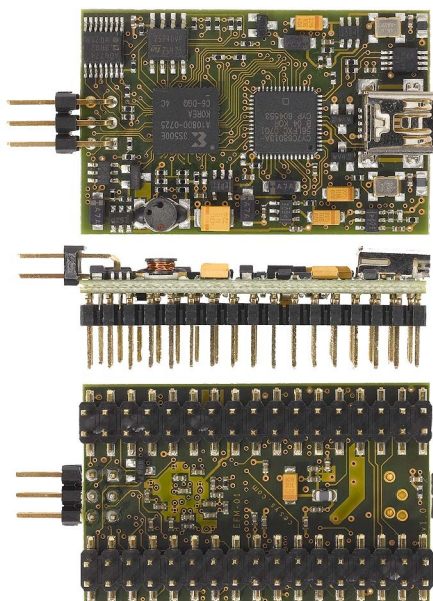


**SPARTAN-3E™ FPGA module with USB2.0,
SPI FLASH and JTAG interface.**

Order number: C1050-4107



Copyright information

Copyright © 2010 CESYS GmbH. All Rights Reserved. The information in this document is proprietary to CESYS GmbH. No part of this document may be reproduced in any form or by any means or used to make derivative work (such as translation, transformation or adaptation) without written permission from CESYS GmbH.

CESYS GmbH provides this documentation without warranty, term or condition of any kind, either express or implied, including, but not limited to, express and implied warranties of merchantability, fitness for a particular purpose, and non-infringement. While the information contained herein is believed to be accurate, such information is preliminary, and no representations or warranties of accuracy or completeness are made. In no event will CESYS GmbH be liable for damages arising directly or indirectly from any use of or reliance upon the information contained in this document. CESYS GmbH will make improvements or changes in the product(s) and/or program(s) described in this documentation at any time.

CESYS GmbH retains the right to make changes to this product at any time, without notice. Products may have minor variations to this publication, known as errata. CESYS GmbH assumes no liability whatsoever, including infringement of any patent or copyright, for sale and use of CESYS GmbH products.

CESYS GmbH and the CESYS logo are registered trademarks.

All product names are trademarks, registered trademarks, or service marks of their respective owner.

⇒ Please check www.cesys.com to get the latest version of this document.

CESYS Gesellschaft für angewandte Mikroelektronik mbH
Zeppelinstrasse 6a
D – 91074 Herzogenaurach
Germany

Overview

Summary of EFM01

The **Embedded FPGA Module EFM 01** is a very small low-cost module with SPARTAN-3E FPGA and USB 2.0 Interface. 50 I/O balls of the FPGA are available on standard 0,1 inch headers. It offers multiple configuration options and can also be used without the USB interface.

Feature list

Form factor	Module, 45x30x15mm
XILINX SPARTAN-3ETM	XC3S500E-4CPG132C
USB2.0 Controller	CYPRESS™ CY7C68013A
FPGA configuration	Using USB2.0, JTAG or SPI-Flash
Expansion connectors	Two 2x17-Pin standard RM2.54mm headers
Clock	Onboard 48MHz clock signal, external clock sources possible.
Example code	Sample VHDL and C++ code can be used as starting point for user designs.

Included in delivery

The standard delivery, order no. C1050-4107, includes:

- One EFM01
- One USB cable 1,5m
- One CD-ROM containing the user's manual (English), drivers, libraries, tools and example source code.

All parts are ROHS compliant.

Single modules and very low quantities can be ordered in this configuration only. OEM customers may have a different scope of supply based on individual agreements. If you have questions, please call.

Hardware

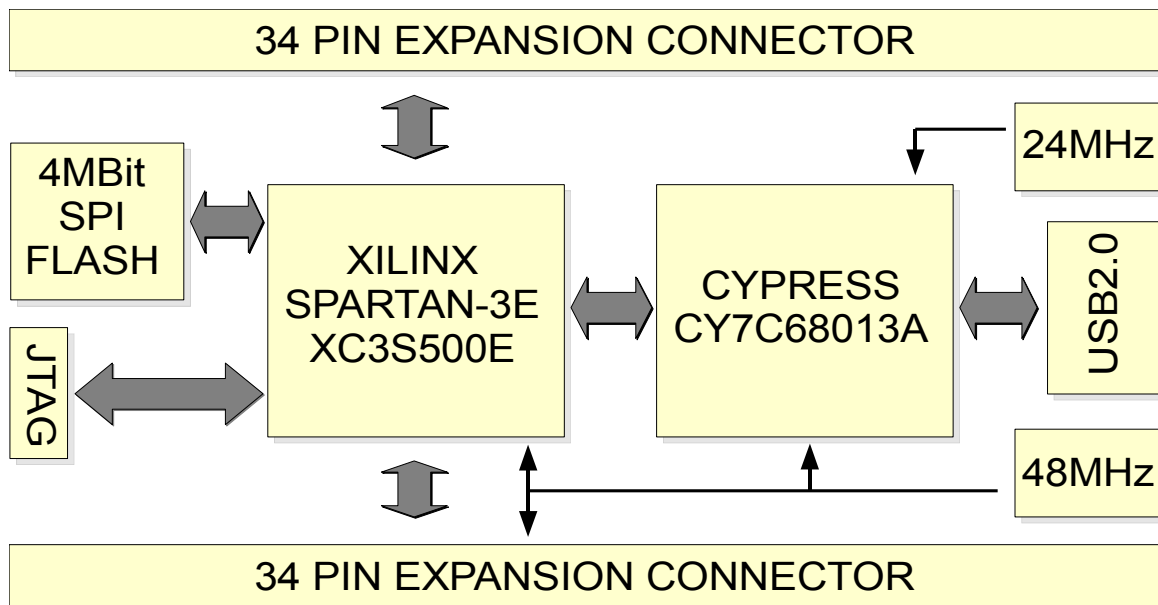


Figure 1: EFM01 block diagram

SPARTAN-3E FPGA

XC3S500E-4CPG132C FPGA features:

Configurable logic blocks (CLB)	1,164
Equivalent logic cells	10,476
Slices	4,656
RAM16 / SRL16	4,656
Max distributed RAM	74,496

For details of the SPARTAN -3E™ FPGA device, please look at the data sheet at:
<http://direct.xilinx.com/bvdocs/publications/ds312.pdf>

Module size

EFM01 is sized as small as 43x28mm. So even in systems where space is limited it is possible to implement a fast USB2.0 connection with only little effort.

Connectors and FPGA pinout

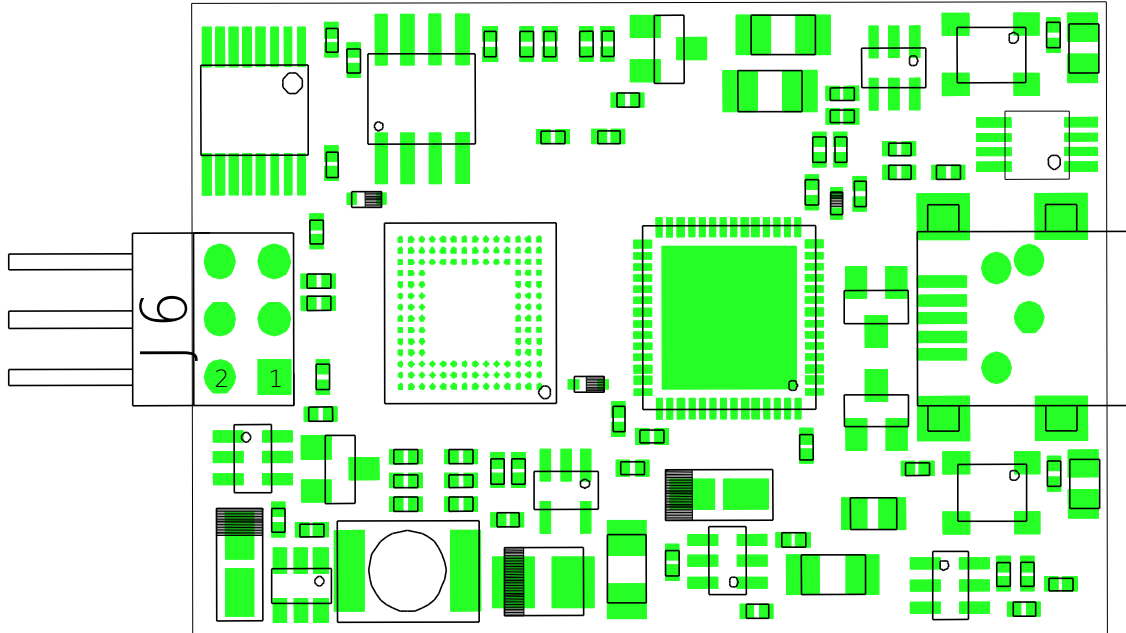


Figure 2: EFM01 Top View

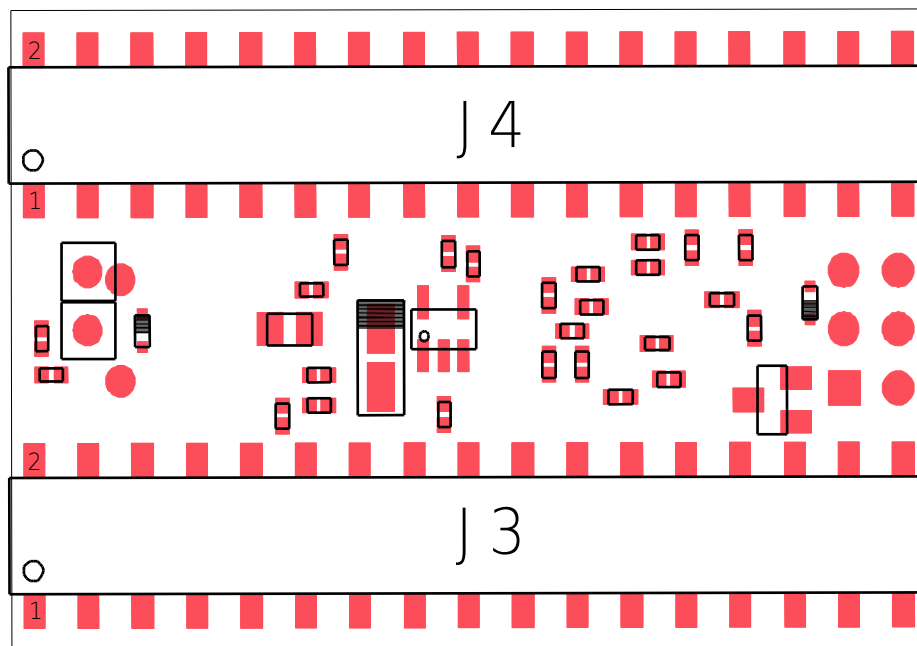


Figure 3: EFM01 Bottom View

Powering EFM01

There are two ways to power *EFM01*. First, *EFM01* may be used bus-powered without the need of any external power supply other than USB. In this mode J3,PIN7 must connect to J3,PIN8. If *EFM01* is run bus-powered, 3.3VCCO on J3,PIN21 can source up to 500mA, but keep in mind that USB power supply current is limited depending on which system is used as host and may even be less than 500mA in case of some laptops. 5VUSB supply on J3,Pin8 can be used to power external devices up to the limits of the used USB2.0 host interface. NEVER connect external voltages to 5VUSB, as this may result in serious damage of the attached USB2.0 host interface.

Modes of operation			
Mode	J3,PIN7	J3,PIN8	3.3VCCO limit
Bus- powered	Connect to J3,Pin8	Connect to J3,Pin7	500 mA
Self- powered	External 5V supply	Must be left floating	1000 mA

Secondly, it is possible to use *EFM01* self-powered if the attached USB2.0 host interface should not be used as power supply or if USB2.0 is not connected at all. In this case an external 5V power supply must connect to J3, PIN7, while J3,PIN8 must be left floating. If USB2.0 is not used, J4,PIN18 “PWR_ENA” must be driven HIGH externally to enable power-up of the onboard power supplies of the FPGA. If *EFM01* is run self-powered, current supplied on J3,PIN21 (3.3VCCO) mainly is limited by the external power supply, but should not exceed 1000mA.

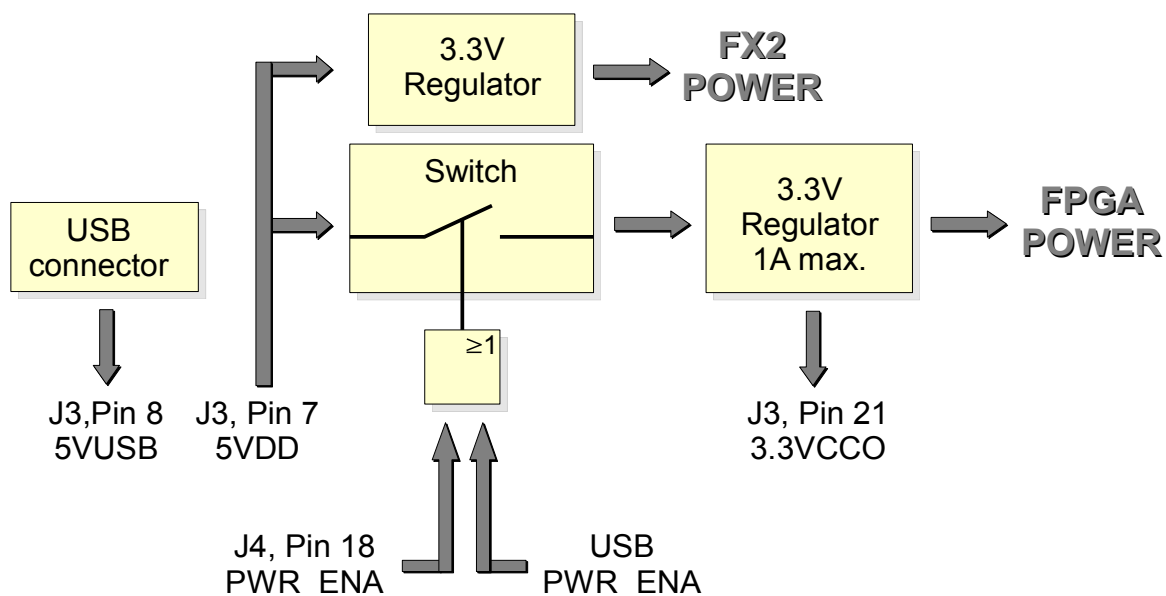


Figure 4: EFM01 Power scheme

EFM01 configuration

Configuration of EFM01 can be accomplished in several ways: JTAG, FLASH or USB. The default configuration mode is booting from SPI FLASH. After powering on the FPGA, EFM01 always tries to configure itself from the attached FLASH using SPI Master mode. If no valid design is stored in the SPI FLASH the FPGA has to be configured via JTAG or USB. For further information on configuration via USB please take a look at chapter C. JTAG configuration is supported at any time after the FPGA is properly powered on. For downloading designs via JTAG ISE WebPACK¹ from XILINX™ is recommended. The tool can be downloaded from XILINX web page free of charge.

