00        ; Modified. Not legacy (do not use Irq for primary and Irq+1 for secondary)
      "Irq"=dword:55                  ; (LOGINTR_BASE_PIOA = 64) + (pin PA21 on PIOA = 21)  = 85
      "SysIntr"=dword:00              ; SysIntr not specified
      "DMAAlignment"=dword:04         ; default DMA alignment
      "SoftResetTimeout"=dword:03     ; ATA/ATAPI spec defines 31s ceiling; this is too long
      "StatusPollCycles"=dword:100    ; Status register DRQ/BSY polling; 256 poll cycles
      "StatusPollsPerCycle"=dword:20  ; Status register DRQ/BSY polling; 32 polls per cycle
      "StatusPollCyclePause"=dword:05  ;  Status  register  DRQ/BSY  polling;  5  milliseconds  between  poll
cycles
      "SpawnFunction"="CreateATAPIEBI" ; controller-specific instantiation
      "IsrDll"=""                     ; required if 2 hard disks share the same interrupt
      "IsrHandler"=""                 ; required if 2 hard disks share the same interrupt
      "RegisterStride"=dword:01       ; ATA register stride; register block is contiguous
      "InterfaceType"="Internal"
      "FSD"="fatfsd.dll"
      "IoBase"=multi_sz:"50C00000", "50E00004"
      "IoLen"=multi_sz:"8", "8"
; IoBase1 = True IDE mode in Memory space NCS4, address of Status register of Hard disk
; IoBase2 = Alternate True IDE mode in Memory space NCS4, address of Alternate Status register of Hard disk
;           offset of 4 bytes for ATA_REG_ALT_STATUS because we're not in PCI bus case
; IoLen   = 8 because only 3 bits are required to adress an IDE hard-disk

   [HKEY_LOCAL_MACHINE\Drivers\BuiltIn\ATAPI\Device0]
      "IClass"=multi_sz:"{A4E7EDDA-E575-4252-9D6B-4195D48BB865}"
      "Prefix"="DSK"
      "Dll"="AT91SAM9263EK_atapiebi.dll"
      "InterruptDriven"=dword:01      ; 1 = enable interrupt driven I/O (DRQ)
      "DMA"=dword:00                  ; disable DMA
      "DoubleBufferSize"=dword:10000  ; 128 sector (65536 byte) double buffer
      "DrqDataBlockSize"=dword:200    ; 1 sector (512 byte) DRQ data block
      "WriteCache"=dword:01           ; enable on-disk write cache
      "LookAhead"=dword:01            ; enable on-disk look-ahead
      "DeviceId"=dword:00             ; device 0, i.e., primary master
      "TransferMode"=dword:08         ; use PIO flow control transfer mode, Mode 0=08
      "Profile"="HDProfile"

   [HKEY_LOCAL_MACHINE\System\StorageManager\Profiles\HDProfile]
       "Name"="HARDDISK"
       "Folder"="HardDisk"
```

```
    "AutoMount"=dword:1          ; Automatically mounts each detected partition
    "AutoPart"=dword:1           ; Automatically partitions the HD with largest creatable partition
    "AutoFormat"=dword:1         ; Formats the HD automatically when the HD is unformatted
    "MountFlags"=dword:0
    "FileSystem"="fatfsd.dll"
    "PartitionDriver"="mspart.dll"

; This block driver profile has its own FATFS settings
[HKEY_LOCAL_MACHINE\System\StorageManager\Profiles\HDProfile\FATFS]
    "EnableCache"=dword:1        ; 1 = Enable cache
    "EnableCacheWarm"=dword:0    ; cache write-back
;   "Flags"=dword:00280014       ; for a hard drive with non-atomic sector write operations
    "Flags"=dword:00000014       ; for a hard drive with atomic sector write operations
```

3.11 EEPROM diver

3.11.1 Description

The EEPROM driver is a stream driver that provides access to an EEPROM on I2C as if it was a file.

This allows an application to write or read data in EEPROM by calling the standard File API.

3.11.2 Code Location

The driver is divided into three parts :

- %_TARGETPLATROOT%\SRC\DRIVERS\EEPROM : this directory contains the code specific to the AT91SAM9263EK board,
- %_PLATFORMROOT%\COMMON\SRC\ARM\ATMEL\AT91SAM9263\DRIVERS\EEPROM : this directory contains the code specific to the AT91SAM9263 family.
- %_PLATFORMROOT%\COMMON\SRC\ARM\ATMEL\AT91SAM926x\DRIVERS\EEPROM : this directory contains the code specific to the AT91SAM926x family.

3.11.3 Description of the implementation

The driver uses the I2C driver for low level access.

It takes information about the EEPROM in the registry (page size, size of the EEPROM and address of the EEPROM on the I2C bus).

The driver takes care of buffer going over multiple pages.

3.11.4 Registry Settings

