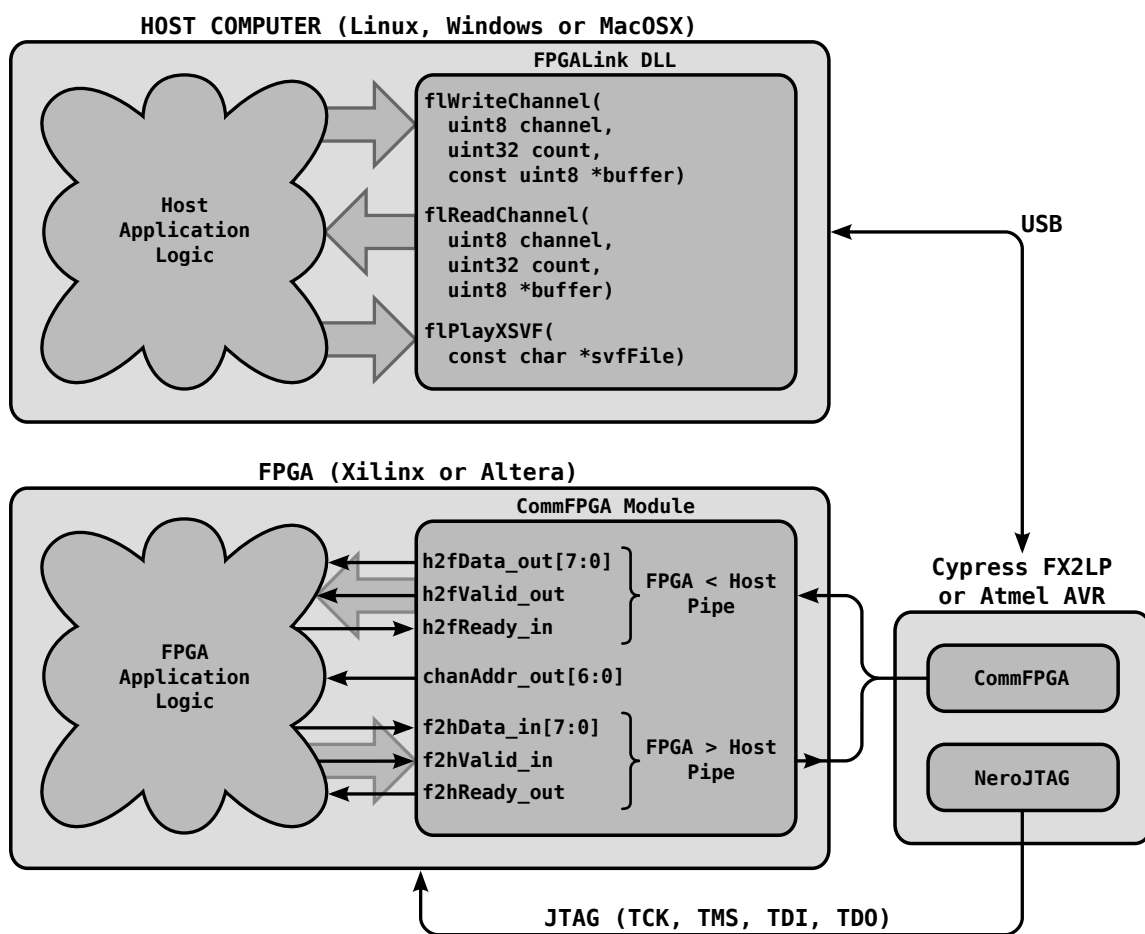


FPGALink User Manual

(Verilog Edition)

©2012 Chris McClelland/MakeStuff.eu



This work is licensed under the Creative Commons Attribution-ShareAlike 3.0 Unported License

Contents

| | | |
|----------|--|-----------|
| 1 | Introduction | 3 |
| 1.1 | Justification | 3 |
| 1.2 | Overview | 3 |
| 1.3 | Document Conventions | 4 |
| 1.4 | How to Get Help | 4 |
| 1.5 | Licences & Disclaimers | 5 |
| 2 | Getting Started | 6 |
| 2.1 | Installation | 6 |
| 2.2 | Supported Boards | 7 |
| 2.2.1 | Cypress FX2LP-Based Boards | 7 |
| 2.2.2 | Atmel AVR-Based Boards | 9 |
| 2.3 | The <code>flcli</code> Utility | 9 |
| 2.4 | Programming the FPGA | 12 |
| 2.5 | Interacting with the FPGA (Part 1) | 14 |
| 2.6 | Interacting with the FPGA (Part 2) | 16 |
| 2.7 | Summary | 18 |
| 3 | Host Application Development | 19 |
| 3.1 | Language Bindings | 19 |
| 3.1.1 | C | 19 |
| 3.1.2 | Python | 20 |
| 3.1.3 | Excel/VBA | 20 |
| 3.2 | API Overview | 20 |
| 3.2.1 | Firmware Operations [FX2LP-Specific] | 21 |
| 3.2.2 | Connection Lifecycle Operations | 21 |

| | | |
|----------|---|-----------|
| 3.2.3 | Device Capabilities and Status Operations | 22 |
| 3.2.4 | NeroJTAG Operations | 22 |
| 3.2.5 | CommFPGA Operations | 22 |
| 3.2.6 | Miscellaneous Operations | 23 |
| 4 | FPGA Application Development | 24 |
| 4.1 | Internal Port | 24 |
| 4.2 | FX2 External Port | 25 |
| 4.3 | EPP External Port | 26 |
| 4.4 | Build Infrastructure | 27 |
| 4.4.1 | MacOSX | 27 |
| 4.4.2 | Windows | 27 |
| 4.4.3 | Linux | 28 |
| 4.4.4 | Specifying the JTAG Chain | 28 |
| 4.4.5 | Location Constraints | 28 |
| 4.4.6 | The xsvf2csvf Utility | 28 |
| A | Custom Boards | 30 |
| A.1 | Minimus/EP2C5 | 30 |
| A.2 | Minimus/Nexys2 | 31 |

1 Introduction

1.1 Justification

Development kits for Field Programmable Gate Arrays (FPGAs) are ubiquitous, with offerings from a plethora of manufacturers, with prices ranging from the tens of dollars to well into the thousands, and featured FPGAs ranging from a few thousand logic cells to a few million. Whereas the high-end boards tend to be PCIx plug-in cards, the cheaper boards tend either to be designed around a USB interface chip (e.g Cypress FX2LP, Atmel AVR, Microchip PIC or FTDI chip), or lack direct host interfacing altogether, requiring a standalone JTAG cable for programming.

