CONTENTS

CONTENTS 1
1 INTRODUCTION 2
1.1 About This Documentation 2
1.2 Prerequisites 2
1.3 Distribution 2
1.3.1 “doc” Directory 2
1.3.2 “driver” Directory 3
1.3.3 “bin” directory 3
1.3.4 “inc” Directory 3
1.3.5 “lib” Directory 4
1.3.6 “src” Directory 4
1.3.7 “fpga” Directory 5
1.4 Minimum Requirements 5
1.5 Benchmarking 5
1.6 CeUsb2 Overview 6
2 CEUSB2 COMPONENTS AND FEATURES 7
2.1 Hardware 7
2.1.1 USB Bus 8
2.1.2 Cypress FX2 USB 2.0 controller chip 10
2.1.3 Serial Interface Engine (SIE) 10
2.1.4 8051 Microcontroller 11
2.1.5 Serial Input Output Interface (SIO) 11
2.1.6 Serial EEPROM 12
2.1.7 IO Ports Interface 13
2.1.8 General Purpose Interface (GPIF) 15
2.1.9 XILINX FPGA 17
2.1.10 External RAM 1 22
2.1.11 External RAM 2 23
2.2 Firmware 25
2.3 Loading process 26
2.4 CeUsb2 Configuration Program and Registry 27
2.5 Drivers 29
2.6 Application Programming Interface (CeUsb2 API) 31
2.7 Graphical User Interface (CeUsb2 GUI) 31
2.8 CeUsb2 Diagnostic Program 32

1 INTRODUCTION

1.1 About This Documentation

This documentation is a general description of CeUsb2 Software Development Kit (SDK) with explanation of its drivers, firmware applications, interfaces, APIs, GUIs, diagnostic programs and some other components. CeUsb2 hardware is also discussed in sight of software. You can find more detailed documents for each component in the distribution, also with setup information.

1.2 Prerequisites

CeUsb2 type products are USB 2.0 compatible devices; therefore at least a basic knowledge of USB protocol is necessary to understand this document and SDK's functionality.

CeUsb2 firmwares are designed using ANSI C. This document references some firmware code, therefore an intermediate or upper level knowledge of C programming language is necessary.

Some CeUsb2 type of devices has an on-board FPGA chip. This document references some sample hardware designs for the FPGA written in VHDL hardware description language. Therefore, an intermediate or upper level of VHDL language is also required.

1.3 Distribution

This section will provide a detailed description and explanation of the structure and content of the CeUsb2 Software distribution as it exists on the users PC after the installation from the CeUsb2 CD. For installation instructions see the CeUsb2 Installation Guide, Install.pdf, located within the CeUsb2 Development Kit.

1.3.1 "doc" Directory

Contains support files for the documentation of the CeUsb2 Development Kit. These files are:

Install.pdf: CeUsb2 Installation documentation.
CeUsb2dg.pdf (This document): General CeUsb2 design guide.
CeUsb2drv.pdf: CeUsb2 driver specifications documentation.
CeUsb2ld.pdf: CeUsb2 loader driver specifications documentation.

CeUsb2gfw.pdf: CeUsb2 generic firmware specifications documentation.
CeUsb2api.pdf: CeUsb2 Application Programming Interface (API) documentation.
CeUsb2gui.pdf: CeUsb2 Graphical User Interface (GUI) documentation.
CeUsb2com.pdf: CeUsb2 COM API documentation
CeUsb2diag.pdf: CeUsb2 diagnostic program (CeUsb2Diag.exe) documentation.

1.3.2 “driver” Directory

This directory contains driver executable files, firmware applications and other necessary files for installing CeUsb2 drivers.

CeUsb2.inf: Driver installation information file.
CeUsb2Ld.sys: CeUsb2 loader driver executable.
CeUsb2.sys: CeUsb2 main driver executable.
CeU2Ld.hex: Loader firmware application file.
CeU2g001.hex: CeUsb2 generic firmware application file.
CeUsb2pp.dll: This dynamic link library is used with the driver installation and puts an additional property page to the property sheet of the CeUsb2, which is displayed in the Device Manager applet of the system (Will be available in the next version).

1.3.3 “bin” directory

This directory contains utility & diagnostics programs and dynamic link libraries (DLLs) utilized within the CeUsb2 Development Kit. Note that copies of these executables may exist in other directories within distribution.

CeUsb2Cfg.exe: Configuration program for CeUsb2 devices, drivers and firmware applications.
CeUsb2Diag.exe: Diagnostic program for CeUsb2 devices.
CeError.exe: Helper error detection utility.
CeUsb2Api.dll: CeUsb2 API executable.
CeUsb2Mfc.dll: CeUsb2 GUI executable.
GPIF.exe: A utility program from Cypress which allows the user to generate GPIF programs for the FX2 integrated circuit. Please check Cypress website for more information (http://www.cypress.com).

1.3.4 “inc” Directory

Contains C/C++ header files (*.h). Customer projects which will use the CeUsb2 IOCTL interface, API or GUI should include these files with the required libraries found in the lib directory.

CeError.h: Contains status and error code definitions of the drivers and APIs.
CeUsb2If.h: Interface definition header file for CeUsb2 driver.
CeUsb2LdIf.h: Interface definition file for CeUsb2 loader driver.
Guids.h: Guid definitions for CeUsb2 programming interface.
Usbd.h: USB definitions header file.
UsbDef.h : USBD definitions header file.
Version.h : Contains constant definitions for API versioning.
CeFirmware.h: CeUsb2 generic firmware interface header file.
CeUsb2Api.h: CeUsb2 API main header file.
CeUsb2Wr.h : CeUsb2 GUI main header file.
CeVector.h : Contains template list & vector classes.
CeVector.cpp : Implementations of utility list & vector classes.
CeUsb2ComApi.idl: CeUsb2 COM API interface definition file.

1.3.5 "lib" Directory

Contains libraries for source compilation and re-build process. Projects in the src directory, which are distributed with source codes share their libraries (*.lib), dynamic link libraries (*.dll) and executable files (*.sys, *.exe) in this directory.

i - Subdirectory lib\chk\i386
CeUsb2Api.lib : Debug version of CeUsb2 API library file.
CeUsb2Mfc.lib : Debug version of CeUsb2 GUI library file.
CeUsb2Api.dll : Debug version of CeUsb2 API executable (dynamic link library).
CeUsb2Mfc.dll : Debug version of CeUsb2 GUI library file (dynamic link library).
CeUsb2ComApi.dll: Debug version of CeUsb2 COM API executable (dynamic link library).

ii - Subdirectory lib\fre\i386
CeUsb2Api.lib : Release version of CeUsb2 API library file.
CeUsb2Mfc.lib : Release version of CeUsb2 GUI library file.
CeUsb2Api.dll : Release version of CeUsb2 API executable (dynamic link library).
CeUsb2Mfc.dll : Release version of CeUsb2 GUI library file (dynamic link library).

1.3.6 "src" Directory

This directory includes source distribution of CeUsb2 SDK.

GPIFTest : GPIF test program written with MFC. It is used with gpif_test sample FPGA design.

CeUsb2ComApi : Includes CeUsb2 COM API type library file, CeUsb2ComApi.tlb.

CeUsb2ComApiTest: Includes CeUsb2 COM API test projects written in different programming languages. It's subdirectories are

- MfcTest : Visual C++ 6.0 test project with MFC.
- VbTest : Visual Basic.NET test project.
- CSharpTest : Visual C# test project.
- Delphi : Borland Delphi 7.0 test

1.3.7 "fpga" Directory

Contains sample FPGA designs (currently only in VHDL).

LedBlink : Simple VHDL design which blinks the LEDs on the board.
mem_test: External memory test design in VHDL.
gpif_test: GPIF test design in VHDL.

1.4 Minimum Requirements

CeUsb2 software minimum requirements are:

- Windows 2000 or Windows XP operating systems.
- Pentium 266 MHz CPU
- 16 Mbytes of RAM
- 10 Mbytes of hard disk space

Since CeUsb2 uses the enhanced data transfer features of USB2.0 protocol, it might be better to use a faster CPU and larger RAM.

1.5 Benchmarking

Maximum data transfer rates of CeUsb2 devices:

Bulk-write transfer: Not available yet.
Bulk-read transfer: Not available yet.
Isochronous-read transfer: Not available yet.
Isochronous-write transfer: Not available yet.
Control read-transfer: Not available yet.
Control write-transfer: Not available yet.

