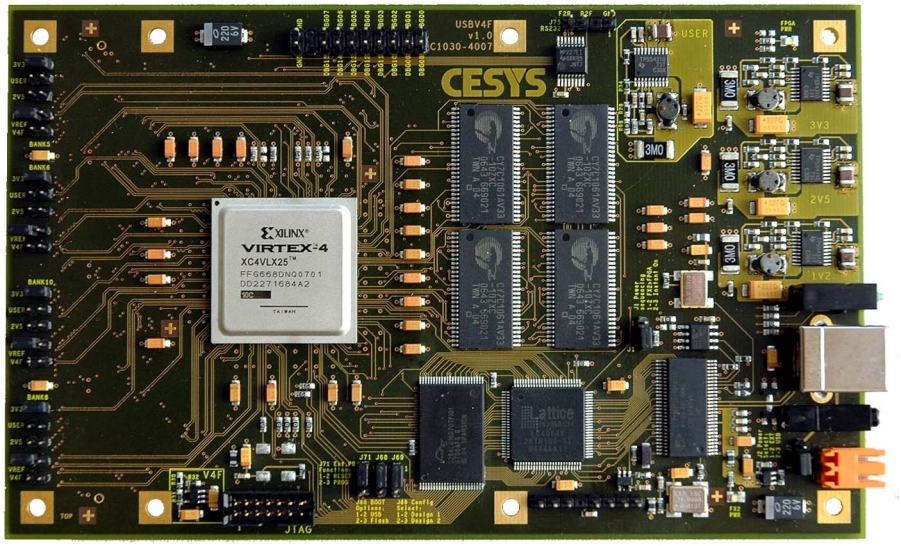
VIRTEX-4 FPGA board with USB 2.0 interface.

Order number: C1030-4007



Copyright information

⇒ 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 USBV4F

The USBV4F is a VIRTEX-4™ LX (Logic) FPGA-board with USB2.0 Interface and 8 MByte SRAM. It offers multiple configuration options and can also be used without the USB interface.

Feature list

- XILINX™ Virtex-4 FPGA (XC4VLX25-10FFG668C)
- USB 2.0 controller (CYPRESS FX2LP)
- USB 2.0 compliant device (Plug-and-Play)
- Self-powered or bus-powered
- 2 MByte high performance FLASH (2M x8 Bit/1M x16 Bit, 70ns access time)
- FPGA configuration via JTAG, USB2.0 or FLASH
- 2 configuration bitstreams can be stored in FLASH and selected for boot-up
- Up to 8 MByte FAST SRAM (2M x 32 Bit, 10ns)
- Expansion connector with more than 200 user I/Os
- 4 ports with selectable bank I/O voltage (VCCO)
- 4 ports with selectable bank reference voltage (VREF)
- 16 FPGA I/Os routed on standard 2,54mm pitch connector for debugging
- RS-232 transceiver
- 4 user configurable LEDs

Included in delivery

- *USBV4F* board
- One USB-cable 1,5m
- One CD-ROM containing the user's manual (English), drivers, libraries, tools and example source code.

Hardware

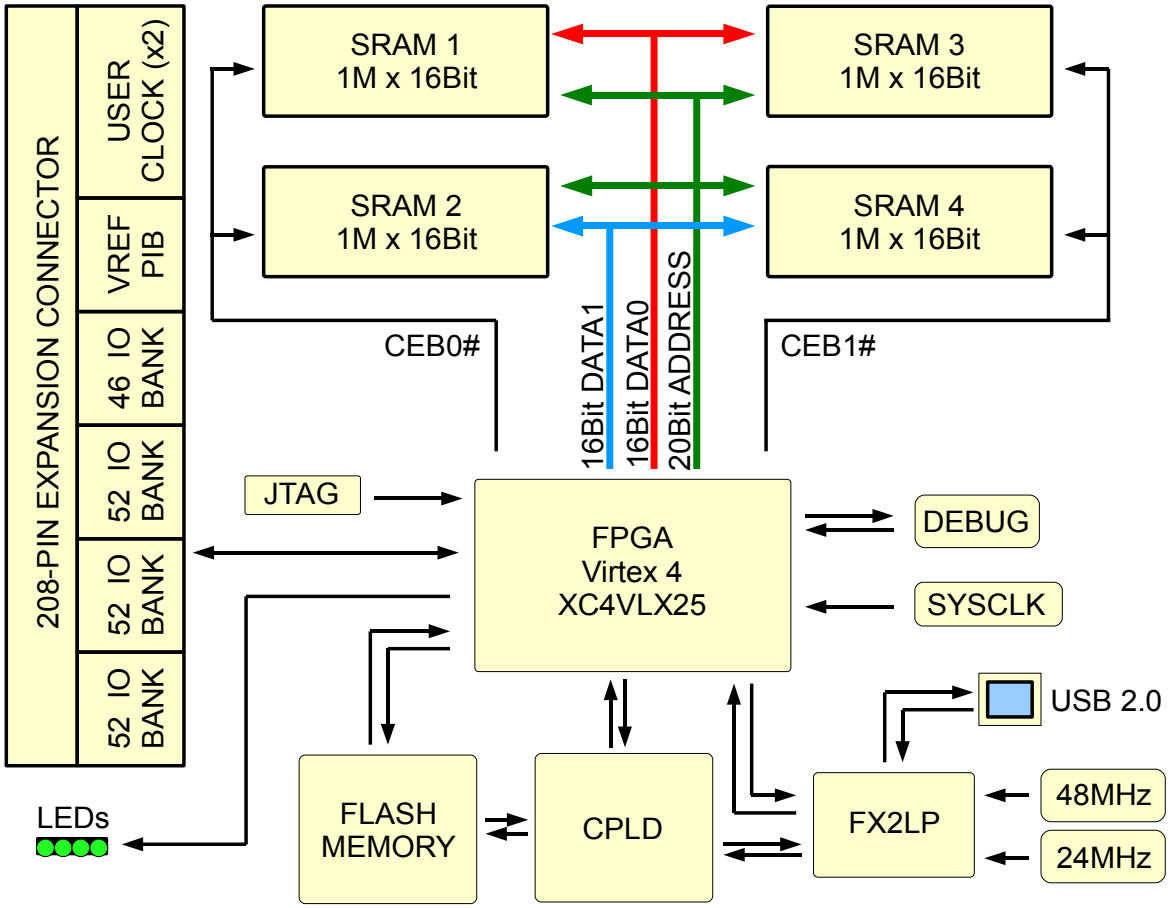


Figure 1: USBV4F block diagram

Virtex-4 FPGA

XC4VLX25-10FFG668C FPGA features:

Configurable Logic Blocks (CLBs)	96 x 26
Logic Cells	24,192
Max Distributed Ram	168 kBit
XtremeDSP Slices	48
Block RAM	72 blocks (1,296 kBit)
DCMs	8
PMCDs	4

For details on Virtex-4™, please refer to data sheet available at Xilinx.com.