Unfortunately, even for those boards designed around a USB interface, there is a general lack of good integrated solutions for exchanging arbitrary data between the host computer and the FPGA, once it has been programmed.

1.2 Overview

FPGALink is an end-to-end, open-source, cross-platform solution designed to do a couple of simple jobs, and do them well:

- Program an FPGA with JTAG, either from an onboard configuration source or over USB.
- Allow the host and/or microcontroller to exchange arbitrary binary data with the FPGA.

It provides a host-side API, firmware for several USB interface microcontrollers, and 128 addressable eight-bit read/write FIFOs on the FPGA side.

- On the host side there is a dynamic-link library with a straightforward API. Library and example application binaries are provided for MacOSX (x86_64 & i386), Windows (i686) and Linux (x86_64, i686, ARM & PowerPC). Bindings are provided for C/C++, Python and Excel/VBA, but binding other languages is straightforward.
- For the USB interface there are firmwares for the Cypress FX2LP (used on most Digilent, KNJN, ZTEX and Opal Kelly boards) and Atmel AVR (used by some AVNet and Digilent boards). Support for the FTDI chips is planned.
- The Cypress FX2LP firmware supports a synchronous FIFO interface with a sustained bandwidth of around 26MiB/s. The Atmel AVR firmware supports

an asynchronous interface¹ with a sustained bandwidth of around 1.2MiB/s. Other microcontroller-to-FPGA protocols such as SPI would be straightforward to implement.

- For the FPGA there is a simple interface module which when instantiated in your design gives the host a FIFO-style read/write interface, supporting up to 128 separate logical “channels” into your design. A couple of fully-functional example designs are provided to get you started.

Everything is licensed under the GNU Lesser General Public Licence²; you are therefore free to distribute unmodified copies of FPGALink with your products. The library has no commercial or hardware platform usage restrictions, so you can prototype your design with an inexpensive devkit, and then use the same software tools on your custom-built PCBs. In this way you can easily distribute updated FPGA designs to your customers just as you would with regular firmware updates, with no special programming cables required, making your FPGA truly “field-programmable”.

1.3 Document Conventions

Whilst describing interactive console sessions, I will use **monospace bold** for characters entered by a human and `monospace regular` for the computer’s responses.

Remember:

- 1MB = 1 megabyte = 10^6 bytes.
- 1MiB = 1 mebibyte = 2^{20} bytes.
- 1Mb = 1 megabit = 10^6 bits.
- 1Mib = 1 mebibit = 2^{20} bits.

1.4 How to Get Help

The only place you’re *guaranteed* to get a response to FPGALink-related queries is the FPGALink Users Group at <http://groups.google.com/group/fpgalink-users>.

¹Actually IEEE 1284 in Enhanced Parallel Port mode.

²<http://www.gnu.org/copyleft/lesser.html>

1.5 Licences & Disclaimers

The FPGALink library, firmware & Verilog code is licensed under the LGPLv3³:

Copyright ©2009-2012 Chris McClelland

FPGALink is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

FPGALink is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

The FLCLI utility is licensed under the GPLv3⁴:

Copyright ©2009-2012 Chris McClelland

FLCLI is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

FLCLI is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

³<http://www.gnu.org/copyleft/lesser.html>

⁴<http://www.gnu.org/copyleft/gpl.html>

2 Getting Started

2.1 Installation

First, download the FPGALink binary distribution. This manual assumes you're using [libfpgalink-20121106.tar.gz](#).

Linux:

Just download the binary distribution and unpack it into your home directory. Things will work out-of-the-box on most modern distributions.

Separate sets of binaries for x86_64, i686, ARM and PowerPC architectures are provided in the `linux.*` directories.

To grant regular users permission to access the USB devices you'll be using, you will need to add `udev` rules. First check which groups you're in by running “`groups`”, choose a group (I chose “`users`”) and then for each USB device, add a line to `/etc/udev/rules.d/10-local.rules`:

```
wotan$ sudo tee -a /etc/udev/rules.d/10-local.rules > /dev/null <<EOF
> ACTION=="add", SUBSYSTEM=="usb", \
> ATTR{idVendor}=="1443", ATTR{idProduct}=="0005", \
> GROUP="users", MODE="0660"
> EOF
wotan$
```

You may need to restart the `udev` service with `sudo service udev restart`, but you will definitely need to unplug and reconnect the device(s) before the new permissions will be in effect.

Windows:

You will need to install the [VC++ 2010 redistributable package](#) and a USB driver for your board:

- Be sure to uninstall any existing driver for your board before you start.
- Download [LibUSB-Win32](#) and run `bin/inf-wizard.exe`.
- Click “Next”, select your FPGA board, make a note of the vendor and product IDs and click “Next” twice.
- Choose a location for the driver and click “Save”.
- Click “Install Now”.

Also, although it's not strictly required, all the command-line examples in this manual assume you're using the [MakeStuff Build Infrastructure](#). It's also *the*

only supported build platform on Windows, so whilst you're free to use something else, you're on your own if you do.

One set of binaries for the i386 architecture is supplied in the **win32** directory.

MacOSX:

You just need to install LibUSB⁵ and you're good to go.

One set of universal binaries for x86_64 and i386 architectures is provided in the **darwin** directory.

2.2 Supported Boards

By providing multiple “board support packages” (BSPs), FPGALink is able to support several different FPGA development kits, based on Xilinx or Altera FPGAs and Atmel AVR or Cypress FX2LP USB microcontrollers⁶.

A pair of fully-functional examples is provided, with ready-to-go programming files for each of the supported FPGA boards. The first example is **ex_cksum**, which we will play with in the next section, and the second is **ex_fifo**. Both use various bits of board I/O to provide some interaction and visual feedback:

- Eight slider-switches, providing eight bits of input data.
- Eight LEDs, providing visual feedback of an eight-bit binary value.
- Four seven-segment displays, wired with common cathodes and an anode for each digit, providing visual feedback of a sixteen-bit hexadecimal value (and an additional four-bit binary value via the four decimal point LEDs).

Obviously, not all of the supported FPGA boards include all of these I/O features, and may have other special requirements before the supplied examples will work.

2.2.1 Cypress FX2LP-Based Boards

The Cypress FX2LP is a Hi-Speed (480Mb/s) USB interface. It is capable of transferring data between an FPGA and the host at 26MiB/s, using an eight-bit synchronous FIFO interface. Although fast and fairly cheap, it has a rather eccentric internal architecture, so custom firmware development is not recommended.