1.6 CeUsb2 Overview

Figure 1.1 shows a simple diagram of the CeUsb2 environment. Hardware part displays the CeUsb2 board with its three basic components: firmware, FPGA and all other possible on-board and external components. Firmware has the ability to communicate with the external components and the FPGA. Moreover, it is the interface between the hardware and the kernel mode drivers of CeUsb2.

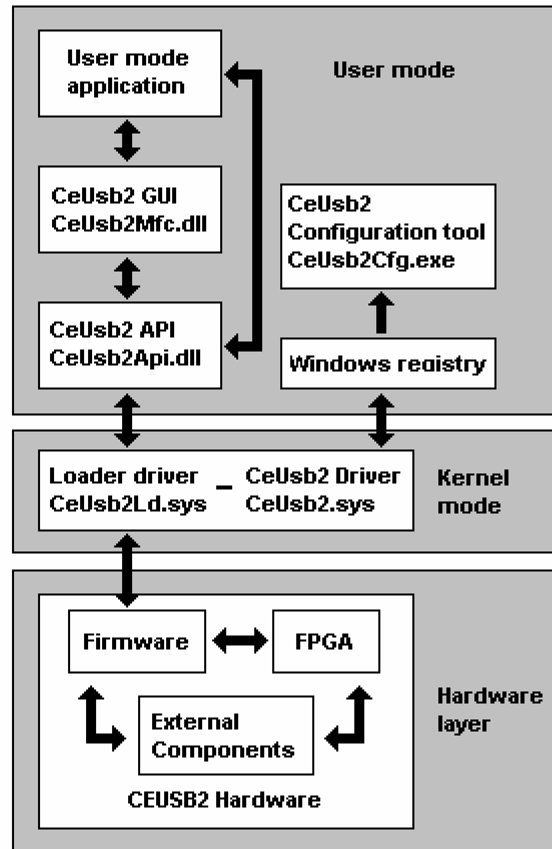


Figure 1.1 – CeUsb2 SDK diagram.

There are more than one CeUsb2 (or CeUsb2 compatible) boards on the market. All of them use the same drivers, API and GUI dynamic link libraries (DLL), configuration program and Windows registry entries.

Configuration program configures the system through the registry so that CeUsb2 software can distinguish between different types of CeUsb2 devices. For more information see 2.4, Configuration Program and CeUsb2 Registry.

During installation of the drivers, registry entries are written and configured in a way that a generic CeUsb2 board (CeUsb2 Evaluation board) can be recognized. Later, these registry entries can be changed by some user mode programs and configuration program. CeUsb2 drivers retrieve the information about a CeUsb2 device by reading these values. After the drivers and firmware executables are

loaded (for loading process see chapter 2.3, Loading Process), CeUsb2 kernel mode software is ready to communicate with user mode APIs and programs.

CeUsb2 driver (CeUsb2.sys) gives an IOCTL (input output control) interface to the user mode applications. The user is free to use this interface to control CeUsb2 hardware. However, CeUsb2 API (CeUsb2Api.dll) simplifies the usage of the IOCTL interface and serves some upper level C++ classes for device input output and controlling of the external components. User mode applications can communicate directly with the API DLL, instantiate member classes and call exported functions from it. Another way to access CeUsb2 hardware from the user mode is to use CeUsb2 GUI (CeUsb2Mfc.dll). This module serves some dialog functions that the user can add to his own project.

2 CEUSB2 COMPONENTS AND FEATURES

2.1 Hardware

This section gives a brief explanation of CeUsb2 hardware in sight of software.

Each CeUsb2 board may have some different features than the others; therefore, you should check your board's hardware document to get more detailed information about its specific hardware features. The most important things that you should know about your hardware before you start your design process are:

- Check if your board has an integrated FPGA chip or another programmable digital component such as CPLD, or ASIC.
- If your board has an FPGA, check if it is configured automatically by a serial PROM. Otherwise, you should learn from firmware specifications if how your FPGA is configured.
- All CeUsb2 boards use FX2 USB controller chip from Cypress Semiconductors. This chip is shipped as 56, 100 and 128 pin packages. Each package has slightly different features. For example, 56 pin packages don't have serial input output interface. Therefore, it is useful to know which of these chips is used on your board.
- Get some information (type, timing diagram, control interface etc.) about the RAMs or other memory storage elements your board has.
- Check if how your board uses I/O pins of the FX2.

Figure 2.1 is the simple hardware diagram of a typical CeUsb2 board. Throughout this section all components compromising CeUsb2 hardware will be explained more detailed.

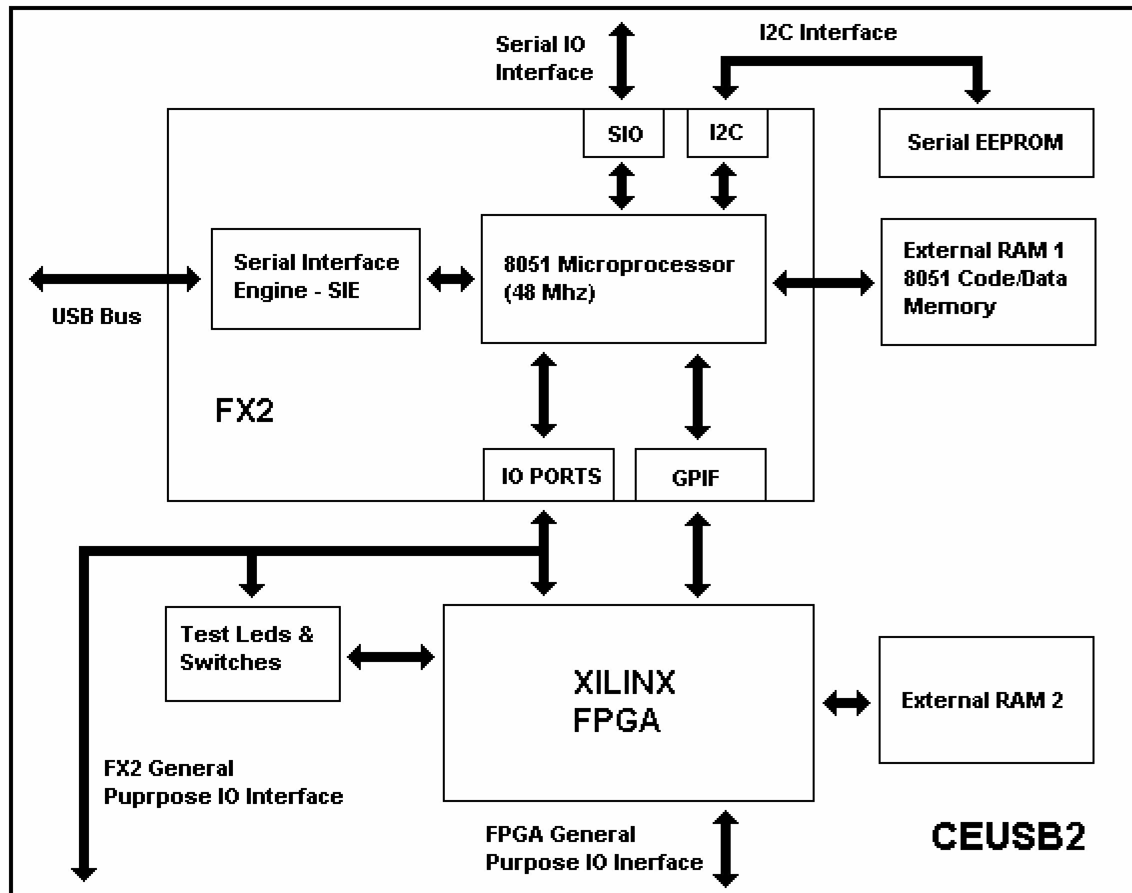


Figure 2.1 – CeUsb2 hardware diagram.

2.1.1 USB Bus

Although CeUsb2 is a USB 2.0 board, it can be connected to an USB 1.1 port. If connected to the USB 2.0 port, it operates in high-speed mode enabling 480 MBits /second data rate. If the firmware running on 8051 processor handles protocol differences, CeUsb2 can be also connected to an USB 1.1 port enabling full-speed mode with 12 MBits/second data rate.

