phyCORE-i.MX35

BINARY BSP DOCUMENTATION

WINCE 6.0

Edition August 2010

Table of Contents

1 Scope of this Document

This document is for developer who have experience in Windows Embedded CE programming. The document describes how to communicate with your phyCore-i.MX35 device in Windows Embedded CE.

2 BSP Contents

The Binary BSP contains the following folders and files:

- **WINCE600**:
  contains the Libraries and Build files needed to build an Windows CE 6.0 Image for the phyCore i.MX35 Evaluation Kit. For installation instructions, refer to chapter 3.

Folder structure of the BSP:

```
PLATFORM
|--COMMON
|       |--SRC
|               |--SOC
|                       |--COMMON_FSL_V2
|                               |-- INC
|
|--iMX35Phytec
        |--Binaries
        |--CATALOG
        |--FILES
        |--OalLibs
        |--SDKLibs
        |--SRC
                |--COMMON
                |       |--BSPCMN
                |       |--OTHER
                |--INC
                |--KITL
                |--OAL
```

3 Requirements

Visual Studio 2005 with Microsoft Windows Embedded CE 6.0 Plugin.

Updates:

- Visual Studio 2005 Service Pack 1
- Windows Embedded CE 6.0 Platform Builder Service Pack 1
- Windows Embedded CE 6.0 R2
- Windows Embedded CE 6.0 R3:
    - Windows Embedded CE 6.0 Monthly Update (January 2010)
    - Windows Embedded CE 6.0 Monthly Update (February 2010)
    - Windows Embedded CE 6.0 Monthly Update (March 2010)
    - Windows Embedded CE 6.0 Monthly Update (April 2010)
    - Windows Embedded CE 6.0 Monthly Update (May 2010)

, all available on the Microsoft Website.

4 Installation and Build – Instructions

4.1 Preparation

Before you install the BSP, be sure to perform the following steps:
In order to avoid problems, it is recommended to remove any previously installed i.MX35 BSPs. To do so, go to the windows control panel, click “Add or Remove Programs” and select “PHYTEC WinCe i.MX35-Kit” to remove.
If you decide to not remove an installed i.MX35 BSP, be warned that the installation of the Phytec i.MX35 BSP will override all changes you made to files contained in this BSP. So either save your changed files to another location or consider removing the previous BSP version.

4.2 BSP Installation

Unzip the “BinaryBSP_ExampleProject.zip” in the same directory, where the WINCE600 folder is located on your hard drive.

4.3 Building an Image

The best way to start building an image for the Phytec i.MX35 platform is to use the provided example project. To do so, open platform builder and perform File -> Open -> Project/Solution and then choose the iMX35Phytec.sln file from the example project directory. After that, choose Build -> Build iMX35Phytec.

CAUTION: as mentioned above, this step will override any files associated with a previously installed i.MX35 BSP!

5 Supported Features

The following table lists all features supported by this release of the BSP.

Feature	Supported?	Comment
OAL		
KITL	YES	at the moment, the used KITL-Connection is fixed to 'active KITL'
Serial debug	YES	Serial debug message show up on UART2
I²C	YES	Used to read and write to the external RTC on I²C
External RTC	YES	• Used to save the time during power-off • External RTC is synchronized, when time is changed in controlpanel
Splashscreen	NO	Display of a custom bitmap on the display during startup.
Drivers		
Display	YES	Support for: • Hitachi TX09D70VM1CCA • Hitachi TX18D16VM1CBB • Primeview PD050VL1
Camera	NO	
SPI	YES	Bus Driver for CSPI interface
Serial	YES	UART1 for serial communication
GPT	YES	General Purpose Timer
PWM	YES	
NANDFC	YES	File System, Hive-based registry
One Wire	NO	

Audio	YES	Playback and Recording via Line-In
MMC/SD	YES	File System
PCMCIA	NO	
USB Host	YES	USB HighSpeed Host Port for Mass-Storage and HID (Mouse, Keyboard)
USB OTG	YES	OTG Port to establish an Active Sync connection to PC
I²C	YES	Bus Driver for e.g. Camera and EEPROM
Ethernet	YES	• Download WinCE-Image over Ethernet in Eboot • WinCE NDIS driver
CAN	YES	Use of the intern CAN-Controller
Battery	NO	
EEPROM	YES	

6 Driver Documentation

6.1 Display Driver

The Display Driver is a Direct Draw Driver using the Synchronous Display Controller (SDC) of the i.MX35 Image Processing Unit (IPU). It supports the Hitachi TX09D70VM1CCA QVGA panel (default panel), the Hitachi TX18D16VM1CBB WVGA panel and the Prime View PD050VL1 VGA panel.

Driver Attribute	Definition
Import Library	ddgpe.lib, gpe.lib
Driver DLL	ddraw_ipu.dll
Required Catalog Items	Third Party → BSPs → iMX35Phytec → Device Drivers → Display → TX09D70VM1CCA (QVGA) Third Party → BSPs → iMX35Phytec → Device Drivers → Display → TX18D16VM1CBB (WVGA) Third Party → BSPs → iMX35Phytec → Device Drivers → Display → Prime View PD050VL1 (VGA)
SYSGEN Dependency	SYSGEN_DDRAW=1
BSP Environment Variable	BSP_DISPLAY_TX09_35 for Hitachi TX09D70VM1CCA BSP_DISPLAY_TX18_81 for Hitachi TX18D16VM1CBB BSP_DISPLAY_PRIMEVIEW_PD050VL1 for PrimeView PD050VL1

Software Operation

Communication with the Display is done using the Graphics Device Interface (GDI) API and the DirectDraw API, both defined by Microsoft (please refer to ***Windows Embedded CE Features –> Shell, GWES, and User Interface –> Graphics, Windowing and Events –> GWES Application Development –> Graphics Device Interface*** for help on GDI and to ***Windows CE Features –> Graphics –> DirectDraw*** for help on DirectDraw).