⁵See <http://www.makestuff.eu/wordpress/?p=1760>

⁶Porting to other boards based on these components is usually fairly trivial; if you have such a board, ask on the [mailing list](#).

Digilent Nexys3:

There are no special considerations for the Nexys3.

Digilent Nexys2 (500K & 1200K versions):

Separate BSPs are provided for the 500K and 1200K gate versions. Also, when the Nexys2 “power select” jumper is set to “USB”, the FPGA is supplied with power via a little FET on the board which is under software control. Therefore before programming the FPGA it’s necessary to turn this FET on.

Digilent Atlys:

The Atlys has the requisite LEDs and switches, but does not have a display of any kind, so on Atlys the provided examples map the seven-segment display signals to the first twelve pins on the board’s VHDCI connector. Unless you know exactly what you’re doing, please ensure you have nothing connected to the VHDCI port before you begin.

KNJN Xylo-L:

Since the Xylo-L has no onboard peripherals, the provided examples map the switches, LEDs and seven-segment displays to the board’s expansion connector marked H4. Unless you know exactly what you’re doing, please ensure you have nothing connected to the H4 port before you begin.

Digilent S3BOARD:

There is only partial support for the S3BOARD, because it does not include a USB interface of any kind; instead, it is supplied with a parallel-port JTAG cable. To use the S3BOARD with FPGALink, you will need an external FX2LP board with the appropriate connections⁷.

Hypothetical LX9 Board:

I have provided the LX9 BSP for a custom board based on the Spartan-6 LX9 FPGA. Since I don’t even have a schematic for it, it’s purely hypothetical, but may be of use to someone.

⁷FX2FPGA serves as a reference design, but suitable commercial boards are also available.

2.2.2 Atmel AVR-Based Boards

The USB-capable Atmel AVR8 microcontrollers (AT90USB* & ATmega*U*) are very cheap, and they incorporate a Full-Speed (12Mb/s) USB interface. They are capable of transferring data between an FPGA and the host at between 330KiB/s and about 1.2MiB/s, using an eight-bit asynchronous EPP-style interface. Although slower than the FX2LP, they are much more popular as general-purpose microcontrollers, and much easier to program, making custom firmware development fairly straightforward.

Minimus/EP2C5 Board:

You can buy⁸ a Minimus AT90USB162 board for £4, and an EP2C5 Mini Board for £17. With suitable interconnects, the pair make up by far the cheapest FPGALink-capable hardware solution.

Minimus/Nexys2 Board:

There is also a BSP for a Minimus board attached to a Nexys2-1200. It is unlikely to be of interest because the Nexys2-1200 already has a superior FX2LP-based USB interface.

Both of these boards require custom wiring, which is described in Appendix A.

2.3 The flcli Utility

Since FPGALink is a library, you would normally just embed it into your application, but in order to get you started, the FPGALink binary distribution includes **flcli**⁹, a small command-line utility which provides access to some of the library's features.

```
wotan$ linux.x86_64/rel/flcli --help
```

```
FPGALink Command-Line Interface Copyright (C) 2012 Chris McClelland
```

```
Usage: flcli [-psch] [-i <VID:PID>] -v <VID:PID> [-j <portSpec>] [-x <fileName>]
        [-a <actionString>]
```

Interact with an FPGALink device.

| | |
|-----------------------|--|
| -i, --ivp=<VID:PID> | vendor ID and product ID (e.g 04B4:8613) |
| -v, --vp=<VID:PID> | vendor ID and product ID (e.g 04B4:8613) |
| -j, --jtag=<portSpec> | JTAG port config (e.g D0234) |
| -x, --xsvf=<fileName> | SVF, XSVF or CSVF file to load |
| -p, --power | FPGA is powered from USB (Nexys2 only!) |
| -s, --scan | scan the JTAG chain |

⁸e.g from ModTraders and eBay, respectively

⁹Unlike the rest of FPGALink, the **flcli** utility is licensed under the terms of the GNU General Public License. See <http://www.gnu.org/copyleft/gpl.html>

-a, --action=<actionString> a series of CommFPGA actions
-c, --cli start up an interactive CommFPGA session
-h, --help print this help and exit

--ivp=<VID:PID> [FX2LP-specific]

The FX2LP microcontroller has no onchip nonvolatile storage, so it loads its firmware from an external EEPROM and/or has its firmware loaded over USB. The **flcli** utility first tries to connect to the device specified by **--vp**; if it fails, it tries to load firmware into the device specified by **--ivp**.

--jtag=<portSpec> [FX2LP-specific]

By default the FX2LP firmware's JTAG interface is compatible with the wiring on Digilent boards (**TDO**, **TDI**, **TMS** & **TCK** connected to port D bits 0, 2, 3 & 4 respectively.) If your board has different connections, you will need to specify which port lines to use along with **--ivp**. The port specification is the port ("C" or "D") followed by four digits for the port bits to use for **TDO**, **TDI**, **TMS** & **TCK**. For example, the Xylo-L board uses a JTAG port of "D1240".

--vp=<VID:PID>

The FPGALink device to connect.

--power [Nexys2-specific]

If your Nexys2's power jumper is set to "USB", it's necessary to switch on the FPGA's power manually using this option.

--scan

Print an IDCODE for each device in the JTAG chain.

--xsvf

Play the specified **.svf**, **.xsvf** or **.csvf** file into the JTAG chain. This is typically (but not necessarily) used for FPGA programming.

--action=<actionString>

Execute the semicolon-separated list of CommFPGA commands, for reading and writing FPGA channels.

--cli

Start a command-line interface for executing CommFPGA commands.

The `--cli` and `--action=<actionString>` options support three commands, “**r**” (read), “**w**” (write) and “**q**” (quit). The syntax of the read command is as follows:

r<channel> [<count> [<fileName>]]

channel:

The FPGA channel, 0-7f.

count:

How many bytes to read from the FPGA channel, default 1.

fileName:

A binary file (in “quotes”) to write the FPGA’s data to.

If you don’t specify a **fileName**, the FPGA’s data is printed to stdout as a hex dump.

The syntax of the write command is as follows:

w<channel> <byteSeq | fileName>

channel:

The FPGA channel, 0-7f.

byteSeq:

A sequence of bytes to be written to the FPGA channel,
e.g **0123456789abcdef**.