```
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\EEPROM]
    "Prefix"="EEP"
    "Dll"="AT91SAM9263EK_eeprom.dll"
    "Index"=dword:1
    "Order"=dword:2
    "PageSize"=dword:100          ;256 Bytes
    "TotalSize"=dword:20000 ;128 kBytes
    "I2CAddr"=dword:50
    "I2CDevice"="I2C1:"
```

3.12 The Serial driver

3.12.1 General description of a Windows CE serial driver

This is a layered driver. This Serial driver is implemented as a stream driver, stream means that this driver is accessed through a Win32 file IO interface.

The standard stream driver functions exported by the driver are:

COM_Init

COM_Deinit

COM_Open

COM_Close

COM_Read

COM_Write

COM_Seek

COM_PowerDown

COM_PowerUp

COM_IOControl

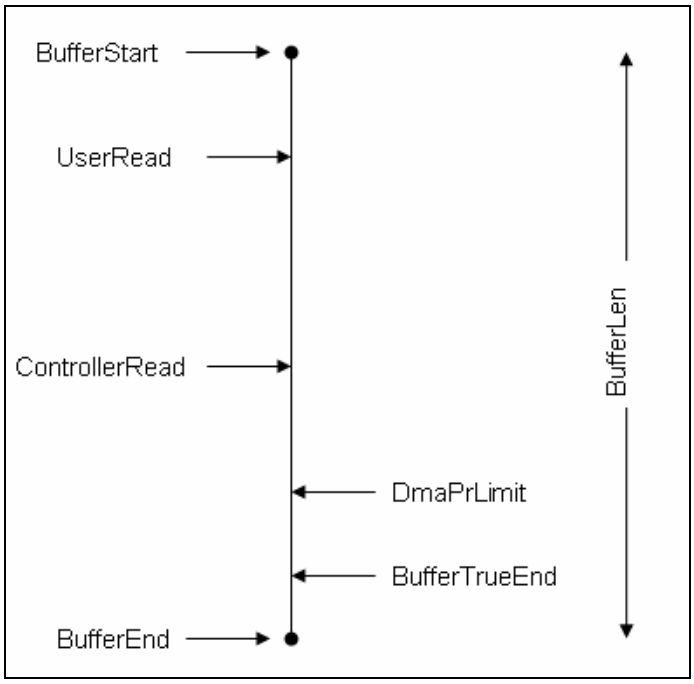


Figure 1 Structure of the buffer management system

The serial driver used a linear memory block for receive. The driver manage 2 pointers, one used to know what is already read by the user (UserRead), the second is used by the driver to know where write line in data.

The PDC managed all the receive/transmit data. It puts the receive data in the buffer and when this buffer is full, the oldest data are deleted. For the transmission, the PDC takes data in a write buffer. It reads from the current buffer and when this buffer is empty, it reads automatically from the next buffer.

The serial driver should initialize the current buffer (RPR for the reception and TPR for the transmission) and the next buffer (RNPR and TNPR) in the PDC.

For the reception, the configuration of the PDC is done in the interrupt thread and in COM_Read function. To do not loose characters, a limit is fixed for the DMA next buffer register.

For the transmission, the configuration of the PDC is done in the Write function after the reception of a TxEvent.