J6 JTAG connector			
PIN	Signal name	FPGA IO	Comment
1	TDI	A2	Test Data In
2	TMS	B14	Test Mode Select
3	TCK	B13	Test Clock
4	TDO	A14	Test Data Out
5	2.5VCCAUX	--	Auxiliary supply voltage of the FPGA
6	GND	--	Corresponding ground signal

For further information on the different configuration solutions for XILINX™ SPARTAN-3E FPGA the reader is encouraged to take a look at the user guide [UG332](#) on XILINX™ web page.

How to store configuration data in SPI Flash

To allow configuration of the FPGA via onboard SPI Flash on power-up first an appropriate configuration file has to be stored in the SPI Flash. There are several ways to accomplish this.

Loading SPI Flash via USB

The easiest way to get data into SPI Flash surely is to use CESYS software UDKLab. With the help of this easy to use tool raw binary FPGA configuration bitstreams (*.bin) can be downloaded to onboard SPI Flash via USB. Further information about usage of UDKLab can be found in Software Chapter.

SPI Flash Indirect Programming Using FPGA JTAG Chain

Since Xilinx™ ISE WebPACK¹ version 10.1 it is possible to configure SPI Flashes attached to the FPGA via JTAG interface. Before starting to download a design to SPI Flash with

¹ [ISE WebPACK](#) at the XILINX web page from

iMPACT programming software it is necessary to generate the required *.mcs PROM file. Xilinx™ provides a good guide how to accomplish that using iMPACT or PROMGen software tools in their Spartan-3 Generation Configuration User Guide [UG332](#) in chapter Master SPI Mode, Preparing an SPI PROM File. Select **4M SPI PROM Density** when asked. After creation of *.mcs file connect JTAG adapter to *EFM01* JTAG connector J6 as described in the spreadsheet above. **Only 2.5V signaling levels are supported.** Then power-up *EFM01*. Be sure to also enable FPGA power-up. If external power supply is used and no USB connection is available, driving PWR_ENA (J4, pin18) HIGH externally will enable FPGA power supplies to ramp up. If Bus- powered mode is used it is necessary to start a software tool like **cesys-Monitor** after plugging *EFM01* to USB to enable FPGA power-up. With Xilinx™ parallel cable IV the led lights green if FPGA is powered on. With *EFM01* properly powered now start Xilinx™ iMPACT programming tool and select Boundary Scan mode. Follow the manual provided by Xilinx™ in user guide [UG332](#) chapter Master SPI Mode, Indirect SPI Programming using iMPACT. Select **M25P40 SPI Flash PROM Type** when asked.

SPI Flash				
M25P40	FPGA Connection	FPGA IO	FPGA Direction	Comment
D	MOSI	N2	Output	Master SPI Serial Data Output
Q	DIN	N8	Input	Master SPI Serial Data Input
S#	CSO_B	M2	Output	Master SPI Chip Select Output
C	CCLK	N12	Output	Configuration Clock
W#	WP#	--		Externally pulled HIGH
HOLD#	HOLD#	--		Externally pulled HIGH

SPI Flash Direct Programming using iMPACT

Out of the box Direct SPI Programming via Xilinx™ download cable and iMPACT programming software is not supported. But with the help of some tiny FPGA design which only has to bypass SPI signals to external IO pins on connectors J3 and/or J4 it is possible to access all needed SPI Flash pins. Connect JTAG adapter to external IO pins as described in the following chart.

SPI Flash Direct Programming – necessary connections to JTAG cable		
M25P40	FPGA Connection	JTAG signal name
D	MOSI	TDI
Q	DIN	TDO
S#	CSO_B	TMS
C	CCLK	TCK

SPI Flash Direct Programming – necessary connections to JTAG cable		
M25P40	FPGA Connection	JTAG signal name
VCC	3.3VCCO	VREF
GND	GND	GND

Make sure that 3.3VCCO (J3,pin21) power supply is connected to the JTAG adapter. Do not forget to also enable FPGA power-up. If external power supply is used and no USB connection is available, driving PWR_ENA (J4,pin18) HIGH externally will enable FPGA power supplies to ramp up. If Bus- powered mode is used it is necessary to start a software tool like **cesys-Monitor** after plugging *EFM01* to USB to enable FPGA power-up. With Xilinx™ parallel cable IV the led lights green if FPGA is powered on. Before starting to download a design to SPI Flash with iMPACT programming software it is necessary to generate the required *.mcs PROM file. Xilinx™ provides a good guide how to accomplish that using iMPACT or PROMGen software tools in their Spartan-3 Generation Configuration User Guide [UG332](#) in chapter Master SPI Mode, Preparing an SPI PROM File. Select **4M SPI PROM Density** when asked. Now programming of SPI Flash can be started by clicking Direct SPI Configuration from within iMPACT. Follow the manual provided by Xilinx™ in user guide [UG332](#) chapter Master SPI Mode, Direct SPI Programming using iMPACT. Select **M25P40 SPI Flash PROM Type** when asked.

Program SPI Flash using external microcontrollers

Similar to Direct SPI Flash Programming described in the previous section another method of SPI Flash configuration is possible. Instead of using iMPACT software together with a JTAG download cable, a small microcontroller could be used to address the SPI Flash. The FPGA has to be configured with a small helper-design that bypasses SPI signals of the Flash to external IO on connectors J3 and/or J4 which then can be driven by the microcontroller. Bypassing should also be enabled in the design downloaded to the SPI Flash or else an update of Flash contents is only possible after reconfiguration of the FPGA with the small helper-design using another configuration method like USB or JTAG. Keep in mind that with this procedure maximum clock frequency of SPI Flash may be degraded due to pin-to-pin delays in the FPGA which depend on what pin pair is used and may lead to intolerable phase difference. To stimulate the FPGA to reconfigure from SPI Flash it is necessary to pull PROG_B pin (J4, Pin5) low for a short period of time. After releasing PROG_B again, the FPGA will automatically start to configure itself from the attached Flash.

External expansion connectors

On connectors J3 and J4 up to 50 general purpose FPGA IO are accessible. All IO banks of the FPGA are configured for 3.3V signaling level. Additionally some configuration pins are routed to these connectors to enable even more flexible use of *EFM01*.

! IO on connectors J3 and J4 are directly connected to FPGA IO and therefore are only 3.3 Volt tolerant. NEVER apply voltages outside the interval [-0,2V..3,45V] as this may lead to severe damage of FPGA and attached components.

J3 External expansion connector I			
PIN	Signal name	FPGA IO	Comment
1	GND	--	
2	EXT1_IO0	C12	IO
3	EXT1_IO1	F12	IO
4	EXT1_IO2	C11	IO*
5	EXT1_IO3	H3	IO, LHCLK7 [#]
6	EXT1_IO4	A12	IO**
7	5VDD	--	External 5V power supply may be connected at this pin. ATTENTION: If no external power source is used, 5VUSB MUST connect to 5VDD .
8	5VUSB	--	5V USB power supply output to power external logic devices. ATTENTION: Do not connect any external power supply. ATTENTION: If no external power source is used, 5VUSB MUST connect to 5VDD.
9	EXT1_IO5	G13	IO, RHCLK7 ^{##}
10	EXT1_IO6	F3	IO, LHCLK0 [#]
11	EXT1_IO7	H13	IO, RHCLK4 ^{##}
12	GND	--	
13	EXT1_IO8	B12	IO**
14	EXT1_IO9	C2	IO
15	EXT1_IO10	F13	IO
16	EXT1_IO11	D2	IO
17	EXT1_IO12	D12	IO, LDC0, low during configuration
18	EXT1_IO13	C4	IO**
19	EXT1_IO14	A13	IO
20	EXT1_IO15	B11	IO
21	3.3VCCO	--	3.3 Volt power supply output. VCCO power supply for all FPGA IO banks. May be used to optionally power external logic devices. ATTENTION: Do not connect any external power supply.

* IO for XC3S250E,XC3S500E, INPUT ONLY for XC3S100E

LHCLK inputs optionally clock the left-half of the SPARTAN-3E device

** IO for XC3S250E,XC3S500E, N.C. for XC3S100E

RHCLK inputs optionally clock the right-half of the SPARTAN-3E device

J3 External expansion connector I			
PIN	Signal name	FPGA IO	Comment
22	EXT1_IO16	D13	IO, HDC, high during configuration
23	EXT1_IO17	H12	IO; RHCLK5 ^{##}
24	SYSCLK	A10	GCLK4; System clock (48MHz) driven by onboard oscillator.
25	EXT1_IO19	J14	IO, RHCLK3 ^{##}
26	GND	--	
27	EXT1_IO20	L13	IO
28	EXT1_IO21	J12	IO, RHCLK1 ^{##}
29	EXT1_IO22	M12	IO
30	EXT1_IO23	M10	IO ^{**}
31	EXT1_IO24	M9	IO ^{**}
32	EXT1_IO25	C14	IO, LDC1, low during configuration
33	GND	--	
34	EXT1_IO26	C13	IO, LDC2, low during configuration

J4 External expansion connector II			
PIN	Signal name	FPGA IO	Comment
1	GND	--	
2	EXT2_IO0	C5	IO
3	EXT2_IO1	G3	IO, LHCLK4 [#]
4	EXT2_IO2	J3	IO
5	EXT_PROGB	A1	PROG_B, Active Low asynchronous reset to configuration logic of FPGA. Internal 4.7kOhm pull-up. Leave open, if not used.
6	EXT2_IO4	H1	IO, LHCLK5 [#]
7	EXT2_IO5	H2	IO, LHCLK6 [#] , green LED
8	GND	--	
9	EXT2_IO6	G1	IO, LHCLK3 [#]
10	EXT2_IO7	L1	IO
11	EXT2_IO8	F2	IO, LHCLK1 [#]
12	EXT2_IO9	K3	IO [*]
13	DONE		FPGA DONE pin. Low during configuration.

^{##} RHCLK inputs optionally clock the right-half of the SPARTAN-3E device

[#] LHCLK inputs optionally clock the left-half of the SPARTAN-3E device

^{*} IO for XC3S250E, XC3S500E, INPUT ONLY for XC3S100E

J4 External expansion connector II			
PIN	Signal name	FPGA IO	Comment
14	EXT2_IO10	L2	IO
15	EXT2_IO11	F1	IO, LHCLK2 [#]
16	GND	--	
17	GND	--	
18	PWR_ENA		If USB is not used, this pin must be driven HIGH externally to power-up FGPA. Internal 10kOhm pull-down. Leave open, if not used.
19	#Flash_Inhibit	--	Active Low Flash Inhibit signal. Drive Low on power-up to prevent FPGA configuration from SPI-Flash. Internal 1kOhm pull-up. Leave open, if not used. Do not drive Low constantly, as this signal is used as #Reset- signal in USB applications.
20	EXT2_IO13	L3	IO
21	EXT2_IO14	M1	IO
22	EXT2_IO15	N9	IO ^{**}
23	EXT2_IO16	P11	IO [*]
24	EXT2_IO17	N10	IO ^{**}
25	EXT2_IO18	M13	IO
26	GND	--	
27	EXT2_IO19	J13	IO, RHCLK2 ^{##}
28	EXT2_IO20	N14	IO
29	EXT2_IO21	L14	IO
30	EXT2_IO22	K13	IO
31	EXT2_IO23	G14	IO, RHCLK6 ^{##}
32	EXT2_IO24	K14	IO, RHCLK0 ^{##}
33	GND	--	
34	EXT2_IO25	F14	IO

! It is strongly recommended to check the appropriate data sheets of SPARTAN-3E devices about special functionality IO like LHCLK, RHCLK, LDC, HDC ...