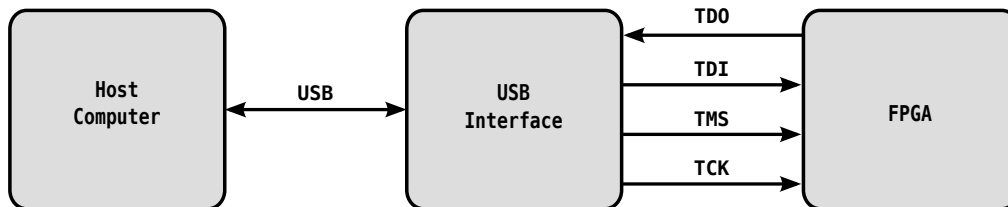
fileName:

An existing binary file (in “quotes”) to dump into the FPGA.

All numbers are in hexadecimal. Since a byte is two hex digits, the **byteSeq** must have an *even* number of digits. Filenames must be quoted using double-quotes (“”). You may put several read and/or write commands on one line, separated by semicolons (;).

2.4 Programming the FPGA

Using the `flcli` utility, we can now program the FPGA for the first time. Programming is done by playing a scripted set of JTAG operations into the FPGA's JTAG port.



You can program the FPGA like this:

```
wotan$ ./<PLATFORM>/rel/flcli -v <VID:PID> -i <VID:PID> \
      -s -x gen_csvf/ex_cksum_<BOARD>_<PROTOCOL>_verilog.csvf
Attempting to open connection to FPGALink device 04B4:8613...
Loading firmware into 04B4:8613...
Awaiting renumeration.....
Attempting to open connection to FPGALink device 04B4:8613 again...
The FPGALink device at 04B4:8613 scanned its JTAG chain, yielding:
    0x01414093
    0x05045093
Playing "gen_csvf/ex_cksum_s3board_fx2_verilog.csvf" into the JTAG chain on FPGA-
Link device 04B4:8613...
wotan$
```

You will need to replace `<PLATFORM>`, `<VID:PID>`, `<BOARD>` and `<PROTOCOL>` in the above command-line with values appropriate for your specific host platform and FPGA board:

`<PLATFORM>`

Your host platform. This will be one of `win32`, `darwin`, `linux.x86_64`, `linux.i686`, `linux.armel` or `linux.ppc`.

`<VID:PID>`

The correct vendor and product IDs for your board. This will be:

- `1443:0005` for Nexys2
- `1443:0007` for Nexys3 & Atlys
- `04B4:8613` for Xylo-L

`<BOARD>`

The name of your board, which will be one of `nexys2-500`, `nexys2-1200`,

nexys3, atlys, xylo-1, s3board, ep2c5 or lx9.

<PROTOCOL>

The protocol to use to talk to the FPGA, which will be one of **fx2** or **epp**.

There are some caveats:

Nexys2 Users:

If your board’s “power select” jumper is set to “USB”, it will be necessary to switch on the FPGA’s power by supplying an additional **-p** command-line option.

Xylo-L Users:

The FX2LP port lines used for JTAG differ from the default, so you will need to pass an additional **-j D1240** command-line option.

AVR-Based Boards:

The **-i <VID:PID>** option is only needed for FX2LP-based boards. It is not needed for AVR-based boards.

If successful, you should see the “Done” light on your board switch on, and “0000” appear on your board’s seven-segment display. If your board does not have either of these, don’t worry; as long as the **flcli** command completed without error, you can proceed to the next section.

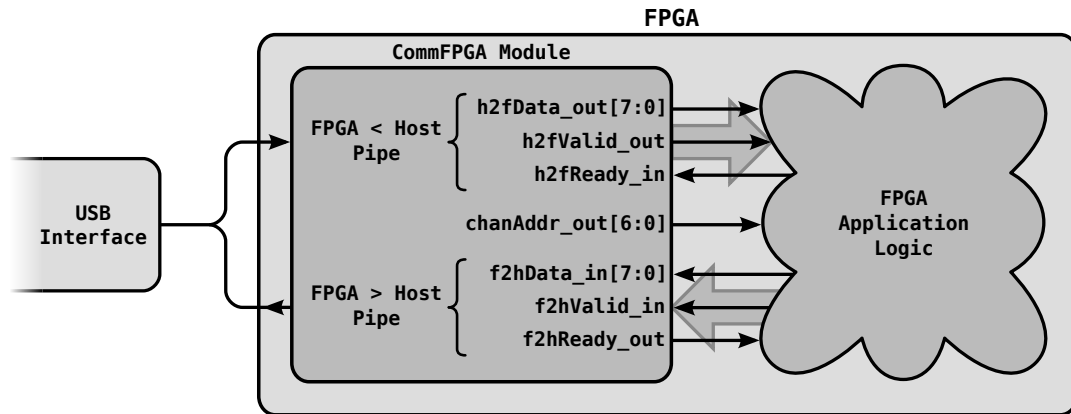
So what just happened? Well, **flcli** loads new firmware if necessary (FX2LP-based boards only), then powers up the FPGA (Nexys2 only), then scans the board’s JTAG chain for attached devices, and finally loads a pre-built design file for the **ex_cksum** example into the FPGA.

In this case we used a **.csvf** file¹⁰ to program the FPGA, but FPGALink directly supports the **.svf** files generated by the Xilinx and Altera tools, as well as the Xilinx-specific **.xsvf** format.

¹⁰The CSVF format is similar to Xilinx’s XSUF, but it’s better suited for playback by small microcontrollers, and is much more space-efficient.

2.5 Interacting with the FPGA (Part 1)

A couple of interface modules are supplied, “`comm_fpga_fx2`”¹¹ and “`comm_fpga_epp`”¹² for you to instantiate in your application’s Verilog code. The former has an external interface compatible with the Cypress FX2LP’s slave FIFO signals and the latter has an external interface compatible with the EPP signals provided by the Atmel AVR firmware. However, both have identical *internal* interfaces, each providing a read pipe, a write pipe and a seven-bit address specifying which of 128 logical *channels* is to be read or written.



Each channel is eight bits wide. Each channel may be read from or written to by the host. Each read or write operation can deal with single bytes or many hundreds of megabytes. Applications are free to choose how these channels are implemented in their Verilog code. Two such implementations are given in the `ex_cksum` and `ex_fifo` examples.

In the previous section, we loaded the `ex_cksum` example into the FPGA. The `ex_cksum` example implements the FPGA channels using ordinary registers, meaning that writes to channel N update `regN`, and reads from channel N typically return the current value of `regN`.