The Display Driver supports the following DirectDraw features:

- page flipping (one backbuffer)
- Overlay surfaces using RGB or the FOURCC UYVY pixel format.
- Overlaying using a color key for the overlay surface for RGB colors.
- Overlaying using a color key for the non-overlay graphics surface for RGB colors.
- Stretching of overlay surfaces.

The Display Driver also uses Post-Processing features of the i.MX35 IPU to provide the following possibilities:

- Color space conversion of UYVY overlay data to RGB. This conversion is required in order to combine the overlay data with RGB graphics plane data in the IPU SDC.
- Resizing of the overlay surface.
- Rotation of the overlay surface (used when the screen orientation is rotated, see 'Button Operation' above).
- Resizing and rotation of the primary graphics surface.

In order to communicate directly with the display driver, an escape code mechanism is provided. Descriptions of standard escape codes can be found under ***Developing a Device Driver → Windows Embedded CE Drivers → Display Drivers → Display Driver Development Concepts → Display Driver Escape Codes*** in the Platform Builder Help.

Configuration

The Display Driver is configured using the registry keys described below. Depending on the selected display in the catalog, the corresponding registry settings are included.

If the '*Hitachi TX09D70VM1CCA'* panel is selected, the following registry settings are included:

```
[HKEY_LOCAL_MACHINE\Drivers\Display\DDIPU]
      "Bpp"=dword:10                ; 16bpp
      "CxScreen"=dword:F0           ; 240
      "CyScreen"=dword:140          ; 320
      "VideoBpp"=dword:10           ; RGB565
      "PanelType"=dword:4           ; TX09 QVGA Panel (3,5 Inch)
      "VideoMemSize"=dword:600000
```

If the '*Hitachi TX18D16VM1CBB*' panel is selected, the following registry keys are included:

```
[HKEY_LOCAL_MACHINE\Drivers\Display\DDIPU]
      "Bpp"=dword:10                ; 16bpp
      "CxScreen"=dword:320          ; 800
      "CyScreen"=dword:1E0          ; 480
      "VideoBpp"=dword:10           ; RGB565
      "PanelType"=dword:5           ; TX18 QVGA Panel (8,1 Inch)
      "VideoMemSize"=dword:600000
```

If the '*PrimeView PD050VL1'* panel is selected, the following registry keys are included:

```
[HKEY_LOCAL_MACHINE\Drivers\Display\DDIPU]
      "Bpp"=dword:10                ; 16bpp 10
      "CxScreen"=dword:280          ; 640
      "CyScreen"=dword:1E0          ; 480
      "VideoBpp"=dword:10           ; RGB666
      "PanelType"=dword:6           ; PRIME VIEW 640x480 Panel
      "VideoMemSize"=dword:200000   ;
```

VideoMemSize determines the amount of memory used by the driver for the various surfaces.

Bpp sets the bytes per pixel of the display, this is set to 16-bit RGB pixel data for all supported displays.

6.2 SPI Driver

The SPI Driver allows rapid serial communication and can be configured to master- or slave-mode. You can access following SPI Busses and Chipselects:

- SPI1_SS0
- SPI1_SS1
- SPI2_SS0

Driver Attribute	Definition
Import Library	spisdk.lib
Driver DLL	Cspi.dll
Required Catalog Items	Third Party → BSPs → iMX35Phytec → Device Drivers → CSPI Bus → CSPI 1/2
SYSGEN Dependency	N/A
BSP Environment Variable	BSP_CSPIBUS1=1 BSP_CSPIBUS2=1

Software Operation

The CSPI Driver is a stream interface driver but can be accessed over extern declared functions. For successfully access to the driver, you have to include following Library and Header-File:

Include Library:

/WINCE600/PLATFORM/iMX35Phytec/SDKLibs/spisdk.lib

Include Header:

/WINCE600/PLATFORM/COMMON/SRC/SOC/COMMON_FSL_V2/INC/cspibus.h

Before working with SPI devices, a handle to the driver must be obtained using the **`CSPIOpenHandle`** (special device SPI1:) function. As always, this handle has to be closed using the **`CSPICloseHandle`** function, when no longer needed. Configuring the CSPI and exchanging data is done using the following Functions and Structures:

Functions:

BOOL CSPIExchange(HANDLE hCSPI, PCSPI_XCH_PKT_T pCspiXchPkt)

relevant Parameters:

hCSPI

Handle to the CSPI stream driver that you have created with the **CSPIOpenHandle.**

pCspiXchPkt:

The pointer of CSPI_XCH_PKT_T structure that contains a pointer to the CSPI_BUSCONFIG_T structure.

```
// CSPI bus configuration
typedef struct
{
    UINT8   chipselect;
    UINT32  freq;
    UINT8   bitcount;
    BOOL    sspol;
    BOOL    ssctl;
    BOOL    pol;
    BOOL    pha;
    UINT8   drctl;
    BOOL    usedma;
    BOOL    usepolling;
} CSPI_BUSCONFIG_T, *PCSPI_BUSCONFIG_T;
```

Members

chipselect

In master mode, these byte select the external slave devices by asserting the SSn outputs. Only the selected SSn signal will be active while the remaining chipselects signals will be negated.

freq

Frequency to which CSPI bus has to be configured.

bitcount

Number of bits to be transmitted at a time.

sspol

Selects the polarity of the chipselect signal.

sspol = TRUE	sspol = FALSE
Clock polarity Active High	Clock polarity Active Low

ssctl

SS Signal waveform select the polarity of the chipselect signal.

ssctl = TRUE	ssctl = FALSE
SS Signal insert pulse	SS Signal stays asserted

pol

SPI clock polarity control.

pol = TRUE	pol = FALSE
Clock polarity Active Low	Clock polarity Active High

pha

SPI Clock/Data Control

pha = TRUE	pha = FALSE
Phase 1 operation	Phase 0 operation

drctl

SPI Data Ready Control.