CESYS PIB² slot

Board size

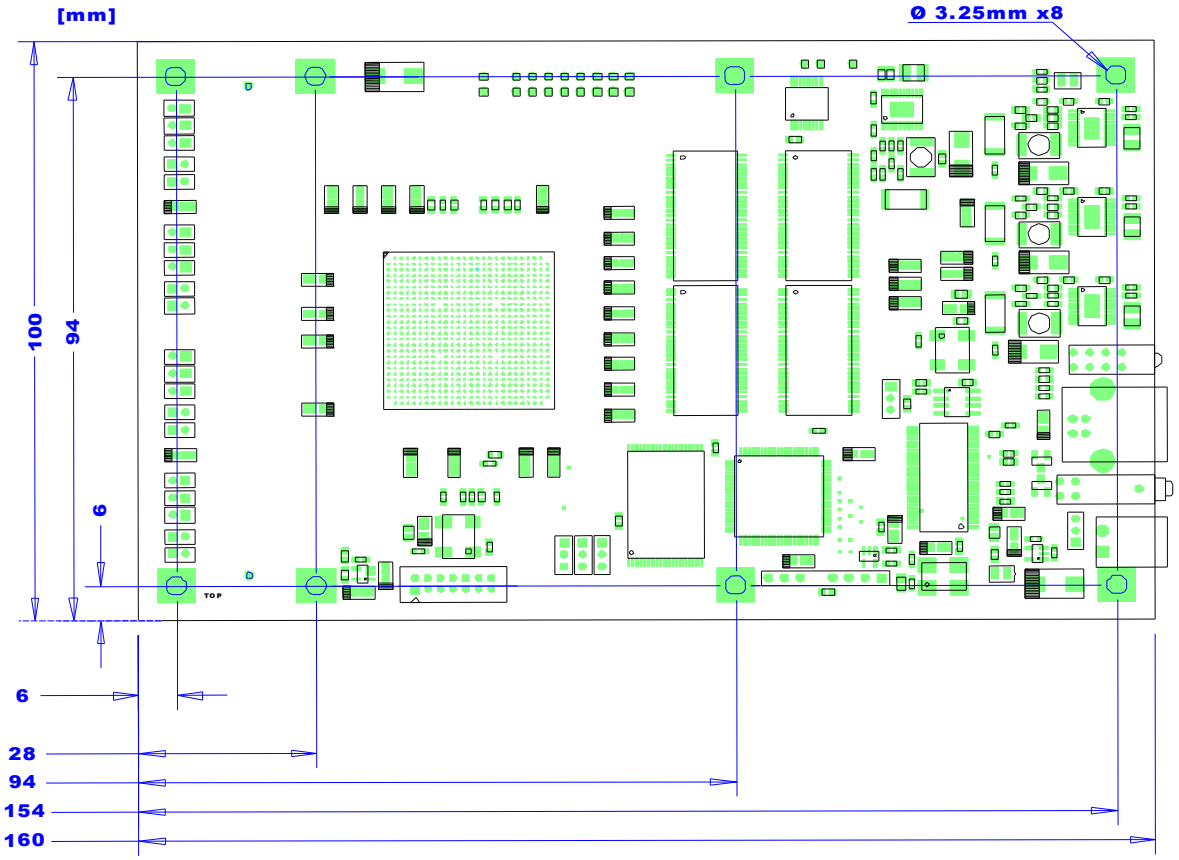


Figure 2: C1030-4007 outline drawing and dimensions

Connectors and FPGA pinout

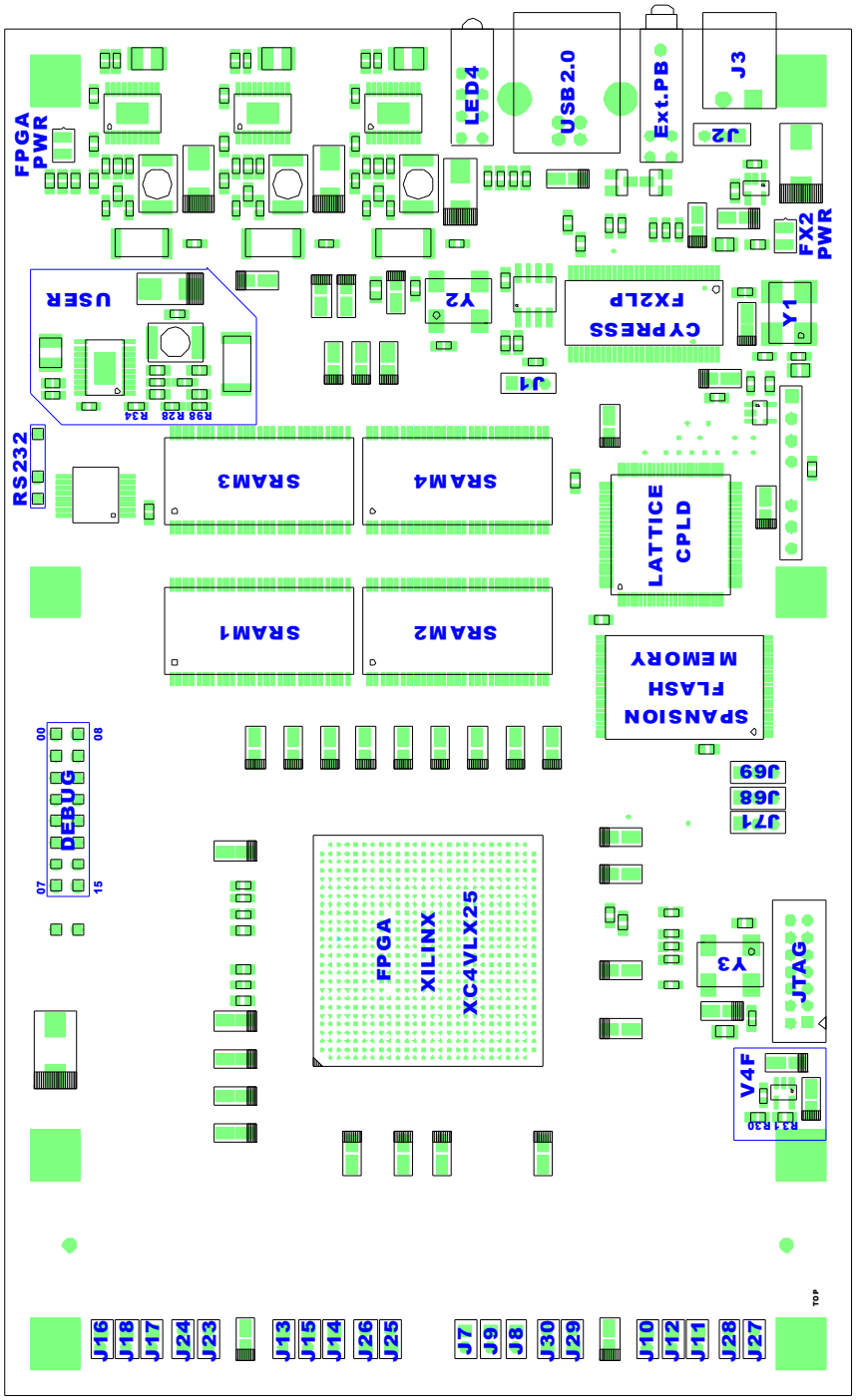


Figure 3: C1030-4007 connector diagram (TOP view)

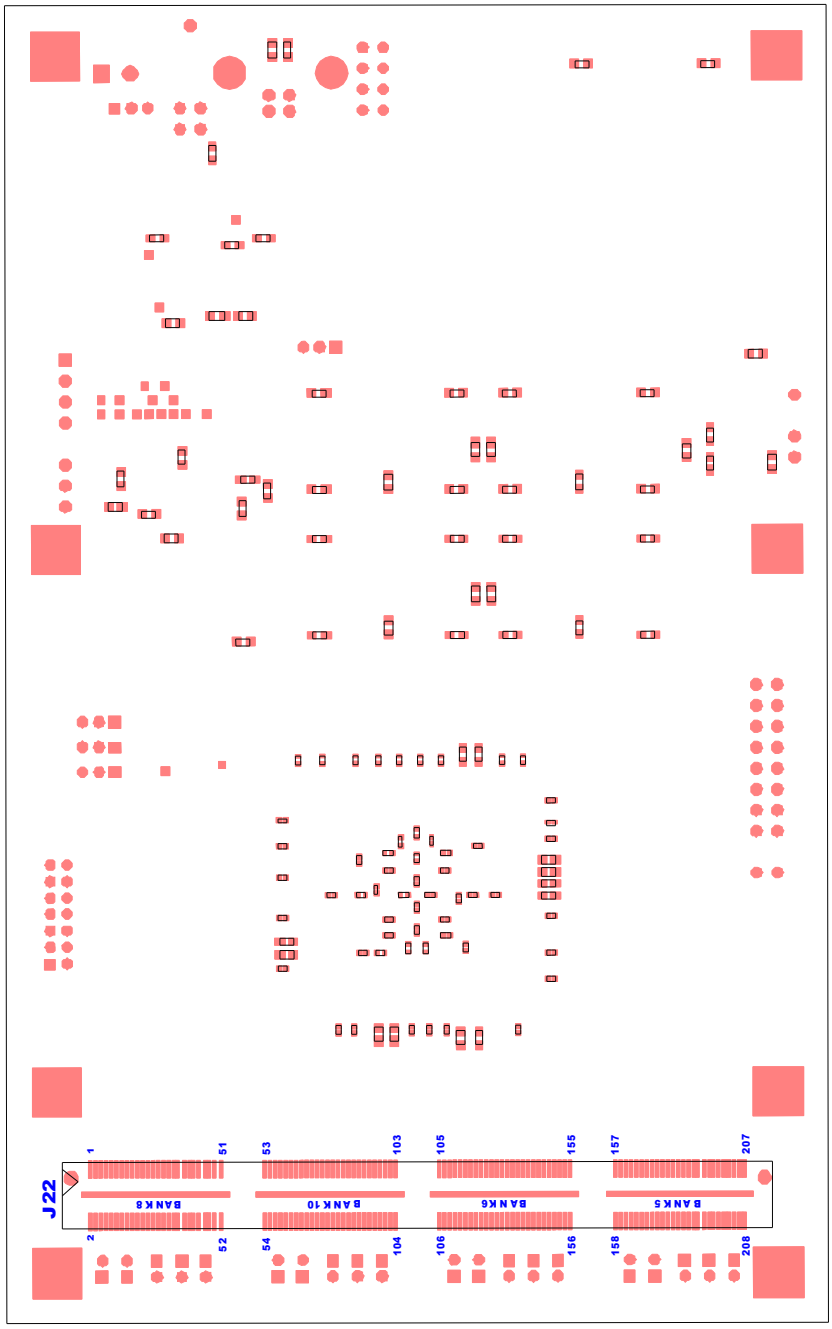


Figure 4: C1030-4007 connector diagram (BOTTOM View)

Clock signals

USBV4F provides several clock source possibilities for user designs. First of all users can implement the interface clock IFCLK provided by the USB 2.0 interface in their designs. IFCLK defaults to 48 MHz. If IFCLK does not comply with design requirements one extra clock oscillator can be populated to source SYSCLK. If it is necessary to have even more clock sources available, users can generate up to two clocks on their PIB- compatible plug-in board and connect them to the FPGA global clock net via PIBCLK0/1. PIBCLK0/1 are +3,3 Volt only I/Os and can optionally be used as differential clock inputs PIBCLK0P/N and PIBCLK1P/N. If not used as clock inputs PIBCLK0P/N and PIBCLK1P/N are usable as standard +3,3 Volt I/Os.

Clock signals				
Signal name	**Clock rate**	**I/O Standard**	**FPGA I/O**	**Comment**
CCLK[1]	48MHz	LVCMOS33	CCLK	FPGA Configuration clock
IFCLK	48MHz	LVCMOS33	C15	USB2.0 interface clock
SYSCLK	--	LVCMOS33	AE10	Optional extra oscillator
PIBCLK0P[2]	--	LVCMOS33	A12	Clock input for user clock from PIB
PIBCLK0N	--	LVCMOS33	A11	Complementary clock signal input
PIBCLK1P[2]	--	LVCMOS33	A10	Clock input for user clock from PIB
PIBCLK1N	--	LVCMOS33	B10	Complementary clock signal input

FX2LP

FX2LP			
Signal name	**I/O Standard**	**FPGA I/O**	**Comment**
FD0	LVCMOS33	AC24	Bidirectional FIFO data bus
FD1	LVCMOS33	AC25	Bidirectional FIFO data bus
FD2	LVCMOS33	AC26	Bidirectional FIFO data bus
FD3	LVCMOS33	AB23	Bidirectional FIFO data bus
FD4	LVCMOS33	AC23	Bidirectional FIFO data bus
FD5	LVCMOS33	AD26	Bidirectional FIFO data bus

1 CCLK is directly connected to IFCLK
2 PIBCLKP/N can be used as standard +3,3Volt I/Os if not used as clock input

FX2LP			
Signal name	**I/O Standard**	**FPGA I/O**	**Comment**
FD6	LVCMOS33	AD25	Bidirectional FIFO data bus
FD7	LVCMOS33	AE24	Bidirectional FIFO data bus
FD8	LVCMOS33	AA24	Bidirectional FIFO data bus
FD9	LVCMOS33	AA23	Bidirectional FIFO data bus
FD10	LVCMOS33	AB26	Bidirectional FIFO data bus
FD11	LVCMOS33	AB25	Bidirectional FIFO data bus
FD12	LVCMOS33	AB24	Bidirectional FIFO data bus
FD13	LVCMOS33	AA20	Bidirectional FIFO data bus
FD14	LVCMOS33	AF21	Bidirectional FIFO data bus
FD15	LVCMOS33	AE21	Bidirectional FIFO data bus
FLAGA	LVCMOS33	AE23	Programmable slave-FIFO output status flag signal
FLAGB	LVCMOS33	AD23	Programmable slave-FIFO output status flag signal
FLAGC	LVCMOS33	AF24	Programmable slave-FIFO output status flag signal
SLRD/RDY0	LVCMOS33	AC21	Input-only read strobe for the slave FIFOs connected to FD[7..0] or FD[15..0].
SLWR/RDY1	LVCMOS33	AD21	Input-only write strobe for the slave FIFOs connected to FD[7..0] or FD[15..0].
FLAGD/PA7/SLCS#	LVCMOS33	AB21	Programmable slave-FIFO output status flag signal.
PA0/INT0#	LVCMOS33	AF23	Bidirectional I/O
PWR_ENA/ PA1	LVCMOS33	AA26[3]	Active high enable power output
SLOE/PA2	LVCMOS33	AB22	Input-only output enable for the slave FIFOs connected to FD[7..0] or FD[15..0].
WU/PA3	LVCMOS33	AC22	Bidirectional I/O
FIFOADR0/ PA4	LVCMOS33	AD22	Input-only address select for the slave FIFOs connected to FD[7..0] or FD[15..0].
FIFOADR1/ PA5	LVCMOS33	AF22	Input-only address select for the slave FIFOs connected to FD[7..0] or FD[15..0].
PKTEND/ PA6	LVCMOS33	Y21	Input used to commit the FIFO packet data to the endpoint.
IFCLK	LVCMOS33	C15	Interface Clock, used for synchronously clocking data into or out of the slave FIFOs.