For details of the SPARTAN -3E™ FPGA device, please look at the data sheet at:
<http://direct.xilinx.com/bvdocs/publications/ds312.pdf>

** IO for XC3S250E,XC3S500E, N.C. for XC3S100E

* IO for XC3S250E,XC3S500E, INPUT ONLY for XC3S100E

RHCLK inputs optionally clock the right-half of the SPARTAN-3E device

FPGA design

Cypress FX-2 LP and USB basics

Several data transfer types are defined in USB 2.0 specification. High-speed bulk transfer is the one and only mode of interest to end users. USB transfers are packet oriented and have a time framing scheme. USB packets consist of USB protocol and user payload data. Payload could have a variable length of up to 512 bytes per packet. Packet size is fixed to the maximum value of 512 bytes for data communication with CESYS *EFM01* USB card to achieve highest possible data throughput. USB peripherals could have several logical channels to the host. The data source/sink for each channel inside the USB peripheral is called the USB endpoint. Each endpoint can be configured as “IN”- (channel direction: peripheral => host) or “OUT”-endpoint (channel direction: host => peripheral) from host side perspective. CESYS *EFM01* USB card supports two endpoints, one for each direction. FX-2 has an integrated USB SIE (Serial Interface Engine) handling USB protocol and transferring user payload data to the appropriate endpoint. So end users do not have to care about USB protocol in their own applications. FX-2 endpoints are realized as 2 kB buffers. These buffers can be accessed over a FIFO-like interface with a 16 bit tristate data bus by external hardware. External hardware acts as a master, polling FIFO flags, applying read- and write-strobes and transferring data. Therefore this FX-2 data transfer mechanism is called “slave FIFO mode”. As already mentioned, all data is transferred in multiples of 512 bytes. External hardware has to ensure, that the data written to IN-endpoint is aligned to this value, so that data will be transmitted from endpoint buffer to host. The 512 byte alignment normally causes no restrictions in data streaming applications with endless data transfers. Maybe it is necessary to fill up endpoint buffer with dummy data, if some kind of host timeout condition has to be met. Another FX-2 data transfer mechanism is called “GPIF (General Programmable InterFace) mode”. The GPIF engine inside the FX-2 acts as a master to endpoint buffers, transferring data and presenting configurable handshake waveforms to external hardware. CESYS USB card supports “slave FIFO mode” for data communication only. “GPIF mode” is exclusively used for downloading configuration bitstreams to FPGA.

Clocking FPGA designs

The 48 MHz SYCLK oscillator is the only onboard clock source for the FPGA. It is used as interface clock (IFCLK) between FX-2 slave FIFO bus and FPGA I/Os, too. So this clock source must be used for data transfers to and from FPGA over USB! Appropriate timing constraints can be found in “*.ucf”-files of design examples included in delivery.

It is strictly recommended to use a single clock domain whenever possible. Using a fully synchronous system architecture often results in smaller, less complex and more performant FPGA designs (compare Xilinx™ white paper [WP331](#) “Timing Closure/Coding Guidelines”).

In FPGA designs with multiple clock domains asynchronous FIFOs have to be used for transferring data from one clock domain to the other and comprehensive control signals have to be resynchronized.

Other clock sources can be added internally by using Spartan-3E™ onchip digital clock managers (DCMs) or externally by connecting clock sources to FPGA-I/Os. Recommended I/Os are additionally labeled with “RHCLK<>” and “LHCLK<>”. A wide range of clock frequencies can be synthesized with DCMs. For further details on DCMs please see “[Spartan-3™ Generation FPGA User Guide UG331](#)”, “[Spartan-3E™ FPGA Family: Complete Data Sheet DS312](#)” and Xilinx™ application note “[Using Digital Clock Managers \(DCMs\) in Spartan-3™ FPGAs XAPP462](#)”.

FX-2/FPGA slave FIFO connection

Only the logical behavior of slave FIFO interface is discussed here. For information about the timing behavior like setup- and hold-times please see FX-2 datasheet (cy7c68013a_8.pdf).

All flags and control signals are active low (postfix “#”). The whole interface is synchronous to IFCLK. The asynchronous FIFO transfer mode is not supported.

- SLWR#: FX-2 input, FIFO write-strobe
- SLRD#: FX-2 input, FIFO read-strobe
- SLOE#: FX-2 input, output-enable, activates FX-2 data bus drivers
- PKTEND#: FX-2 input, packet end control signal, causes FX-2 to send data to host at once, ignoring 512 byte alignment (so called “short packet”)
- Short packets sometimes lead to unpredictable behavior at host side. So *EFM01* does not support short packets! This signal has to be statically set to HIGH! Dummy data should be added instead of creating short packets. There is normally no lack of performance by doing this, because transmission of USB packets is bound to a time framing scheme, regardless of amount of payload data.
- FIFOADR[1:0]: FX-2 input, endpoint buffer addresses, *EFM01* uses only two endpoints EP2 (OUT, ADR[1:0] = b”00”) and EP6 (IN, ADR[1:0] = b”10”)
- Switching FIFOADR[1] is enough to select data direction. FIFOADR[0] has to be statically set to LOW!
- FLAG#-A/-B/-C: FX-2 outputs, A => EP2 “empty” flag, B => EP2 “almost empty” flag, meaning one 16 bit data word is available, C => EP6 “almost full” flag, meaning one 16 bit data word can still be transmitted to EP6, there is no real “full” flag for EP6, “almost full” could be used instead
- FD[15:0]: bidirectional tristate data bus

Introduction to example FPGA designs

The CESYS *EFM01* Card is shipped with some demonstration FPGA designs to give you an easy starting point for own development projects. The whole source code is written in VHDL. Verilog and schematic entry design flows are not supported.

- The design “efm01” demonstrates the implementation of a system-on-chip (SOC) with host software access to the peripherals like GPIOs, Flash Memory and BlockRAM over USB. This design requires a protocol layer over the simple USB bulk transfer (see CESYS application note “Transfer Protocol for CESYS USB products” for details), which is already provided by CESYS software API.
- The design “efm01_perf” allows high speed data transfers from and to the FPGA over USB and can be used for software benchmarking purposes. This design uses 512 byte aligned USB bulk transfer without additional protocol layer only.

The Spartan-3E XC3S500E Device is supported by the free Xilinx™ ISE Webpack development software. You will have to change some options of the project properties for own applications.

A bitstream in the “*.bin”-format is needed, if you want to download your FPGA design with the CESYS software API-functions `LoadBIN()` and `ProgramFPGA()`. The generation of this file is disabled by default in the Xilinx™ ISE development environment. Check “create binary configuration file” at right click “generate programming file”=>properties=>general options:

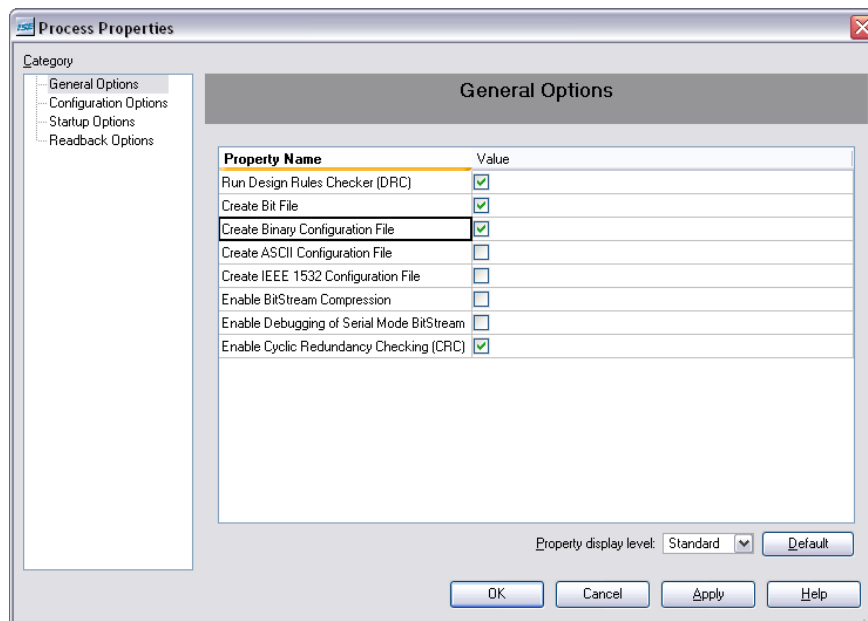


Figure 5: ISE Generate Programming File Properties (Gen. Opt.)

After `ProgramFPGA()` is called and the FPGA design is completely downloaded, the pin `#FPGA_RESET` (note: the prefix `#` means, that the signal is active low) is automatically pulsed (HIGH/LOW/HIGH). This signal can be used for resetting the FPGA design. The API-function `ResetFPGA()` can be called to initiate a pulse on `#FPGA_RESET` at a user given time.

The following sections will give you a brief introduction about the data transfer from and to the FPGA over the Cypress FX-2 USB peripheral controller's slave FIFO interface, the WISHBONE interconnection architecture and the provided peripheral controllers.

The *EFM01* uses only slave FIFO mode for transferring data.

For further information about the FX-2 slave FIFO mode see Cypress FX-2 user manual (EZ-USB_TRM.pdf) and datasheet (cy7c68013a_8.pdf) and about the WISHBONE architecture see specification B.3 (wbspec_b3.pdf).

FPGA source code copyright information

This source code is copyrighted by CESYS GmbH / GERMANY, unless otherwise noted.

FPGA source code license

THIS SOURCECODE IS NOT FREE! IT IS FOR USE TOGETHER WITH THE CESYS EFM01 USB CARD (ARTICLE-NR.: C1050-4107) ONLY! YOU ARE NOT ALLOWED TO MODIFY AND DISTRIBUTE OR USE IT WITH ANY OTHER HARDWARE, SOFTWARE OR ANY OTHER KIND OF ASIC OR PROGRAMMABLE LOGIC DESIGN WITHOUT THE EXPLICIT PERMISSION OF THE COPYRIGHT HOLDER!

Disclaimer of warranty

THIS SOURCECODE IS DISTRIBUTED IN THE HOPE THAT IT WILL BE USEFUL, BUT THERE IS NO WARRANTY OR SUPPORT FOR THIS SOURCECODE. THE COPYRIGHT HOLDER PROVIDES THIS SOURCECODE "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THIS SOURCECODE IS WITH YOU. SHOULD THIS SOURCECODE PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

IN NO EVENT WILL THE COPYRIGHT HOLDER BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THIS SOURCECODE (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THIS SOURCECODE TO OPERATE WITH ANY OTHER SOFTWARE-PROGRAMS, HARDWARE-CIRCUITS OR ANY OTHER KIND OF ASIC OR PROGRAMMABLE LOGIC DESIGN), EVEN IF THE COPYRIGHT HOLDER HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Design “efm01”

An on-chip-bus system is implemented in this design. The VHDL source code shows you, how to build a 32 Bit WISHBONE based shared bus architecture. All devices of the WISHBONE system support only SINGLE READ / WRITE Cycles. Files and modules having something to do with the WISHBONE system are labeled with the prefix “wb_”. The WISHBONE master is labeled with the additional prefix “ma_” and the slaves are labeled with “sl_”.

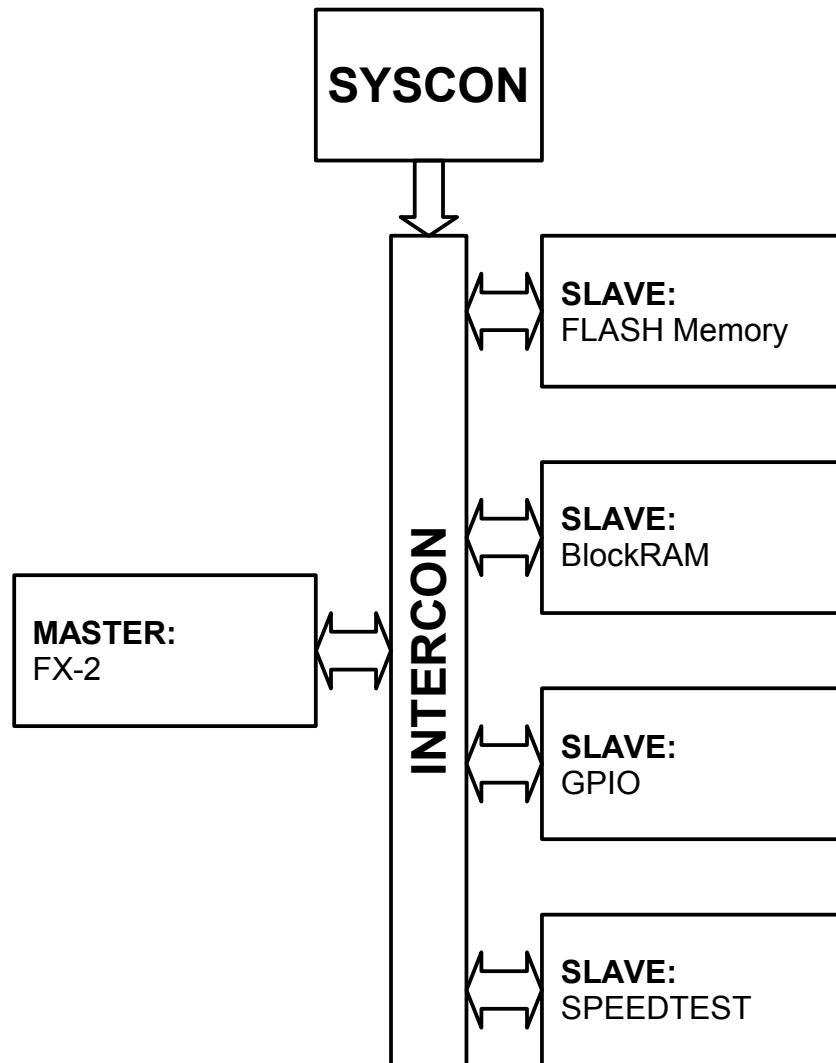


Figure 6: WISHBONE system overview

Files and modules

src/wishbone.vhd:

A package containing datatypes, constants, components, signals and information for software developers needed for the WISHBONE system. You will find C/C++-style “#define”s with important addresses and values to copy and paste into your software source code after VHDL comments (“--”).

src/efm01_top.vhd:

This is the top level entity of the design. The WISHBONE components are instantiated here.

src/wb_syscon.vhd:

This entity provides the WISHBONE system signals RST and CLK. It uses #FPGA_RESET and SYSCCLK as external reset and clock source. SYSCCLK is identically to FX2_IFCLK. That means FX-2 slave FIFO interface and WISHBONE system are fully synchronous.

src/wb_intercon.vhd:

All WISHBONE devices are connected to this shared bus interconnection logic. Some MSBs of the address are used to select the appropriate slave.

src/wb_ma_fx2.vhd:

This is the entity of the WISHBONE master, which converts the CESYS USB protocol into one or more 32 Bit single read/write WISHBONE cycles. The low level FX-2 slave FIFO controller (fx2_slfifo_ctrl.vhd) is used and 16/32 bit data width conversion is done by using special FIFOs (sfifo_hd_a1Kx18b0K5x36.vhd).

src/wb_sl_bram.vhd:

A internal BlockRAM is instantiated here and simply connected to the WISHBONE architecture. It can be used for testing address oriented data transactions over USB.

src/wb_sl_speedtest.vhd:

A single register with zero delay slave handshake response. It can be used for benchmarking purposes. Auto address increment must be deactivated.

src/wb_sl_gpio.vhd:

This entity controls the signals at connectors J3 and J4. 50 I/Os can be used as general purpose I/Os. Each of these I/Os can be configured as an in- or output. Additional pinout information is provided by an embedded comma separated values file after VHDL

comments (“--”).

src/wb_sl_flash.vhd:

The module encapsulates the low level FLASH controller flash_ctrl.vhd. The integrated command register supports the BULK ERASE command, which erases the whole memory by programming all bits to '1'. In write cycles the bit values can only be changed from '1' to '0'. That means, that it is not allowed to have a write access to the same address twice without erasing the whole flash before. The read access is as simple as reading from any other WISHBONE device. Please see the SPI-FLASH data sheet (m25p40.pdf) for details on programming and erasing. It is used for programming FPGA configuration bitstream to SPI-FLASH.

src/fx2_slfifo_ctrl.vhd:

This controller copies data from FX-2 endpoints to internal FPGA buffers (sync_fifo16.vhd) and vice versa.

src/sync_fifo16.vhd:

This entity is a general purpose synchronous FIFO buffer with 15 data entries. It is build of FPGA distributed RAM.

src/sfifo_hd_a1Kx18b0K5x36.vhd:

This entity is a general purpose synchronous FIFO buffer with mismatched port widths. It is build of a FPGA BlockRAM.

src/flash_ctrl.vhd:

The low level FLASH controller for the 4MBit SPI FLASH memory. It supports reading and writing of four bytes of data at one time as well as erasing the whole memory.

efm01.ise:

Project file for Xilinx™ ISE

efm01.ucf:

User constraint file with timing and pinout constraints

WISHBONE transactions

The software API-functions `ReadRegister()`, `WriteRegister()` lead to one and `ReadBlock()`, `WriteBlock()` to several consecutive WISHBONE single cycles. Bursting is not allowed in the WISHBONE demo application. The address can be incremented automatically in block transfers. You can find details on enabling/disabling the

burst mode and address auto-increment mode in the CESYS application note “Transfer Protocol for CESYS USB products” and software API documentation.

CESYS USB transfer protocol is converted into one or more WISHBONE data transaction cycles. So the FX-2 becomes a master device in the internal WISHBONE architecture. Input signals for the WISHBONE master are labeled with the postfix “_I”, output signals with “_O”.

WISHBONE signals driven by the master:

- STB_O: strobe, qualifier for the other output signals of the master, indicates valid data and control signals
- WE_O: write enable, indicates, if a write or read cycle is in progress
- ADR_O[31:0]: 32-Bit address bus, the software uses BYTE addressing, but the WISHBONE system uses DWORD (32-Bit) addressing. The address is shifted two bits inside the WISHBONE master module
- DAT_O[31:0]: 32-Bit data out bus for data transportation from master to slaves

WISHBONE signals driven by slaves:

- DAT_I[31:0]: 32-Bit data in bus for data transportation from slaves to master
- ACK_I: handshake signal, slave devices indicate a successful data transfer for writing and valid data on bus for reading by asserting this signal, slaves can insert wait states by delaying this signal, it is possible to assert ACK_I in first clock cycle of STB_O assertion using a combinatorial handshake to transfer data in one clock cycle (recommendation: registered feedback handshake should be used in applications, where maximum data throughput is not needed, because timing specs are easier to meet)

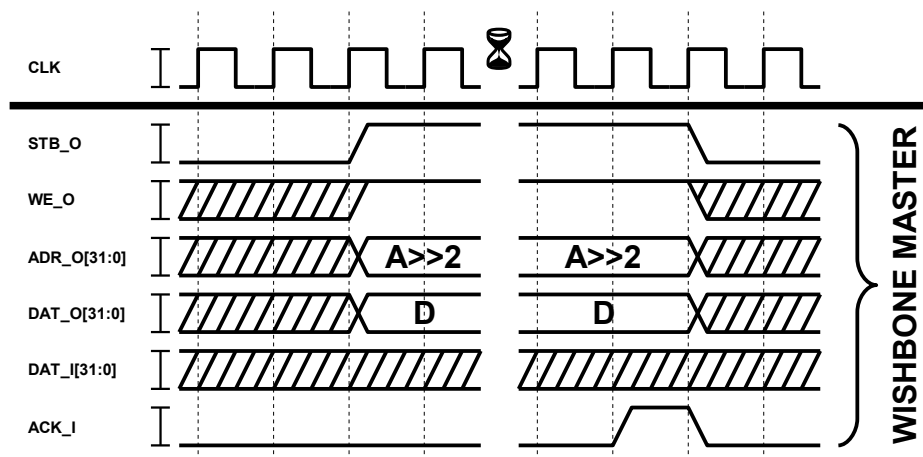


Figure 7: WISHBONE transactions with `WriteRegister()` and `WriteBlock()`

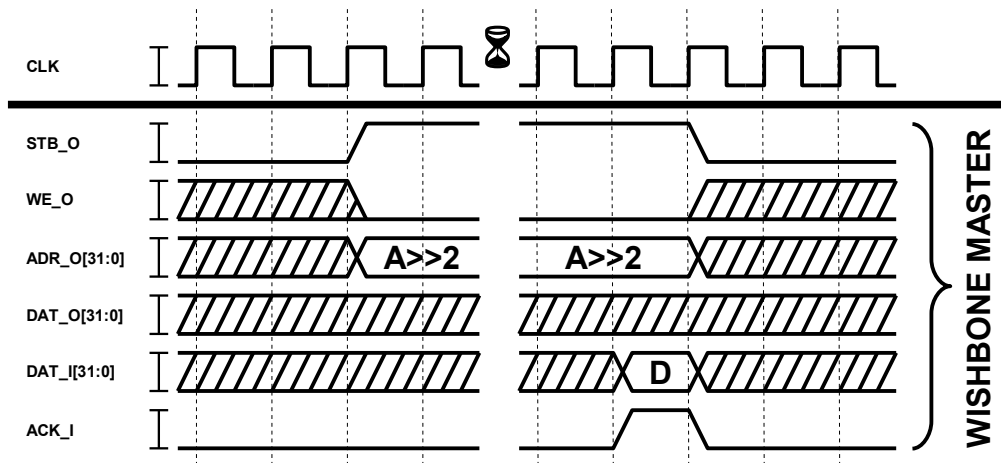


Figure 8: WISHBONE transactions with `ReadRegister()` and `ReadBlock()`

The WISHBONE signals in these illustrations and explanations are shown as simple bit types or bit vector types, but in the VHDL code these signals could be encapsulated in extended data types like arrays or records.

Example:

```
...
port map
(
    ...
    ACK_I => intercon.masters.slave(2).ack,
    ...
)
```

Port ACK_I is connected to signal ack of element 2 of array slave, of record masters, of record intercon.

Design “efm01_perf”

This design is intended to demonstrate behavior of low level slave FIFO controller entity `fx2_slfifo_ctrl`. It handles the FX-2 slave FIFO interface. It can be synthesized in two modes, data loopback mode and infinite data source/sink mode with 16 bit counting data source. Ports of `fx2_slfifo_ctrl` connected to FX-2 are labeled with prefix `fx2_` and ports connected to user logic are labeled with prefix `app_`. Sometimes the abbreviations `_h2p_` (host to peripheral) and `_p2h_` (peripheral to host) are used in signal names to indicate data flow direction.

Files and modules

src/efm01_perf.vhd:

This is the top level module. It instantiates the low level slave FIFO controller (`fx2_slfifo_ctrl.vhd`). A generic variable selects between data loopback and infinite data mode at synthesis time.

src/fx2_slfifo_ctrl.vhd:

See chapter “Design *efm01*”

src/sync_fifo16.vhd:

See chapter “Design *efm01*”

efm01_perf.ise:

Project file for Xilinx™ ISE.

efm01_perf.ucf:

User constraint file with timing and pinout constraints.

Slave FIFO transactions

The software API functions `ReadBulk()` and `WriteBulk()` lead to 512 byte aligned USB bulk transfers without CESYS USB transfer protocol. So it is possible to achieve maximum data rates over USB. `fx2_slfifo_ctrl` checks FX-2 FIFO flags and copies data from FX-2 endpoint buffers to FPGA and vice versa. So the USB data link looks like any other FPGA FIFO buffer to user logic.

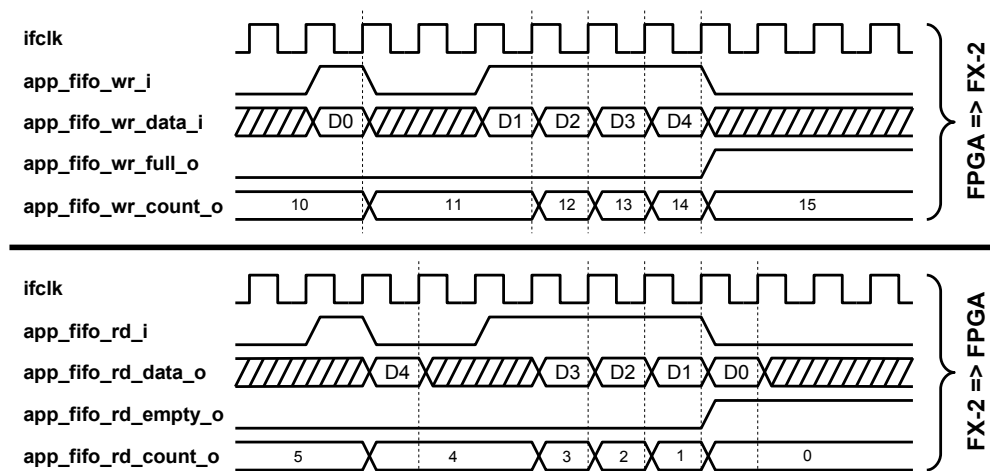


Figure 9: FIFO transactions with `ReadBulk()` and `WriteBulk()` at user logic side

The upper waveform demonstrates the behavior of `app_fifo_wr_full_o` and `app_fifo_wr_count_o` when there is no transaction on the slave FIFO controller side of the FIFO. During simultaneous FIFO-read- and FIFO-write-transactions, the signals do not change. The signal `app_fifo_wr_full_o` will be cleared and `app_fifo_wr_count_o` will decrease, if there are read-transactions at the slave FIFO controller side, but no write-transactions at the application side.

The lower waveform demonstrates the behavior of `app_fifo_rd_empty_o` and `app_fifo_rd_count_o` when there is no transaction at the slave FIFO controller side of the FIFO. During simultaneous FIFO-read- and FIFO-write-transactions, the signals do not change. The signal `app_fifo_rd_empty_o` will be cleared and `app_fifo_rd_count_o` will increase, if there are write-transactions on the slave FIFO controller side, but no read-transactions at the application side. Please note the one clock-cycle delay between `app_fifo_rd_i` and `app_fifo_rd_data_o`!

The signals `app_usb_h2p_pktcount_o[7:0]` and `app_usb_p2h_pktcount_o[7:0]` (not shown in figure 6) are useful to fit the 512 byte USB bulk packet alignment. They are automatically incremented, if the appropriate read- (`app_fifo_rd_i`) or write-strobe (`app_fifo_wr_i`) is asserted. These signals count 16 bit data words, not data bytes! 512 byte alignment is turned into a 256 16 bit word alignment at this interface.

Software

Introduction

The UDK (Unified Development Kit) is used to allow developers to communicate with Cesium's USB and PCI(e) devices. Older releases were just a release of USB and PCI drivers plus API combined with some shared code components. The latest UDK combines all components into one single C++ project and offers interfaces to C++, C and for .NET (Windows only). The API has functions to mask-able enumeration, unique device identification (runtime), FPGA programming and 32bit bus based data communication. PCI devices have additional support for interrupts.

Changes to previous versions

Beginning with release 2.0, the UDK API is a truly combined interface to Cesium's USB and PCI devices. The class interface from the former USBUni and PCIBase API's was saved at a large extend, so porting applications from previous UDK releases can be done without much work.

Here are some notes about additional changes:

- Complete rewrite
- Build system cleanup, all UDK parts (except .NET) are now part of one large project
- 64 bit operating system support
- UDK tools combined into one application (UDKLab)
- Updated to latest PLX SDK (6.31)
- Identical C, C++ and .NET API interface (.NET ⇒ Windows only)
- Different versions of components collapsed to one UDK version
- Windows only:
 - Microsoft Windows Vista / Seven(7) support (PCI drivers are not released for Seven at the moment)
 - Driver installation / update is done by an installer now
 - Switched to Microsoft's generic USB driver (WinUSB)
 - Support moved to Visual Studio 2005, 2008 and 2010(experimental), older Visual Studio versions are not supported anymore
- Linux only:
 - Revisited USB driver, tested on latest Ubuntu distributions (32/64)
 - Simpler USB driver installation

Windows

Requirements

To use the UDK in own projects, the following is required:

- Installed drivers
- Microsoft Visual Studio 2005 or 2008; 2010 is experimental
- CMake 2.6 or higher ⇒ <http://www.cmake.org>
- wxWidgets 2.8.10 or higher (must be build separately) ⇒ <http://www.wxwidgets.org>
[optionally, only if UDKLab should be build]

Driver installation

The driver installation is part of the UDK installation but can run standalone on final customer machines without the need to install the UDK itself. During installation, a choice of drivers to install can be made, so it is not necessary to install i.e. PCI drivers on machines that should run USB devices only or vice versa. If USB drivers get installed on a machine that has a pre-2.0 UDK driver installation, we prefer the option for USB driver cleanup offered by the installer, this cleanly removes all dependencies of the old driver installation.