Drctl = 0	→	SPI_RDY signal is a don't care.
Drctl = 1	→	Burst will be triggered by the falling edge of SPI_RDY signal.
Drctl = 2	→	Burst will be triggered by a low level of the SPI_RDY signal.

usedma

If "TRUE" driver use DMA otherwise "FALSE"

usepolling

If "TRUE" driver is polling the status otherwise waiting for an interrupt request.

```
// CSPI exchange packet
typedef struct
{
    PCSPI_BUSCONFIG_T pBusCnfg;
    LPVOID pTxBuf;
    LPVOID pRxBuf;
    UINT32 xchCnt;
    LPWSTR xchEvent;
    UINT32 xchEventLength;
} CSPI_XCH_PKT_T, *PCSPI_XCH_PKT_T;
```

Members

pBusCnfg

Object points to the bus configuration structure `PCSPI_BUSCONFIG_T` of the SPI bus.

pTxBuf

A pointer to a buffer of bytes to transmit in a write operation.

pRxBuf

A pointer to a buffer that will be read into during a read operation.

xchCnt

The number of packets to transmit from pTxBuf / read to pRxBuf.

xchEvent

An event name string that is signalled whenever the packet operation (read or write) has completed. **CreateEvent** must be called to create the event before the packet can be passed to the CSPI.

xchEventLength

Size of the event name.

Bus Configuration

In order to successfully communicate with a SPI device, the CSPI module first has to be configured at any write or read access. This is done by setting up a **CSPI_BUSCONFIG_T** structure according to the intended configuration. After that, this structure is set as the **pBusCnfg** -field of a **CSPI_XCH_PKT_T** structure.
For any write or read operation you must fill the complete structure **CSPI_XCH_PKT_T**.

After all this is set up, the configuration is actually done by every write or read access by calling **CSPIExchange** with the previously set up **CSPI_XCH_PKT_T** structure as the input parameter. To wait on the operation to complete, a WaitForSingleObject on the assigned event handle has to be done.

6.3 Serial Driver

The Serial Driver provides support for the internal UART1 of the i.MX35 (on default UART2 is used as debug output). The Driver is implemented as a Stream Interface Driver. It implements the PDD Layer (platform dependent) and is then linked to the Microsoft - provided serial MDD library (com_mdd2.lib) to form the Driver.

Driver Attribute	Definition
Import Library	com_mdd2.lib
Driver DLL	csp_serial.dll
Required Catalog Items	Third Party → BSPs → iMX35Phytec → Device Drivers → Serial → UART1 / UART2
SYSGEN Dependency	N/A
BSP Environment Variable	BSP_SERIAL_UART1=1

Software Operation

The Serial Driver implements the architecture recommended by Microsoft. For detailed information on this architecture, refer to ***Developing a Device Driver → Windows Embedded CE Drivers → Serial Port Drivers → Serial Driver Development Concepts*** in the Windows CE Help**.**

Configuration

The UART1 Serial Driver is configured by the registry settings described below:

```
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\COM2]
      "DeviceArrayIndex"=dword:0
      "IoBase"=dword:43F90000
      "IoLen"=dword:D4
      "Prefix"="COM"
      "Dll"="csp_serial.dll"
      "Index"=dword:2
      "Order"=dword:0
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\COM2\Unimodem]
      "Tsp"="Unimodem.dll"
      "DeviceType"=dword:0
      "FriendlyName"="i.MX35 COM2 UNIMODEM"
      "DevConfig"=hex: 10,00, 00,00, 05,00,00,00, 10,01,00,00,
                       00,4B,00,00, 00,00, 08, 00, 00, 00,00,00,00
```

The UART2 Serial Driver is configured by the registry settings described below:

```
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\COM3]
      "DeviceArrayIndex"=dword:0
      "IoBase"=dword:43F94000
      "IoLen"=dword:D4
      "Prefix"="COM"
      "Dll"="csp_serial.dll"
      "Index"=dword:3
      "Order"=dword:0
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\COM3\Unimodem]
      "Tsp"="Unimodem.dll"
      "DeviceType"=dword:0
      "FriendlyName"="i.MX35 COM3 UNIMODEM"
      "DevConfig"=hex: 10,00, 00,00, 05,00,00,00, 10,01,00,00,
                       00,4B,00,00, 00,00, 08, 00, 00, 00,00,00,00
```

Configure as debug output

You can configure any UART as output for system and debug messages. You have to edit the “DEBUG_PORT” on “bsp_cfg.h” in the Binary BSP under ../WINCE600/PLATFROM/iMX35Phytec/SRC/INC/. The following example show you how to configure UART2 as debug output:

```
#define DEBUG_PORT              DBG_UART2
```

Notice:

A “Rebuild Current BSP and Subprojects” is necessary for changing debug port.

Serial PDD Functions

The Serial PDD Functions defined by Microsoft are mapped to Serial Driver internal functions as follows:

```
const HW_VTBL IoVTbl = {
        SerSerialInit,
        SerPostInit,
        SerDeinit,
        SerOpen,
        SerClose,
        SL_GetIntrType,
        SL_RxIntrHandler,
        SL_TxIntrHandler,
        SL_ModemIntrHandler,
```

```
    SL_LineIntrHandler,
    SL_GetRxBufferSize,
    SerPowerOff,
    SerPowerOn,
    SL_ClearDTR,
    SL_SetDTR,
    SL_ClearRTS,
    SL_SetRTS,
    SerEnableIR,
    SerDisableIR,
    SL_ClearBreak,
    SL_SetBreak,
    SL_XmitComChar,
    SL_GetStatus,
    SL_Reset,
    SL_GetModemStatus,
    SerGetCommProperties,
    SL_PurgeComm,
    SL_SetDCB,
    SL_SetCommTimeouts,
};
```

Refer to ***Developing a Device Driver → Windows Embedded CE Drivers → Serial Port Drivers → Serial Port Driver Reference → Serial Port Driver Structures → HW_VTBL*** to determine which internal function maps to which Serial PDD function.