You can see this by using `flcli`’s command-line mode (triggered by the `-c` option):

```
wotan$ ./<PLATFORM>/rel/flcli -v <VID:PID> -c
Attempting to open connection to FPGALink device 04B4:8613...

Entering CommFPGA command-line mode:
> w1 12;w2 34;w3 abcdef56;r1;r2;r3 04
      00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000 12 34 56 56 56 56                                     .4VVVV
> w0 01
> w0 02
> w0 04
> w0 08
```

¹¹hdl/fx2/verilog/comm_fpga/comm_fpga_fx2.v.

¹²hdl/epp/verilog/comm_fpga/comm_fpga_epp.v.

```
> r0 10
      00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000 35 35 35 35 35 35 35 35 35 35 35 35 35 35 35 35 5555555555555555
> q
wotan$
```

The instructions accepted by the command-line are terse, but simple. The “**w1 12;w2 34;w3 abcdef56;r1;r2;r3 04**” command line performs six operations sequentially:

- Write single byte **0x12** to channel 1.
- Write single byte **0x34** to channel 2.
- Write **four** bytes **0xAB**, **0xCD**, **0xEF**, **0x56** to channel 3.
- Read one byte from channel 1.
- Read one byte from channel 2.
- Read **four** bytes from channel 3.

The output shows “12 34 56 56 56 56” because in the `ex_cksum` example, the data channels inside the FPGA are implemented with simple registers, so although four distinct bytes are written to channel 3, the four bytes that are later read back from channel 3 are just four copies of the *last value written*.

You can see how this is implemented in the FPGA by taking a look at the Verilog¹³:

```
1 // Infer registers
2 always @(posedge fx2Clk_in)
3 begin
4     checksum <= checksum_next;
5     reg0 <= reg0_next;
6     reg1 <= reg1_next;
7     reg2 <= reg2_next;
8     reg3 <= reg3_next;
9 end
10
11 // Drive register inputs for each channel when the host is writing
12 assign checksum_next =
13     (chanAddr == 7'b00000000 && h2fValid == 1'b1) ?
14         checksum + h2fData :
15     (chanAddr == 7'b00000001 && h2fValid == 1'b1 && h2fData[0] == 1'b1) ?
16         16'h0000 :
17     checksum;
18 assign reg0_next =
19     (chanAddr == 7'b00000000 && h2fValid == 1'b1) ? h2fData : reg0;
20 assign reg1_next =
```

¹³See `hdl/{fx2,epp}/verilog/ex_cksum/top_level.v` for the complete Verilog module.


```

21 | (chanAddr == 7'b00000001 && h2fValid == 1'b1) ? h2fData : reg1;
22 | assign reg2_next =
23 | (chanAddr == 7'b00000010 && h2fValid == 1'b1) ? h2fData : reg2;
24 | assign reg3_next =
25 | (chanAddr == 7'b00000011 && h2fValid == 1'b1) ? h2fData : reg3;
26 |
27 | // Select values to return for each channel when the host is reading
28 | assign f2hData =
29 | (chanAddr == 7'b00000000) ? sw_in :
30 | (chanAddr == 7'b00000001) ? reg1 :
31 | (chanAddr == 7'b00000010) ? reg2 :
32 | (chanAddr == 7'b00000011) ? reg3 :
33 | 8'h00;
34 |
35 | // Assert that there's always data for reading, and always room for writing
36 | assign f2hValid = 1'b1;
37 | assign h2fReady = 1'b1;

```

So when you enter “w1 12”, a single byte is written to channel 1, which sets **chanAddr=0x01** and **h2fData=0x12**, and drives **h2fValid** high for one clock cycle, updating register **reg1** with the value **0x12**.

Similarly when you enter “r1”, it initiates a single byte read of channel 1, which sets **chanAddr=0x01** and drives **f2hReady**¹⁴ high for one clock cycle, which samples the current value of register **reg1** and returns it to the host.

Register **reg0** is a little more interesting. Any value written to it (e.g “w0 aa”) is displayed on the eight board LEDs, and is added to a running sixteen-bit checksum which is displayed on the seven-segment display. Reads from register **reg0** return the current state of the eight switches. You can clear the **reg0** checksum by writing a ‘1’ to **reg1**.

Notice that reads and writes never block: there is always data available in the read pipe and there is always room available in the write pipe.

2.6 Interacting with the FPGA (Part 2)

In the previous section, we used the **flcli** utility to read and write simple registers implemented in the FPGA by the **ex_cksum** example. The **ex_fifo** example is more interesting. You can load it like this:

```

wotan$ ./<PLATFORM>/rel/flcli -v <VID:PID> -i <VID:PID> -s \
      -x gen_csvf/ex_fifo_<BOARD>_<PROTOCOL>_verilog.csvf
Attempting to open connection to FPGALink device 04B4:8614...

```

¹⁴The value stored in a simple register like this is unaffected by a read operation, so it’s not necessary for this code to consider **f2hReady** - it need only select the correct register value with which to drive **f2hData**, and do so continuously.

```

Loading firmware into 04B4:8614...
Awaiting reenumeration.....
Attempting to open connection to FPGALink device 04B4:8614 again...
Connecting USB power to FPGA...
The FPGALink device at 04B4:8614 scanned its JTAG chain, yielding:
    0x01414093
    0x05045093
Playing "gen_csvf/ex_fifo_s3board_fx2_verilog.csvf" into the JTAG chain on FPGA-
Link device 04B4:8614...
wotan$ ./<PLATFORM>/rel/flcli -v 04B4:8614 -c
Attempting to open connection to FPGALink device 04B4:8614...

Entering CommFPGA command-line mode:
> r0 10;w0 01020408102040804020100804020100
      00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F .....
> q
wotan$

```

Now, channel 0 is connected to a pair of FIFOs inside the FPGA, a read FIFO and a write FIFO. Separate producer and consumer processes periodically insert upcount data into the read FIFO and drain data from the write FIFO, respectively. The speed at which these processes work is selectable by setting different values on the eight switches: **sw[7:4]** control the speed of the consumer and **sw[3:0]** control the speed of the producer.