Note: There are separate installers for 32 and 64 bit systems.

Important: At least one device should be present when installing the drivers !

Build UDK

Prerequisites

The most components of the UDK are part of one large CMake project. There are some options that need to be fixed in *msvc.cmake* inside the UDK installation root:

- **BUILD_UI_TOOLS** If 0, UDKLab will not be part of the subsequent build procedure, if 1 it will. This requires an installation of an already built wxWidgets.
- **WX_WIDGETS_BASE_PATH** Path to wxWidgets build root, only needed if **BUILD_UI_TOOLS** is not 0.
- **USE_STATIC_RTL** If 0, all projects are build against the dynamic runtime libraries. This requires the installation of the appropriate Visual Studio redistributable pack on every machine the UDK is used on. Using a static build does not create such dependencies, but will conflict with the standard wxWidgets build configuration.

Solution creation and build

The preferred way is to open a command prompt inside the installation root of the UDK,

lets assume to use *c:\udkapi*.

```
c:
cd \udkapi
```

CMake allows the build directory separated to the source directory, so it's a good idea to do it inside an empty sub-directory:

```
mkdir build
cd build
```

The following code requires an installation of CMake and at least one supported Visual Studio version. If CMake isn't included into the **PATH** environment variable, the path must be specified as well:

```
cmake ..
```

This searches the preferred Visual Studio installation and creates projects for it. Visual Studio Express users may need to use the command prompt offered by their installation. If multiple Visual Studio versions are installed, CMake's command parameter '-G' can be used to specify a special one, see CMake's documentation in this case. This process creates the solution files inside *c:\udkapi\build*. All subsequent tasks can be done in Visual Studio (with the created solution), another invocation of cmake isn't necessary under normal circumstances.

Important: The UDK C++ API must be build with the same toolchain and build flags like the application that uses it. Otherwise unwanted side effects in exception handling will occur ! (See example in *Add project to UDK build*).

Info: It is easy to create different builds with different Visual Studio versions by creating different build directories and invoke CMake with different '-G' options inside them:

```
c:
cd \udkapi
mkdir build2005
cd build2005
cmake -G"Visual Studio 8 2005" ..
cd ..
mkdir build2008
cd build2008
cmake -G"Visual Studio 9 2008" ..
```

Linux

There are too many distributions and releases to offer a unique way to the UDK installation. We've chosen to work with the most recent Ubuntu release, 9.10 at the moment. All commands are tested on an up to date installation and may need some tweaking on other systems / versions.

Requirements

- GNU C++ compiler toolchain
- zlib development libraries
- CMake 2.6 or higher ⇒ <http://www.cmake.org>
- wxWidgets 2.8.10 or higher ⇒ <http://www.wxwidgets.org> [optionally, only if UDKLab should be build]

```
sudo apt-get install build-essential cmake zlib1g-dev libwxbase2.8-dev  
libwxgtk2.8-dev
```

The Linux UDK comes as gzip'ed tar archive, as the Windows installer won't usually work. The best way is to extract it to the home directory:

```
tar xzvf UDKAPI-x.x.tgz ~/
```

This creates a directory */home/[user]/udkapi[version]* which is subsequently called *udkroot*. The following examples assume an installation root in *~/udkapi2.0*.

Important: Commands sometimes contain a ``` symbol, have attention to use the right one, refer to command substitution if not familiar with.

Drivers

The driver installation on Linux systems is a bit more complicated than on Windows systems. The drivers must be build against the installed kernel version. Updating the kernel requires a rebuild.

USB

As the USB driver is written by Cesys, the installation procedure is designed to be as simple and automated as possible. The sources and support files reside in directory *<udkroot>/drivers/linux/usb*. Just go there and invoke *make*.

```
cd ~/udkapi2.0/drivers/linux/usb  
make
```

If all external dependencies are met, the build procedure should finish without errors. Newer kernel releases may change things which prevent success, but it is out of the scope of our possibilities to be always up-to-date with latest kernels. To install the driver, the

following command has to be done:

```
sudo make install
```

This will do the following things:

- Install the kernel module inside the module library path, update module dependencies
- Install a new udev rule to give device nodes the correct access rights (0666) (/etc/udev/rules.d/99-ceusbuni.rules)
- Install module configuration file (/etc/dev/modprobe.d/ceusbuni.conf)
- Start module

If things work as intended, there must be an entry */proc/ceusbuni* after this procedure.

The following code will completely revert the above installation (called in same directory):

```
sudo make remove
```

The configuration file, */etc/modprobe.d/ceusbuni.conf*, offers two simple options (Read the comments in the file):

- Enable kernel module debugging
- Choose between firmware which automatically powers board peripherals or not

Changing these options require a module reload to take affect.

PCI

The PCI drivers are not created or maintained by Cesium, they are offered by the manufacturer of the PCI bridges that were used on Cesium PCI(e) boards. So problems regarding them can't be handled or supported by us.

Important: If building PlxSdk components generate the following error / warning:

```
/bin/sh [[: not found
```

Here's a workaround: The problem is Ubuntu's default usage of *dash* as *sh*, which can't handle command *[[*. Replacing *dash* with *bash* is accomplished by the following commands that must be done as root:

```
sudo rm /bin/sh
sudo ln -s /bin/bash /bin/sh
```

Installation explained in detail:

PlxSdk decompression:

```
cd ~/udkapi2.0/drivers/linux
tar xvf PlxSdk.tar
```

Build drivers:

```
cd PlxSdk/Linux/Driver
PLX_SDK_DIR=`pwd`/../../ ./buildalldrivers
```

Loading the driver manually requires a successful build, it is done using the following commands:

```
cd ~/udkapi2.0/drivers/linux/PlxSdk
sudo PLX_SDK_DIR=`pwd` Bin/Plx_load Svc
```

PCI based boards like the **PCIS3Base** require the following driver:

```
sudo PLX_SDK_DIR=`pwd` Bin/Plx_load 9056
```

PCIe based boards like the **PCleV4Base** require the following:

```
sudo PLX_SDK_DIR=`pwd` Bin/Plx_load 8311
```

Automation of this load process is out of the scope of this document.

Build UDK

Prerequisites

The whole UDK will be build using CMake, a free cross platform build tool. It creates dynamic Makefiles on unix compatible platforms.

The first thing should be editing the little configuration file *linux.cmake* inside the installation root of the UDK. It contains the following options:

- **BUILD_UI_TOOLS** If 0 UDKLab isn't build, if 1 UDKLab is part of the build, but requires a compatible wxWidgets installation.
- **CMAKE_BUILD_TYPE** Select build type, can be one of *Debug*, *Release*, *RelWithDebInfo*, *MinSizeRel*. If there should be at least 2 builds in parallel, remove this line and specify the type using command line option *-DCMAKE_BUILD_TYPE=....*

Makefile creation and build

Best usage is to create an empty build directory and run cmake inside of it:

```
cd ~/udkapi2.0
mkdir build
cd build
cmake ..
```

If all external dependencies are met, this will finish creating a Makefile. To build the UDK, just invoke make:

```
make
```

Important: The UDK C++ API must be build with the same toolchain and build flags like

the application that uses it. Otherwise unwanted side effects in exception handling will occur ! (See example in *Add project to UDK build*).

Use APIs in own projects

C++ API

- Include file: `udkapi.h`
- Library file:
 - Windows: `udkapi_vc[ver]_[arch].lib`, [ver] is 8, 9, 10, [arch] is *x86* or *amd64*, resides in *lib/[build]/*
 - Linux: `libusbapi.so`, resides in *lib/*
- Namespace: `ceUDK`

As this API uses exceptions for error handling, it is really important to use the same compiler and build settings which are used to build the API itself. Otherwise exception based stack unwinding may cause undefined side effects which are really hard to fix.

Add project to UDK build

A simple example would be the following. Let's assume there's a source file *mytest/mytest.cpp* inside UDK's root installation. To build a *mytestexe* executable with UDK components, those lines must be appended:

```
add_executable(mytestexe mytest/mytest.cpp)
target_link_libraries(mytestexe ${UDKAPI_LIBNAME})
```

Rebuilding the UDK with these entries in Visual Studio will create a new project inside the solution (and request a solution reload). On Linux, calling *make* will just include *mytestexe* into the build process.

C API

- Include file: `udkapi.h`
- Library file:
 - Windows: `udkapi_vc[ver]_[arch].lib`, [ver] is 8, 9, 10, [arch] is *x86* or *amd64*, resides in *lib/[build]/*
 - Linux: `libusbapi.so`, resides in *lib/*
- Namespace: Not applicable

The C API offers all functions from a dynamic link library (Windows: *.dll*, Linux: *.so*) and uses standardized data types only, so it is usable in a wide range of environments.

Adding it to the UDK build process is nearly identical to the C++ API description, except that `${UDKAPIC_LIBNAME}` must be used.

.NET API

- Include file: -
- Library file: `udkapinet.dll`, resided in *bin/[build]*
- Namespace: `cesys.ceUDK`

The .NET API, as well as its example application is separated from the normal UDK build. First of all, CMake doesn't have native support .NET, as well as it is working on Windows systems only. Building it has no dependency to the standard UDKAPI, all required sources are part of the .NET API project. The Visual Studio solution is located in directory *dotnet/* inside the UDK installation root. It is a Visual Studio 8/2005 solution and should be convertible to newer releases. The solution is split into two parts, the .NET API in mixed native/managed C++ and an example written in C#.

To use the .NET API in own projects, it's just needed to add the generated DLL *udkapinet.dll* to the projects references.

API Functions in detail

Notice: To prevent overhead in most usual scenarios, the API does not serialize calls in any way, so the API user is responsible to serialize call if used in a multi-threaded context !

Notice: The examples for .NET in the following chapter are in C# coding style.

API Error handling

Error handling is offered very different. While both C++ and .NET API use exception handling, the C API uses a classical return code / error inquiry scheme.

C++ and .NET API

UDK API code should be embedded inside a try branch and exceptions of type *ceException* must be caught. If an exception is raised, the generated exception object offers methods to get detailed information about the error.

C API

All UDK C API functions return either *CE_SUCCESS* or *CE_FAILED*. If the latter is returned, the functions below should be invoked to get the details of the error.

Methods/Functions

GetLastErrorCode

API	Code
C++	unsigned int ceException::GetErrorCode()
C	unsigned int GetLastErrorCode()
.NET	uint ceException.GetLastErrorCode()

Returns an error code which is intended to group the error into different kinds. It can be one of the following constants:

Error code	Kind of error
ceE_TIMEOUT	Errors with any kind of timeout.
ceE_IO_ERROR	IO errors of any kind, file, hardware, etc.
ceE_UNEXP_HW_BEH	Unexpected behavior of underlying hardware (no response, wrong data).
ceE_PARAM	Errors related to wrong call parameters (NULL pointers, ...).
ceE_RESOURCE	Resource problem, wrong file format, missing dependency.
ceE_API	Undefined behavior of underlying API.
ceE_ORDER	Wrong order calling a group of code (i.e. deinit()→init()).
ceE_PROCESSING	Occurred during internal processing of anything.
ceE_INCOMPATIBLE	Not supported by this device.
ceE_OUTOFMEMORY	Failure allocating enough memory.

GetLastErrorText

API	Code
C++	const char *ceException::GetLastErrorText()
C	const char *GetLastErrorText()
.NET	string ceException.GetLastErrorText()

Returns a text which describes the error readable by the user. Most of the errors contain problems meant for the developer using the UDK and are rarely usable by end users. In most cases unexpected behavior of the underlying operation system or in data transfer is reported. (All texts are in english.)

Device enumeration

The complete device handling is done by the API internally. It manages the resources of all enumerated devices and offers either a device pointer or handle to API users. Calling Init() prepares the API itself, while Delnit() does a complete cleanup and invalidates all device pointers and handles.

To find supported devices and work with them, Enumerate() must be called after Init(). Enumerate() can be called multiple times for either finding devices of different types or to find newly plugged devices (primary USB at the moment). One important thing is the following: Enumerate() does **never** remove a device from the internal device list and so invalidate any pointer, it just add new ones or does nothing, even if a USB device is removed. For a clean detection of a device removal, calling Delnit(), Init() and Enumerate() (in exactly that order) will build a new, clean device list, but invalidates all previous created device pointers and handles.

To identify devices in a unique way, each device gets a UID, which is a combination of device type name and connection point, so even after a complete cleanup and new enumeration, devices can be exactly identified by this value.

Methods/Functions

Init

API	Code
C++	static void ceDevice::Init()
C	CE_RESULT Init()
.NET	static void ceDevice.Init()

Prepare internal structures, must be the first call to the UDK API. Can be called after invoking Delnit() again, see top of this section.

Delnit

API	Code
C++	static void ceDevice::Delnit()
C	CE_RESULT Delnit()
.NET	static void ceDevice.Delnit()

Free up all internal allocated data, there must no subsequent call to the UDK API after this call, except Init() is called again. All retrieved device pointers and handles are invalid after this point.