3 As default PWR_ENA/PA2 is used to enable power-up of all logic other than FX2LP to keep supply current below 100mA during enumeration processes for USB powered devices. It is possible to connect PWR_ENA/PA2 to FPGA I/O AA26 by the use of jumper J1. Be aware, that this is not a standard operation mode and therefore is not officially supported by CESYS.

FLASH

FLASH			
Signal name	**I/O Standard**	**FPGA I/O**	**Comment**
DQ0	LVCMOS33	AD13	Data I/O
DQ1	LVCMOS33	AC13	Data I/O
DQ2	LVCMOS33	AC15	Data I/O
DQ3	LVCMOS33	AC16	Data I/O
DQ4	LVCMOS33	AA11	Data I/O
DQ5	LVCMOS33	AA12	Data I/O
DQ6	LVCMOS33	AD14	Data I/O
DQ7	LVCMOS33	AC14	Data I/O
DQ8	LVCMOS33	AA13	Data I/O
DQ9	LVCMOS33	AB13	Data I/O
DQ10	LVCMOS33	AA15	Data I/O
DQ11	LVCMOS33	AA16	Data I/O
DQ12	LVCMOS33	AC11	Data I/O
DQ13	LVCMOS33	AC12	Data I/O
DQ14	LVCMOS33	AB14	Data I/O
DQ15 / A0	LVCMOS33	AA14	Data I/O, optional address input A0 for byte mode
A1	LVCMOS33	AF20	Address input
A2	LVCMOS33	AB10	Address input
A3	LVCMOS33	AC10	Address input
A4	LVCMOS33	AD10	Address input
A5	LVCMOS33	AF10	Address input
A6	LVCMOS33	AD11	Address input
A7	LVCMOS33	AF11	Address input
A8	LVCMOS33	AD12	Address input
A9	LVCMOS33	AD17	Address input
A10	LVCMOS33	AC18	Address input
A11	LVCMOS33	AB18	Address input
A12	LVCMOS33	AA18	Address input
A13	LVCMOS33	AF19	Address input
A14	LVCMOS33	AD19	Address input
A15	LVCMOS33	AC19	Address input
A16	LVCMOS33	AA19	Address input

FLASH			
Signal name	**I/O Standard**	**FPGA I/O**	**Comment**
A17	LVCMOS33	AB20	Address input
A18	LVCMOS33	AE12	Address input
A19	LVCMOS33	AF12	Address input
A20	LVCMOS33	AD16	Address input
A21	LVCMOS33	AC17	Address input only for 16MBit type
RESET#	LVCMOS33	AF18	Active low reset input, externally pulled high
CE#	LVCMOS33	AE20	Active low chip enable input, externally pulled high
WE#	LVCMOS33	AE18	Active low write enable input, externally pulled high
OE#	LVCMOS33	AD20	Active low output enable input, externally pulled high
BYTE#	LVCMOS33	AC20	Active low byte mode enable input 0 DQ0-DQ7, A0-A21 1 DQ0-DQ15, A1-A21
RY_BY#	LVCMOS33	AE13	Ready / Busy output
WP#_ACC[4]	LVCMOS33	AE14	Active Low write protect input

JTAG interface

In addition to configuration via USB 2.0 or FLASH it is possible to download configuration data using a JTAG interface. The *USBV4F* is equipped as standard with a 2-row 14-pin connector to plug-in the *Parallel Cable IV* [5] from Xilinx™. But the JTAG interface is not only suitable to download designs for testing purposes but enables the user to check a running design by the help of software tools provided by Xilinx™, for instance ChipScope[6].

CON1 JTAG connector	
Pin	**Comment**
Pin 1, 3, 5, 7, 9, 11, 13	**GND**
Pin 2	**+3,3 Volt**
Pin 4	TMS
Pin 6	TCK

4 Accelerated programming is not supported
5 Parallel Cable IV is not included
6 ChipScope is not included. A demo version is available at the Xilinx™ webpage. (http://www.xilinx.com/ise/optional_prod/cspro.htm)

CON1 JTAG connector	
Pin	**Comment**
Pin 8	TDO
Pin 10	TDI
Pin 12, 14	Not connected

SRAM

As default *USBV4F* is equipped with two CY7C1061AV33 (1M x 16 Bit, 10ns) SRAM devices from CYPRESS™ with common address bus and chip enable signal #CEB0. 16 Bit data bus, control inputs #WE, #OE and bank enable signals are routed independently to each device to allow several memory configurations to be established, for instance 1M x 32 Bit. Optionally two additional SRAM devices can be populated to increase available SRAM memory to 8 MByte (2M x 32 Bit). For further information about SRAM routing on *USBV4F* please refer to Figure 5 on page 15. Information about how to use SRAM successfully in user designs can be found in chapter D. The complete listing of FPGA I/Os routed to the different SRAM devices is given in the following table:

SRAM			
Signal name	**I/O Standard**	**FPGA I/O**	**Comment**
CEB0#	LVCMOS33	M26	Active low chip enable bank 0, externally pulled high
CEB0	LVCMOS33	--	Active high chip enable bank 0, externally pulled high, not connected to FPGA
CEB1#	LVCMOS33	R26	Active low chip enable bank 1, externally pulled high
CEB1	LVCMOS33	--	Active high chip enable bank 1, externally pulled high, not connected to FPGA
A0	LVCMOS33	K24	Address input: 20Bit Address bus
A1	LVCMOS33	K23	Address input: 20Bit Address bus
A2	LVCMOS33	K22	Address input: 20Bit Address bus
A3	LVCMOS33	K21	Address input: 20Bit Address bus
A4	LVCMOS33	J26	Address input: 20Bit Address bus
A5	LVCMOS33	V22	Address input: 20Bit Address bus
A6	LVCMOS33	V21	Address input: 20Bit Address bus
A7	LVCMOS33	W26	Address input: 20Bit Address bus
A8	LVCMOS33	W25	Address input: 20Bit Address bus
A9	LVCMOS33	W24	Address input: 20Bit Address bus

SRAM			
Signal name	**I/O Standard**	**FPGA I/O**	**Comment**
A10	LVCMOS33	W21	Address input: 20Bit Address bus
A11	LVCMOS33	J20	Address input: 20Bit Address bus
A12	LVCMOS33	K20	Address input: 20Bit Address bus
A13	LVCMOS33	L20	Address input: 20Bit Address bus
A14	LVCMOS33	N22	Address input: 20Bit Address bus
A15	LVCMOS33	M21	Address input: 20Bit Address bus
A16	LVCMOS33	L26	Address input: 20Bit Address bus
A17	LVCMOS33	L24	Address input: 20Bit Address bus
A18	LVCMOS33	L23	Address input: 20Bit Address bus
A19	LVCMOS33	L21	Address input: 20Bit Address bus
WE0#	LVCMOS33	K26	Active low write enable bus 0
OE0#	LVCMOS33	P24	Active low output enable bus 0
BE0.0#	LVCMOS33	P23	Active low enable lower byte bus 0
BE0.1#	LVCMOS33	K25	Active low enable upper byte bus 0
DQ0.0	LVCMOS33	M22	Data I/O: 16Bit data bus 0
DQ0.1	LVCMOS33	M23	Data I/O: 16Bit data bus 0
DQ0.2	LVCMOS33	M24	Data I/O: 16Bit data bus 0
DQ0.3	LVCMOS33	M25	Data I/O: 16Bit data bus 0
DQ0.4	LVCMOS33	R22	Data I/O: 16Bit data bus 0
DQ0.5	LVCMOS33	R23	Data I/O: 16Bit data bus 0
DQ0.6	LVCMOS33	R24	Data I/O: 16Bit data bus 0
DQ0.7	LVCMOS33	R25	Data I/O: 16Bit data bus 0
DQ0.8	LVCMOS33	P25	Data I/O: 16Bit data bus 0
DQ0.9	LVCMOS33	N25	Data I/O: 16Bit data bus 0
DQ0.10	LVCMOS33	N24	Data I/O: 16Bit data bus 0
DQ0.11	LVCMOS33	N23	Data I/O: 16Bit data bus 0
DQ0.12	LVCMOS33	J21	Data I/O: 16Bit data bus 0
DQ0.13	LVCMOS33	J22	Data I/O: 16Bit data bus 0
DQ0.14	LVCMOS33	J23	Data I/O: 16Bit data bus 0
DQ0.15	LVCMOS33	J25	Data I/O: 16Bit data bus 0
WE1#	LVCMOS33	U26	Active low write enable bus 1
OE1#	LVCMOS33	W23	Active low output enable bus 1
BE1.0#	LVCMOS33	W22	Active low enable lower byte bus 1
BE1.1#	LVCMOS33	T21	Active low enable upper byte bus 1

SRAM			
Signal name	**I/O Standard**	**FPGA I/O**	**Comment**
DQ1.0	LVCMOS33	U25	Data I/O: 16Bit data bus 1
DQ1.1	LVCMOS33	U24	Data I/O: 16Bit data bus 1
DQ1.2	LVCMOS33	U23	Data I/O: 16Bit data bus 1
DQ1.3	LVCMOS33	U22	Data I/O: 16Bit data bus 1
DQ1.4	LVCMOS33	U20	Data I/O: 16Bit data bus 1
DQ1.5	LVCMOS33	T20	Data I/O: 16Bit data bus 1
DQ1.6	LVCMOS33	N21	Data I/O: 16Bit data bus 1
DQ1.7	LVCMOS33	P22	Data I/O: 16Bit data bus 1
DQ1.8	LVCMOS33	V23	Data I/O: 16Bit data bus 1
DQ1.9	LVCMOS33	V25	Data I/O: 16Bit data bus 1
DQ1.10	LVCMOS33	V26	Data I/O: 16Bit data bus 1
DQ1.11	LVCMOS33	U21	Data I/O: 16Bit data bus 1
DQ1.12	LVCMOS33	R21	Data I/O: 16Bit data bus 1
DQ1.13	LVCMOS33	T26	Data I/O: 16Bit data bus 1
DQ1.14	LVCMOS33	T24	Data I/O: 16Bit data bus 1
DQ1.15	LVCMOS33	T23	Data I/O: 16Bit data bus 1