USB Bus enables the host to communicate with the CeUsb2 devices by means of:

- Control transfers
- Bulk transfers
- Isochronous transfers
- Vendor & class requests (internally control transfer is used)

through USB endpoints. CeUsb2 has a default endpoint 0 and additional 5 endpoints.

USB endpoints reserves buffers in the upper 7.5 Kbytes of the internal memory for data transfer. FX2 endpoint buffers appear to have different sizes depending on whether the FX2 is operating at full or high speed. This is due to the difference in maximum packet sizes allowed by the USB specification for the two modes. An USB endpoint can be configured as IN or OUT endpoint but not both (except EP0, it has separate buffers for IN and OUT transfers).

Endpoints 2, 4, 6 and 8 have buffering features which refers to the number of RAM blocks available to the endpoint. With double buffering, for example, USB data can fill or empty an endpoint buffer at the same time that another packet from the same endpoint fills or empties from the external logic. This technique maximizes performance by saving each side, USB and external-logic interface, from waiting for the other side. FX2 CPU can only access the active buffer of a multiple-buffered endpoint. In other words, firmware must treat a double-buffered 512-byte endpoint as being only 512 bytes wide, even though the double-buffered endpoint actually occupies 1024 bytes of RAM.

Endpoint 0 is a control endpoint and required for all USB devices. It is special since it can perform vendor and class requests. Endpoint 0 has two separate buffers for IN and OUT transfers with maximum package size, 64.

EP1 has a small buffer of 64 bytes and can be configured for only bulk transfers. It doesn't support buffering.

EP4 and EP8 are identical double buffered endpoints. They support both bulk and isochronous transfers with a fixed 512 byte buffer on USB2.0. On USB 1.1 during bulk transfers only 64 bytes of these buffers are usable.

EP2 and EP6 are most powerful and configurable endpoints for CeUsb2. They can be configured for single, double, triple and quad buffering. Their buffers can grow up to 1024 bytes on USB2.0.

Table 2.1 summarizes the features of CeUs2 endpoints.

Endpoint Number	Transfer Type	USB 1.1 max. Buffer size (bytes)	USB 2.0 max. Buffer Size (bytes)	Supported Buffering
0	Control	8,16,32,64	64	Single
1	Bulk	64	64	Single
2	Bulk, Isochronous	64 for bulk, 1023 for isochronous	1-1024	Single, Double, Triple, Quad
4	Bulk, Isochronous	64 for bulk, 512 for isochronous	512	Double
6	Bulk, Isochronous	64 for bulk, 1023 for isochronous	1-1024	Single, Double, Triple, Quad
8	Bulk, Isochronous	64 for bulk, 512 for isochronous	512	Double

Table 2.1 - CeUsb2 endpoints.

Another standard USB communication type, Interrupt transfer, is not supported by CeUsb2.

If an endpoint is configured by the firmware, an USB pipe is reserved for it. Host communicates with the target device by means of this pipe. You should check the documentation of your firmware, to see which endpoints or pipes are implemented and the features they support.

2.1.2 Cypress FX2 USB 2.0 controller chip

All CeUsb2 devices have a type of FX2 USB chip from Cypress semiconductor to handle USB communications and data transfers. FX2 combines an USB serial engine with an integrated 8051 compatible microprocessor (48 MHz) and some other peripherals such as GPIF, IO PORTS, I2C and serial interfaces (RS232). It can also communicate with an on-board or external FPGA chip through GPIF and IO PORTS.

FX2 chips are shipped in three different packages: 56, 100, 128 pin packages. Different CeUsb2 devices may have different type of FX2s. Also some packages may have different features than the others.

For more information about FX2 check Cypress website (http://www.cypress.com).

2.1.3 Serial Interface Engine (SIE)

FX2 USB chip has a Serial Interface Engine (SIE) which connects to the USB data lines and delivers data to and from the USB device. It decodes the packet PIDs, performs error checking on the data using the transmitted CRC bits, and delivers payload data to the USB device.

One of the most important features of the FX2 family is that its configuration is soft over serial interface. It doesn’t require ROM or other fixed memory, but it contains internal code & data RAM which can be loaded and cleared over the USB. This makes modifications, and updates easier and faster.

The FX2’s SIE can also perform full enumeration by itself, which allows the FX2 to connect as a USB device and download code into its RAM while its CPU is in reset state. This SIE functionality is also accessible from the software, to make development easier and save code and processing time. During enumeration

serial interface engine is responsible for detecting full-speed or high speed USB bus, and handling protocol differences.

2.1.4 8051 Microcontroller

FX2 USB chip has an integrated and enhanced 8051 microprocessor, which runs the firmware code. It operates at 12/24/48 MHz speeds and with 4 processor clock cycles per instruction.

The FX2 uses the standard 8051 instruction set, so it's supported by industry-standard 8051 compilers and assemblers. Instructions execute faster on the FX2 than on the standard 8051.

Like the standard 8051, the FX2 contains 128 bytes of Internal Data RAM at addresses 0x00-0x7F and a partially populated SFR space at addresses 0x80-0xFF. An additional 128 indirectly addressed bytes of Internal Data RAM (sometimes called "IDATA") are also available at addresses 0x80-0xFF.

All other on-chip FX2 RAM (program/data memory, endpoint buffer memory, and the FX2 control registers) is addressed as though it were off-chip 8051 memory. FX2 can also use an external off chip memory (maximum 64KByte RAM) as external code and data storage (see chapter 2.1.10, External RAM 1).

The role of the CPU in a typical FX2-based USB peripheral is:

- It implements the high-level USB protocol by servicing host requests over the control endpoint 0.
- Configures the interface for high-bandwidth data transfers. On CeUsb2 these kinds of transfers (bulk, isochronous) is mostly handled by the internal master General Purpose Interface (GPIF).
- Control and monitor GPIF activity.
- Handle all application-specific tasks using its serial IO interface SIO, counter-timers, interrupts, I/O pins, etc.

Some operation modes and speed of the FX2 CPU may configurable from the user mode through the firmware. Check the documentation of your firmware for more information.

2.1.5 Serial Input Output Interface (SIO)

The FX2 provides two serial ports which are almost identical with the 8051 serial ports (only 100 and 128 pin packages). Each serial port can operate in synchronous or asynchronous mode. In synchronous mode, the FX2 generates the serial clock and the serial port operates in half-duplex mode. In asynchronous mode, the serial port operates in full-duplex mode. In all modes, FX2 buffers the

incoming data (double buffering) so that a byte of incoming data can be received while firmware is reading the previously received byte.

A CeUsb2 firmware application may or may not implement serial interface for the user mode APIs and programs. If serial interface is implemented it can be used for debug or diagnostic output and it can control an external peripheral or perform data transfer with an external device.

2.1.6 Serial EEPROM

CeUsb2 devices stores their vendor, product and Cesys specific device Id in the on board serial EEPROM. This EEPROM is accessed by the CPU through I2C protocol, and only the first 8 bytes are used for board configuration.

Table 2.2 displays the contents of CeUsb2 EEPROM.

EEPROM Address	Value	Description
0	0xC0	Always 0xC0 for a configured CeUb2 device.
1	0xF8	Lower byte of Cesys USB vendor id (0x10F8).
2	0x10	Upper byte of Cesys USB vendor id (0x10F8).
3	0x00 – 0x01	0x00 for loader driver, 0x01 for CeUsb2 real driver.
4	0xC2	Always 0xC2 for a configured CeUsb2.
5	0xXX	Lower byte of the Cesys specific device identifier.
6	0xXX	Upper byte of the Cesys specific device identifier.
7	0x00	FX2 specific configuration byte, Always 0x00.

Table 2.2 – CeUsb2 EEPROM Contents.

The first byte of the EEPROM is always 0xC0 for a configured CeUsb2 device. If it is not all other ids are not read and the device is put in an unconfigured state staying with the loader driver in the system.

Cesys GmbH has its own USB vendor Id (0x10F8), which is written in the 1st and 2nd bytes of the EEPROM.

The 3rd and 4th bytes of the EEPROM are the USB product id. CeUsb2 devices use slightly different logic for product id from other USB devices. If product id is 0xC200, then the CeUsb2 is configured and the current firmware should load the loader driver (CeUsb2Ld.sys) in the system. Else if the product id is 0xC201, CeUsb2 is configured and the current firmware should load the real driver (CeUsb2.sys) in the system. Notice that after the loader firmware starts running

on the board, it will change the product id from 0xC200 to 0xC201 which will trigger the loading process of the real driver. For more information check section 2.3, Loading Process. All other values of product ids describe an unconfigured CeUsb2 device.