6.4 NAND-Flash Driver

The NAND-Flash driver is used to make our NAND-Flash available as a block device in Windows CE. Therefore the Flash Abstraction Layer (FAL) is implemented.
Further more the hive-based registry is stored on the NAND-Flash device.

Driver Attribute	Definition
Import Library	N/A
Driver DLL	flashpdd_nand.dll
Required Catalog Items	Third Party → BSPs → iMX35Phytec → Storage Drivers → MSFlash Drivers → Nand Flash Driver
SYSGEN Dependency	N/A
BSP Environment Variable	BSP_NAND_FMD=1

Software Operation

The NAND Flash Driver implements the Flash Media Drivers (FMD) architecture. Refer to ***Developing a Device Driver > Windows CE Drivers > Flash Media Drivers*** for more information.

Hive-based registry

The NAND-Flash is also used as storage device for the hive-based registry. For saving the registry you have to execute „SaveReg.exe“.

Notes:

„SaveReg.exe“ deletes automatically the touch calibration tool startup entry.

If you have build an own WinCE-Image it is necessary to delete the hole NAND-Flash area.

6.5 Audio Driver

The audio driver is used to playback or record audio using the Wolfson WM9712 IC.

Driver Attribute	Definition
SOC static library	N/A
Import Library	N/A
Driver DLL	Wolfson_audio.dll
Required Catalog Items	Third Party → BSPs → iMX35Phytec → Device Drivers → Audio → Wolfson WM9712 Audio CoreOS → CEBASE → Graphics and Multimedia Technologies → Audio → Waveform Audio
SYSGEN Dependency	SYSGEN_AUDIO=1
BSP Environment Variable	BSP_WOLFSON_AUDIO=1

Software Operation

The audio driver conforms to the architecture for audio drivers recommended by Microsoft. For a detailed description, read ***Developing a Device Driver → Windows CE Drivers → Audio Drivers → Audio Driver Development Concepts*** in the WindowsCE help.

6.6 MMC/SD Driver

The Secure Digital Host Controller (SDHC) supports Multimedia Cards (MMC) and Secure Digital Cards (SD).
The SDHC driver provides the interface between Microsoft's SD Bus Driver and the i.MX35 SDHC.

Driver Attribute	Definition
SOC static library	esdhcbase_common_fsl_v2.lib
Import Library	N/A
Driver DLL	esdhc.dll
Required Catalog Items	Third Party → BSPs → iMX35Phytec → Device Drivers → SD Controller → Enhanced SD Host Controller 1
SYSGEN Dependency	SYSGEN_SD_MEMORY=1

	SYSGEN_SDBUS IMGSDBUS2
BSP Environment Variable	BSP_ESDHC1=1

Software Operation

The SDHC Driver follows the architecture for Secure Digital Host Controller drivers recommended by Microsoft. For more information, look at ***Developing a Device Driver → Windows Embedded CE Drivers → Secure Digital Card Drivers → Secure Digital Card Driver Development Concepts*** in the Windows CE Help.

Required Catalog Items

for SD and MMC Memory Card Support:

Catalog –> Device Drivers –> SDIO –> SD Memory

Registry Settings

```
 [HKEY_LOCAL_MACHINE\Drivers\BuiltIn\ESDHC_MX35]
      "Order"=dword:21
      "Dll"="esdhc.dll"
      "Prefix"="SHC"
      "Index"=dword:1
      "DisableDMA"=dword:0  ; Use this reg setting to disable both
                              internal and external DMA
      "MaximumClockFrequency"=dword:3197500  ; 52 MHz max clock speed
      "UseExternalDMA"=dword:0

[HKEY_LOCAL_MACHINE\System\StorageManager\Profiles\MMC]
      "Name"="MMC Card"
      "Folder"="MMCMemory"

[HKEY_LOCAL_MACHINE\System\StorageManager\Profiles\SDMemory]
      "Name"="SD Memory Card"
      "Folder"="SDMemory"

[HKEY_LOCAL_MACHINE\Comm\PegasSDN1\Parms]
      "SdioBitMode"=dword:00000001

[HKEY_LOCAL_MACHINE\Comm\PegasSDN1\Parms]
      "DisablePowerManagement"=dword:1
      "ResetOnResume"=dword:0
      "RebindOnResume"=dword:1
```

6.7 GPT Driver

The GPT module can be used to implement various time-related tasks with high accuracy such as external triggered timestamps and interrupts or generating periodic output. Therefore, it supports input-capture and output-compare functionality.

Driver Attribute	Definition
Static Library	gpt_common_fsl_v2.lib, gpt_mx35_fsl_v2.lib
Import Library	gptsdk.lib
Driver DLL	gpt.dll
Required Catalog Items	Third Party → BSPs → iMX31_Phytec → Device Drivers → Timers → GPT
SYSGEN Dependency	N/A
BSP Environment Variable	BSP_GPT=1

Software Operation

For successfully access to the driver, you have to include following Library and Header-File:

Include Library:

/WINCE600/PLATFORM/iMX35Phytec/SDKLibs/gptsdk

Include Header:

/WINCE600/PLATFORM/COMMON/SRC/SOC/COMMON_FSL_V2/INC/gpt.h

The GPT Driver is a stream - interface driver and therefore can be accessed using the file system API (special device GPT1:). After opening a handle to the driver using the GptOpenHandle(L"GPT1:") call, communication with the device is done by issuing the available wrapper - functions:

HANDLE GptOpenHandle(LPCWSTR lpDevName)

Creates a handle to the GPT stream driver.

Return value:

Handle to the GPT or INVALID_HANDLE_VALUE if there was an error.

BOOL GptCloseHandle(HANDLE hGpt)

Closes a GPT-handle previously created by **GptOpenHandle.**

HANDLE GptCreateTimerEvent(HANDLE hGpt, LPTSTR eventName)

Returns an event handle, which is triggered when a timer period has elapsed.
relevant parameters:

eventName: string containing the name of the event to be created.

BOOL GptReleaseTimerEvent(HANDLE hGpt, LPTSTR eventName)

Closes a handle to the GPT timer event.
relevant parameters:

eventName: string containing the name of the event to be closed.

