usb_serial: USB 1.1 / 2.0 serial data transfer core

Version:	2009-10-06
Author:	Joris van Rantwijk
Language:	VHDL
License:	GPL – GNU General Public License
Website:	`http://www.xs4all.nl/~rjoris/fpga/usb.html`

usb_serial is a synthesizable VHDL core, implementing serial data transfer over USB. Combined with a UTMI-compatible USB transceiver chip, this core acts as a USB device that transfers a byte stream in both directions over the bus.

Features

- Very simple application interface in terms of byte-at-a-time FIFOs. Complications involving the USB protocol (packets, endpoints, descriptors) are handled within the core. This is in fact the whole point of this package.
- UTMI[1]-compatible transceiver interface.
- Supports USB 1.1 and USB 2.0; full-speed (12 Mbit/s) or high-speed (480 Mbit/s).
- Compatible with the standard Communication Device Class as a CDC-ACM device. This implies that *no special drivers* are needed on the computer. Linux, Windows and Mac OS X all have built-in support for CDC-ACM devices. (For Windows, a special `.INF` file is needed.)
- Tested on a Xilinx Spartan-3. Works out-of-the-box on a Trenz TE0146 micromodule.

Applications

- *Drop-in replacement for RS-232 communication with a PC.*
  Many modern PCs lack a serial port. Use **usb_serial** plus an UTMI chip on your FPGA board instead of an RS-232 UART. On the PC, the device will appear as a TTY or virtual COM port. No need to modify PC software.
- *Fast data exchange with a PC.*
  Enable high speed support in **usb_serial** and pump 30 MByte/s between your FPGA and PC. Note that USB transfer rates are very dependent on the PC mainboard specs and CPU load. Custom software will be needed on the PC; virtual COM port drivers are not made for such high data rates.

Operation

At power on and after a reset pulse, the core attaches to the USB bus. If high speed support is enabled, the core tries to negotiate high speed mode with the host. Otherwise, or in case high speed negotiation fails, the core will operate in full speed mode. It responds to standard device requests from the host to obtain an address and to publish its device descriptor. The descriptor indicates CDC-ACM compatibility. Product ID, Vendor ID and Product Version can be configured at compile time.

1 *UTMI = USB 2.0 Transceiver Macrocell Interface; a standard interface for USB transceivers.*

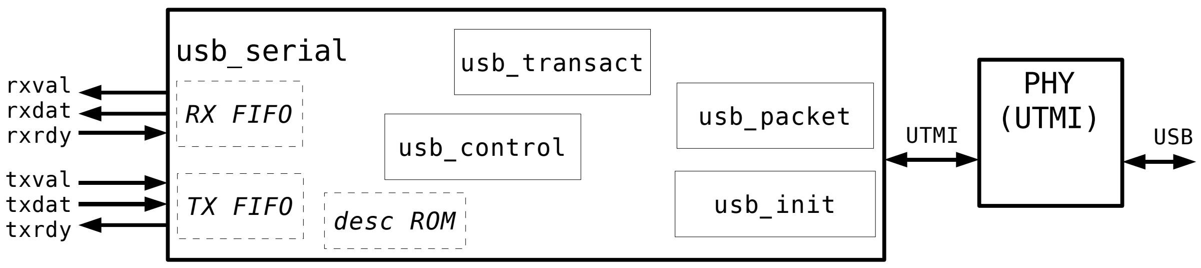

*Figure 1: **usb_serial** core with external interface and sub-entities*

Two bulk endpoints are used to transfer data; one for sending and the other for receiving. An additional interrupt endpoint is implemented as required by the CDC-ACM specification, but traffic on that endpoint is ignored.

Received data are held in a FIFO until they are accepted by the application. Likewise, the application puts data in a transmit FIFO until they can be sent on the USB bus. The application interacts with these FIFOs one byte at a time.

The core requires a 60 MHz system clock signal, synchronized to the UTMI interface signals. This clock signal is normally generated by the UTMI transceiver.