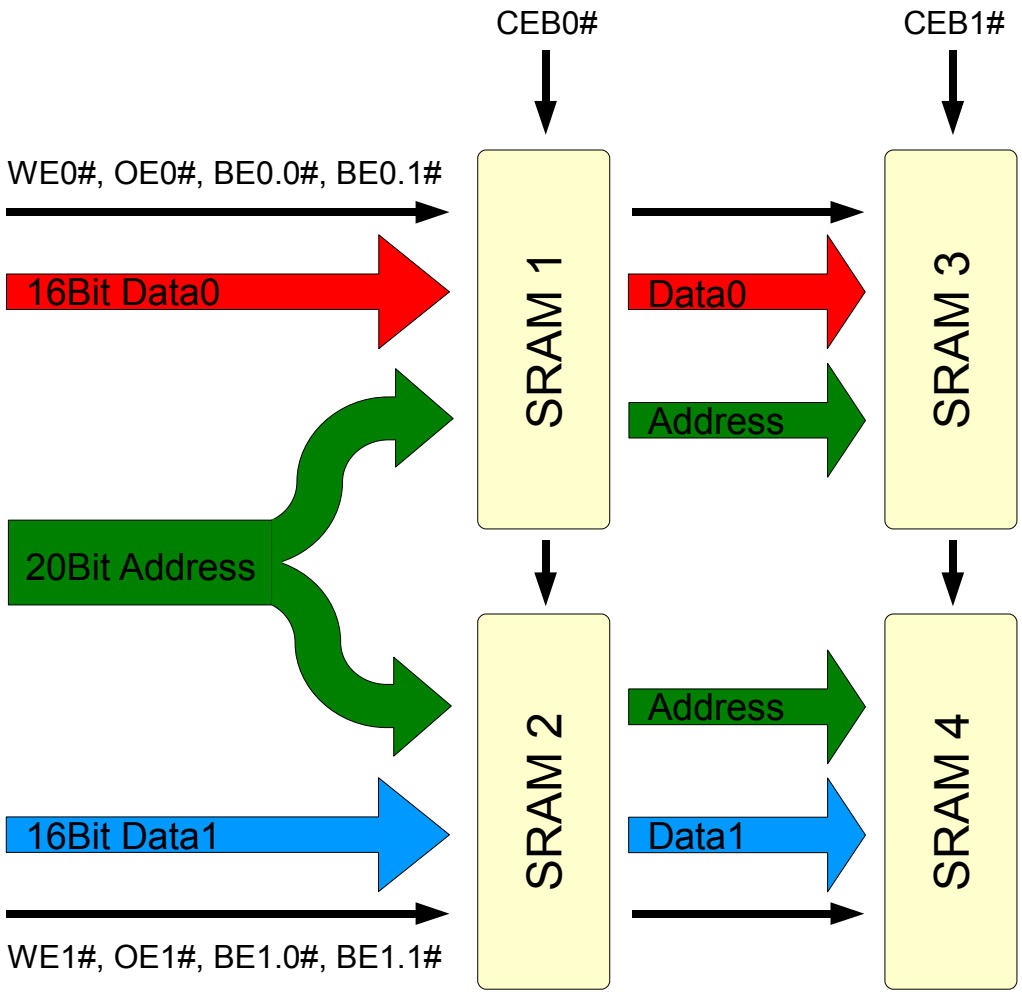


Figure 5: SRAM block diagram

DEBUG

16 FPGA I/Os are directly routed to a standard 2x8 2,54mm pitch connector. 2 additional ground pins in the immediate neighborhood simplify connection of logic analyzers for debugging purposes. A complete listing of the associated FPGA I/Os can be found in the following table:

DEBUG			
Debug pin	**FPGA I/O ball**	**Debug pin**	**FPGA I/O ball**
DBG00	F16	DBG08	D16
DBG01	C16	DBG09	F15
DBG02	D15	DBG10	F14
DBG03	E14	DBG11	D14

DEBUG			
Debug pin	**FPGA I/O ball**	**Debug pin**	**FPGA I/O ball**
DBG04	D13	DBG12	E13
DBG05	F13	DBG13	D12
DBG06	F12	DBG14	C11
DBG07	D11	DBG15	F11

! All debug I/Os are directly connected to FPGA I/Os in bank 1 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.

RS232

USBV4F provides a standard RS232 port with one receiver and one transmitter routed to a 4-pin 2,54mm pitch connector. As RS232- transceiver a MAX3227E from Maxim ™ is used. MAX3227E features EIA/ TIA-232 communications interfaces with automatic shutdown/wakeup, high data rates (up to 1MBaud) and enhanced electro-static discharge protection. A complete data sheet of this device can be downloaded from Maxim ™ website. Some more information about RS232 implementation in user designs can be found chapter D.

J75 RS232			
Pin	**Signal name**	**FPGA I/O**	**Comment**
1	**GND**	**GND**	--
2	--	--	Missing pin for reverse polarity protection
3	RX	A17	RS232 receiver input
4	TX	B15	RS232 transmitter output
--	#FORCEOFF	B14	Active low force-off input to shutdown transmitter and receiver. Overrides AutoShutdown and FORCEON
--	FORCEON	A15	Active high force-on input to override AutoShutdown (#FORCEOFF must be high)
--	#INVALID	A16	Active low valid signal detector output
--	READY	B17	Active high ready to transmit output

PIB

Like some other CESYS boards, *USBV4F* provides an extension bus to connect to user peripheral logic. In case of *USBV4F* with J22 a 208-pin high performance SAMTEC ™ QFS-connector (QFS-104-01-L-D-A) is used to give access to 206 FPGA I/Os. The PIB interface connector J22 mainly is divided into 4 ports, each connected to a different FPGA bank. Single-ended and differential signaling standards are supported. Even DCI can be enabled, as the appropriate FGPA I/Os are connected to GND or VCCO respectively with 50R- series-resistors. By adjusting jumpers, it is possible to select one out of three

voltages as VCCO for each bank individually to adjust I/O voltage. As VCCO +3,3 Volt, +2,5 Volt or a user- configurable voltage V_USER (default: +1,8 Volt) can be used. Additionally for each bank a reference voltage can be selected via jumpers to support differential signaling standards requiring reference voltages. Possible reference voltages are the onboard user- configurable reference voltage (V4F, default:+1,25 Volt) or a user generated voltage (PIB VREF) which has to be supplied on Pin 205 of extension connector J22. Further details on jumper settings can be obtained from table “Jumper settings for VCCO and VREF selection on extension bus” on page B16. A complete list of all FPGA I/Os connected to extension bus connector J22 and the corresponding FPGA bank as well as differential pairing are summarized in table “J22 Extension Bus connector”.

J22 Extension Bus connector[7]					
Pin	**Signal name**	**FPGA I/O**	**Pin**	**Signal name**	**FPGA I/O**
1	PIB000_08_27P	Y10	2	PIB001_08_31P	AF9
3	PIB002_08_27N	AA10	4	PIB003_08_31N	AE9
5	PIB004_08_29P	AC9	6	PIB005_08_25P^{CC}	AF8
7	PIB006_08_29N	AB9	8	PIB007_08_25N^{CC}	AF7
9	PIB008_08_28P^{SE}	AC7	10	PIB009_08_32P	AD8
11	PIB010_08_20P^{SE}	AA7	12	PIB011_08_32N	AC8
13	PIB012_08_19P	AF6	14	PIB013_08_30P	AE6
15	PIB014_08_19N	AF5	16	PIB015_08_30N	AD6
17	PIB016_08_24P^{CC}	AC6	18	PIB017_08_17P	AF4
19	PIB018_08_24N^{CC}	AB6	20	PIB019_08_17N	AE4
21	PIB020_08_22P	AD5	22	PIB021_08_13P	AC5
23	PIB022_08_22N	AD4	24	PIB023_08_13N	AB5
25	PIB024_08_15P	AF3	26	PIB025_08_16P	AD2
27	PIB026_08_15N	AE3	28	PIB027_08_16N	AD1
29	PIB028_08_14P	AC2	30	PIB029_08_18P	AD3
31	PIB030_08_14N	AC1	32	PIB031_08_18N	AC3
33	PIB032_08_10P	AB1	34	PIB033_08_11P	AC4
35	PIB034_08_10N	AA1	36	PIB035_08_11N	AB4
37	PIB036_08_12P^{SE}	AB3	38	PIB037_08_07P	AA4
39	PIB038_08_04P^{SE}	W4	40	PIB039_08_07N	AA3
41	PIB040_08_06P	Y2	42	PIB041_08_08P^{CC}	Y4
43	PIB042_08_06N	Y1	44	PIB043_08_08N^{CC}	Y3
45	PIB044_08_09P^{CC}	Y6	46	PIB045_08_01P	W2
47	PIB046_08_09N^{CC}	Y5	48	PIB047_08_01N	W1
49	PIB048_08_05P	W6	50	PIB049_08_02P	V6
51	PIB050_08_05N	W5	52	PIB051_08_02N	V5

7 PIBxxx_bb_ii: xxx=PIB-IO; bb=FPGA bank; ii=FPGA bank I/O; P=positive logic, N=negative logic for differential I/O standards

SE Single ended only I/O

CC Clock capable IO

J22 Extension Bus connector					
Pin	**Signal name**	**FPGA I/O**	**Pin**	**Signal name**	**FPGA I/O**
53	PIB052_10_30P	V2	54	PIB053_10_29P	V4
55	PIB054_10_30N	V1	56	PIB055_10_29N	U4
57	PIB056_10_26P	T4	58	PIB057_10_24P^{CC}	U1
59	PIB058_10_26N	T3	60	PIB059_10_24N^{CC}	T1
61	PIB060_10_28P^{SE}	U3	62	PIB061_10_32P	U6
63	PIB062_10_20P^{SE}	R4	64	PIB063_10_32N	U5
65	PIB064_10_27P	T7	66	PIB065_10_22P	R2
67	PIB066_10_27N	T6	68	PIB067_10_22N	R1
69	PIB068_10_25P^{CC}	R8	70	PIB069_10_16P	P3
71	PIB070_10_25N^{CC}	R7	72	PIB071_10_16N	P2
73	PIB072_10_18P	P5	74	PIB073_10_21P	P7
75	PIB074_10_18N	P4	76	PIB075_10_21N	P6
77	PIB076_10_14P	N3	78	PIB077_10_15P	N5
79	PIB078_10_14N	N2	80	PIB079_10_15N	N4
81	PIB080_10_11P	M2	82	PIB081_10_13P	M6
83	PIB082_10_11N	M1	84	PIB083_10_13N	M5
85	PIB084_10_09N^{CC} [8]	L8	86	PIB085_10_10P	L1
87	PIB086_10_09P^{CC} [8]	M8	88	PIB087_10_10N	K1
89	PIB088_10_05P	L7	90	PIB089_10_08P^{CC}	L4
91	PIB090_10_05N	L6	92	PIB091_10_08N^{CC}	L3
93	PIB092_10_06P	K5	94	PIB093_10_07P	K3
95	PIB094_10_06N	K4	96	PIB095_10_07N	K2
97	PIB096_10_12P^{SE}	M4	98	PIB097_10_03P	K7
99	PIB098_10_04P^{SE}	J2	100	PIB099_10_03N	K6
101	PIB100_10_01P	J7	102	PIB101_10_02P	J5
103	PIB102_10_01N	J6	104	PIB103_10_02N	J4
105	PIB104_06_32P	H2	106	PIB105_06_31P	H4
107	PIB106_06_32N	H1	108	PIB107_06_31N	H3
109	PIB108_06_29P	H6	110	PIB109_06_30P	G2
111	PIB110_06_29N	H5	112	PIB111_06_30N	G1
113	PIB112_06_28P^{SE}	G4	114	PIB113_06_26P	E1
115	PIB114_06_20P^{SE}	C2	116	PIB115_06_26N	F1
117	PIB116_06_27P	F4	118	PIB117_06_24P^{CC}	E3
119	PIB118_06_27N	F3	120	PIB119_06_24N^{CC}	E2
121	PIB120_06_25P^{CC}	D2	122	PIB121_06_22P	D3
123	PIB122_06_25N^{CC}	D1	124	PIB123_06_22N	E4
125	PIB124_06_16P	C4	126	PIB125_06_14P	A3
127	PIB126_06_16N	D4	128	PIB127_06_14N	B3
129	PIB128_06_15P	A4	130	PIB129_06_18P	E6
131	PIB130_06_15N	B4	132	PIB131_06_18N	E5