BOOL GptStart(HANDLE hGpt, pGPT_Config pTimerConfig)

Enables the GPT Interrupt and starts the GPT timer with the given configuration.
Relevant parameters:

pTimerConfig: pointer to a pGPT_Config structure containing information for setting GPT timer delay (see structure description below)

BOOL GptStop(HANDLE hGpt)

Disables the GPT Interrupt and stops the GPT timer.

BOOL GptResume(HANDLE hGpt)

Reactivates the GPT, usually called after a Stop.

Structures

```
typedef enum timerMode{
    timerModePeriodic,
    timerModeFreeRunning
} timerMode_c;
```

```
typedef enum timerSrc{
    GPT_NOCLK,
    GPT_IPGCLK,
    GPT_HIGHCLK,
    GPT_EXTCLK,
    GPT_32KCLK
} timerSrc_c;


typedef struct
{
    timerMode_c timerMode;
    UINT32 period;
    timerSrc_c timerSrc;
} GPT_Config, *pGPT_Config;
```

Members

timeMode_c timerMode:

You can configure the GPT in two different modes:

Timer Mode	Description
timerModeFreeRunning	In free-running-mode, the counter is not reset when an compare value event occurs. The counter continues upt to 0xFFFFFFFF and roll over to 0x00000000.
timerModePeriodic	If the timer reaches the compare value it reset itself and starts again from 0x00000000.

UINT32 period:

Stores the timer value. The value is dependent on the selected timer source.

timerSrc_c timerSrc:

Select the clock source of the general purpose timer.

Clock Source	**Frequency**
GPT_NOCLK	66,5 MHz
GPT_IPGCLK	66,5 MHz
GPT_HIGHCLK	66,5 MHz
GPT_EXTCLK	66,5 MHz
GPT_32KCLK	32 kHz

6.8 USB Host Driver

The USB Host Driver provides USB 2.0 host support using the USB High Speed Host (H2) of the i.MX35. The driver has class support for mass storage, HID, printer and RNDIS clients.

Driver Attribute	Definition
SOC static library	Usbh_usb2com_common_fsl_v2.lib Usbh_ehcdmdd_common_fsl_v2.lib Usbh_ehcdpdd_common_fsl_v2.lib
Import Library	N/A
Driver DLL	hcd_hsh2.dll
Required Catalog Items	Third Party → BSPs → iMX35Phytec → Device Drivers → USB Devices → USB Host Devices → High Speed Host intern
SYSGEN Dependency	SYSGEN_USB=1
BSP Environment Variable	BSP_USB=1 BSP_USB_HSH2=1

The host driver requires one or more class drivers to be loaded (see below).

Required Catalog Items

For Human Interface Devices (HID):

Core OS → Windows CE devices → Core OS Services → USB Host Support → USB Human Input Device (HID) Class Driver

For Printers:

Core OS → Windows CE devices → Core OS Services → USB Host Support → USB Printer Class Driver

For RNDIS:

Core OS → Windows CE devices → Core OS Services → USB Host Support → USB Remote NDIS Class Driver

For Mass Storage:

Core OS → Windows CE devices → Core OS Services → USB Host Support → USB (mass) Storage Class Driver

Note that sometimes additional Catalog Items have to be included in order to get the desired functionality (i.e. filesystem drivers, printer protocols, keyboard layouts etc...)

Software Operation

The driver implements the Windows USB Software Architecture. For more information, refer to ***Developing a Device Driver → Windows CE Drivers → USB Host Drivers*** and ***Developing a Device Driver → Windows CE Drivers → USB Host Drivers → USB Host Controller Drivers → USB Host Controller Driver Development Concepts*** in the Windows CE help.

6.9 USB OTG Driver

The OTG driver provides USB host and device support for the i.MX35 USB "On The Go" (OTG) port. It will automatically select host or device functionality depending on the USB cable configuration. The driver consists of three parts: the USB OTG host controller driver, the USB client driver and the USB transceiver controller driver (for host/client switching).

OTG Client

Driver Attribute	Definition
SOC static library	N/A
Import Library	N/A
Driver DLL	usbfn.dll
Required Catalog Items	Third Party → BSPs → iMX35Phytec → Device Drivers → USB Devices → USB High Speed OTG Device → USB High Speed OTG Port Full OTG Function
SYSGEN Dependency	SYSGEN_USBFN=1
BSP Environment Variable	BSP_USB=1 BSP_USB_HSOTG_CLIENT

When in client mode, there must be configured a function driver to use (mass storage, serial and rndis function drivers are supported by Windows CE). In order to use ActiveSync, the serial function driver must be selected.

OTG Host

Driver Attribute	Definition
SOC static library	N/A
Import Library	Usbh_usb2com_common_fsl_v2.lib Usbh_ehcdmdd_common_fsl_v2.lib Usbh_ehcdpdd_common_fsl_v2.lib
Driver DLL	hcd_hsotg.dll
Required Catalog Items	Third Party → BSPs → iMX35Phytec → Device Drivers → USB High Speed OTG Device → USB High Speed OTG Port Full OTG Function or Third Party → BSPs → iMX35Phytec → Device Drivers → USB High Speed OTG Device → USB High Speed OTG Pure Host Function
SYSGEN Dependency	SYSGEN_USBFN=1
BSP Environment Variable	BSP_USB=1 BSP_USB_HSOTG_HOST

The OTG host driver requires one or more class drivers to be loaded, see section 5.13.

OTG Transceiver

Driver Attribute	Definition
SOC static library	usb_xvc_mx35_fsl_v2.lib
Import Library	N/A
Driver DLL	imx_xvc.dll
Required Catalog Items	Third Party → BSPs → iMX35Phytec → Device Drivers → USB High Speed OTG Device → USB High Speed OTG Port Full OTG Function
SYSGEN Dependency	SYSGEN_USBFN=1
BSP Environment Variable	BSP_USB=1 BSP_USB_HSOTG_CLIENT BSP_USB_HSOTG_HOST BSP_USB_HSOTG_XVC

Software Operation

USB OTG Host Controller Driver

The USB OTG host controller driver is part of the Windows USB software standard. See section 5.13 for help on getting more information.