The Figure2 shows a global description of the read and the write operation.

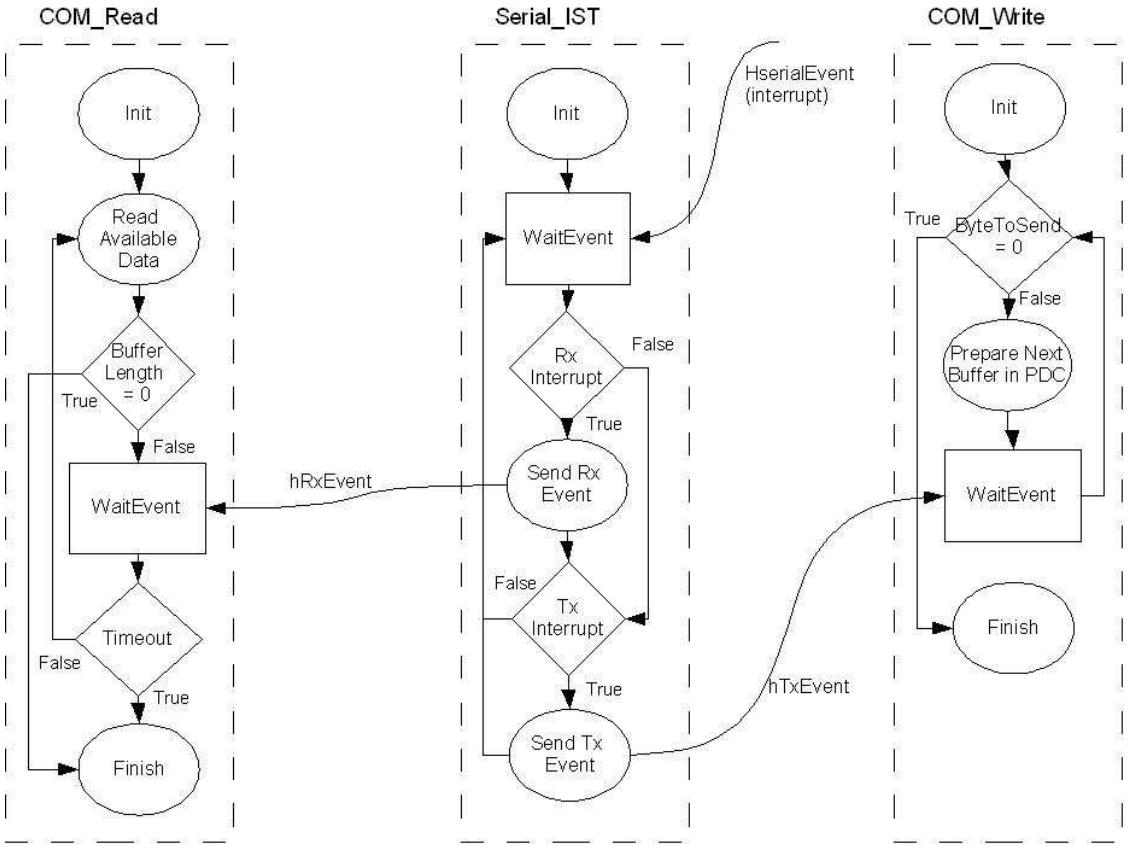


Figure 2 Operating sequence of the serial driver

3.12.2 Code Location

The driver is divided in three parts:

- %_TARGETPLATROOT%\SRC\DRIVERS\Serial : this directory contains the code specific to the AT91SAM9263EK board,
- %_PLATFORMROOT%\COMMON\SRC\ARM\ATMEL\AT91SAM9263\DRIVERS\Serial: this directory contains the code specific to the AT91SAM9263processor.
- %_PLATFORMROOT%\COMMON\SRC\ARM\ATMEL\AT91SAM926x\DRIVERS\Serial: this directory contains the code specific to the AT91SAM926x family.

3.12.3 Implementation

3.12.3.1 PDC and buffers management

The serial driver uses one 128 bytes-long buffer for transmission and four 512 bytes-long buffers for receiving. All these buffers are managed by the PDC.

When a transmission is needed, the driver stores in the transmission buffer the bytes to transmit and sets in the PDC registers the size of the data and the beginning of the buffer. When the transmission is

enabled, the PDC sends to the UART the bytes to send. When all the bytes were sent, the PDC launches an interruption. The driver handles it and stores in the buffer the new data to send.

Regarding the reception, there is just one buffer. Its size is configurable by editing the driver registry file (see 1.1.4).

3.12.3.2 PIOs and other board specific settings:

Here are the PIOs which are used by UART:

UART	Signal	PIO	Peripheral
UART0	TXD	PA26	A
	RXD	PA27	A
	RTS	PA28	A
	CTS	PA29	A
UART1	TXD	PD0	A
	RXD	PD1	A
	RTS	PD7	B
	CTS	PD8	B
UART2	TXD	PD2	A
	RXD	PD3	A
	RTS	PD5	B
	CTS	PD6	B

If the platform is not designed to drive RTS and CTS signals, it is possible to override the functions which set and clear them in the platform specific implementation “PLATFORM\AT91SAM9263EK\SRC\DRIVERS\SERIAL\serial.c”.

3.12.4 Registry Settings