Here's the code¹⁵:

```

1 // Infer registers
2 always @(posedge fx2Clk_in)
3     count <= count_next;
4
5 // Wire up write FIFO to channel 0 writes:
6 //   flags(2) driven by writeFifoOutputValid
7 //   writeFifoOutputReady driven by consumer_timer
8 //   LEDs driven by writeFifoOutputData
9 assign writeFifoInputData = h2fData;
10 assign writeFifoInputValid =
11     (h2fValid == 1'b1 && chanAddr == 7'b00000000) ? 1'b1 : 1'b0;
12 assign h2fReady =
13     (writeFifoInputReady == 1'b0 && chanAddr == 7'b00000000) ? 1'b0 : 1'b1;
14
15 // Wire up read FIFO to channel 0 reads:
16 //   readFifoInputValid driven by producer_timer
17 //   flags(0) driven by readFifoInputReady
18 assign count_next = (readFifoInputValid == 1'b1) ? count + 1'b1 : count;
19 assign readFifoInputData = count;
20 assign f2hValid =
21     (readFifoOutputValid == 1'b0 && chanAddr == 7'b00000000) ? 1'b0 : 1'b1;
22 assign readFifoOutputReady =

```

¹⁵See `hdl/{fx2,epp}/verilog/ex_fifo/top_level.v` for the complete Verilog module.

```

23     (f2hReady == 1'b1 && chanAddr == 7'b00000000) ? 1'b1 : 1'b0;
24
25     // Select values to return for each channel when the host is reading
26     assign f2hData =
27         (chanAddr == 7'b00000000) ? readFifoOutputData : // get data from read FIFO
28         (chanAddr == 7'b00000001) ? fifoCount[15:8] : // read depth of the write FIFO
29         (chanAddr == 7'b00000010) ? fifoCount[7:0] : // read depth of the write FIFO
30         8'h00;

```

When the host writes to channel 0, each byte is clocked into the write FIFO. When the write FIFO fills up, **h2fReady** is deasserted which tells the host to stop sending until the consumer process has freed up some room for more data.

When the host reads from channel 0, each byte is clocked out of the read FIFO. When the read FIFO empties, **f2hValid** is deasserted to stop sending data to the host until the producer process has inserted some more data into the FIFO.

When the example is first loaded, you will see the leftmost pair of digits on the seven-segment display incrementing, telling you how many bytes there are in the read FIFO. When you issue the command **r0 10**, the host reads sixteen bytes from the read FIFO, causing the count to decrement by sixteen. If there are as yet insufficient bytes in the read FIFO to fulfill the request, the host blocks until bytes become available.

Similarly, the rightmost pair of digits on the seven-segment display tells you how many bytes there are in the write FIFO. When you issue the command **w0 01020408102040804020100804020100**, sixteen bytes are written to the write FIFO. The consumer process then begins to drain the data, displaying each byte in turn on the eight LEDs. If there is as yet insufficient room in the write FIFO to fulfill the request, the host blocks until enough room is available. This is not always noticeable because the microcontrollers have additional bytes of FIFO space.

If your board does not have a seven-segment display, you can read the current depth of the write FIFO by reading from channel 1, and you can read the current depth of the read FIFO by reading from channel 2.

2.7 Summary

In this chapter we used the **flcli** utility to program an FPGA with a couple of pre-built examples, and to communicate with the data channels in the FPGA.

3 Host Application Development

The `flcli` utility is a great way to get started with FPGALink, and for simple testing of the behaviour of your Verilog code, but sooner or later you will want to dive in and write your own host-side applications. There is out-of-the-box support for C/C++, Python and Excel/VBA, but the core library itself is just a C DLL, so calling into it from other languages is straightforward¹⁶.

In order to enable an application to communicate with potentially many FPGALink devices, the library uses the concept of a “handle” to refer to an FPGALink device. An opaque handle is returned when a connection is first established to a device, and that handle is supplied for all subsequent operations on that device.

3.1 Language Bindings

There is obviously a significant cross-language semantic overlap in the API; the only differences are those imposed by the languages themselves.

3.1.1 C

Because C has no concept of exceptions, wherever a function *can* fail, its return value is just a status code; the actual result (if any) is provided in an “out” parameter instead. An optional error message is also provided in an “out” parameter. Conversely, wherever a function *cannot* fail, its return value (if any) is actually the result.

Each function that *can* fail will return `FL_SUCCESS` on success, or something else on failure. Each will also accept a pointer to a `const char*` which will be set to a more or less meaningful message if an error occurs. The memory for this message is dynamically allocated and must subsequently be deallocated by application code with `flFreeError()`. If you do not wish to receive error messages you can just set this parameter to `NULL`.

It’s useful to define a macro to handle this:

```
1 | #define CHECK(x) \
2 |     if ( status != FL_SUCCESS ) { \
3 |         returnCode = x; \
4 |         fprintf(stderr, "%s\n", error); \
5 |         flFreeError(error); \
6 |         goto cleanup; \
7 |     } \
8 |     :
9 | status = flWriteChannel(handle, 1000, 0x01, 1, &byte, &error);
10| CHECK(21);
```

¹⁶Or at least it *ought* to be straightforward!

To get you started, there is an example in C for you to study in the **examples/c** subdirectory. See **examples/c/README** for details of how to compile the code on your platform.

3.1.2 Python

Python has exceptions, so the API in Python is written such that each function's return value (if any) is actually its result, with an exception thrown when an error occurs.

Separate bindings are provided for Python2.x and Python3.x. These reside in the **examples/python** subdirectory. The bindings are themselves executable, and offer similar functionality to the C example. The Python binding may be imported into your own code, or used from an interactive Python session. For more details, see **examples/python/README**.

3.1.3 Excel/VBA

VBA has exceptions, so the API in VBA is written such that each function's return value (if any) is actually its result, with an exception thrown when an error occurs.

The **examples/excel/fpgalink.xls** spreadsheet incorporates the VBA binding, and exposes a simple graphical user interface. Before opening it, please ensure you have unpacked the FPGALink distribution to a *local* drive, not a network drive, otherwise Excel will consider the FPGALink DLL to be untrusted and will refuse to load it.

3.2 API Overview

See http://www.swaton.ukfsn.org/apidocs/libfpgalink_8h.html for more detailed API documentation.

The library consists of five classes of functions:

- Firmware operations (FX2LP-specific)
- Connection lifecycle operations
- NeroJTAG operations (programming the FPGA)
- CommFPGA operations (interacting with the FPGA)
- Miscellaneous operations

Each will now be covered in turn; please refer also to the detailed API docs.