8 Please note that orientation of differential pair 10_09N/10_09P differs from other differential pairs

J22 Extension Bus connector					
Pin	**Signal name**	**FPGA I/O**	**Pin**	**Signal name**	**FPGA I/O**
133	PIB132_06_06P	A6	134	PIB133_06_08P[CC]	B6
135	PIB134_06_06N	A5	136	PIB135_06_08N[CC]	C6
137	PIB136_06_12P[SE]	C5	138	PIB137_06_17P	E7
139	PIB138_06_04P[SE]	D8	140	PIB139_06_17N	D6
141	PIB140_06_11P	B7	142	PIB141_06_03P	A8
143	PIB142_06_11N	C7	144	PIB143_06_03N	A7
145	PIB144_06_02P	D9	146	PIB145_06_13P	A9
147	PIB146_06_02N	C8	148	PIB147_06_13N	B9
149	PIB148_06_07P	E9	150	PIB149_06_01P	D10
151	PIB150_06_07N	F9	152	PIB151_06_01N	C10
153	PIB152_06_09P[CC]	G10	154	PIB153_06_05P	F10
155	PIB154_06_09N[CC]	G9	156	PIB155_06_05N	E10
157	PIBCLK1N	B10	158	PIB156_05_01P	C17
159	PIBCLK1P[9][10]	A10	160	PIB157_05_01N	D17
161	PIBCLK0N	A11	162	PIB158_05_09P[CC]	G18
163	PIBCLK0P[9]	A12	164	PIB159_05_09N[CC]	G17
165	PIB160_05_03P	B18	166	PIB161_05_07P	C19
167	PIB162_05_03N	A18	168	PIB163_05_07N	D18
169	PIB164_05_13P	A20	170	PIB165_05_02P	C20
171	PIB166_05_13N	A19	172	PIB167_05_02N	B20
173	PIB168_05_15P	A22	174	PIB169_05_06P	C21
175	PIB170_05_15N	A21	176	PIB171_05_06N	B21
177	PIB172_05_04P[SE]	D20	178	PIB173_05_14P	D22
179	PIB174_05_12P[SE]	E21	180	PIB175_05_14N	C22
181	PIB176_05_08P[CC]	A24	182	PIB177_05_10P	B24
183	PIB178_05_08N[CC]	A23	184	PIB179_05_10N	B23
185	PIB180_05_21P	D23	186	PIB181_05_16P	D24
187	PIB182_05_21N	C23	188	PIB183_05_16N	C24
189	PIB184_05_20P[SE]	C26	190	PIB185_05_25P[CC]	D26
191	PIB186_05_28P[SE]	G24	192	PIB187_05_25N[CC]	D25
193	PIB188_05_18P	E23	194	PIB189_05_27P	E25
195	PIB190_05_18N	E22	196	PIB191_05_27N	E24
197	PIB192_05_29P	F26	198	PIB193_05_24P[CC]	F24
199	PIB194_05_29N	E26	200	PIB195_05_24N[CC]	F23
201	PIB196_05_31P	G26	202	PIB197_05_32P	H26
203	PIB198_05_31N	G25	204	PIB199_05_32N	H25
205	**VREF_PIB**	--	206	PIB200_05_30P	H24
207	**+5 Volt**	--	208	PIB201_05_30N	H23

9 +3,3 Volt only clock input pins, optionally differential; usable as standard I/O if not used as clock

VCCO SELECTION ON FPGA BANKS 5, 6, 8 AND 10

With USBV4F it is possible to select between several VCCO on 4 banks individually to support multiple IO standards. Only one setting is allowed per bank. To support differential signaling standards which require reference voltages a user configurable +1,25 Volt reference voltage is available. Optional an external voltage can be used as reference (VREF_PIB) which must be provided on Extension Bus Connector J22, Pin205.

The following chart describes jumper settings for VCCO voltage selection on FPGA banks routed to Extension Bus Connector J22.

Jumper settings for VCCO and VREF selection on extension bus		
BANK5		
J16	**+3,3 Volt**	
J17	**+2,5 Volt**	
J18	**V_User**	per default V_USER is set to +1,8 Volt
J23	**VREF_onboard**	per default VREF_onboard is set to +1,25 Volt
J24	**VREF_PIB**	VREF set to external. Power supply must be provided on Extension Bus Connector J22,Pin205.
BANK6		
J13	**+3,3 Volt**	
J14	**+2,5 Volt**	
J15	**V_User**	per default V_USER is set to +1,8 Volt
J25	**VREF_onboard**	per default VREF_onboard is set to +1,25 Volt
J26	**VREF_PIB**	VREF set to external. Power supply must be provided on Extension Bus Connector J22,Pin205.
BANK8		
J10	**+3,3 Volt**	
J11	**+2,5 Volt**	
J12	**V_User**	per default V_USER is set to +1,8 Volt
J27	**VREF_onboard**	per default VREF_onboard is set to +1,25 Volt
J28	**VREF_PIB**	VREF set to external. Power supply must be provided on Extension Bus Connector J22,Pin205.
BANK10		
J7	**+3,3 Volt**	
J8	**+2,5 Volt**	
J9	**V_User**	per default V_USER is set to +1,8 Volt
J29	**VREF_onboard**	per default VREF_onboard is set to +1,25 Volt
J30	**VREF_PIB**	VREF set to external. Power supply must be provided on Extension Bus Connector J22,Pin205.

! Do not change jumper settings when board is powered ON. Never set more than one VCCO- jumper per bank at a time. Only one VREF- jumper per bank is allowed. Violation of

these restrictions will lead to severe damage of *USBV4F*.

! It is strongly recommended to check the appropriate data sheets of Virtex4™ devices about supported signaling standards and allowed maximum voltages.

LEDs

USBV4F brings along several LEDs to give further information about board states. FX2 PWR LED gives notice upon the power state of the USB- controller chip FX2LP and lights up when the +3,3 Volt supply of FX2LP is ramped up successfully. FPGA PWR LED starts to light after all necessary supply voltages to power the FPGA are ramped up successfully. USBV4F also provides an external pushbutton with an integrated LED. The PUSHBUTTON LED is connected to the FPGA DONE pin and lights up after successful device configuration. To make user design states visible it is possible to connect these to one out of 4 LEDs (LED0..LED3). Applying a logic high level to the FPGA I/O will trigger the corresponding LED on. Further details are available in the following chart.

LEDs	
LED	**Comment**
LED0 Green	FPGA I/O Ball Y22
LED1 Green	FPGA I/O Ball Y23
LED2 Green	FPGA I/O Ball Y24
LED3 Green	FPGA I/O Ball Y25
FX2 PWR LED	Power supply of USB controller is active
FPGA PWR LED	Power supply of FPGA is active
PUSHBUTTON LED	FPGA configuration done

Powering USBV4F

J2 Power Source Select	
Pin 1 shorts to Pin 2	Self powered (External +5 Volt power supply must be attached to connector J3)
Pin 3 shorts to Pin 2	Bus powered (USB power supply)

The default setting is “bus powered”. No external power supply is needed in this case, but only current to the limits of the USB bus is available (typically 500 mA). If more current is needed jumper J2 can be set to “self-powered” mode. With this setting power must be provided by an external +5 Volt power supply (not included) connected to connector J3.

J3 External Power Connector		
Pin 1	**GND**	Connect GND of external power source
Pin 2	**+5 Volt**	Connect +5 Volt external power source

J1 Select Optional Power Sequencing	
Pin 1 shorts to Pin 2	At startup only FX2LP power supply will ramp up. Only after the attached USB host grants more than 100mA FX2LP will enable power up of other onboard power supplies.[10]
Pin 3 shorts to Pin 2	All power supplies will ramp up as soon as +5 Volt are attached.[10]

If “bus powered” mode is enabled it is recommended to also enable optional power sequencing. At startup only FX2LP power supply will ramp up. With this option set it is guaranteed that USB startup power supply limits are met.

10 VIRTEX-4 power supplies will ramp up sequentially

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 *USBV4F* 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 USBV4F 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 USBV4F USB card supports “slave FIFO mode” for data communication only. “GPIF mode” is exclusively used for downloading configuration bitstreams to FPGA.

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 USBV4F 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, USBV4F 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 *USBV4F* 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 “usbv4f” demonstrates the implementation of a system-on-chip (SOC) with host software access to the peripherals like GPIOs, RS232, Flash Memory and SRAM 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 “usbv4f_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 Virtex4 XCV4LX25 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:

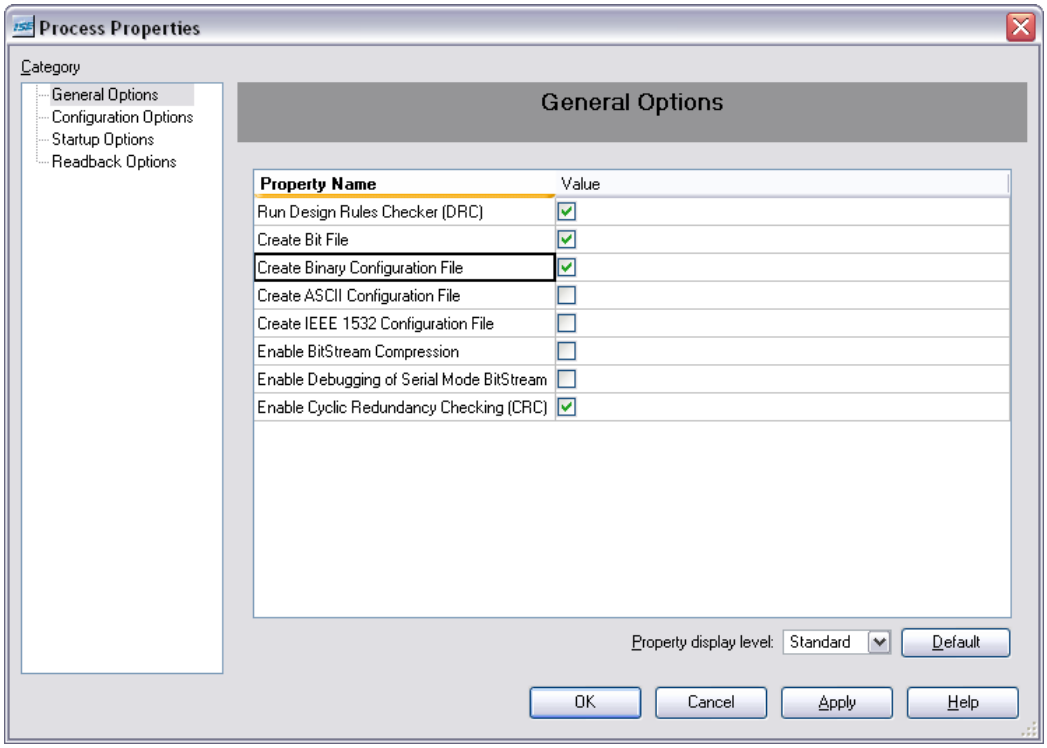


Figure 6: ISE Generate Programming File Properties (Gen. Opt.)