```
;=============== UART1 (COM2) (Serial) ===============
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\Serial1]
   "Index"=dword:2
   "Prefix"="COM"                       ; COM
   "Dll"="AT91SAM9263EK_serial.dll"     ; in DriverSerialV2.dll
   "SerialPortIndex"=dword:0            ; USART0
   "TxBufferSize"=dword:400             ;
   "RxBufferSize"=dword:2800            ;
```

The above settings described UART1. Up to 3 UART can be set. There is one registry file per UART. The only specific parameter is “SerialPortIndex”. It permits to indicate which UART will be attached to the COM port. In this example UART0 is used when COM2 is opened.

Other fields are common to all other serial drivers and are fully described in the MSDN. See http://msdn.microsoft.com/library/en-us/wceddk40/html/cxconserialportdriverregistrysettings.asp

3.13 The Audio driver

3.13.1 General description of a Windows CE Audio driver

This is a layered driver: most of the code is provided by Microsoft, only the hardware specific part (PDD) is implement. The MDD is the wavemdd.lib.

The MDD interface is the standard Stream Interface.

The PDD interface is:

- **PDD_AudioDeinitialize** : called to deinitialize the driver
- **PDD_AudioGetInterruptType** : called whenever an audio interrupt occurs in order to determine the actual cause of the interrupt and take the adequate actions.
- **PDD_AudioInitialize** : called to initialize the driver
- **PDD_AudioMessage** : called to handle any custom WAVEIN or WAVEOUT messages
- **PDD_AudioPowerHandler** : called to power up or down the audio part.
- **PDD_WaveProc** : called to process WPDM_xxx messages from the MDD (like START PLAYBACK, STOP PLAYBACK, CONTINUE PLAYBACK messages). This is the main message handler for the PDD.

3.13.2 Code Location

The driver is located in “PLATFORM\AT91SAM9263EK\SRC\DRIVERS\ WAVEDEV”.

3.13.3 Implementation

This driver supports the AC97 controller of the AT91SAM9263.

This driver supports only 1 output flow 1 input flow.

It supports the following properties :

- Sample rate : 8KHz, 16KHz, 22.5 KHz, 32 KHz, 44100 KHz and 48 KHz
- Stereo and Mono
- 8 Bits/sample and 16 Bits/sample.

However the hardware doesn’t supports 8 bits/sample and mono channel (it doesn’t allow to output the mono stream on both Left and right channels). In consequence the driver handle the conversion. That is why the software mixer (if used) should always be configured for 16 bits/sample Stereo.

3.13.3.1 PDC and buffers management

The audio data is transported over the AC97 bus. The transfer is managed by a PDC for both input and output.

For both input and output a fixed number of buffers is used (4). These buffers are managed through FIFOs.

Output :

- FreeOutBufferFifo : Contains the list of buffers that are available.
- BufferToPlayFifo : contains the list of buffers that contains data to be played
- BufferBeingPlayedFifo : contains the list of buffers that contains data being transfered by the PDC.

Input :

- FreeInBufferFifo : Contains the list of buffers that are available.

- BufferRecordedFifo : contains the list of buffers that contains data already recorded
- BufferBeingRecordedFifo: contains the list of buffers that contains data being transfered by the PDC.

3.13.3.2 Codec and other board specific features.

The driver implements a generic codec management.

3.13.4 Additional and custom features

The driver implements 4 additional functions that are not part of a standard audio driver.

- WPDM_PRIVATE_WRITE_AC97 : Used to write a value in a register of the codec.
- WPDM_PRIVATE_READ_AC97 : Used to read a value of a register of the codec.
- WPDM_PRIVATE_GET_INPUT_VOLUME : Used to retrieve information about the volume of the microphone. The parameters are pointer to 2 UINT16 that receives the value of MIC_VOLUME and RECORD_GAIN registers of the codecs.
- WPDM_PRIVATE_SET_INPUT_VOLUME : Used to set the volume of the microphone. The parameters are 2 UINT16 that will be stored in MIC_VOLUME and RECORD_GAIN registers of the codec.

Example :