Enumerate

API	Code
C++	static void ceDevice::Enumerate(ceDevice::ceDeviceType DeviceType)
C	CE_RESULT Enumerate(unsigned int DeviceType)
.NET	static void ceDevice.Enumerate(ceDevice.ceDeviceType DeviceType)

Search for (newly plugged) devices of the given type and add them to the internal list. Access to this list is given by GetDeviceCount() / GetDevice(). DeviceType can be one of the following:

DeviceType	Description
ceDT_ALL	All UDK supported devices.
ceDT_PCI_ALL	All UDK supported devices on PCI bus.
ceDT_PCI_PCIS3BASE	Cesys PCIS3Base
ceDT_PCI_DOB	DOB (*)
ceDT_PCI_PCIEV4BASE	Cesys PCIeV4Base
ceDT_PCI_RTC	RTC (*)
ceDT_PCI_PSS	PSS (*)
ceDT_PCI_DEFLECTOR	Deflector (*)
ceDT_USB_ALL	All UDK supported devices.
ceDT_USB_USBV4F	Cesys USBV4F
ceDT_USB_EFM01	Cesys EFM01
ceDT_USB_MISS2	MISS2 (*)
ceDT_USB_CID	CID (*)
ceDT_USB_USBS6	Cesys USBS6

* Customer specific devices.

GetDeviceCount

API	Code
C++	static unsigned int ceDevice::GetDeviceCount()
C	CE_RESULT GetDeviceCount(unsigned int *puiCount)
.NET	static uint ceDevice.GetDeviceCount()

Return count of devices enumerated up to this point. May be larger if rechecked after calling Enumerate() in between.

GetDevice

API	Code
C++	static ceDevice *ceDevice::GetDevice(unsigned int uidx)
C	CE_RESULT GetDevice(unsigned int uidx, CE_DEVICE_HANDLE *pHandle)
.NET	static ceDevice ceDevice.GetDevice(uint uidx)

Get device pointer or handle to the device with the given index, which must be smaller than the device count returned by GetDeviceCount(). This pointer or handle is valid up to the point Delnit() is called.

Information gathering

The functions in this chapter return valuable information. All except `GetUDKVersionString()` are bound to devices and can be used after getting a device pointer or handle from `GetDevice()` only.

Methods/Functions

GetUDKVersionString

API	Code
C++	<code>static const char *ceDevice::GetUDKVersionString()</code>
C	<code>const char *GetUDKVersionString()</code>
.NET	<code>static string ceDevice.GetUDKVersionString()</code>

Return string which contains the UDK version in printable format.

GetDeviceUID

API	Code
C++	<code>const char *ceDevice::GetDeviceUID()</code>
C	<code>CE_RESULT GetDeviceUID(CE_DEVICE_HANDLE Handle, char *pszDest, unsigned int uiDestSize)</code>
.NET	<code>string ceDevice.GetDeviceUID()</code>

Return string formatted unique device identifier. This identifier is in the form of *type@location* while type is the type of the device (i.e. *EFM01*) and location is the position the device is plugged to. For PCI devices, this is a combination of bus, slot and function (PCI bus related values) and for USB devices a path from device to root hub, containing the port of all used hubs. So after re-enumeration or reboot, devices on the same machine can be identified exactly.

Notice C API: `pszDest` is the buffer where the value is stored to, it must be at least of size `uiDestSize`.

GetDeviceName

API	Code
C++	<code>const char *ceDevice::GetDeviceName()</code>
C	<code>CE_RESULT GetDeviceName(CE_DEVICE_HANDLE Handle, char *pszDest, unsigned int uiDestSize)</code>
.NET	<code>string ceDevice.GetDeviceName()</code>

Return device type name of given device pointer or handle.

Notice C API: pszDest is the buffer where the value is stored to, it must be at least of size uiDestSize.

GetBusType

API	Code
C++	ceDevice::ceBusType ceDevice::GetBusType()
C	CE_RESULT GetBusType(CE_DEVICE_HANDLE Handle, unsigned int *puiBusType)
.NET	ceDevice.ceBusType ceDevice.GetBusType()

Return type of bus a device is bound to, can be any of the following:

Constant	Bus
ceBT_PCI	PCI bus
ceBT_USB	USB bus

GetMaxTransferSize

API	Code
C++	unsigned int ceDevice::GetMaxTransferSize()
C	CE_RESULT GetMaxTransferSize(CE_DEVICE_HANDLE Handle, unsigned int *puiMaxTransferSize)
.NET	uint ceDevice.GetMaxTransferSize()

Return count of bytes that represents the maximum in one transaction, larger transfers must be split by the API user.

Using devices

After getting a device pointer or handle, devices can be used. Before transferring data to or from devices, or catching interrupts (PCI), devices must be accessed, which is done by calling `Open()`. All calls in this section require an open device, which must be freed by calling `Close()` after usage.

Either way, after calling `Open()`, the device is ready for communication. As of the fact, that Cesys devices usually have an FPGA on the device side of the bus, the FPGA must be made ready for usage. If this isn't done by loading contents from the on-board flash (not all devices have one), a design must be loaded by calling one of the `ProgramFPGA*()` calls. These call internally reset the FPGA after design download. From now on, data can be transferred.

Important: All data transfer is based on a 32 bit bus system which must be implemented inside the FPGA design. PCI devices support this natively, while USB devices use a protocol which is implemented by Cesys and sits on top of a stable bulk transfer implementation.

Methods/Functions

Open

API	Code
C++	<code>void ceDevice::Open()</code>
C	<code>CE_RESULT Open(CE_DEVICE_HANDLE Handle)</code>
.NET	<code>void ceDevice.Open()</code>

Gain access to the specific device. Calling one of the other functions in this section require a successful call to `Open()`.

Notice: If two or more applications try to open one device, PCI and USB devices behave a bit different. For USB devices, `Open()` causes an error if the device is already in use. PCI allows opening one device from multiple processes. As PCI drivers are not developed by Cesys, it's not possible to us to prevent this (as we see this as strange behavior). The best way to share communication of more than one application with devices would be a client / server approach.

Close

API	Code
C++	<code>void ceDevice::Close()</code>
C	<code>CE_RESULT Close(CE_DEVICE_HANDLE Handle)</code>
.NET	<code>void ceDevice.Close()</code>

Finish working with the given device.

ReadRegister

API	Code
C++	unsigned int ceDevice::ReadRegister(unsigned int uiRegister)
C	CE_RESULT ReadRegister(CE_DEVICE_HANDLE Handle, unsigned int uiRegister, unsigned int *puiValue)
.NET	uint ceDevice.ReadRegister(uint uiRegister)

Read 32 bit value from FPGA design address space (internally just calling ReadBlock() with size = 4).

WriteRegister

API	Code
C++	void ceDevice::WriteRegister(unsigned int uiRegister, unsigned int uiValue)
C	CE_RESULT WriteRegister(CE_DEVICE_HANDLE Handle, unsigned int uiRegister, unsigned int uiValue)
.NET	void ceDevice.WriteRegister(uint uiRegister, uint uiValue)

Write 32 bit value to FPGA design address space (internally just calling WriteBlock() with size = 4).

ReadBlock

API	Code
C++	void ceDevice::ReadBlock(unsigned int uiAddress, unsigned char *pucData, unsigned int uiSize, bool blncAddress)
C	CE_RESULT ReadBlock(CE_DEVICE_HANDLE Handle, unsigned int uiAddress, unsigned char *pucData, unsigned int uiSize, unsigned int uiLncAddress)
.NET	void ceDevice.ReadBlock(uint uiAddress, byte[] Data, uint uiLen, bool blncAddress)

Read a block of data to the host buffer which must be large enough to hold it. The size should never exceed the value retrieved by GetMaxTransferSize() for the specific device. blncAddress is at the moment available for USB devices only. It flags to read all data from the same address instead of starting at it.

WriteBlock

API	Code
C++	void ceDevice::WriteBlock(unsigned int uiAddress, unsigned char *pucData, unsigned int uiSize, bool blncAddress)
C	CE_RESULT WriteBlock(CE_DEVICE_HANDLE Handle, unsigned int uiAddress,

	unsigned char *pucData, unsigned int uiSize, unsigned int uiIncAddress)
.NET	void ceDevice.WriteBlock(uint uiAddress, byte[] Data, uint uiLen, bool blncAddress)

Transfer a given block of data to the 32 bit bus system address uiAddress. The size should never exceed the value retrieved by GetMaxTransferSize() for the specific device. blncAddress is at the moment available for USB devices only. It flags to write all data to the same address instead of starting at it.

WaitForInterrupt

API	Code
C++	bool ceDevice::WaitForInterrupt(unsigned int uiTimeOutMS)
C	CE_RESULT WaitForInterrupt(CE_DEVICE_HANDLE Handle, unsigned int uiTimeOutMS, unsigned int *puiRaised)
.NET	bool ceDevice.WaitForInterrupt(uint uiTimeOutMS)

(PCI only) Check if the interrupt is raised by the FPGA design. If this is done in the time specified by the timeout, the function returns immediately flagging the interrupt is raised (return code / *puiRaised). Otherwise, the function returns after the timeout without signaling.

Important: If an interrupt is caught, EnableInterrupt() must be called again before checking for the next. Besides that, the FPGA must be informed to lower the interrupt line in any way.

EnableInterrupt

API	Code
C++	void ceDevice::EnableInterrupt()
C	CE_RESULT EnableInterrupt(CE_DEVICE_HANDLE Handle)
.NET	void ceDevice.EnableInterrupt()

(PCI only) Must be called in front of calling WaitForInterrupt() and every time an interrupt is caught and should be checked again.

ResetFPGA

API	Code
C++	void ceDevice::ResetFPGA()
C	CE_RESULT ResetFPGA(CE_DEVICE_HANDLE Handle)
.NET	void ceDevice.ResetFPGA()

Pulses the FPGA reset line for a short time. This should be used to sync the FPGA design with the host side peripherals.

ProgramFPGAFromBIN

API	Code
C++	void ceDevice::ProgramFPGAFromBIN(const char *pszFileName)
C	CE_RESULT ProgramFPGAFromBIN(CE_DEVICE_HANDLE Handle, const char *pszFileName)
.NET	void ceDevice.ProgramFPGAFromBIN(string sFileName)

Program the FPGA with the Xilinx tools .bin file indicated by the filename parameter. Calls ResetFPGA() subsequently.

ProgramFPGAFromMemory

API	Code
C++	void ceDevice::ProgramFPGAFromMemory(const unsigned char *pszData, unsigned int uiSize)
C	CE_RESULT ProgramFPGAFromMemory(CE_DEVICE_HANDLE Handle, const unsigned char *pszData, unsigned int uiSize)
.NET	void ceDevice.ProgramFPGAFromMemory(byte[] Data, uint Size)

Program FPGA with a given array created with UDKLab. This was previously done using fpgaconv.

ProgramFPGAFromMemoryZ

API	Code
C++	void ceDevice::ProgramFPGAFromMemoryZ(const unsigned char *pszData, unsigned int uiSize)
C	CE_RESULT ProgramFPGAFromMemoryZ(CE_DEVICE_HANDLE Handle, const unsigned char *pszData, unsigned int uiSize)
.NET	void ceDevice.ProgramFPGAFromMemoryZ(byte[] Data, uint Size)

Same as ProgramFPGAFromMemory(), except the design data is compressed.

SetTimeOut

API	Code
C++	void ceDevice::SetTimeOut(unsigned int uiTimeOutMS)
C	CE_RESULT SetTimeOut(CE_DEVICE_HANDLE Handle, unsigned int uiTimeOutMS)
.NET	void ceDevice.SetTimeOut(uint uiTimeOutMS)

Set the timeout in milliseconds for data transfers. If a transfer is not completed inside this timeframe, the API generates a timeout error.

EnableBurst

API	Code
C++	void ceDevice::EnableBurst(bool bEnable)
C	CE_RESULT EnableBurst(CE_DEVICE_HANDLE Handle, unsigned int uiEnable)
.NET	void ceDevice.EnableBurst(bool bEnable)

(PCI only) Enable bursting in transfer, which frees the shared address / data bus between PCI(e) chip and FPGA by putting addresses on the bus frequently only.

UDKLab

Introduction

UDKLab is a replacement of the former cesys-Monitor, as well as cesys-Lab and fpgaconv. It is primary targeted to support FPGA designers by offering the possibility to read and write values from and to an active design. It can further be used to write designs onto the device's flash, so FPGA designs can load without host intervention. Additionally, designs can be converted to C/C++ and C# arrays, which allows design embedding into an application.

The main screen

The following screen shows an active session with an EFM01 device. The base view is intended to work with a device, while additional functionality can be found in the tools menu.

The left part of the screen contains the device initialization details, needed to prepare the FPGA with a design (or just a reset if loaded from flash), plus optional register writes for preparation of peripheral components.

The right side contains elements for communication with the FPGA design:

- Register read and write, either by value or bit-wise using checkboxes.
- Live update of register values.
- Data areas (like RAM or Flash) can be filled from file or read out to file.
- Live view of data areas.
- More on these areas below.