Note: An external USB 2.0 transceiver chip is required between the core and the actual USB lines. This transceiver must be compatible with the UTMI standard (USB 2.0 Transceiver Macrocell Interface). It must use an 8-bit data interface at 60 MHz.

Application interface

The application receives data by reading from the RX FIFO. When the application is ready to read data, it asserts the RXRDY signal. In response to RXRDY, if the FIFO is not empty, the core asserts RXVAL and puts a data byte on RXDAT. Whenever both RXRDY and RXVAL are high on a rising clock edge, the core assumes that a byte has been consumed and moves on to the following byte. If the application does not want to read more data, it should deassert RXRDY before the following rising clock edge.

The application sends data by putting it in the TX FIFO. As long as the FIFO is not full, the core asserts TXRDY. When the application is ready to send data, it asserts TXVAL and puts a data byte on TXDAT. Whenever both TXRDY and TXVAL are high on a rising clock edge, the core accepts a byte from TXDAT and puts it in the FIFO. The application must either provide the next data byte on TXDAT or deassert TXVAL before the following rising clock edge.

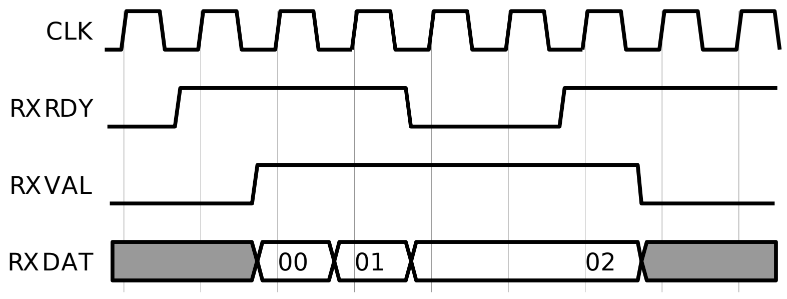

Figure 2: timing of RX FIFO interface (application accepts 3 bytes)

Configuration options

Generic	Function	Type	Default
VENDORID	Vendor ID reported in device descriptor.	16 bits	
PRODUCTID	Product ID reported in device descriptor.	16 bits	
VERSIONBCD	Product version reported in device descriptor.	16 bits	
HSSUPPORT	Include support for high speed mode.	boolean	false
SELFPOWERED	Enable self-powered bit in descriptor and status word.	boolean	false

RXBUFSIZE_BITS	Size of receive FIFO, expressed as the base-2 logarithm of the number of bytes. Must be at least 10 (1024 bytes) if high speed support is enabled.	7 - 12	11
TXBUFSIZE_BITS	Size of transmit FIFO, expressed as the base-2 logarithm of the number of bytes.	7 - 12	10

Signal descriptions

Signal name	Type	Function
CLK	Input	System clock, 60 MHz, acts on rising edge. Must be synchronized to the UTMI interface.
RESET	Input	Synchronous reset, active high. Clears buffers and re-attaches to the USB bus.
USBRST	Output	Pulsed high when a reset signal is detected on the USB bus. **Note:** Do not wire this signal to RESET; this is not needed and would lock the core in reset.
HIGHSPEED	Output	High when the device is in high speed mode (possibly suspended).
SUSPEND	Output	High while the device is suspended. **Note:** This signal is not synchronized to CLK; it may be used to combinatorially drive the UTMI SuspendM pin.
ONLINE	Output	High when the device is in *Configured* state.
RXVAL	Output	High if valid received data is available on RXDAT.
RXDAT	Out 8 bits	Received data byte.
RXRDY	Input	High if the application is ready to accept the next byte.
RXLEN	Out unsigned	Number of bytes currently available in the receive FIFO.
TXVAL	Input	High if the application is ready to send data.
TXDAT	In 8 bits	Data byte to send, must be valid if TXVAL is high.
TXRDY	Output	High if the core is ready to accept the next byte.
TXROOM	Out unsigned	Number of free byte positions in the transmit FIFO.
TXCORK	Input	Temporarily suppress transmissions on the USB bus. This may be used to pre-fill the transmit buffer before starting transmission.
PHY_DATAIN	In 8 bits	UTMI DataOut (confusing signal name; this is output from the transceiver and input to the core).
PHY_DATAOUT	Out 8 bits	UTMI DataIn (confusing signal name; this is output from the core and input to the transceiver).
PHY_TXVALID	Output	UTMI Transmit Valid.
PHY_TXREADY	Input	UTMI Transmit Data Ready.
PHY_RXACTIVE	Input	UTMI Receive Active.
PHY_RXVALID	Input	UTMI Receive Data Valid.
PHY_RXERROR	Input	UTMI Receive Error.
PHY_LINESTATE	In 2 bits	UTMI Line State.
PHY_OPMODE	Out 2 bits	UTMI Operational Mode.
PHY_XCVRSELECT	Output	UTMI Transceiver Select: selects between HS and FS mode.
PHY_TERMSELECT	Output	UTMI Termination Select: selects between HS and FS termination.
PHY_RESET	Output	UTMI Reset.