After `ProgramFPGA()` is called and the FPGA design is completely downloaded, the pin #RESET_FPGA (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 #RESET_FPGA 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 USBV4F 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 *USBV4F USB* CARD 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 "usbv4f"

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_".

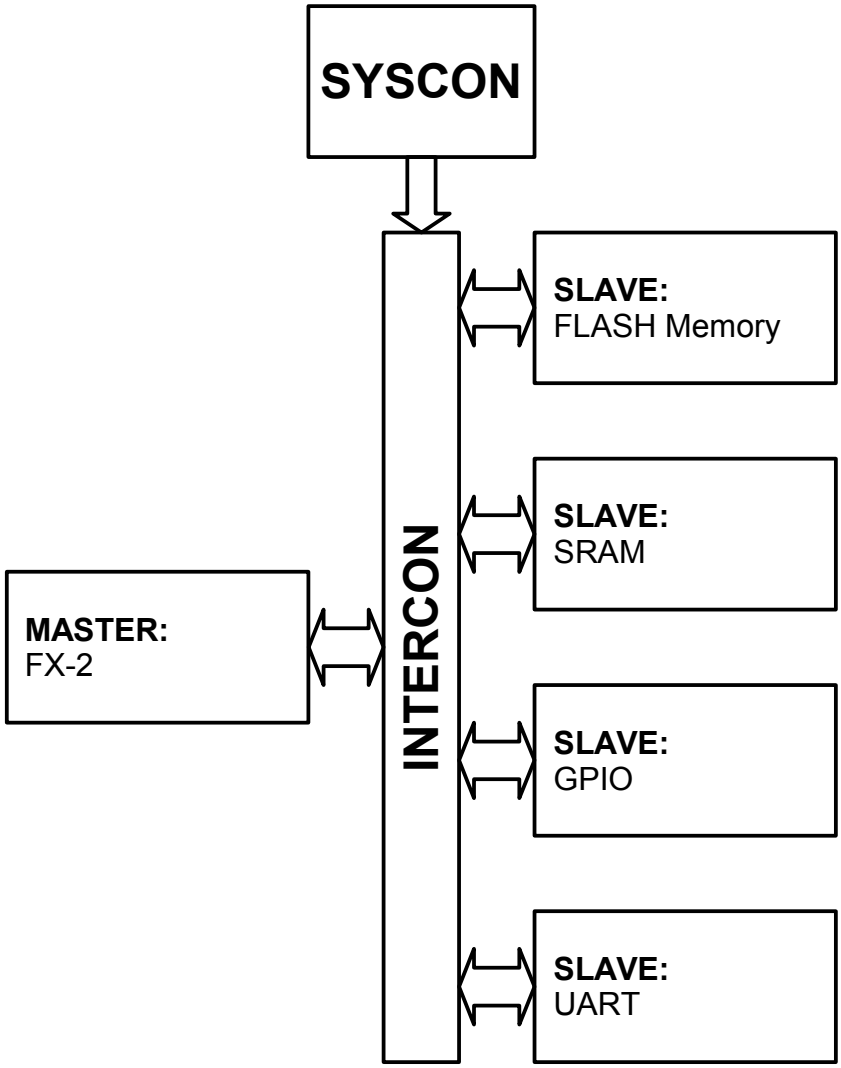


Figure 7: WISHBONE system overview (snippet)

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/usbv4f_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 #RESET_FPGA and FPGA_CCLK as external reset and clock source. FPGA_CCLK 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 the high density connector J22 for plug in boards. All 206 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 (“--”).The four LEDs are controlled and a demonstration of 16 debug signals/ports is done by this module as well.

src/wb_sl_flash.vhd:

The module encapsulates the low level FLASH controller flash_ctrl.vhd. The integrated command register supports an 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 FLASH data sheet for details on programming and erasing. It is used for programming FPGA configuration bitstreams to FLASH. Two bitstreams can be programmed, one to lower and one to upper memory space. A jumper setting selects, which one to boot.

src/wb_sl_sram.vhd:

This module allows access to onboard SRAM circuits.

src/wb_sl_uart.vhd:

This entity is a simple UART transceiver with 16 byte buffer for each direction connected to USBV4F RS232 interface. Xilinx™ UART transceiver macros are used as physical layer. Baudrate and data format are fixed at 115200/8-N-1. Reading from UART pipe is always non-blocking. A data present flag provided along with received bytes indicates, if current RX value is valid. Writing to UART pipe is blocking, if TX buffer gets full. So that loss of transmitted data can easily be avoided.

src/xil_uart_macro/:

This directory contains VHDL source code files of Xilinx™ UART transceiver macros. Note that these source code files are copyrighted by Xilinx™ and are absolutely not supported by CESYS! For details on these macros see the user manual provided by Xilinx™.

fx2_slfifo_ctrl.vhd:

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

sync_fifo16.vhd:

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

sfifo_hd_a1Kx18b0K5x36.vhd:

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

flash_ctrl.vhd:

This module is a low level parallel FLASH memory controller. It handles basic timing requirements of the asynchronous memory interface.

usbv4f.ise:

Project file for Xilinx™ ISE

usbv4f.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”.

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)

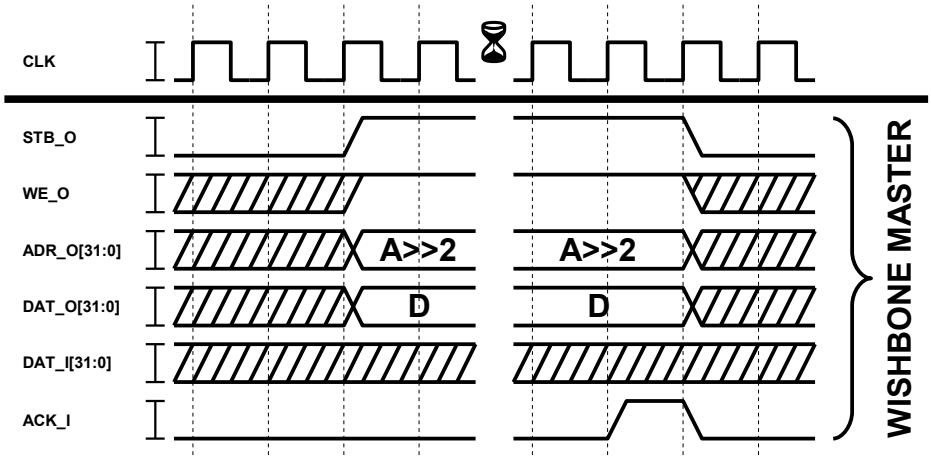


Figure 8: WISHBONE transactions with `WriteRegister()` *and* `WriteBlock()`

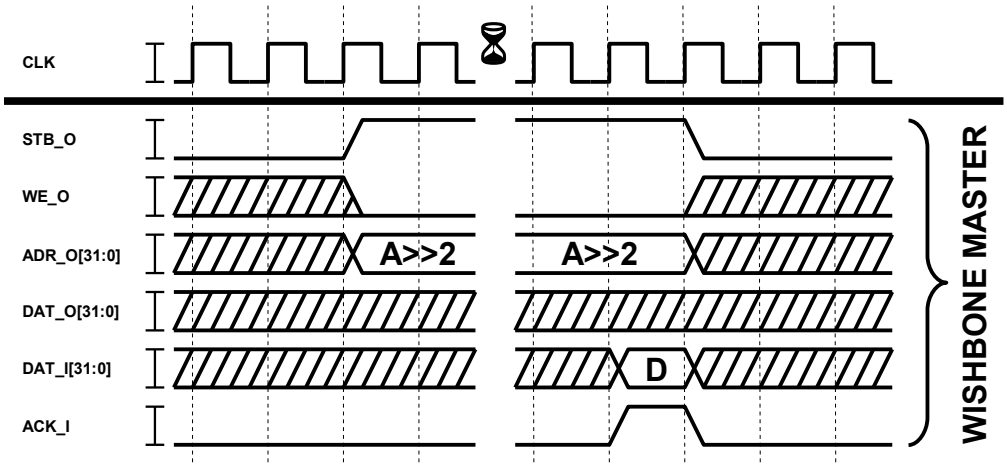


Figure 9: WISHBONE transactions with `ReadRegister()` *and* `ReadBlock()`

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 "usbv4f_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/usbv4f_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.

fx2_slfifo_ctrl.vhd:

See chapter "Design USBV4F"

sync_fifo16.vhd:

See chapter "Design USBV4F"

usbv4f_perf.ise:

Project file for Xilinx™ ISE.

usbv4f_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 possibly 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.

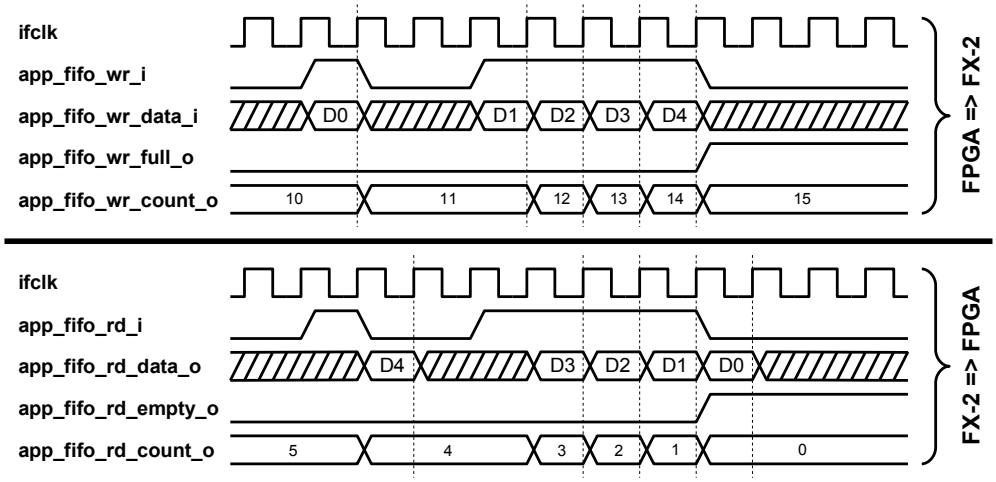


Figure 10: FIFO transactions with `ReadBulk()` and `WriteBulk()` at user logic side

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 10) 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 Cesys'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 Cesys'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 Cesys, they are offered by the manufacturer of the PCI bridges that were used on Cesys 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 **PCIeV4Base** 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: udkapic.h
- Library file:
  - Windows: udkapic_vc[ver]_[arch].lib, [ver] is *8, 9, 10*, [arch] is *x86* or *amd64*, resides in *lib/[build]/*
  - Linux: libusbapic.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 it 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 DeInit() 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 DeInit(), 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 DeInit() again, see top of this section.