The 5th and 6th bytes of the EEPROM are the Cesys specific device identifier, which characterizes a different board produced by Cesys and supports CeUsb2 hardware and software architecture. Loader driver reads this value from the EERPOM and using this value it searches the registry for a matching firmware. If it finds one it loads and runs this firmware. If it can not find a matching firmware the loading process is finished with error.

2.1.7 IO Ports Interface

The 56-pin FX2 package provides input-output systems as:

- A set of programmable I/O pins (described in this section).
- A programmable I2C-compatible bus controller (CeUsb2 uses I2C bus only to access the EEPROM, see chapter 2.1.6 Serial EEPROM).
- Two serial input output ports (see chapter 2.1.5 Serial Input Output Interface (SIO))

The I/O pins may be configured either for general-purpose I/O or for alternate functions (GPIF address and data; FIFO data; USART, timer, and interrupt signals; etc.). When they are used as general purpose I/0 pins they can control an on-board or external component, or can be connected to an FPGA device. FPGA design together with the firmware defines an I/O ports interface.

The FX2's I/O ports are implemented differently than those of a standard 8051. The FX2 has up to five eight-pin bidirectional I/O ports, labeled A, B, C, D, and E. Individual I/O pins are labeled P*x.n*, where *x* is the port (A, B, C, D, or E) and *n* is the pin number (0 to 7). The 100- and 128-pin FX2 packages provide all five ports; the 56-pin package provides only ports A, B, and D.

Following codes are examples which display the usage of a single IO port. Code example 2.1 is firmware code written in C, which implements a simple FPGA reset process and Code example 2.2 is VHDL code snippet of an FPGA design which uses the reset signal to initialize its variables.

```
// Code example 2.1
// 8051 firmware C code with FPGA reset process

// For this CeUsb2 device the first pin of port E (PE.0) is the reset pin that is connected to the FPGA

// Helper macros
#define RESET_HIGH()                IOE |= 0x00      // switches PE.0 HIGH
#define RESET_LOW()                 IOE &= 0xFE      // switches PE.0 LOW
```

```
// imaginary firmware initialization function
void Init(void)
{
	//…
	PORTECFG |= 0x00; // PE.0 doesn't use alternate function ( 1 is alternate function)
	OEE |= 0x01; // PE.0 is OUTOUT (1 is output)
	// …
}

void FPGA_ResetConfig( )
{
	RESET_HIGH(); // reset signal high
	DelayUs(10);       // delay function (in micro seconds)
	RESET_LOW();  // reset signal low
}


-- Code example 2.2
-- FPGA VHDL code which uses the reset pin

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

--
-- counter entity
-- Implements a 8 bit counter which changes its value with rising
-- edge of the clock (SCLK). RESET signal resets the counter value to 0.
--
entity counter is
	port (
	SCLK : in std_logic; -- system clock
	RESET: in std_logic  -- reset signal  controlled by FPGA_ResetConfig()
	-- function of the firmware.

	--
	-- some more signal definitions go here
	--
	);
end counter;

architecture bhv of counter is

signal count_val : std_logic_vector(7 downto 0); -- counter signal

begin

-- counter process
process(RESET,SCLK)
begin
	if(RESET = '1') then
		count_val <= "00000000"; -- RESET resets the counter value
	elsif rising_edge(SCLK) then -- rising edge of the clock is detected
		count_val <= count_val + "00000001"; -- counter is incremented
	end if;
end process;
```

```
end bhv;
```

2.1.8 General Purpose Interface (GPIF)

Some CeUsb2 devices may use the FX2's CPU to process USB data directly, and handle data transfers by their own logic (with address, data, read, write, chip select, interrupt and I/O pins). However, most will use the FX2 simply as a conduit between the USB and external data-processing logic (e.g., an FPGA or DSP).

CeUsb2 enables external data processing logic for data transfers. USB data flows between host and the external logic through the endpoint FIFOs without any participation of the FX2 CPU. To the external logic, these endpoint FIFOs look like most others; they provide the usual timing signals, handshake lines (full, empty, programmable-level), read and write strobes, output enable, etc.

Although these FIFOs can be controlled by an external logic, CeUsb2 uses General Purpose Interface (GPIF) as an internal master for the FIFOs. Therefore external logic isn't supposed to define a FIFO interface.

GPIF allows the FX2 to connect directly to external peripherals such as ASICs, FPGAs, DSPs, or other digital logic that uses an 8- or 16-bit parallel interface. It provides external pins that can operate as outputs (CTL[5:0]), inputs (RDY[5:0]), Data bus (FD[15:0]), and Address Lines (GPIFADR[8:0]).

Data transfer through CeUsb2 device starts with a request from a user mode program. Firmware may reply this request immediately or performs data transfer with an external component before it replies the user. As stated before firmwares are free to use their own logic for data transfer, however GPIF is the most powerful and fastest way for communicate with the external world for CeUsb2. Following is 4 different data transfer processes through GPIF:

1 – Reading data with GPIF single read transitions:

a. User mode application sends a GPIF single read vendor request to the firmware.
b. Firmware checks the amount of data requested with the starting address (offset) to read. Single GPIF read transactions use address lines.
c. With the address and data count value it triggers the GPIF to read data.
d. GPIF sets the address line (GPIF_Adr), toggles its single read signal. This signal toggling depends on the GPIF waveform programmed.
e. External component connected to the GPIF replies this read request.
f. After a while GPIF returns with requested amount of data.

2 – Writing data with GPIF single write transitions:

a. User mode application sends a GPIF single write vendor request to the firmware.
b. Firmware checks the amount of data sent with the starting address (offset) to write. Single GPIF write transactions use address lines.
c. With the address and data count value it triggers the GPIF to write data.
d. GPIF sets the address line (GPIF_Adr), puts data on the bus (GPIF_Data) and toggles its write signal with. This signal toggling depends on the GPIF waveform programmed.
e. External component connected to the GPIF reads the data.
f. After a while GPIF returns.

3 – Reading data with GPIF FIFO read transitions:

a. User mode application sends a GPIF FIFO read request to the firmware with bulk or isochronous data transfer request.
b. Firmware should have the ability to start GPIF for FIFO read, starting the transition, checking the ready signals if available, checking the amount of data retrieved, ending a request either with error or with some data and so on. If so, firmware reads data from the external component if the ready signal(s) requirement is met and GPIF internal FIFOs are not full, regardless of the user request.
   For doing this, GPIF toggles FIFO read signal. Toggling depends on the GPIF waveform programmed. Notice that FIFO operations do not use addressing.
c. External component connected to the GPIF replies read requests and puts data on the data bus sequentially. GPIF also takes this data and puts it into its internal FIFOs.
d. Firmware replies to the user mode request if there is data to be sent in the GPIF FIFOs. Some firmwares have the ability to skip a request and return with 0 byte or less than the required amount of data.

4 – Writing data with GPIF FIFO write transitions:

a. User mode application sends a GPIF FIFO write request to the firmware with bulk or isochronous data transfer request.
b. Firmware should have the ability to start GPIF for FIFO write, starting the transition, checking the ready signals if available, checking the amount of data retrieved, ending a request either with error or success and so on. If so, firmware sends data to the external component if the ready signal(s) requirement is met and GPIF internal FIFOs are not empty, regardless of the user request.
   For doing this GPIF toggles FIFO write signal. Whit every toggle it puts the next data byte on the bus. Toggling depends on the GPIF waveform programmed. Notice that FIFO operations do not use addressing.
c. External component connected to the GPIF replies write requests and gets the data on the data bus sequentially.
d. Firmware replies to the user mode request with success or error.

A waveform descriptor in internal RAM describes the behavior of each of the GPIF signals. The waveform descriptor is loaded into the GPIF registers by the FX2 firmware during initialization, and it is then used throughout the execution of the code to perform transactions over the GPIF interface. CeUsb2 software enables loading another user supplied waveform descriptor, (see CeUsb2 API or CeUsb2 generic firmware interface documentation) if this feature is implemented in the firmware.

Waveform descriptor can be prepared with GPIF.exe program. It is a utility program from Cypress which allows the user to generate GPIF programs for the FX2 integrated circuit. Please check Cypress website for more information (http://www.cypress.com).