Host software

Data transfer is done using two bulk endpoints. Data sent by the host to the device through endpoint 1_OUT ends up in the receive FIFO. Data queued in the transmit FIFO are sent to the host through endpoint 1_IN.

The device is accessible from software running on the host. This typically requires interaction with the USB driver in the operating system. How this is done depends on the operating system used.

Linux:

Linux 2.6 has a built-in class driver for CDC-ACM devices. This driver, called `cdc-acm`, works with **usb_serial**. If the driver is installed, it should automatically detect the device when it is plugged in. A device node `/dev/ttyACM`*n* will be created (look at the kernel messages for the exact device name). This device node can be accessed by application software as if it were a serial port.

Alternatively, the generic USB serial driver for Linux, called `usbserial`, can be used. The device's Product ID and Vendor ID must be passed as parameters when loading the driver. The device will be accessible as `/dev/ttyUSB`*n*.

Both drivers have problems with flow control: if the device sends more data than can be processed by the host software, some data may be lost. This is especially relevant to high speed mode. At low data rates (a few kByte/s) the problem will not occur.

Alternatively, the host application can directly access the USB device through `libusb` [2]. This approach provides better performance than the serial drivers, especially in high speed mode.

Mac OS X:

On Mac OS X, the device is automatically recognized as a USB modem. It appears as device node `/dev/tty.usbmodem`*XXXX*. This device can be used by application software as if it were a serial port. Flow control does not work properly.

Windows:

The device works with `usbser.sys` (provided with Windows) and will show up as an extra `COM`*n*`:` port. A custom `.INF` file is needed to tell Windows that it must use this driver. Have a look at the sample file `fpgaser.inf` [3] to see how it can be done. Note that Product ID and Vendor ID in the file must match those of the device. Helpful instructions can also be found in `Documentation/usb/gadget_serial.txt` in the Linux kernel source tree.

The Windows 2000 version of `usbser.sys` is notoriously broken and should not be used. Windows XP seems to do better. Flow control has not been tested and probably does not work correctly.

Source code structure

The main VHDL entity of the core is `usb_serial`. It contains four sub-entities, as well as FIFOs, descriptor ROM, buffer management logic and glue logic. The package consists of the following VHDL files:

`usb_serial.vhdl`	Main entity.
`usb_init.vhdl`	Initialization, handshake and reset detection.
`usb_packet.vhdl`	Packet handling.
`usb_transact.vhdl`	Transaction handling.
`usb_control.vhdl`	Default control endpoint.
`usb_pkg.vhdl`	Package with entity declarations.
`usbtest.vhdl`	Top-level entity consisting of usb_serial and a simple test application.

Synthesis

The VHDL code in this package is readily synthesizable for the Xilinx Spartan-3 family of FPGA devices. The code should in principle be portable to other FPGA platforms. A tricky issue could be the inference of Block RAM. The receive FIFO, transmit FIFO and descriptor ROM are coded in VHDL in such a way that the Xilinx synthesis tool automatically infers Block RAM primitives. Some tweaking may be needed to get this to work with different synthesis software.