USB OTG Client Driver

The USB OTG client driver is part of the Windows USB software standard. For more information, refer to ***Developing a Device Driver → Windows CE Drivers → USB Function Drivers → USB Function Controller Drivers*** and ***Developing a Device Driver → Windows CE Drivers → USB Function Client Drivers*** in the Windows CE help.

6.10 I2C Driver

The I2C module can be used to communicate with one or more i2c devices, such as EEPROM, RTC or Camera devices.

Driver Attribute	Definition
SOC static library	I2C_common_fsl_v2.lib
Import Library	I2csdk.lib
Driver DLL	i2c.dll
Required Catalog Items	Third Party -> BSPs -> iMX35Phytec -> Device Drivers -> I2C Bus 1 / 3
SYSGEN Dependency	N/A
BSP Environment Variable	BSP_I2CBus=1

Software Operation

For successfully access to the driver, you have to include following Library and Header-File:

Include Library:

/WINCE600/PLATFORM/iMX35Phytec/SDKLibs/i2csdk.lib

Include Header:

/WINCE600/PLATFORM/COMMON/SRC/SOC/COMMON_FSL_V2/INC/i2cbus.h

The I2C driver is a stream-interface driver and thus can be accessed through the file system APIs. Therefore, a handle to the driver must first be created using the **I2COpenHandle** - function. After that, commands and data can be passed to or from the driver by using the functions explained below. After using the I2C bus, the device handle has to be closed using the **I2CCloseHandle** function.

Functions:

`BOOL I2CSetSlaveMode(HANDLE hI2C)`

Requires a handle to the I2C-device as parameter, sets the I2C device to slave mode.

`BOOL I2CSetMasterMode(HANDLE hI2C)`

Requires a handle to the I2C-device as parameter, sets the I2C device to slave mode.

`BOOL I2CIsMaster(HANDLE hI2C, PBOOL pbIsMaster)`

Requires a handle to the I2C-device as parameter.Determines, if the I2C device is currently in master mode.

Relevant Parameters:

hI2C

Handle to the I2C stream driver that you have created with the **`I2COpenHandle.`**

pbIsMaster

Pointer to a BOOL which will hold the return value: TRUE if device is in master mode, otherwise FALSE.

`BOOL I2CIsSlave(HANDLE hI2C, PBOOL pbIsSlave)`

Same as `I2CIsMaster`**,** but instead checking if the device is in slave mode.

Relevant Parameters:

hI2C

Handle to the I2C stream driver that you have created with the **`I2COpenHandle`**

pbIsSlave

Pointer to a BOOL which will hold the return value: TRUE if device is in slave mode, otherwise FALSE.

`BOOL I2CGetClockRate(HANDLE hI2C, PWORD pwClkRate)`

Retrieves the clock rate divisor. The actual clock rate is platform dependent.

Relevant Parameters:

hI2C

Handle to the I2C stream driver that you have created with the **`I2COpenHandle`**

`pwClkRate`

Pointer to the divisor index. Refer to the Table below for more information.

Table 28-7. IFDR Register Field Values

IC	Divider	IC	Divider	IC	Divider	IC	Divider
0x00	30	0x10	288	0x20	22	0x30	160
0x01	32	0x11	320	0x21	24	0x31	192
0x02	36	0x12	384	0x22	26	0x32	224
0x03	42	0x13	480	0x23	28	0x33	256
0x04	48	0x14	576	0x24	32	0x34	320
0x05	52	0x15	640	0x25	36	0x35	384
0x06	60	0x16	768	0x26	40	0x36	448
0x07	72	0x17	960	0x27	44	0x37	512
0x08	80	0x18	1152	0x28	48	0x38	640
0x09	88	0x19	1280	0x29	56	0x39	768
0x0A	104	0x1A	1536	0x2A	64	0x3A	896
0x0B	128	0x1B	1920	0x2B	72	0x3B	1024
0x0C	144	0x1C	2304	0x2C	80	0x3C	1280
0x0D	160	0x1D	2560	0x2D	96	0x3D	1536
0x0E	192	0x1E	3072	0x2E	112	0x3E	1792
0x0F	240	0x1F	3840	0x2F	128	0x3F	2048

`BOOL I2CSetClockRate(HANDLE hI2C, WORD wClkRate)`

This function will initialize the I2C device with the given clock rate. Note that this function does not expect to receive the absolute peripheral clock frequency. Rather, it will be expecting the clock rate divisor index stated in the I2C specification. If absolute clock frequency must be used, please use the function I2CSetFrequency() as described below.

Relevant Parameters:

hI2C

Handle to the I2C stream driver that you have created with the **`I2COpenHandle`**

wClkRate

Pointer to the divisor index. Refer to I2C specification for more information.

`BOOL I2CSetFrequency(HANDLE hI2C, DWORD dwFreq)`

Estimates the nearest clock rate acceptable for the device and initializes the device to use the estimated clock rate divisor.

Relevant Parameters:

hI2C

Handle to the I2C stream driver that you have created with the **`I2COpenHandle`**

dwFreq

Pointer to the desired I2C frequency.

`BOOL I2CSetSelfAddr(HANDLE hI2C, BYTE bySelfAddr)`

Initializes the device with the given address.

Relevant Parameters:

hI2C

Handle to the I2C stream driver that you have created with the **`I2COpenHandle`**

bySelfAddr

Pointer to the expected I2C device address (0x00 to 0x7F).

`BOOL I2CGetSelfAddr(HANDLE hI2C, PBYTE pbySelfAddr)`

Retrieves the address of the I2C device (only meaningful if device is in slave mode).

Relevant Parameters:

hI2C

Handle to the I2C stream driver that you have created with the **`I2COpenHandle`**

pbySelfAddr

Pointer to the current I2C device address (0x00 to 0x7F).