```
MMDRV_MESSAGE_PARAMS msg;
HANDLE hAudioDevice;
UINT16 usMicVolume,usRecordGain;
msg.uDeviceId = 0;
msg.dwUser = 0;

hAudioDevice =      CreateFile (L"WAV1:",
                                GENERIC_READ | GENERIC_WRITE,                   // Access (read-write) mode
                                FILE_SHARE_READ | FILE_SHARE_WRITE,   // Share mode
                                NULL,        // Pointer to the security attribute
                                OPEN_EXISTING,// How to open the serial port
                                0,           // Port attributes
                                NULL);
if (hAudioDevice == INVALID_HANDLE_VALUE)
{
            RETAILMSG(1,(TEXT("Unable to open audio device\r\n")));
            return -1;
}
msg.uMsg = WPDM_PRIVATE_GET_INPUT_VOLULME;
msg.dwParam1 = (DWORD)&usMicVolume;
msg.dwParam2 = (DWORD)&usRecordGain;
if (DeviceIoControl(hAudioDevice,IOCTL_WAV_MESSAGE,&msg,sizeof(msg),NULL,0,NULL,NULL) == FALSE)
{
            RETAILMSG(1,(TEXT("WPDM_PRIVATE_GET_INPUT_VOLULME failed\r\n")));
}
msg.uMsg = WPDM_PRIVATE_SET_INPUT_VOLULME;
msg.dwParam1 = usMicVolume;
msg.dwParam2 = usRecordGain;
```

```
if (DeviceIoControl(hAudioDevice,IOCTL_WAV_MESSAGE,&msg,sizeof(msg),NULL,0,NULL,NULL) == FALSE)
{
        RETAILMSG(1,(TEXT("WPDM_PRIVATE_READ_AC97 failed\r\n")));
}
CloseHandle(hAudioDevice);
```

3.13.5 Registry Settings

```
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\WaveDev\]
  "Prefix"="WAV"
  "Dll"="at91sam9263ek_wavedev.dll"
  "Index"=dword:1
  "Order"=dword:0
  "Priority256"=dword:150

[HKEY_LOCAL_MACHINE\Audio\SoftwareMixer]
   "BufferSize"=dword:1000
   "Priority256"=dword:151
   "SampleRate"=dword:ac44
   "Buffers"=dword:4
   "EnableLowPassFilter"=dword:1