2.1.9 XILINX FPGA

Most of the CeUsb2-type boards have an on-board XILINX FPGA (field programmable gate array) chip (for more information about XILINX FPGAs see http://www.xilinx.com). Moreover some of these boards are FPGA prototyping or evaluation boards which are aimed to give developers a quick way of FPGA programming. A more complex or complete digital system can be also developed whit these evaluation boards.

Following is the two main usages of the FPGAs with CeUsb2 boards:

1.) FPGA is a processing unit. It can work like a second CPU with FX2 and External RAM 2.

2.) FPGA is used for communicating with the external components. It can control the external components and perform data transfer with them.

Before FPGA is used, it must be configured with a byte stream. This byte stream is extracted from an FPGA configuration file and sent to the firmware by the host. CeUsb2 firmwares use two different logics for FPGA configuration: ENACNF controlled and uncontrolled. If ENACNF uncontrolled logic is used, ENACNF signal is held low during the whole configuration process, otherwise it is toggled once during every byte transformation with the CCLK signal (configuration clock). In both cases configuration is done when ENACNF signal is low.

ENCANF controlled FPGA configuration is mostly used in the firmwares which implements a GPIF interface. CCLK is connected to the GPIF interface clock (IFCLK), and ENACNF pin is connected to the corresponding control signal (CTLx) signal of the GPIF. This control signal is the one which is responsible form GPIF FIFO write operation. When the user sends a buffer of configuration bytes over a bulk or isochronous pipe, GPIF toggles its FIFO write CTL pin (high-

low-high) with the interface clock for each byte. This process generates a signal diagram which matches the FPGA configuration signal diagram as seen in figure 2.X, ENACNF controlled configuration.

Initialization and get status steps (see below) are done by toggling PROGRAM pin and checking INIT and DONE pin statuses as it is done in uncontrolled ENACNF configuration.

Figure 2.2 shows the pin connections between FX2 and an FPGA device for configuration. Figure 2.3 shows two different diagrams for controlled and uncontrolled ENACNF configuration.

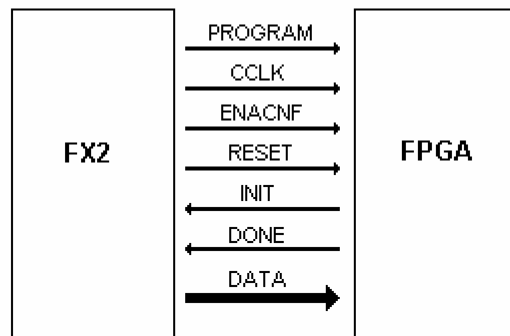
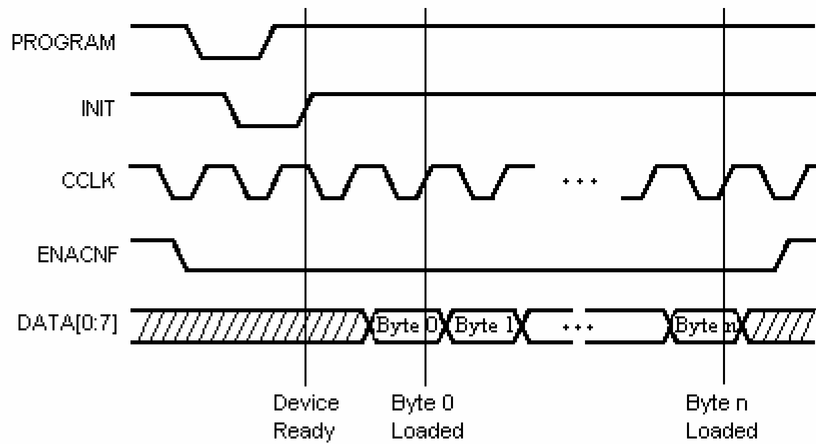


Figure 2.2 – Pin connections for FPGA configuration.



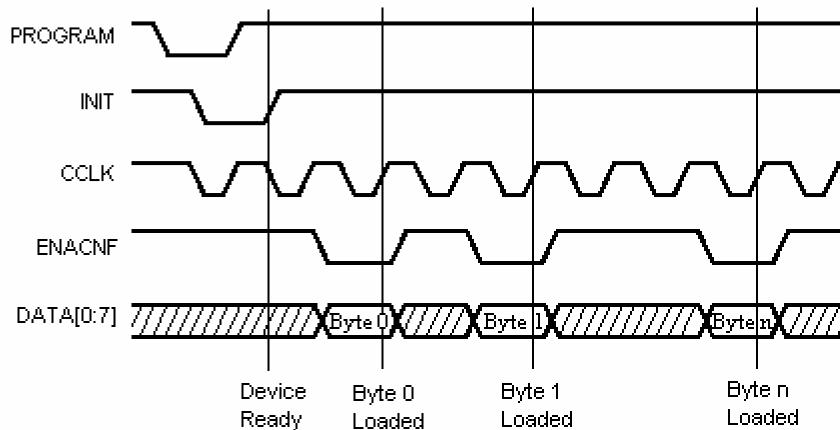


Figure 2.3- FPGA configuration signal diagrams.

Following is a step by step description of an ENACNF uncontrolled configuration of the FPGA:

1.) Host makes a FPGA initialization request.

2.) Firmware starts initialization process by making ENACNF (enable configuration) pin low. Then it holds the PROGRAM pin low for 1 millisecond.

3.) INIT pin of the FPGA is initially high. A logic low on the PROGRAM pin resets the configuration logic and holds the FPGA in the clear configuration memory state. As long as the PROGRAM pin is held low, the FPGA continues to clear its configuration memory while holding INIT low to indicate the configuration memory is being cleared. When PROGRAM is released, the FPGA continues to hold INIT low until it has completed clearing all the configuration memory.

4.) After the firmware holds the PROGRAM pin high it checks with a timer loop if the INIT goes to high to confirm a successful initialization. Later it returns to the host 1 byte status code (0x00 – success, 0xAB – timeout error). If a timeout error occurs configuration process is finished unsuccessfully.

5.) Host extracts configuration bytes form an FPGA configuration file (.exo, .rbt or another format), packs them in smaller buffers and sends them to the Firmware in the right order over a control, bulk or isochronous pipe.

6.) Firmware receives configuration buffers, checks their length and sends the configuration bytes to the FPGA in the right order. When sending configuration bytes it first puts the data on the DATA bus and toggles the CCLK once for each byte. During this process firmware can't detect any errors.

7.) After all data bytes are sent to the firmware; host starts a get FPGA status request.

8.) FPGA performs a CRC check after all configuration data is loaded. Upon successful completion of the final CRC check, the FPGA enters the start-up sequence. This sequence releases DONE (transition high) and activates the FPGA I/O Pins. At this point the FPGA becomes active and functional with the loaded design.

9.) Firmware puts the ENACNF pin in the high state and checks the DONE pin's state with a timer loop. If it sees that DONE pin is high, it returns the status 0x00 to the user to indicate a successful end of FPGA configuration process. Otherwise, it returns 0xAB to indicate operation timeout and finishes FPGA configuration process with error.

Also some FPGA designs need an external RESET signal to initialize their signals, variables, state machines etc. For this reason CeUsb2 defines a I/O pin as RESET signal which is connected to the FPGA.

Code example 2.3 is 8051 firmware C code snippet showing the reset, initialization and get status processes for the FPGA configuration with controlled ENACNF.

```
// Code example 2.3
// 8051 sample firmware C code for FPGA reset, initialization and get status processes.
//
// I/O port pins for this design are:
// PORTE pin 0: RESET
// PORTE pin 4: PROGRAM
// PORTE pin 5: DONE
// PORTA pin 5: INIT
//
// ENACNF pin is connected to the FIFO write signal of the GPIF (ENACNF controlled configuration)
```

```
//


void DelayUs(WORD usDelay); // microsecond delay function
void DelayMs(WORD msDelay); // millisecond delay function

void FPGA_ResetConfig( )
{
        IOE |= 0x01; // RESET high
        DelayUs(10);
        IOE &= 0xFE; // RESET low
}


BYTE FPGA_Init()
{
        WORD xdata Timeout = 0;

        IOE &= 0xEF; // PROGRAM low
        DelayMs(1); // delay 1 ms
        IOE |= 0x10; // PROGRAM high

        while(Timeout <= 1000 && (!(IOA & 0x20)) // check INIT pin
                Timeout++;

        if(Timeout == 1001)
                return 0xAB; // return timeout error

        return 0x00; // return success
}


BYTE FPGA_GetStatus()
{
        WORD xdata Timeout = 0;

        DelayMs(50); // delay 50 ms

        while(Timeout <= 1000 && (!(IOE & 0x02))
                Timeout++;

        if(Timeout == 1001)
                return 0xAB;  // return timeout error
        else
        {
                FPGA_ResetConfig(); // reset the FPGA
                return 0x00; // return success
        }
}
```

2.1.10 External RAM 1

FX2 chips have internal memory spaces for code & data space, USB registers and endpoint buffers. Some CeUsb2 devices have also an external RAM chip connected to the FX2 chip. FX2 CPU can use this RAM as code and data memory. CeUsb2 devices which have 56 – pin FX2 chips cannot have this RAM since these types doesn't have external memory controlling mechanism.