2 `http://www.libusb.org/`

3 *contributed by Hans Hübner*

A Makefile is supplied which invokes the Xilinx tools to synthesize a test design for the Trenz TE0146 XC3S1000 micromodule [4]. With some changes, it should also work for different Xilinx targets. A constraint file will be needed to map I/O signals to specific pins and to specify timing constraints. The file `te0146.ucf` contains constraints that are suitable for the Trenz TE0146 module.

Resource usage

Indication of resource requirements, based on synthesis of **usb_serial** as the top level design for a Xilinx XC3S1000. (Xilinx Webpack 7.1i)

Configuration	Full-speed only, 128 byte FIFOs	High-speed support, 2k RX / 1k TX FIFO.
Nr of slice flip flops	235	285
Nr of 4-input LUTs	841	1062
Nr of occupied slices	479 (6%)	610 (7%)
Nr of Block RAMs	3	3
Total equivalent gate count	204,527	206,559

Testing

The top-level entity `usbtest.vhdl` may be used to test **usb_serial**. It implements a state machine, responding to commands received from the host.

You can use minicom (on Linux) or hyperterminal (on Windows) to connect to the device and play with it. The Python script `testdev.py` thoroughly tests various buffering and flow control scenarios. The C program `perftest.c` uses `libusb` to test the bandwidth of data transfers.

For example, with a TE0146 module connected to a Linux host, you could do this:

1. Run `make` to build `usbtest.bit` (or use the precompiled `usbtest_20091006_hs.bit`).
2. Download the `.bit` file to the FPGA module.
3. Run `lsusb` to check that the device is recognized.
   Also look at the kernel messages to check that CDC-ACM support is recognized. It should say something about a new `ttyACM0` device.
4. Use `minicom` to connect to `/dev/ttyACM0`.
   Type "`Hello`" and press Enter. The device should answer "olleH".
5. Run `python testdev.py /dev/ttyACM0` to start the torture test.
   The test takes about 20 minutes to complete.
6. Run `perftest -d fb9a:fb9a` (as root) to start the performance test.

Performance

Tested under Linux 2.6.31 with libusb-1.0, using asynchronous requests.

Bus speed	**Measured IN bandwidth**	**Measured OUT bandwidth**
full speed	1,088,118 bytes/s	1,088,073 bytes/s
high speed	43,727,704 bytes/s	32,635,212 bytes/s

Known limitations

- The TEST_MODE feature (mandatory for high speed devices) is not implemented. This should not affect normal operation of the device, but the device would fail USB conformance tests.
- The default control pipe does not strictly verify requests from the host. As a result, invalid requests from the host may appear to succeed when they should have returned an error code. This should not be an issue with a correctly working host controller, but it would probably cause the device to fail USB conformance tests.

4 *http://www.trenz-electronic.de/*

- In its configuration descriptor, the core claims to support AT-style commands. This is not true, but it is needed to get the device recognized by the Linux CDC-ACM driver. In reality, the core ignores the mandatory CDC requests SET_ENCAPSULATED_COMMAND and GET_ENCAPSULATED_RESPONSE.
- In certain situations, the device may send more bytes than expected by the host. This is flagged as a *babble error* by the host. This may occur, for example, when the host submits an IN request which is not a multiple of the maximum packet size. To avoid this, always submit IN requests with the transfer size set to a multiple of the maximum packet size. Note that this is not a bug in the device core, rather an inconvenience in the USB software stack.
- Depending on FPGA platform and board design, a bus powered device may not be able to respond to the initial host requests after plugging in. For example, a Xilinx FPGA typically needs more than 100 ms to initialize after power on. By this time the host will have tried to communicate with the device. Most operating systems will successfully retry the initialization procedure.
- The application interface of the core must be synchronized to the 60 MHz UTMI clock. This is inconvenient when the core is embedded in a larger system with its own master clock. To overcome this problem, separate clock domains should be created inside the core to decouple the application clock from the UTMI clock.
- There are several bugs in the Windows 2000 version of `usbser.sys`. It is known to freeze the pipe when it receives an unexpected zero-length packet. This happens if the device sends an exact multiple of the maximum packet size while the host attempted to read exactly that same amount of data.