`BOOL I2CTransfer(HANDLE hI2C, PI2C_TRANSFER_BLOCK pI2CTransferBlock)`

Performs a sequential transfer (read or write) of one or more data-packets to or from a device.

Expects a **I2C_TRANSFER_BLOCK** structure containing an array of **I2C_PACKET** - structures to be transferred.

This array is performed sequentially. An I2C START command is issued before the first packet

transmission. Further, if the transfer direction or slave address changes between different packages of the array, an I2C REPEATED START command is automatically inserted. After transmission of the last packet in the array, an I2C STOP command is issued.
relevant Parameters:

hI2C

Handle to the I2C stream driver that you have created with the **`I2COpenHandle`**

pI2CTransferBlock

Pointer to an I2C_TRANSFER_BLOCK structure (explained below).

Structures

I2C_TRANSFER_BLOCK

```
typedef struct
{
        I2C_PACKET *pI2CPackets;
        INT32 iNumPackets;
} I2C_TRANSFER_BLOCK, *PI2C_TRANSFER_BLOCK;
```

I2C_PACKET

```
typedef struct
{
        BYTE byAddr;
        BYTE byRW;
        PBYTE pbyBuf;
        WORD wLen;
        LPINT lpiResult;
} I2C_PACKET, *PI2C_PACKET;
```

Members

BYTE byAddr:

I2C slave device address for the I2C operation.

BYTE byRW:

Signals, if the operation is an read or write transfer.

Read = I2C_READ Write = I2C_WRITE

PBYTE pbyBuf:

Pointer to a message buffer.

WORD wLen:

Length of the message buffer in Bytes.

LPINT lpiResult:

Contains the result of last operation. Informations about the error can be retrieved by a following call of GetLastError().

`BOOL I2CReset(HANDLE hI2C)`

Performs a hardware reset to the I2C-Bus.

6.11 Ethernet Driver

The Fast Ethernet Driver provides the ability to use a wide range of networking services on the device.

Driver Attribute	Definition
Import Library	ndis.lib
Driver DLL	fec.dll
Required Catalog Items	Third Party → BSPs → iMX35Phytec → Device Drivers → FEC → FEC
SYSGEN Dependency	SYSGEN_NDIS=1 SYSGEN_TCPIP=1 SYSGEN_WINSOC=1
BSP Environment Variable	BSP_ETHER_FEC=1

Software Operation

The FEC driver confirms to the NDIS 4.0 specification by Microsoft for the miniport network drivers.

6.12 CAN Controller Driver

The CAN – Driver provides the ability to connect the Evaluation Board to a CAN-Bus with various Bitrates and standard- or extended frame formats.

Driver Attribute	Definition
Import Library	Cansdk.lib
Driver DLL	Can.dll
Required Catalog Items	Third Party → BSPs → iMX35Phytec → Device Drivers → CAN Bus → CAN Bus 1 / 2
BSP Environment Variable	BSP_CANBUS1=1 BSP_CANBUS2=1

Software Operation

For successfully access to the driver, you have to include following Library and Header-File:

Include Library:

/WINCE600/PLATFORM/iMX35Phytec/SDKLibs/cansdk.lib

Include Header:

/WINCE600/PLATFORM/COMMON/SRC/SOC/COMMON_FSL_V2/INC/canbus.h

HANDLE CANOpenHandle(LPCWSTR lpDevName)

Creates a new instance of the CAN controller structure in the library and opens the port of the driver. This function must be executed before any other function is used. Its return value has to be passed as first parameter to all following functions.

Parameters:

lpDevName

Name of the CAN port that should be opened (L“CAN1:”).

Return values:

If successful, a thread-handle of the new created CAN device will be returned. This value is passed as first parameter to all other functions and must not be changed by the application.
If errors occur, INVALID_HANDLE_VALUE will be returned.

BOOL CANCloseHandle(HANDLE hCAN)

This function closes a valid open handle of the can bus device.

Parameters:

hCAN

Handle of the CAN device, that is created in the library. Will be returned via **CanOpenHandle**.

Return values:

If successful, the returned value is non-zero. If errors have occurred, zero will be returned. Informations about the error can be retrieved by a following call of GetLastError().

BOOL CANGetClockRate(HANDLE hCAN, PUINT32 pwClkRate)

Reads the baudrate of the selected CAN interface.

Parameters:

hCAN

Handle of the CAN device, that is created in the library. Will be returned via **CanOpenHandle**.

pwClkRate

Pointer of an integer variable. The driver assigns the current used Baudrate to this variable. The value is the Baudrate in Bytes.

Return values:

If successful, the returned value is non-zero. If errors have occurred, zero will be returned. Informations about the error can be retrieved by a following call of GetLastError().

BOOL CANSetClockRate(HANDLE hCAN, UINT32 iClkRate)

Sets a new baudrate in the controller.

Parameters:

hCAN

Handle of the CAN device, that is created in the library. Will be returned via CanOpenHandle.

iClkRate

A integer variable with the desired Baudrate in Baud.

Return values:

If successful, the returned value is non-zero. If errors have occurred, zero will be returned. Informations about the error can be retrieved by a following call of GetLastError().

BOOL CANSetMode(HANDLE hCAN,DWORD index,CAN_MODE mode)

Set the buffer interrupt for the desired buffer.

Parameters:

hCAN

Handle of the CAN device, that is created in the library. Will be returned via CanOpenHandle.

index

Index of the desired buffer (0 – 63).

mode

Unused parameter. Should be set to “0”.

Return values:

If successful, the returned value is non-zero. If errors have occurred, zero will be returned. Informations about the error can be retrieved by a following call of GetLastError().

`BOOL CANTransfer(HANDLE hCAN, PCAN_TRANSFER_BLOCK pCANTransferBlock)`

Transmits a message on the CAN-bus.

Parameters:

`hCAN`

Handle of the CAN device, that is created in the library. Will be returned via CanOpenHandle.