3.2.1 Firmware Operations [FX2LP-Specific]

The Cypress FX2LP USB interface has no internal nonvolatile storage for firmware. On startup it typically loads firmware from an external serial EEPROM, which is probably how your board works. You can easily load new firmware over USB, or write your new firmware to the external EEPROM using these operations.

flLoadStandardFirmware():

Load standard FPGALink firmware into the FX2's RAM.

flFlashStandardFirmware():

Flash standard FPGALink firmware into the FX2's EEPROM, optionally appending an SVF, XSVF or CSVF initialisation stream and an FPGA initialisation stream.

flLoadCustomFirmware():

Load custom firmware from a **.hex** file into the FX2's RAM.

flFlashCustomFirmware():

Flash a custom firmware from a **.hex** or **.iic** file into the FX2's EEPROM.

flSaveFirmware():

Save existing EEPROM data to an **.iic** file.

3.2.2 Connection Lifecycle Operations

These two operations enable you to actually establish a connection to an FPGALink device over USB.

flOpen():

Open a connection to the FPGALink device at the specified VID & PID.

flClose():

Close the connection to the FPGALink device.

3.2.3 Device Capabilities and Status Operations

These operations enable you to query the FPGALink device to find out what features it supports. Currently there are only two features: **NeroJTAG** for JTAG-programming, and **CommFPGA**, for communicating with an already-programmed FPGA.

flIsDeviceAvailable():

Check if a given device is actually connected to the system.

flIsNeroCapable():

Check to see if the device supports NeroJTAG.

flIsCommCapable():

Check to see if the device supports CommFPGA.

3.2.4 NeroJTAG Operations

The NeroJTAG operations enable you to examine the JTAG chain and program devices in the chain.

flScanChain():

Scan the JTAG chain and return an array of IDCODEs.

flPlayXSVF():

Play an SVF, XSVF or CSVF file into the JTAG chain.

3.2.5 CommFPGA Operations

The CommFPGA operations enable you to read from and write to up to 128 logical “channels” implemented in the FPGA.

flIsFPGARunning():

Check to see whether or not the FPGA has been programmed.

flReadChannel():

Read bytes from the specified channel into the supplied buffer.

flWriteChannel():

Write bytes from the the supplied read-only buffer to the specified channel.

flAppendWriteChannelCommand():

Append a write command to the end of the write buffer.

flPlayWriteBuffer():

Play the write buffer into the FPGALink device immediately.

flCleanWriteBuffer():

Clean the write buffer (if any).

3.2.6 Miscellaneous Operations

Some operations are useful or just necessary, but don't fit into the other four categories:

flInitialise():

Initialise the library. You must call this before any of the other functions.

flFreeError():

Free an error allocated when one of the other functions fails.

flSleep():

Sleep for the specified number of milliseconds.

flLoadFile():

Return a newly-allocated buffer with the specified binary file loaded into it.

flFreeFile():

Free a buffer previously returned by **flLoadFile()**.

flPortAccess():

Access port lines on the USB interface microcontroller.

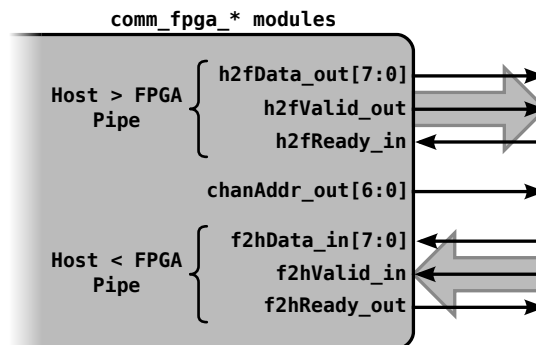
4 FPGA Application Development

FPGALink supports several different CommFPGA protocols for data transfer between the USB microcontroller and the FPGA, with varying throughputs, FPGA pin utilisations and component costs. Note that even though their internal implementation differs, the various CommFPGA implementations present exactly the same host-side API and exactly the same FPGA-side FIFO interfaces.

Whilst it is of course possible to implement one or more of the CommFPGA protocols yourself, there would be little point in doing so, because included in the FPGALink distribution are a couple (with more to follow) of ready-made infrastructure modules for this purpose, `comm_fpga_fx2` and `comm_fpga_epp`, which you can instantiate in your designs. The modules each have two “ports”, an *external* port which is typically wired directly to external pins on the FPGA, and an *internal* port which is connected to application logic. The external port signals of the **fx2** and **epp** flavours differ, but the internal ports are identical.

4.1 Internal Port

The internal port is common to both **fx2** and **epp** implementations. All signals are synchronous to the `comm_fpga_*` module’s clock.



The `comm_fpga_*` modules provide:

- A host-to-FPGA data pipe (`h2fData_out`, `h2fValid_out` & `h2fReady_in`)
- An FPGA-to-host data pipe (`f2hData_in`, `f2hValid_in` & `f2hReady_out`)
- A 7-bit channel address (`chanAddr_out`)

Each data pipe has a standard FIFO interface¹⁷ where the sender drives some data lines `xxxData[7:0]` and a `xxxValid` line, and the receiver asserts `xxxReady` (where

¹⁷See <http://inst.eecs.berkeley.edu/~cs150/Documents/Interfaces.pdf>

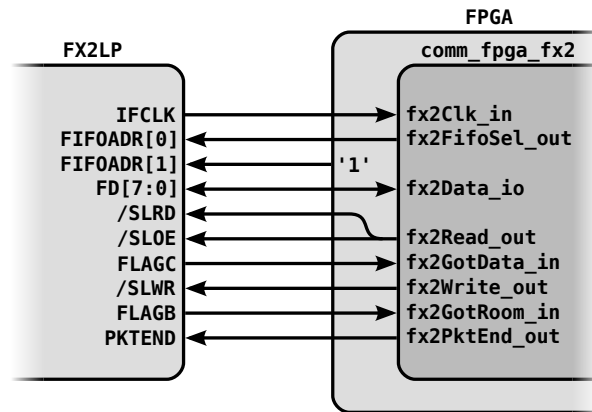
xxx is either **h2f** for host-to-FPGA transfers or **f2h** for FPGA-to-host transfers). If and only if both the **xxxValid** and **xxxReady** signals are asserted when a clock rising edge arrives, the data is registered by the receiver and the next data byte is made available by the sender.

4.2 FX2 External Port

The `comm_fpga_fx2` module's external port clocks data synchronously on every rising edge of `fx2Clk_in`, which is driven at 48MHz by the FX2LP. The internal port is synchronous to the same clock. The interface comprises most of the signals in the FX2LP Slave FIFO Interface¹⁸ with the following changes:

- The data bus is only eight bits wide.
- Only one **FIFOADR** line is supplied, called `fx2FifoSel_out`.
- The other **FIFOADR** line is connected (internally or externally) to Vcc.
- Only **FLAGB** and **FLAGC** are used.
- One signal is supplied for driving both `/SLOE` and `/SLRD`.