If a CeUsb2 device doesn't have external RAM then all the firmware program code and data should be fit into the lower 8Kbytes part of the internal RAM. Otherwise, program code can grow up to 56 Kbytes and memory can grow up to 64 Kbytes.

CeUsb2 loader driver is responsible from firmware downloading. It first downloads a simple loader firmware which is able to access the external memory of the FX2. It reads the raw hex file codes from the final firmware file and extracts the part which will stay in the external RAM and sends this data to the loader firmware with download external memory request. After then it removes the loader firmware from FX2 and downloads the remaining bytes of the final firmware to the internal RAM of the FX2.

Figure 2.4 shows the memory layout of the CeUsb2.

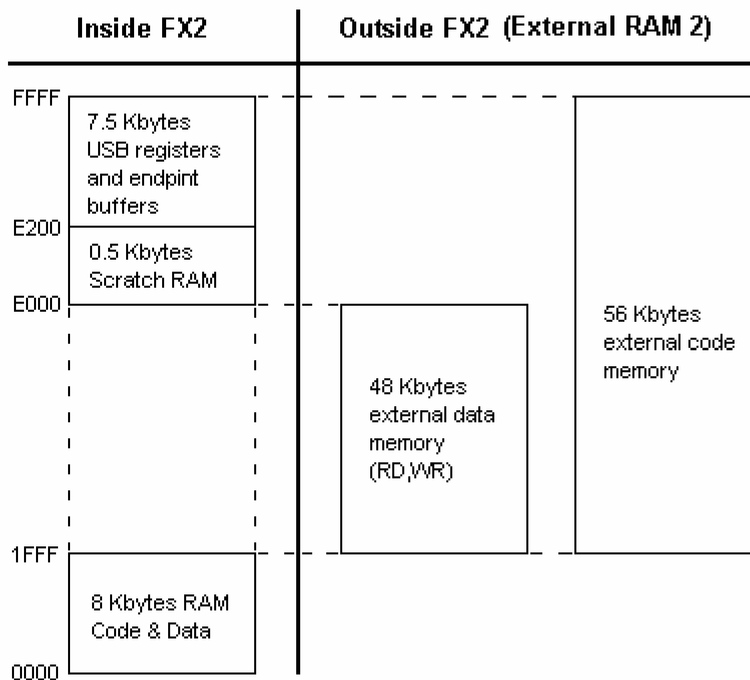


Figure 2.4 – CeUsb2 Memory Layout.

2.1.11 External RAM 2

Some CeUsb2 boards, which have an integrated FPGA chip, may also have one or more external RAM chips connected to the FPGA directly. This RAM can be used as a data buffer or FIFO buffer or for another purpose.

CeUsb2 boards can use external RAMs with different memory technologies such as SRAM, SDRAM or DDR-SDRAM. For example, CeUsb2 evaluation board has 2 SRAMs which are accessible with a simple RAM interface. You should check the documentation of the RAM chip to understand how it is controlled and accessed.

Code example 2.4 is a complete external memory test design written in VHDL.

```
-- Code example 2.4
-- VHDL sample design for CeUsb2 external SRAM access.
--
-- This design is written for a CeUsb2 device with 2 external SRAMs connected to its FPGA.
-- Design writes incrementing data to the RAM by incrementing the address. Later it reads back and
-- compares the data with the written one. If a compare error occurs the corresponding led is OFF,
-- otherwise it blinks. This process is repeated sequentially for both RAMs.
--

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

-- memory test entity
entity mem_test is
	Port (
	sclk : in std_logic; -- system clock
	reset : in std_logic; -- reset signal
	RamCe : out std_logic_vector(1 downto 0); -- chip enable signal for RAMs
	RamWr : out std_logic; -- write signal for both RAMs
	RamOe : out std_logic; -- output enable signal for both RAMs
	RamAdr : out std_logic_vector(18 downto 0); -- address lines for both RAMs
	RamData : inout std_logic_vector(7 downto 0); -- data bus for both RAMs
	Led : out std_logic_vector(1 downto 0) -- debug leds
	);
end mem_test;

architecture Behavioral of mem_test is

-- maximum ram address constant
constant RAM_MAX : std_logic_vector(18 downto 0) := "1111111111111111111";

signal ram_data : std_logic_vector(7 downto 0); -- internal ram data signal
signal ram_data_cmp : std_logic_vector(7 downto 0); -- data stored for comparision
signal ram_adr      : std_logic_vector(18 downto 0); -- internal ram address signal

type STATE_TYPE is (Write0,Write1,Write2,
	Read0,Read1,Read2,Read3,Read4,Restart); --type for state machine states
```

```
signal State : STATE_TYPE; -- variable for state machine states

begin

RamAdr <= ram_adr;

-- test state machine process
test_sm : process(sclk,reset)
variable ram_sw : integer range 0 to 1; -- switches between two RAMs
begin
        if(reset = '1') then -- when reset is active
                -- ram control signals are high initially
                RamCe <= "11";
                RamOe <= '1';
                RamWr <= '1';

                -- internal address and data lines are low
                ram_adr <= (others => '0');
                ram_data <= (others => '0');
                ram_data_cmp <= (others => '0');

                -- release the data bus
                RamData         <= (others => 'Z');

                Led <= "11"; -- both leds are OFF

                ram_sw := 0; -- first RAM is active
                State <= Write0; -- next state
        elsif rising_edge(sclk) then -- when rising edge of the system clock
                case State is
                when Write0 => -- start writing
                        ram_adr <= (others => '0');
                        ram_data <= (others => '0');
                        RamCe(ram_sw) <='0'; -- chip enable for corr. RAM is low
                        Led(ram_sw) <= '1'; -- corr. LED is OFF
                        State <= Write1; -- next state
                when Write1=>
                        RamData <= ram_data; -- put data on the bus
                        RamWr <= '0'; -- trigger RAM writing (1 byte)
                        State <= Write2; -- next state
                when Write2=>
                        RamWr <= '1'; -- finish RAM writing
                        if(ram_adr = RAM_MAX)then -- maximum address is reached
                                ram_adr <= (others => '0'); -- address 0
                                RamData <= (others => 'Z'); -- release data bus
                                ram_data_cmp <= (others => '0');
                                State <= Read0; -- next state, step to reading part
                        else
                                ram_adr <= ram_adr + 1; -- increment address
                                ram_data <= ram_data + 1; -- increment data
                                State <= Write1; -- next state, go on writing
                        end if;

                when Read0 =>
                        RamOe <= '0'; -- triger RAM reading        (1 byte)
                        State <= Read1; -- next state
```

```
                    when Read1 =>
                              RamOe <= '1'; -- finish RAM reading
                              State <= Read2; -- next state
                    when Read2 => --compare state
                              if(RamData /= ram_data_cmp) then --compare error
                                        State <= Read4; -- next state, error state
                              else
                                        State <= Read3; -- next state, repeat state
                              end if;
                    when Read3 =>
                              if(ram_adr = RAM_MAX) then -- maximum address is reached without errors
                                        ram_adr <= (others => '0'); -- ram address is 0
                                        RamCe(ram_sw) <='1';      -- switch the active RAM
                                        Led(ram_sw) <= '0'; -- corresponding LED is ON
                                        State <= Restart; -- next state
                              else
                                        ram_adr <= ram_adr + 1; -- increment address
                                        ram_data_cmp <= ram_data_cmp + 1; -- increment compare data
                                        State <= Read0; -- next state
                              end if;
                    when Read4 => -- compare error
                              RamCe(ram_sw) <='1'; -- switch the active RAM
                              State <= Restart; -- next state

                    when Restart =>
                              if(ram_sw = 0)then -- change ram switch variable
                                        ram_sw := 1;
                              else
                                        ram_sw := 0;
                              end if;
                              State <= Write0; -- next state, go to reading part of the SM

                    when others =>
                              null;
                    end case;
          end if;
end process;

end Behavioral;
```

2.2 *Firmware*

CeUsb2 firmware is a 8051 based embedded application which runs in the integrated CPU of FX2. Its main responsibilities are:

1. Replying USB requests from host, such as:
    - Descriptor request
    - Set & clear feature request
    - Get & set configuration request
    - Get & set interface request

  - § Get status request
  - § Vendor & class request
  - § Control & Bulk & Isochronous transfer requests
2. Initializing the hardware (GPIF, SIO, IO PORTS e.t.c) after USB power on.
3. Handling power requests such as remote wake-up.
4. Disconnecting the device from the USB bus upon request.
5. Performing data transfer with storage components (EEPROM, RAM e.t.c).
6. Configuring the FPGA.
7. Communicating with the FPGA.