`pCANTransferBlock`

Pointer to a **`CAN_TRANSFER_BLOCK`**-structure (see explanation below).

Return values:

If successful, the returned value is non-zero. If errors have occurred, zero will be returned. Informations about the error can be retrieved by a following call of GetLastError().

Structures

```
typedef enum _CAN_FRAME_FORMAT {
    CAN_STANDARD=0,
    CAN_EXTENDED,
} CAN_FRAME_FORMAT;

typedef enum _CAN_RTR_FORMAT {
    CAN_DATA=0,
    CAN_REMOTE,
} CAN_RTR_FORMAT;


typedef struct
{
    BYTE                byIndex;
    BYTE                byRW;
    CAN_FRAME_FORMAT    fromat;
    CAN_RTR_FORMAT      frame;
    WORD                timestamp;
    BYTE                PRIO;
    DWORD               ID;
    PBYTE               pbyBuf;
    WORD                wLen;
    LPINT               lpiResult;
} CAN_PACKET, *PCAN_PACKET;
```

BYTE byIndex:

CAN Bus Message Buffer index (0 – 63) for RX or TX messages.

BYTE byRW:

```
#define CAN_RW_WRITE    0
#define CAN_RW_READ     1
```

For a write access you have to set this Byte to CAN_RW_WRITE.
Otherwise to CAN_RW_READ.

CAN_FRAME_FORMAT fromat:

For the current transmit, it is necessary to determine the frame format.

CAN_STANARD	→	for standard format
CAN_EXTENDED	→	for extended format.

CAN_RTR_FORMAT frame:

Decides if the current frame is an data or remote request frame.

CAN_DATA	→	for data transmitting
CAN_REMOTE	→	for a remote request

WORD timestamp:

Not used.

BYTE PRIO:

Not used.

DWORD ID:

Determines which ID has the current transmit or receive buffer.

PBYTE pbyBuf:

Pointer to a message Buffer data with maximal length of 8 Bytes.

WORD wLen:

Mesage buffer length in Bytes.

LPINT lpiResult:

Contains the result of the last operation.

Define	Value	Description
CAN_NO_ERROR	0	Last operation successful
CAN_ERR_MOPS_CREATE	-1	Mutex Creation failed
CAN_ERR_PA_VA_MISSING	-2	Physical –> Virtual Mapping failed
CAN_ERR_EOPS_CREATE	-3	Event Creation failed
CAN_ERR_IRQ_SYSINTR_MISSING	-4	IRQ -> System Interrupd ID Mapping failed
CAN_ERR_INT_INIT	-5	Interrupt Initialization failed
CAN_ERR_WORKER_THREAD	-6	Worker Thread failed
CAN_ERR_NO_ACK_ISSUED	-7	No Acknowledge Issued
CAN_ERR_NULL_BUF	-8	Buffer is NULL
CAN_ERR_INVALID_BUFSIZE	-9	Invalid Buffer Size
CAN_ERR_NULL_LPIRESULT	-10	lpiResult field is NULL
CAN_ERR_CLOCK_FAILURE	-11	Failed to set the clock
CAN_ERR_TRANSFER_TIMEOUT	-12	Transfer timeout
CAN_ERR_ARBITRATION_LOST	-13	Arbitration lost error
CAN_ERR_TRANSFER_ERR	-14	Undefined transfer error
CAN_ERR_BIT0	-15	One Bit is sent as dominant is received as recessive
CAN_ERR_BIT1	-16	One Bit is sent as recessive is received as dominant
CAN_ERR_ACK	-17	Acknowledge error by the transmitter
CAN_ERR_CRC	-18	CRC Error detected by receiver
CAN_ERR_FRM	-19	Receiver detected a form error
CAN_ERR_STF	-20	Stuffing Error has been detected

```
typedef struct
{
    CAN_PACKET *pCANPackets;
    INT32 iNumPackets;
} CAN_TRANSFER_BLOCK, *PCAN_TRANSFER_BLOCK;
```

CAN_PACKET *pCANPackets:

A pointer to the CAN_PACKET-structure explained above.

INT32 iNumPackets:

Determine how many packets will be transmitted.

6.13 EEPROM Driver

The EEPROM – Driver can be used to store small amounts of data (up to 4096 Byte) in a persistent way using the 24WC32 EEPROM on the Module.

Driver Attribute	Definition
Import Library	N/A
Driver DLL	24wc32.dll
Required Catalog Items	Third Party → BSPs → iMX35Phytec → Device Drivers → EEPROM → 24WC32 EEPROM Driver
BSP Environment Variable	BSP_EEPROM_24WC32=1 BSP_I2CBUS1=1

Software Operation

The EEPROM driver implements the Stream Interface. To use the driver, a handle to the device must be created using the **CreateFile** function with the device name 'EEP1'. All other commands to the device can be issued using the **DeviceIoControl** function with the IOCTLs mentioned below:

IOCTL_READ

Description:

reads a specified amount of data from the EEPROM to a buffer.

Parameters:

pBufIn [IN]

pointer to an EEPROM_TRANSFER structure containing the EEPROM internal address, the length of the buffer and a pointer to the buffer (see below)

pBufOut [OUT]

pointer to an unsigned int, will be set to the amount of Bytes which were actually read

Return Value:
BOOL TRUE if read operation succeeded, else FALSE

IOCTL_WRITE
Description:
writes a specified amount of data from a buffer to EEPROM.

Parameters:
pBufIn [IN]
pointer to an EEPROM_TRANSFER structure containing the EEPROM internal address, the length of the buffer and a pointer to the buffer (see below).

Return Value:
BOOL TRUE if write operation succeeded, else FALSE

Structures

The EEPROM driver defines the following structure to describe a read or write operation:

```
typedef struct
{
        UINT32 Addr; //EEPROM internal address
        UINT8* pBuffer; //pointer to data buffer
        UINT32 Length; //amount of Bytes to read/write
}EEPROM_TRANSFER;
```