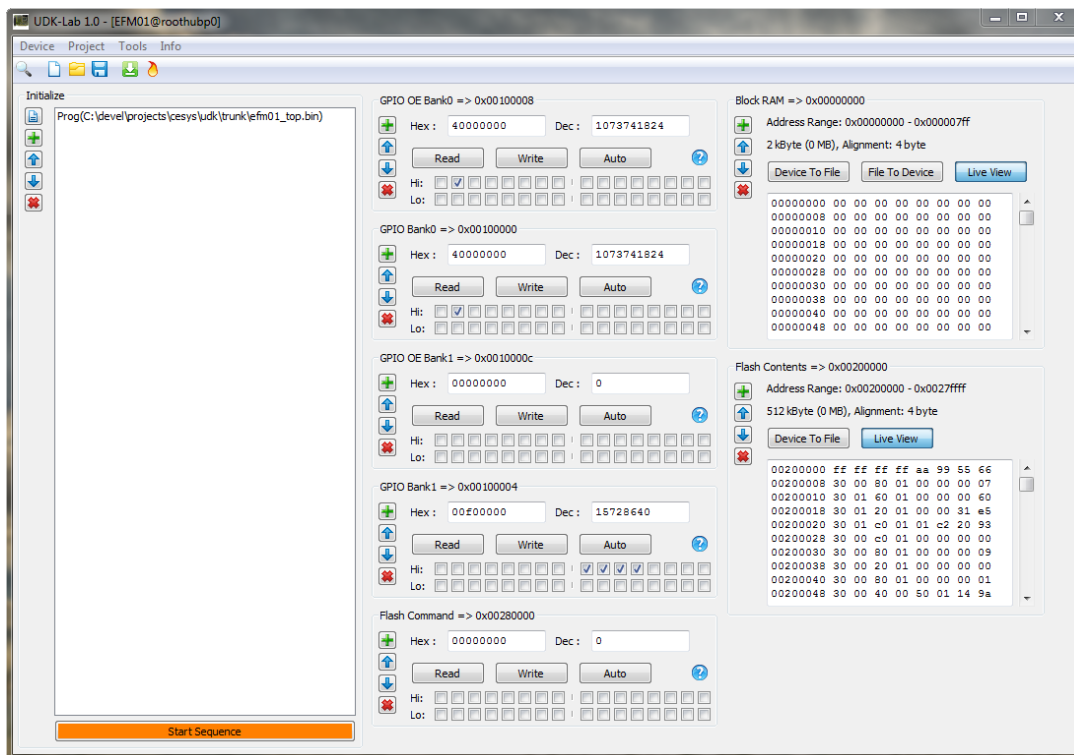


Figure 10: UDKLab Main Screen

Using UDKLab

After starting UDKLab, most of the UI components are disabled. They will be enabled at the point they make sense. As no device is selected, only device independent functions are available:

- The FPGA design array creator
- The option to define USB Power-On behavior
- Info menu contents

All other actions require a device, which can be chosen via the device selector which pops up as separate window:

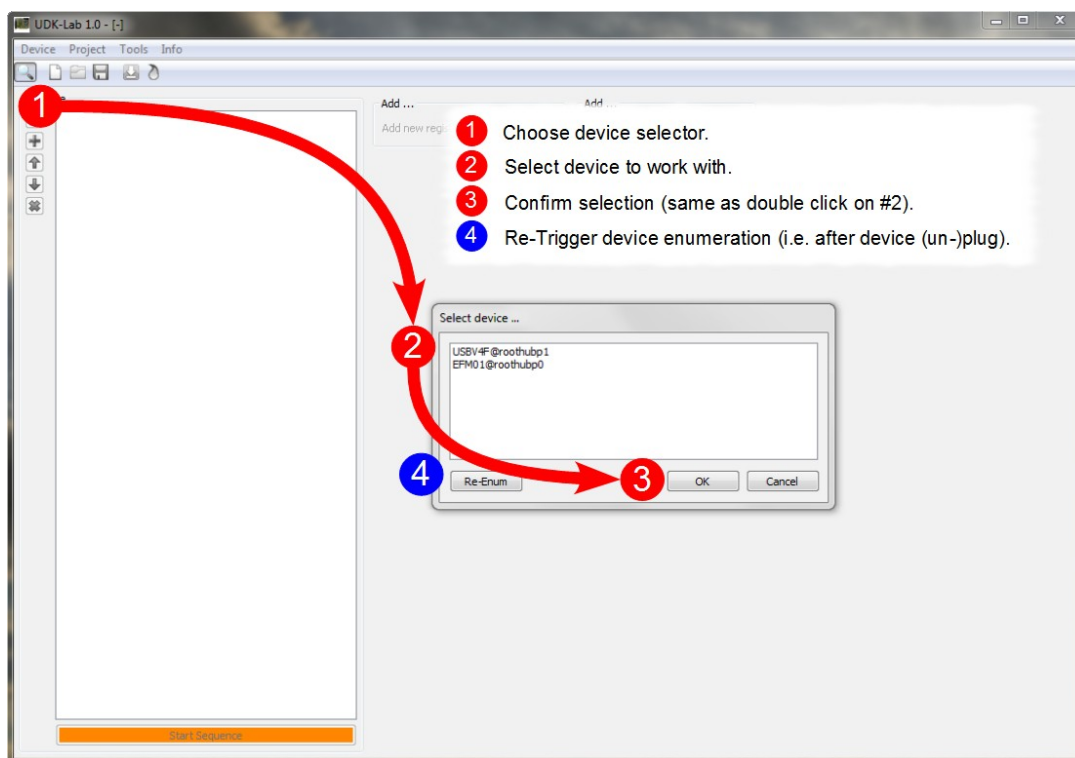


Figure 11: Device selection flow

If the device list is not up to date, clicking Re-Enum will search again. A device can be selected by either double clicking on it or choosing OK.

Important: Opening the device selector again will internally re-initialize the underlying API, so active communication is stopped and the right panel is disabled again (more on the state of this panel below).

After a device has been selected, most UI components are available:

- FPGA configuration
- FPGA design flashing [if device has support]
- Project controls
- Initializer controls (Related to projects)

The last disabled component at this point is the content panel. It is enabled if the initialization sequence has been run. The complete flow to enable all UI elements can be seen below:

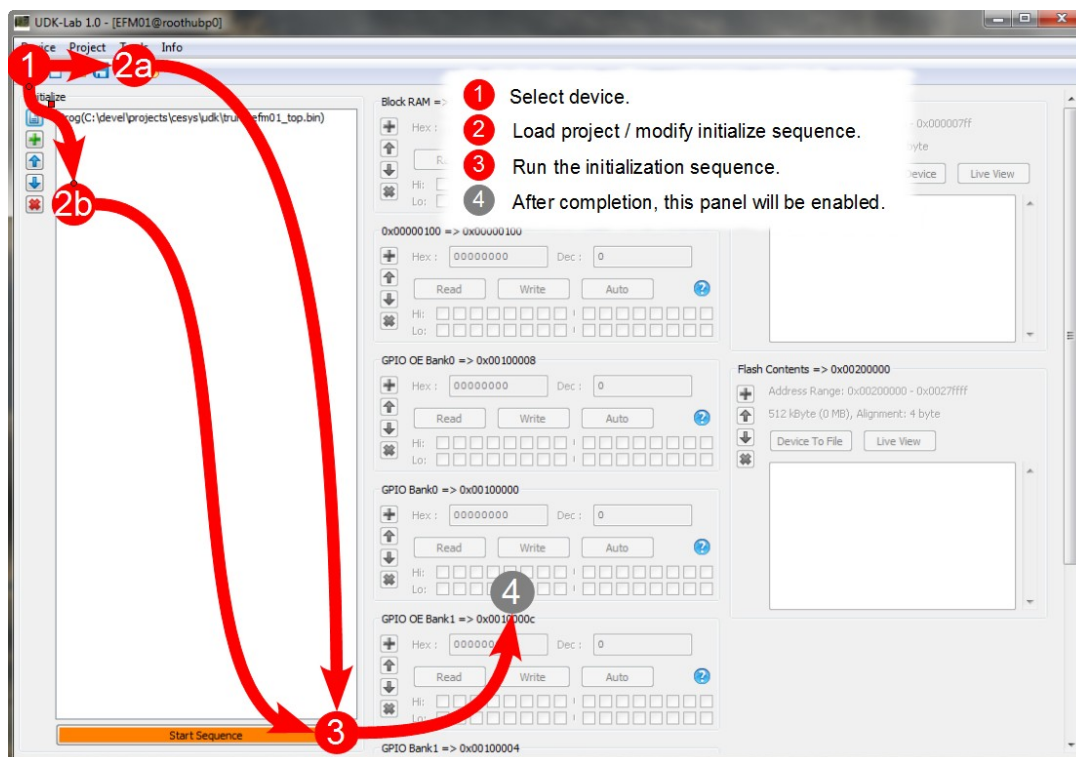


Figure 12: Prepare to work with device

FPGA configuration

Choosing this will pop up a file selection dialog, allowing to choose the design for download. If the file choosing isn't canceled, the design will be downloaded subsequent to closing the dialog.

FPGA design flashing

This option stores a design into the flash component on devices that have support for it. The design is loaded to the FPGA after device power on without host intervention. How and under which circumstances this is done can be found in the hardware description of the corresponding device. The following screen shows the required actions for flashing:

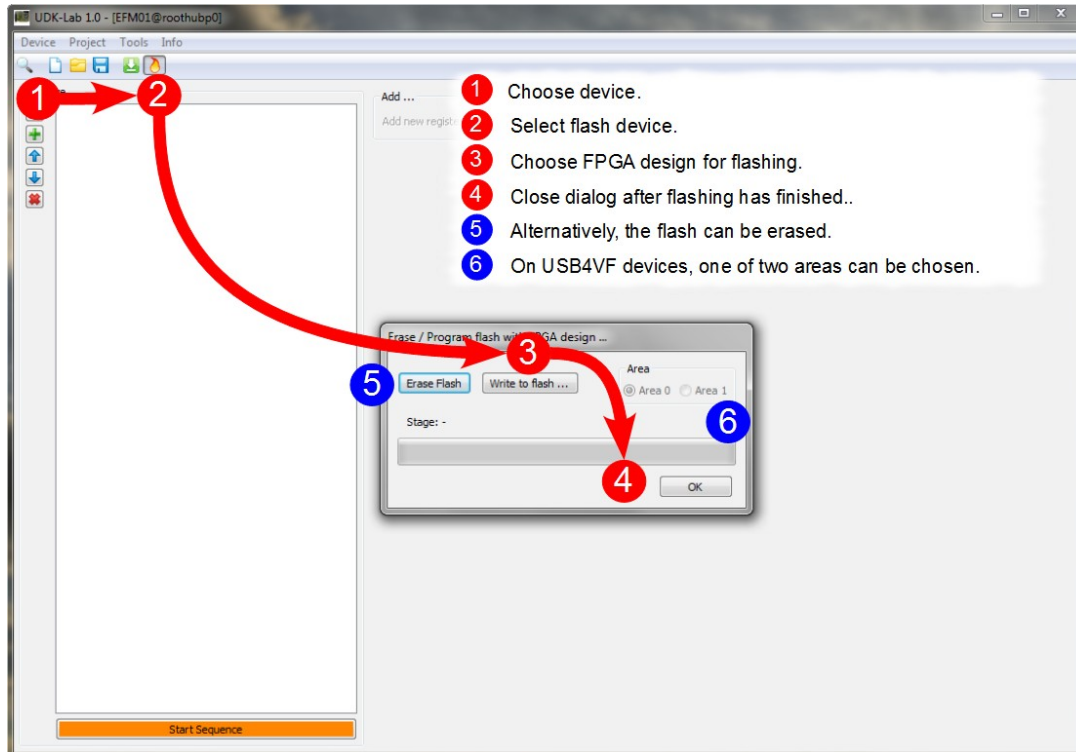


Figure 13: Flash design to device

Projects

Device communication is placed into a small project management. This reduces the actions from session to session and can be used for simple service tasks too. A projects stores the following information:

- Device type it is intended to
- Initializing sequence
- Register list
- Data area list

Projects are handled like files in usual applications, they can be loaded, saved, new

projects can be created. Only one project can be active in one session.

Initializing sequence

The initializing sequence is a list of actions that must be executed in order to work with the FPGA on the device. (The image shows an example initializing list of an EFM01, loading our example design and let the LED blink for some seconds):

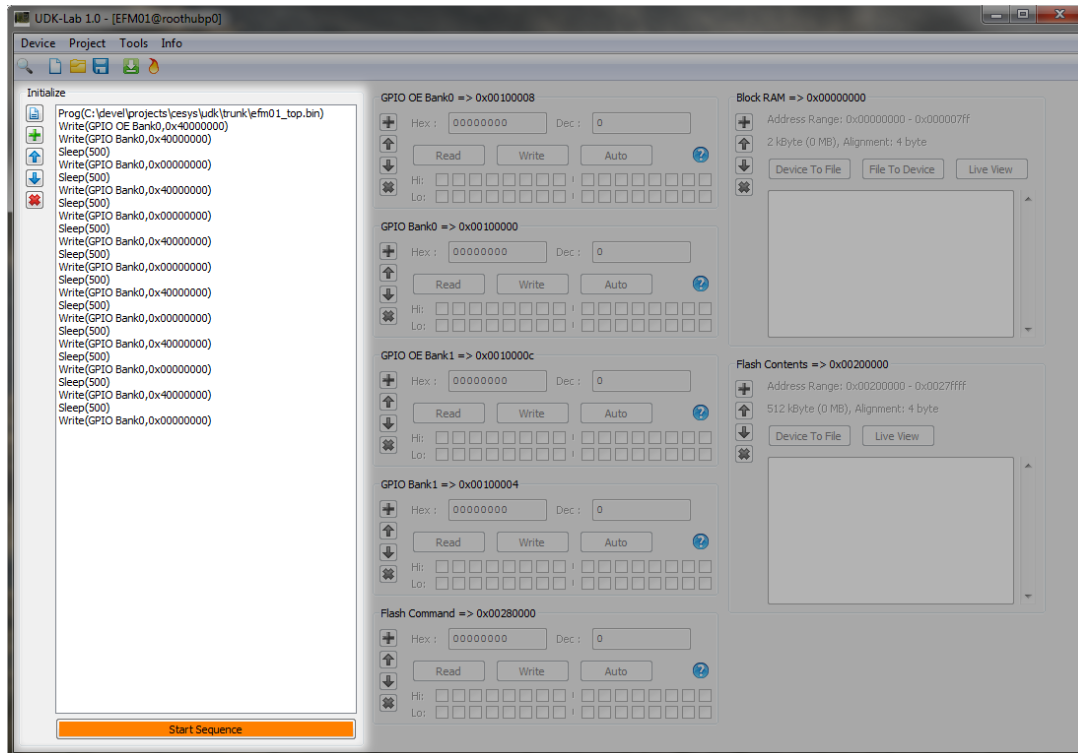


Figure 14: Initializing sequence

Sequence contents

UDKLab supports the following content for initialization:

- FPGA programming
- FPGA reset
- Register write
- Sleep

Without a design, an FPGA does nothing, so it must be loaded before usage. This can be ensured in two ways:

- Download design from host
- Load design from flash (supported on EFM01, USBV4F and USBS6)

So the first entry in the initialize list must be a program entry or, if loaded from flash, a reset entry (To sync communication to the host side). Subsequent to this, a mix of register write and sleep commands can be placed, which totally depends on the underlying FPGA design. This can be a sequence of commands sent to a peripheral component or to fill data structures with predefined values. If things get complexer, i.e. return values must be checked, this goes beyond the scope of the current UDKLab implementation and must be solved by a host process.