Every CeUsb2 device needs two firmwares. The first one which is called as loader firmware is same for all CeUsb2 type devices. It is a small embedded executable which is able to perform minimum USB requests (not bulk and isochronous transfers) and access EEPROM and external memory (external RAM 1) if there is one on the board. Loader firmware fits into the first 8 Kbyte code & data part of the internal memory and doesn't need external memory to run. Indeed it can download executable code to external memory which is send by the host. The other main functionality of the loader firmware is to write the required device, product and Cesys specific device identifiers into the EEPROM, if the device is in an unconfigured state. For more information about this process see chapter 2.1.6, Serial EEPROM.

The second firmware is the real firmware which runs in corporation with the CeUsb2 final driver, if the device is in a configured state. Every CeUsb2 type board has at least one firmware application. Each firmware may define different USB configurations, interfaces and endpoints and may handle data transfer with external components in a different way. You have to check the documentation of your specific firmware to learn its features.

2.3 Loading process

When CeUsb2 board is plugged in, the following events will occur always in the same order:

1- The operating system recognizes CeUsb2 board and asks for its ids (device, product and Cesys specific device identifiers).

2- If the board is not configured it responses with default, unconfigured state ids, else it responses with the ids written in its EEPROM.

3- With this information, host installs the loader driver, CeUsb2ld.sys. This driver first downloads a firmware application CeU2Ld.hex. If the device is not configured it stops loading process leaving the loader driver in the system. After then the loader driver can communicate with user mode programs.

4- Loader driver runs the loader firmware, which is able to change the ids. Therefore it asks once more for ids and searches the registry with these ids for a matching second firmware which will communicate with the device. If it can find one, it loads this firmware otherwise it finishes loading process with error.

5- Loader driver runs the newly loaded firmware.

6- Firmware electronically disconnects your device from the USB bus.

7- The host PC, seeing the original device disappeared, removes the loader driver from the memory.

7- Firmware reconnects the device to the USB bus.

8- Host sees that a new device is plugged in and asks for new device ids. CeUsb2 responds with the new ids, which represent a configured device ready to work and communicate with user mode applications.

9- The host operating system loads the final driver, CeUsb2.sys. After then, the newly loaded driver is ready to perform data transfers over the USB bus.

2.4 CeUsb2 Configuration Program and Registry

Configuration program controls the registry entries of the CeUsb2 and configures the system in a way that the CeUsb2 drivers can distinguish between different devices also same type of devices with different firmwares.

During installation of CeUsb2 software, driver information file (CeUsb2.inf) writes the entries into the registry for a minimum CeUsb2 environment which supports a generic device – firmware combination. Actually generic firmware is the embedded 8051 application for the CeUsb2 FPGA board.

If you have another type of CeUsb2 device or another kind of firmware for your prototyping board you can configure your system for the new environment. However, you will most probably get a batch file (.bat) or a registry file (.reg) from Cesys which will make the necessary changes for you automatically. Yet it is helpful to understand CeUsb2 registry entries and Configuration program to understand the working mechanism of the CeUsb2 software more detailed.

CeUsb2 puts its configuration in the registry under the directory, HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\CeUsb2\ldprm. This directory has further subdirectories for each CeUsb2 device type. Each of these subdirectories has one or more directories for a specific firmware that it supports. Figure 2.5 shows this hierarchy with the sub-keys of the directories.

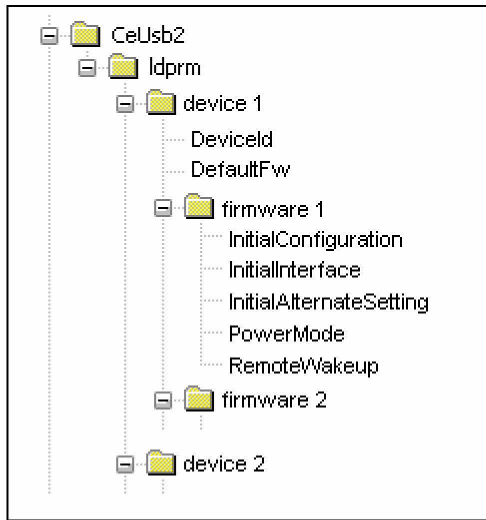


Figure 2.5 – Cesub2 registry directory structure

Device directories sub-keys:

DeviceId: 16 bit (2 byte) Cesys specific device identifier.
DefaultFw: References to a firmware subdirectory which includes information for the default firmware for this device.

Firmware directories sub-keys:

InitialConfiguration: Initial USB configuration number which will be activated by the CeUsb2 driver initially.
InitialInterface: Initial USB interface number in the initial configuration which will be activated by the CeUsb2 driver initially.
InitialAlternateSetting: Initial USB alternate setting number in the initial configuration – interface pair which will be activated by the CeUsb2 driver initially.
PowerMode – Power mode for this firmware. Possible values are:
0: Device is bus powered (uses the USB bus power).
1: Device is self powered (uses an external power supply).
RemoteWakeup – This parameter determines if this firmware supports remote wakeup feature of the USB. Possible values are:
0: Remote wakeup is not supported.
Another value: Remote wakeup is supported.

Figure 2.6 is a snapshot of CeUsb2 configuration program. It is a simple utility tool which gives a graphical interface to the user for configuring CeUsb2 environment. It displays all the entries of CeUsb2 registry directory and serves the features for adding new device types and firmwares. For more information about configuration program check CeUsb2Cfg.pdf documentation.

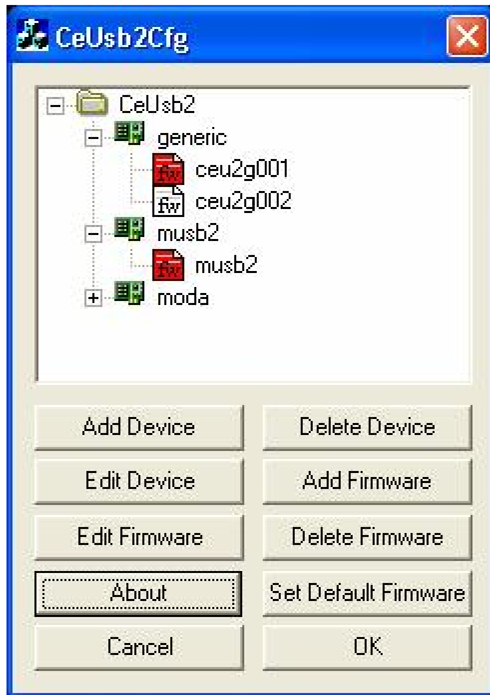


Figure 2.6 – CeUb2 configuration program.

2.5 Drivers

CeUsb2 has two kernel mode Windows drivers which are loaded sequentially by the system. This section describes general features of these drivers. See also the section 2.3, Loading Process, to learn how they are loaded in the system.

CeUsb2 drivers work on top of Windows supplied USB hub driver and host controller drivers. These system supplied drivers may change between operating systems. Figure 2.QQ illustrates the driver stack on Windows 2000 with WDM (windows driver model) layered architecture. Windows XP USB driver stack is almost similar to that one.