DeInit

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

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 uiIdx)
C	CE_RESULT GetDevice(unsigned int uiIdx, CE_DEVICE_HANDLE *pHandle)
.NET	static ceDevice ceDevice.GetDevice(uint uiIdx)

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 DeInit() 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++	static const char *ceDevice::GetUDKVersionString()
C	const char *GetUDKVersionString()
.NET	static string ceDevice.GetUDKVersionString()

Return string which contains the UDK version in printable format.

GetDeviceUID

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

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 were the value is stored to, it must be at least of size uiDestSize.

GetDeviceName

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

Return device type name of given device pointer or handle.

Notice C API: pszDest is the buffer were 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++	void ceDevice::Open()
C	CE_RESULT Open(CE_DEVICE_HANDLE Handle)
.NET	void ceDevice.Open()

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++	void ceDevice::Close()
C	CE_RESULT Close(CE_DEVICE_HANDLE Handle)
.NET	void ceDevice.Close()

Finish working with the given device.

ReadRegister

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

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. bIncAddress 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(unsiged int uiAddress, unsigned char *pucData, unsigned int uiSize, bool bIncAddress)
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 uiAddess, byte[] Data, uint uiLen, bool bIncAddress)

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. bIncAddress 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.

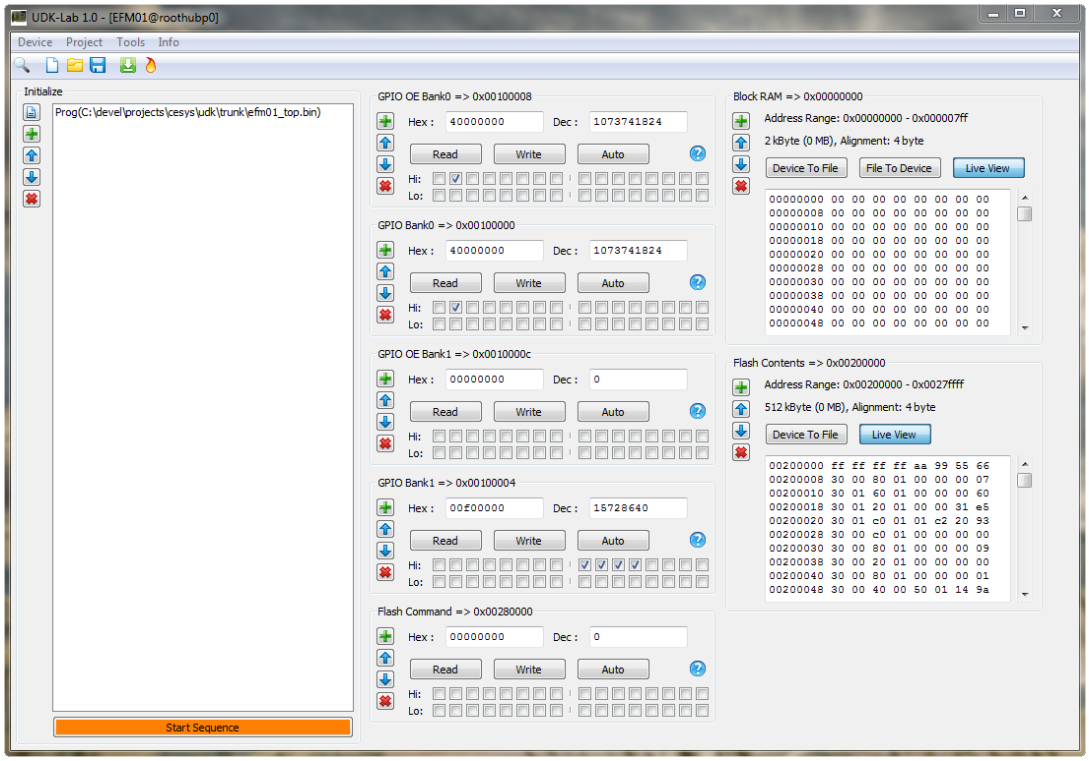


Figure 11: UDKLab Main Screen

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:

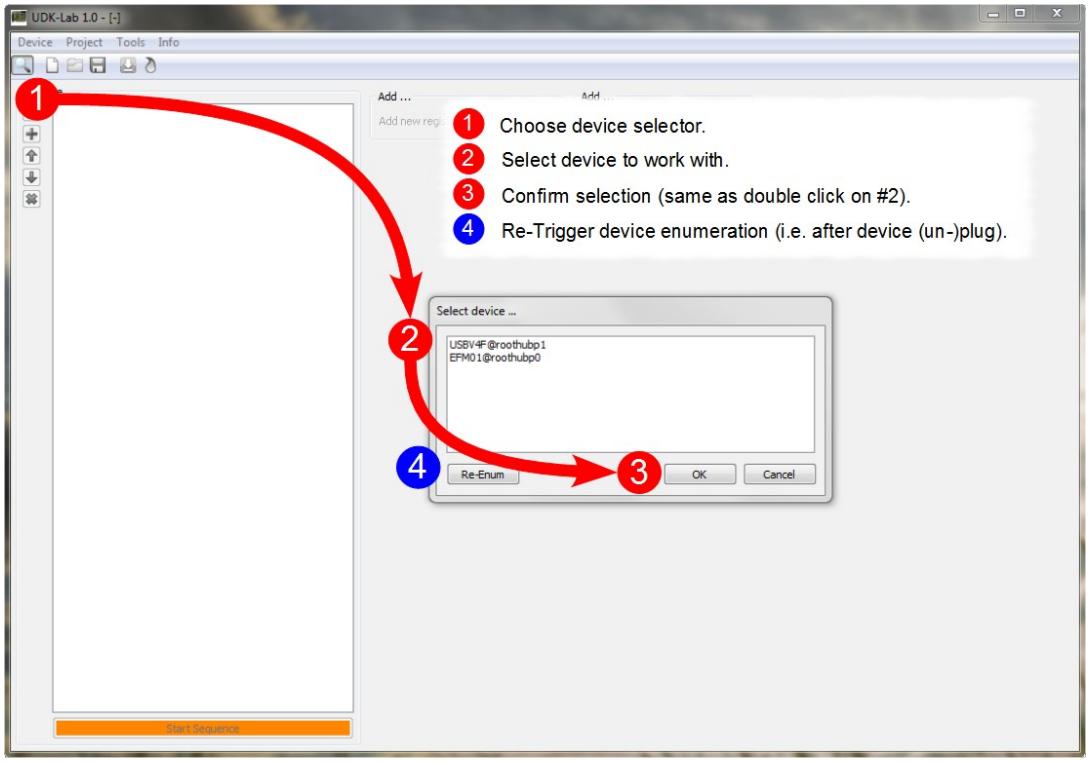


Figure 12: Device selection flow

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:

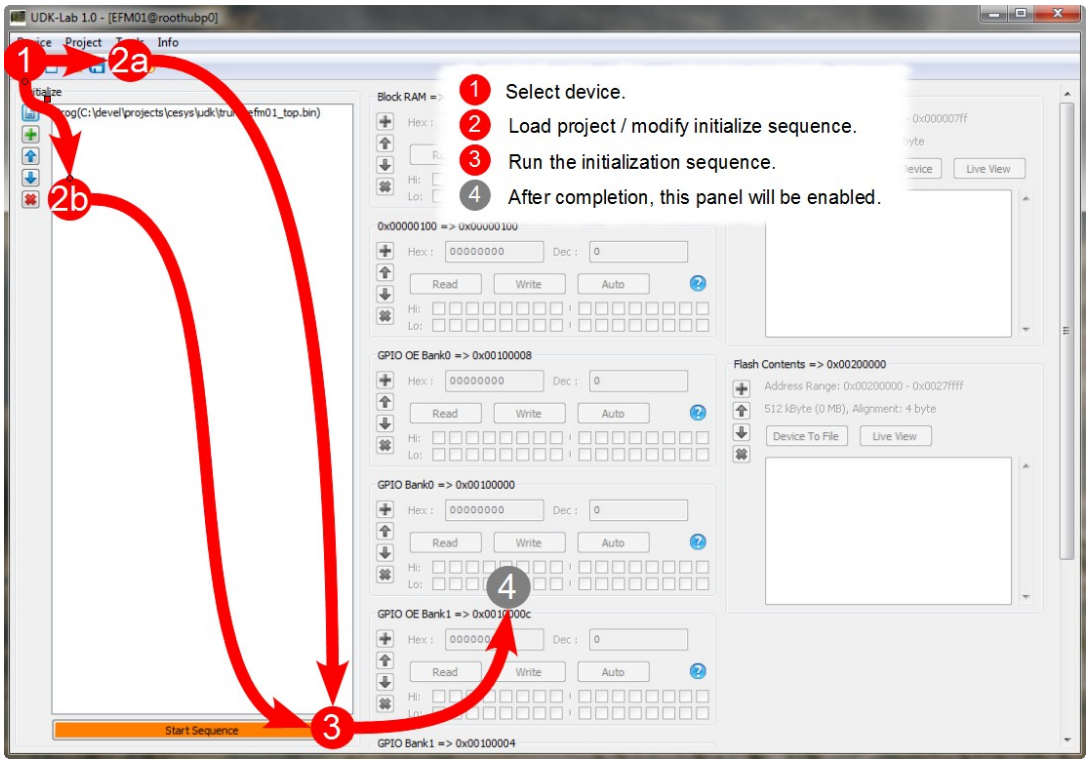


Figure 13: Prepare to work with device

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:

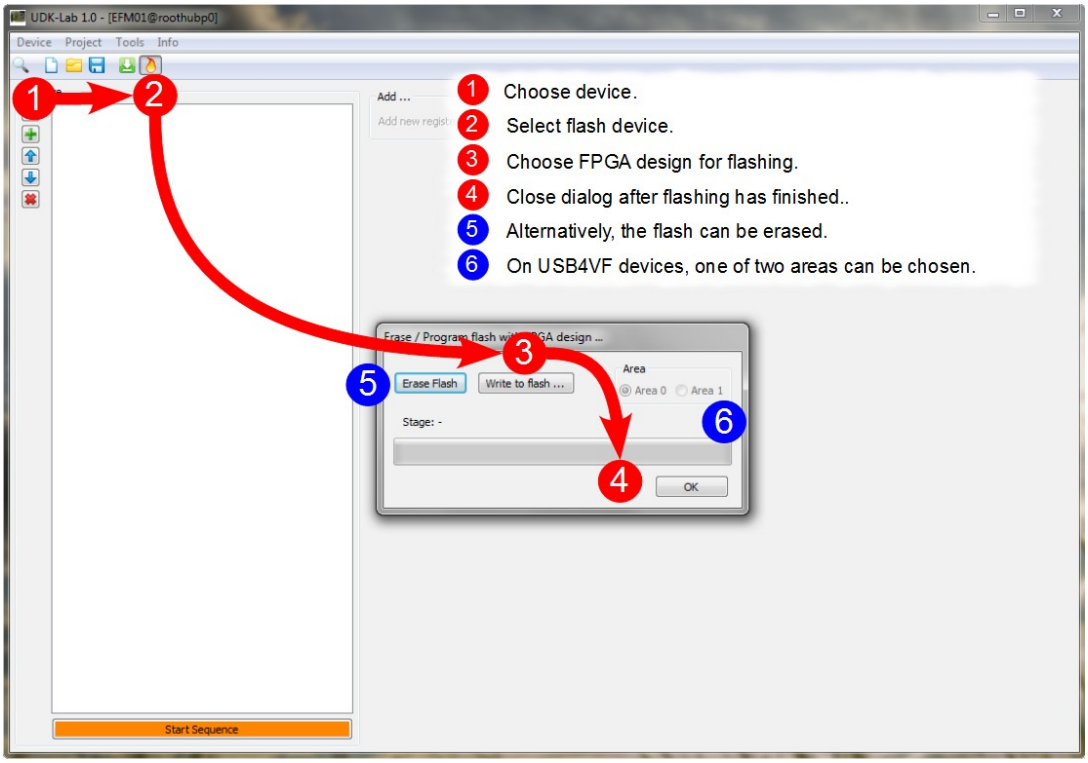


Figure 14: Flash design to device

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):

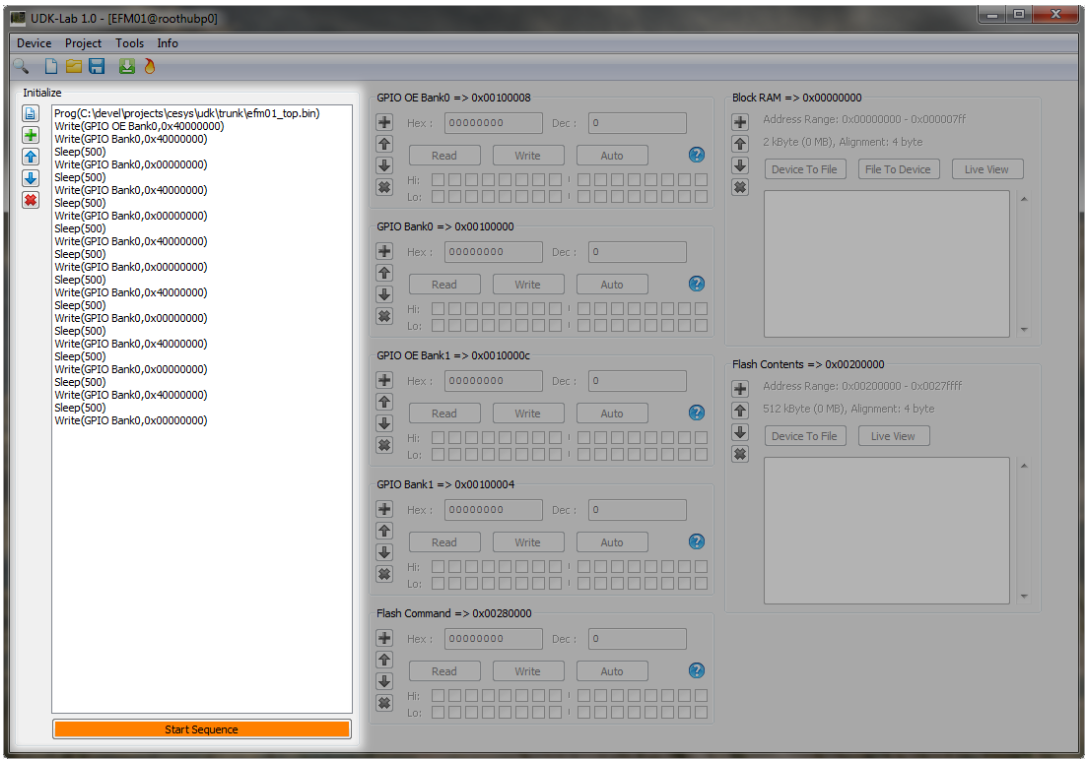


Figure 15: Initializing sequence

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:

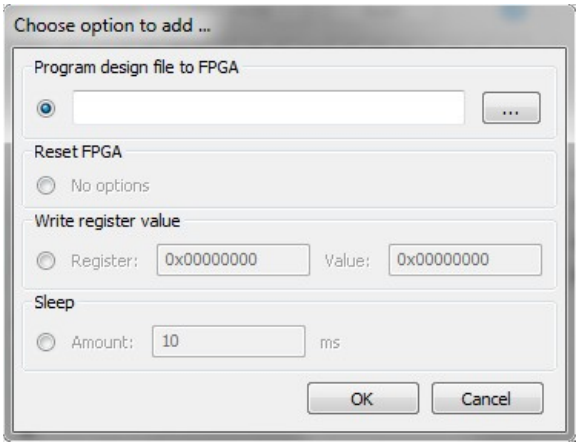


Figure 16: Add new initializing task

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 an 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 visit 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.

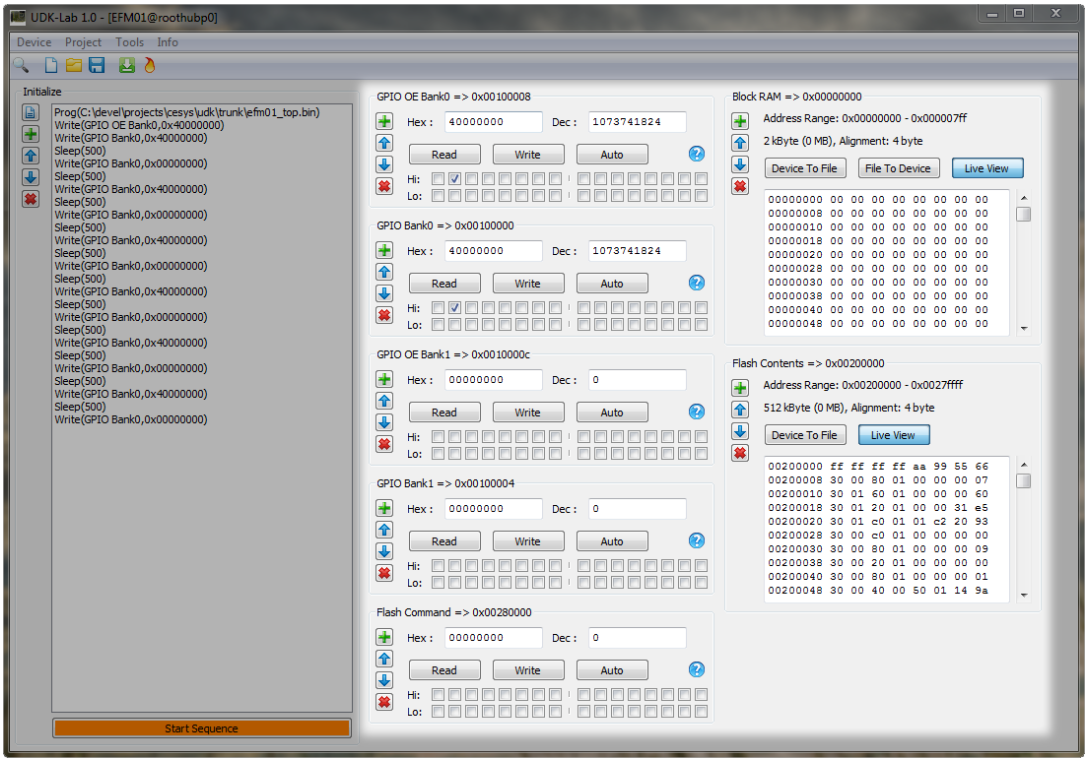


Figure 17: Content panel

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:

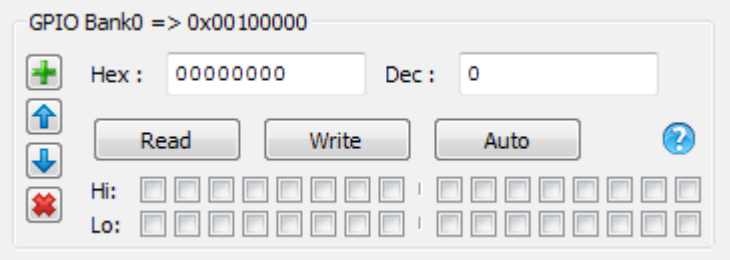


Figure 18: Register panel

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 transfered from and to files, as well as displayed in a live view. An entry constits of the following data:

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

The visual representation is shown below.

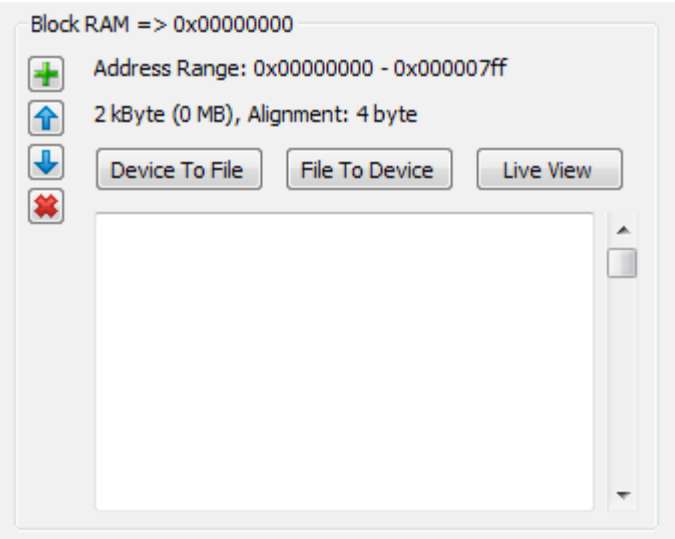


Figure 19: Data area panel

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 *USBV4F* 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 Virtex-4TM FPGA and the free FPGA development environment ISE WebPACK

Table of contents

Table of Contents