To control the sequence, the buttons on the left side can be used. In the order of appearance, they do the following (also indicated by tooltips):

- Clear complete list
- Add new entry (to the end of the list)
- Move currently selected entry on position up
- Move currently selected entry on position down
- Remove currently selected entry

All buttons should be self explanatory, but here's a more detailed look on the add entry, it opens the following dialog:

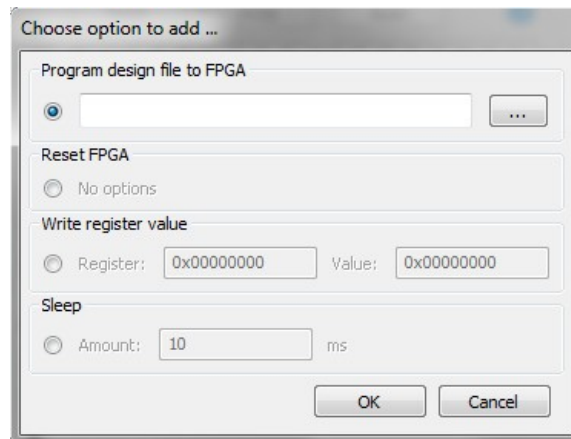


Figure 15: Add new initializing task

One of the four possible entries must be selected using the radio button in front of it. Depending on the option, one or two parameters must be set, **OK** adds the new action to initializer list.

Sequence start

The button sitting below the list runs all actions from top to bottom. In addition to this, the remaining UI components, the content panel, will be enabled, as UDKLab expects a working communication at this point. The sequence can be modified and started as often as wished.

Content panel

The content panel can be a visual representation of the FPGA design loaded during initialization. It consists of a list of registers and data areas, which can be visited and modified using UDKLab. The view is split into two columns, while the left part contains the registers and the right part all data area / block entries.

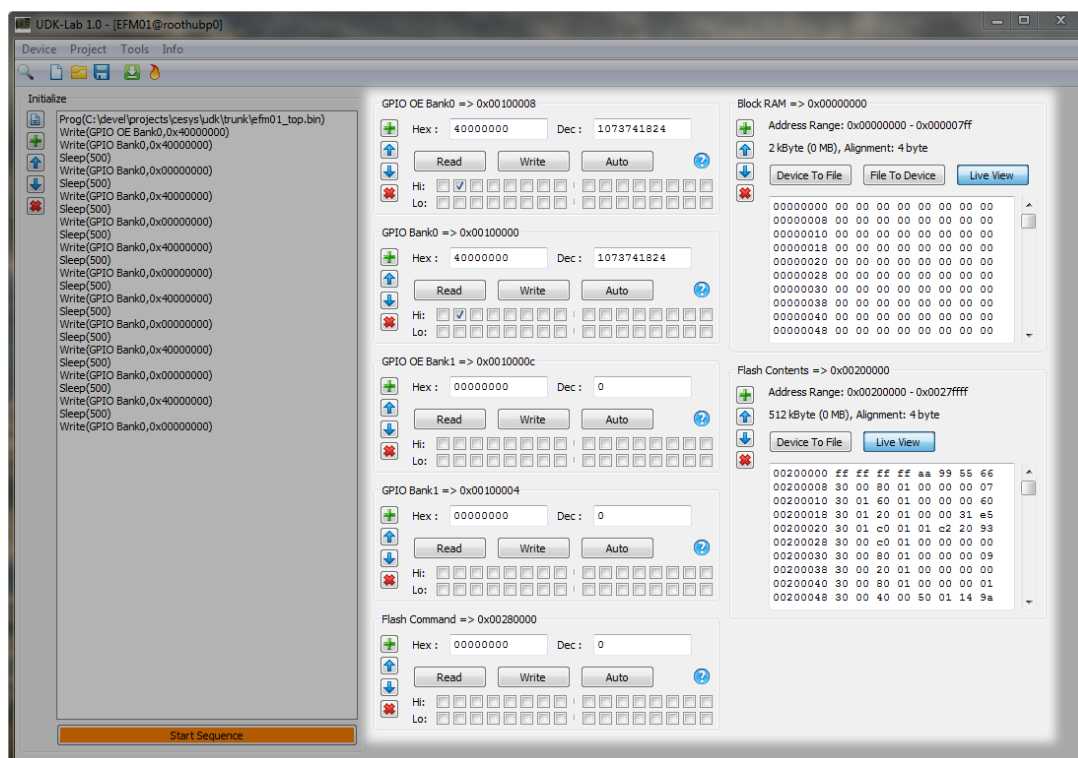


Figure 16: Content panel

Register entry

A register entry can be used to communicate with a 32 bit register inside the FPGA. In UDKLab, a register consists of the following values:

- Address
- Name
- Info text

The visual representation of one register can be seen in the following image:

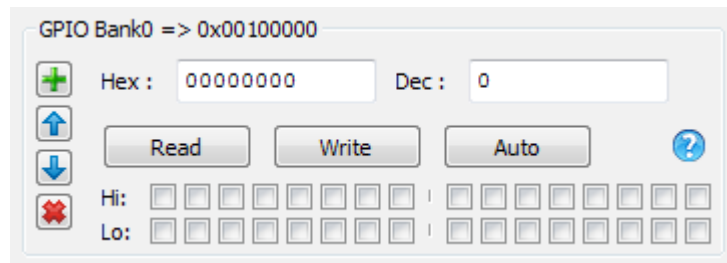


Figure 17: Register panel

The left buttons are responsible for adding new entries, move the entry up or down and removing the current entry, all are self explanatory. The header shows it's mapping name as well as the 32 bit address. The question mark in the lower right will show a tooltip if the mouse is above it, which is just a little help for users. Both input fields can be used to write in a new value, either hex- or decimal or contain the values if they are read from FPGA design. The checkboxes represent one bit of the current value. Clicking the *Read* button will read the current value from FPGA and update both text boxes as well as the checkboxes, which is automatically done every 100ms if the *Auto* button is active. Setting register values inside the FPGA is done in a similar way, clicking the *Write* button writes the current values to the device. One thing needs a bit attention here:

Clicking on the checkboxes implicitly writes the value without the need to click on the *Write* button !

Data area entry

A data area entry can be used to communicate with a data block inside the FPGA, examples are RAM or flash areas. Data can be transferred from and to files, as well as displayed in a live view. An entry consists of the following data:

- Address
- Name
- Data alignment
- Size
- Read-only flag

The visual representation is shown below.

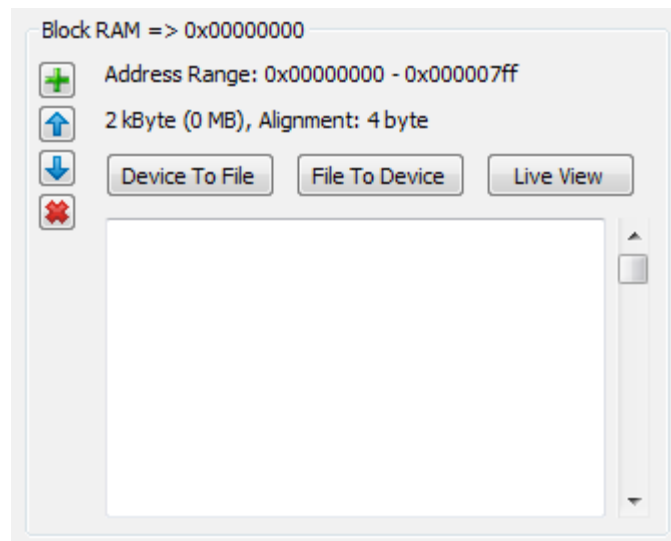


Figure 18: Data area panel

Similar to the register visualization, the buttons on the right side can be used to add, move and remove data area panels. The header shows the name and the address followed by the data area details. Below are these buttons:

- Device To File: The complete area is read and stored to the file which is defined in the file dialog opening after clicking the button.
- File To Device: This reads the file selected in the upcoming file dialog and stores the contents in the data area, limited by the file size or data area size. This button is not shown if the Read-only flag is set.
- Live View: If this button is active, the text view below shows the contents of the area, updated every 100 ms, the view can be scrolled, so every piece can be visited.

Additional information

References

- CESYS *EFM01* software API and sample code
- Cypress FX-2 LP USB peripheral controller datasheet (cy7c68013a_8.pdf) and user manual (EZ-USB_TRM.pdf)
- Specification for the “WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores” Revision B.3, released September 7, 2002 (wbspec_b3.pdf)
- CESYS application note “Transfer Protocol for CESYS USB products”

Links

- <http://www.vhdl-online.de/>
- Informations about the VHDL language, including a tutorial, a language reference, design hints for describing state machines, synthesis and the synthesizable language subset
- <http://www.opencores.org/projects.cgi/web/wishbone/>
- Home of the WISHBONE standard
- <http://www.cypress.com/>
- Provider of the FX-2 LP USB peripheral controller
- <http://www.xilinx.com/>
- Provider of the Spartan-3E™ FPGA and the free FPGA development environment ISE WebPACK

Table of contents

Table of Contents

<u>Copyright information.....</u>	<u>2</u>
<u>Overview.....</u>	<u>3</u>
<u>Summary of EFM01.....</u>	<u>3</u>
<u>Feature list.....</u>	<u>3</u>
<u>Included in delivery.....</u>	<u>3</u>
<u>Hardware.....</u>	<u>4</u>
<u>SPARTAN-3E FPGA.....</u>	<u>4</u>
<u>Module size.....</u>	<u>4</u>
<u>Connectors and FPGA pinout.....</u>	<u>5</u>
<u>Powering EFM01.....</u>	<u>6</u>
<u>EFM01 configuration.....</u>	<u>7</u>
<u>How to store configuration data in SPI Flash.....</u>	<u>8</u>
<u>Loading SPI Flash via USB.....</u>	<u>8</u>
<u>SPI Flash Indirect Programming Using FPGA JTAG Chain.....</u>	<u>8</u>
<u>SPI Flash Direct Programming using iMPACT.....</u>	<u>9</u>
<u>Program SPI Flash using external microcontrollers.....</u>	<u>10</u>
<u>External expansion connectors.....</u>	<u>10</u>
<u>FPGA design.....</u>	<u>14</u>
<u>Cypress FX-2 LP and USB basics.....</u>	<u>14</u>
<u>Clocking FPGA designs.....</u>	<u>14</u>
<u>FX-2/FPGA slave FIFO connection.....</u>	<u>15</u>
<u>Introduction to example FPGA designs.....</u>	<u>16</u>
<u>FPGA source code copyright information.....</u>	<u>18</u>
<u>FPGA source code license.....</u>	<u>18</u>
<u>Disclaimer of warranty.....</u>	<u>18</u>
<u>Design “efm01”.....</u>	<u>19</u>
<u>Files and modules.....</u>	<u>20</u>
<u>src/wishbone.vhd:.....</u>	<u>20</u>
<u>src/efm01_top.vhd:.....</u>	<u>20</u>
<u>src/wb_syscon.vhd:.....</u>	<u>20</u>
<u>src/wb_intercon.vhd:.....</u>	<u>20</u>
<u>src/wb_ma_fx2.vhd:.....</u>	<u>20</u>
<u>src/wb_sl_bram.vhd:.....</u>	<u>20</u>

src/wb_sl_speedtest.vhd:	20
src/wb_sl_gpio.vhd:	20
src/wb_sl_flash.vhd:	21
src/fx2_slfifo_ctrl.vhd:	21
src/sync_fifo16.vhd:	21
src/sfifo_hd_a1Kx18b0K5x36.vhd:	21
src/flash_ctrl.vhd:	21
efm01.isc:	21
efm01.ucf:	21
WISHBONE transactions:	21
WISHBONE signals driven by the master:	22
WISHBONE signals driven by slaves:	22
Example:	23
Design “efm01_perf”	24
Files and modules:	24
src/efm01_perf.vhd:	24
src/fx2_slfifo_ctrl.vhd:	24
src/sync_fifo16.vhd:	24
efm01_perf.isc:	24
efm01_perf.ucf:	24
Slave FIFO transactions:	24
Software:	26
Introduction:	26
Changes to previous versions:	26
Windows:	27
Requirements:	27
Driver installation:	27
Build UDK:	27
Prerequisites:	27
Solution creation and build:	27
Linux:	29
Requirements:	29
Drivers:	29
USB:	29
PCI:	30
Build UDK:	31
Prerequisites:	31
Makefile creation and build:	31
Use APIs in own projects:	33
C++ API:	33
Add project to UDK build:	33
C API:	33
.NET API:	34

<u>API Functions in detail.....</u>	<u>34</u>
<u>API Error handling.....</u>	<u>34</u>
C++ and .NET API.....	34
C API.....	34
Methods/Functions.....	35
Device enumeration.....	36
Methods/Functions.....	36
Information gathering.....	39
Methods/Functions.....	39
Using devices.....	41
Methods/Functions.....	41
UDKLab.....	46
Introduction.....	46
The main screen.....	47
Using UDKLab.....	48
FPGA configuration.....	49
FPGA design flashing.....	50
Projects.....	50
Initializing sequence.....	51
Content panel.....	53
 Additional information.....	 56
References.....	56
Links.....	56
 Table of contents.....	 57