[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\WAPIMAN\]
   "Priority256"=dword:152
```

Notes:

- To disable the software, the developer must provide it with invalid registry settings (SampleRate=1 for example).
- All audio-related system calls go through the software mixer (if enabled). The software mixer opens the audio driver only once and with fixed settings (in the example above : 44100 Hz and the defaults : Stereo 16 Bits). When playing sample with different properties (8KHz mono 8 bits for example), the software mixer will do the conversion in software.

3.14 The Ethernet driver

3.14.1 General description of the driver

This driver supports the EmacB controller of the AT91SAM9263EK. This internal controller is placed in Ethernet stack as a MAC controller. This driver implements the NDIS (“Network Device Interface Specification”) API.

NDIS is part of the networking architecture used in Microsoft Windows operating systems. NDIS provides simplified miniport device driver architecture to enable communication with network adapters using common driver interfaces.

3.14.2 Code Location

The driver is divided into three parts :

- %_TARGETPLATROOT%\SRC\DRIVERS\EmacbNDIS: this directory contains the code specific to the AT91SAM9263EK board,
- %_PLATFORMROOT%\COMMON\SRC\ARM\ATMEL\AT91SAM9263\DRIVERS\EmacbNDIS : this directory contains the code specific to the AT91SAM9263 family.
- %_PLATFORMROOT%\COMMON\SRC\ARM\ATMEL\AT91SAM926x\DRIVERS\EmacbNDIS : this directory contains the code specific to the AT91SAM926x family.

3.14.3 Implementation

3.14.3.1 Specific device settings

EmacB has no internal EEPROM to store Mac address, Vendor ID and Product ID. For final application, external EEPROM is needed to save these specific device settings. During debug time, these settings are loaded from registry settings or bootloader settings by define the bootSettings key in registry.

3.14.3.2 Known issues and limitations

Both Ethernet driver and KITL cannot work at the same time. Indeed, both are using EmacB controller.

3.14.4 Registry Settings

Following registers allow to configure some device parameters, especially “NetworkAddress”, “VendorID” and “ProductID”. These parameters are read from Registry Settings unless the EEPROM is used to save these specific device settings.

```
[HKEY_LOCAL_MACHINE\Comm\EMACB]
  "DisplayName"="EMACB Adapter"
  "Group"="NDIS"
  "ImagePath"="at91sam9260ek_emacbndis.dll"

[HKEY_LOCAL_MACHINE\Comm\EMACB\Linkage]
  "Route"=multi_sz:"EMACB1"

[HKEY_LOCAL_MACHINE\Comm\EMACB1]
  "DisplayName"="EMACB Adapter"
  "Group"="NDIS"
  "ImagePath"="at91sam9260ek_emacbndis.dll"

[HKEY_LOCAL_MACHINE\Comm\Tcpip\Linkage]
  "Bind"="EMACB"

[HKEY_LOCAL_MACHINE\Comm\EMACB1\Parms]
  "BusNumber"=dword:0
  "BusType"=dword:0
  "XmitBuffer"=dword:10
  "RecvBuffer"=dword:F0
; Be carefull with the VramXmitBuffer and VramRecvBuffer values
; These params allocate EMACB1 buffer  into  SRAM or SDRAM....
; Be sure that the place where you allocate buffer is really empty
```

```
; if not this may cause some problems
; See if another driver isn't using the same memory space
; For example: the display driver sometimes allocate the video memory in SRAM
   "VramXmitBuffer"=dword:300000
;  "VramRecvBuffer"=dword:3005F8
   "IrqNumber"=dword:15
   "SYSINTR" = dword:11
   "MemBaseAddress"=dword:30000000
   "NetworkAddress"="02-03-04-05-06-07"
   "VendorID"=dword:1
   "ProductID"=dword:1

 [HKEY_LOCAL_MACHINE\Comm\EMACB1\Parms\TcpIp]
   "EnableDHCP"=dword:0
   "UseZeroBroadcast"=dword:0
   "DefaultGateway"="192.168.0.1"
   "IpAddress"="192.168.100.185"
   "Subnetmask"="255.255.255.0"
   "DNS"="0.0.0.0"
   "WINS"="0.0.0.0"
   "BootSettings"=dword:1

[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\NDIS]
   "Dll"="NDIS.Dll"
   "Prefix"="NDS"
   "Index"=dword:0
   "Order"=dword:1
   ; Indicate NDS is a power manageable interface
   "IClass"="{A32942B7-920C-486b-B0E6-92A702A99B35}"

[HKEY_LOCAL_MACHINE\Drivers\Virtual]
   "Dll"="RegEnum.dll"
   "Order"=dword:1
   "Flags"=dword:1

[HKEY_LOCAL_MACHINE\Drivers\Virtual\NDIS]
   "Dll"="NDIS.dll"
   "Order"=dword:1
   "Prefix"="NDS"
```