At the bottom of the Windows 2000 USB driver stack is the host controller driver. It consists of a class/miniclass driver pair. The host controller class driver, usbd.sys, is paired either with the openhci.sys (Open Host Controller Interface) minidriver or with the uhcd.sys (Universal Host Controller Driver) minidriver, depending on which USB protocol the system is configured to use. Which driver is used depends on the mainboard chip set of the PC.

Immediately above the host controller class driver is the USB bus driver, usbhub.sys, also known as the hub driver. This is the device driver for each hub on the system.

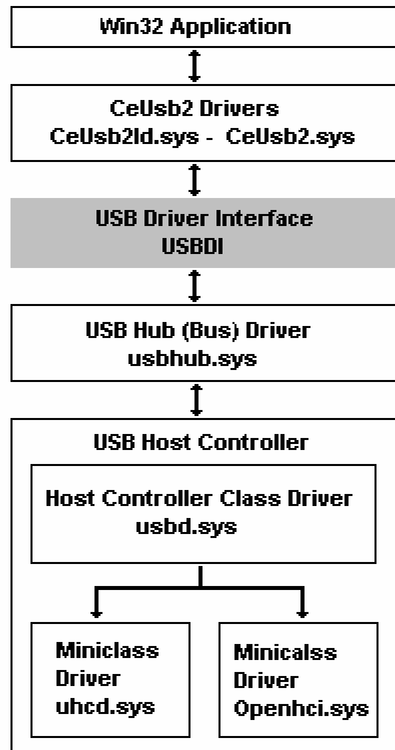


Figure 2.7 – Windows 2000 driver stack and CeUsb2 drivers.

USB Driver Interface (USBDI) is the programming interface which is provided by the operating system. Currently, there is no way to access it from the user mode. It is accessible by the kernel mode executables like CeUsb2 drivers.

Loader driver (CeUsb2Ld.sys) is a small kernel mode system executable which is designed with the WDM model of windows driver development kit (WIN DDK). It supports only minimum requirements of PNP (Plug & Play) protocol of Windows and doesn't support power management and WMI. If a CeUsb2 device is connected to the host, loader driver will be loaded by the system. It downloads the necessary firmware(s) on the board. If the board is unconfigured loader driver stays in the system enabling an input output control code (IOCTL) interface, with the minimum functionality to configure the board, and perform some USB vendor commands.
If the CeUsb2 board is configured previously, loader driver is unloaded and the final driver (CeUsb2.sys) is loaded by the system.

CeUsb2 main driver (CeUsb2.sys) is also developed with WDM model of WIN DDK. It supports almost all PNP requests, handles power management and participates in WMI (Windows management and instrumentation). CeUsb2 driver defines an interface for user mode applications with IOCTL codes, enumerations and structures to control CeUsb2 hardware and perform vendor and class

requests, control, bulk and isochronous transfers and many other USB functionalities.

2.6 Application Programming Interface (CeUsb2 API)

CeUsb2 API is a high level application programming interface which uses the input output control (IOCTL) interface of the CeUsb2 drivers. It has several classes and some global functions for device input output, general USB functionality, USB data transfers, firmware configuration, FPGA configuration, loader driver access, error handling and other features of CeUsb2 devices.

CeUsb2 API is developed with Visual C++ 6.0 and internally doesn't use any software kits other than Win 32 SDK. The final executable of the API is CeUsb2Api.dll.

CeUsb2 API makes the development process faster and easier if you are using C++ programming language for your software project. C++ header file CeUsb2Api.h is the main header file of the API. You should also link your project with the API library file, CeUsb2Api.lib.

For more information about CeUsb2 API, check the API documentation file, CeUsb2Api.pdf.

2.7 Graphical User Interface (CeUsb2 GUI)

CeUsb2 GUI is graphical user interface for programs which runs with the CeUsb2 hardware. It is built on top of CeUsb2 API with Visual C++ 6.0. Unlike the API it uses Microsoft Foundation Class Library (MFC) for its dialog functions. However it is still usable by a non MFC C/C++ program.

CeUsb2 GUI is a high level programming interface which includes several functions that displays some utility and control dialogs for CeUsb2 such as device enumeration, USB descriptors, USB configurations and interfaces, FPGA configuration, serial interface I/O control dialogs. The final executable of the GUI is CeUsb2Mfc.dll. C++ header file CeUsb2Wr.h is the main include file of the API. You should also link your project with the API library file, CeUsb2Mfc.lib.

For more information about CeUsb2 GUI, check CeUsb2gui.pdf. It has all the information about exported global functions with example C/C++ code displaying their usage.

2.8 CeUsb2 COM (Component Object Model) API

CeUsb2 COM API enables software programmers to communicate with the CeUsb2 hardware using another other programming languages than C or C++. It is built on top of CeUsb2 API (CeUsb2Api.dll) and CeUsb2 GUI (CeUsb2Mfc.dll).

CeUsb2 COM API is compatible to all object oriented programming languages which understand Microsoft's COM technology. Currently CeUsb2 COM API is tested with MS Visual C++ 6.0, MS Visual Basic.NET, MS Visual C#, Borland Delphi 7.0.

For more information about CeUsb2 COM API, check CeUsb2com.pdf.

2.9 CeUsb2 Diagnostic Program

CeUsb2 diagnostic program (CeUsb2Diag.exe) is a diagnostic test & control program for CeUsb2 type of devices. It uses both the API and the GUI of the CeUsb2 software kit.

Diagnostic program can be used to test the basic functionality of the CeUsb2 boards either by its control dialogs or its command line executer. Moreover, it has an additional dialog on which the user can write, compile and run text base CeUsb2 command files (with extensions txt and c2c).

For more information about CeUsb2 diagnostic program, check CeUsb2diag.pdf. This document describes the usage of the diagnostic program, together with its simple command language.

Figure 2.8 is a snapshot of CeUsb2 diagnostic program with its command dialog.

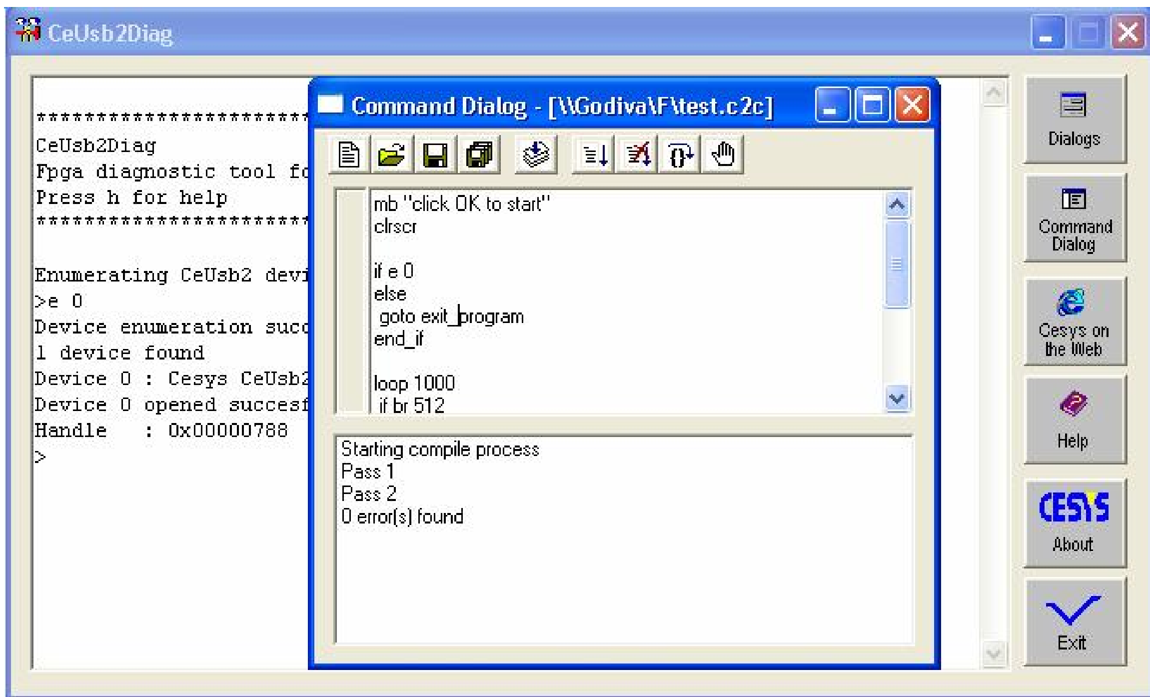


Figure 2.8 – CeUsb2Diag.exe.