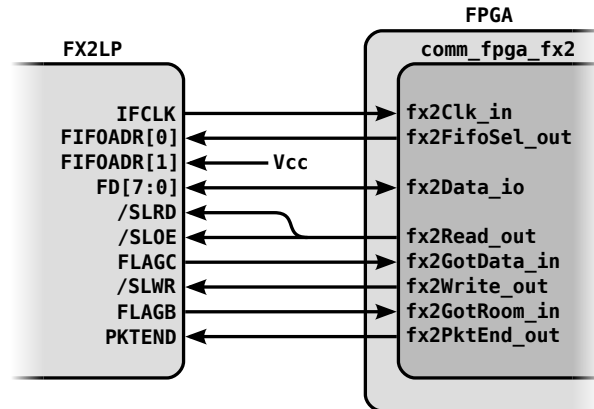
The result is nine signals for control and eight signals for data; most FX2LP-based FPGA devkits just connect all 17 directly to the FPGA:



Notice that the FPGA continuously drives **FIFOADR[1]** high, and that the `fx2Read_out` signal drives *two* external pins, `/SLOE` and `/SLRD`. This is the wiring used by the `ex_cksum` and `ex_fifo` examples.

¹⁸See [EZ-USB TRM Chapter 9](#).

If you're designing your own PCB, you can save two FPGA I/O pins by doing the same thing on the PCB rather than in the FPGA:

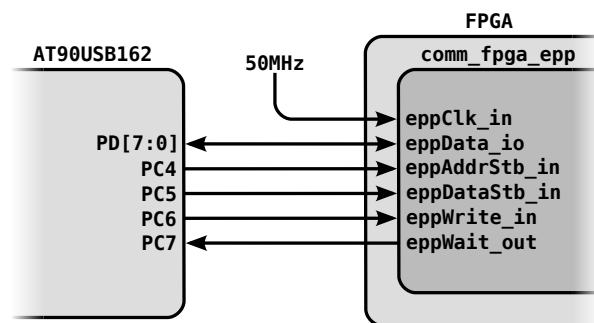


In this case we use eight pins for data as before, but only seven pins for control. Notice that the result is logically the same: **fx2Read_out** signal drives **/SLOE** and **/SLRD**, and **fx2FifoSel_out** drives **FIFOADR[0]**, with **FIFOADR[1]** tied high.

Since the FX2 Slave FIFO interface clocks data synchronously at the end of every 48MHz clock cycle, the theoretical maximum throughput is 48MB/s (45.8MiB/s). In real-world tests with FPGALink, the actual observed throughput is typically about 26MiB/s. The achievable throughput is limited by the speed of the host-side USB code.

4.3 EPP External Port

The Atmel AVR firmware implements the IEEE 1284 Enhanced Parallel Port protocol¹⁹.



The EPP protocol is asynchronous (i.e the control signals are not synchronised to any particular clock). Since FPGAs tend to be synchronous, the EPP signals are internally synchronised to the **comm_fpga_epp** module's **eppClk_in**. The internal port is synchronous to the same clock. The frequency of **eppClk_in** is not important,

¹⁹See <http://www.fapo.com/eppmode.htm>

and neither is its source; the `ex_cksum` and `ex_fifo` examples just use the devkits' onboard 50MHz crystal oscillators.

The EPP protocol is not as efficient as the synchronous FIFO interface of the FX2LP, but in any case the overall throughput of a system based on an Atmel AVR is limited by the USB throughput (unlike the 480Mb/s Hi-Speed FX2LP, the AVRs are Full-Speed 12Mb/s devices) and not the EPP interface. Observed throughput is about 330KiB/s for the AVR firmware running single-buffered on an AT90USB162, ATmega16U2, ATmega32U2 or similar, and about 1.2MiB/s for the firmware running double-buffered on an AT90USB647, ATmega32U4 or similar.

4.4 Build Infrastructure

In the `hdl` directory is a build infrastructure capable of synthesising Verilog code into SVF files suitable for loading into an FPGA with FPGALink. Xilinx and Altera FPGA toolchains are supported. The build infrastructure uses command-line tools rather than the vendors' Integrated Development Environments, but if you prefer to work in an IDE, setting up a project is straightforward.

For Xilinx FPGAs, it's necessary to install ISE WebPACK²⁰. For Altera FPGAs, it's necessary to install Quartus II Web Edition²¹.

4.4.1 MacOSX

Unfortunately, neither Altera's Quartus nor Xilinx's ISE run on MacOSX, so you will need to run either Windows or Linux in a virtual machine under VirtualBox or Parallels.

4.4.2 Windows

The FPGA build infrastructure relies on some UNIX tools like `make`. Whilst it *may* be possible to get builds working with 3rd-party UNIX tools for Windows, the only officially supported set of tools are those available on the MakeStuff website²².

For Xilinx FPGAs, you will need to create a "`XILINX`" environment variable, and set it to the location of the ISE installation (e.g. "`C:/Xilinx/13.2/ISE_DS/ISE`"). No further installation tasks are needed for Altera FPGAs.

²⁰From <http://www.xilinx.com/support/download/index.htm>

²¹From <https://www.altera.com/download/software/software/quartus-ii-we>

²²See http://www.makestuff.eu/wordpress/?page_id=1489

Then from within a Console2 window, you can build the **ex_cksum** example for the Digilent Atlys like this:

```
chris@win7 echo $XILINX
C:/Xilinx/13.2/ISE_DS/ISE
chris@win7$ cd hdl/fx2/verilog/ex_cksum
chris@win7$ ls ../../platforms
atlys  lx9  nexys2-1200  nexys2-500  nexys3  s3board  xylo-1
chris@win7$ make PLATFORM=atlys
```

4.4.3 Linux

For Xilinx FPGAs, you will need to create a “**XILINX**” environment variable, and set it to the location of the ISE installation (e.g “**/opt/Xilinx/13.2/ISE_DS/ISE**”). No further installation tasks are needed for Altera FPGAs.

Then from within a terminal window, you can build the **ex_cksum** example for the Digilent Atlys like this:

```
chris@wotan echo $XILINX
/opt/Xilinx/13.2/ISE_DS/ISE
chris@wotan$ cd hdl/fx2/verilog/ex_cksum
chris@wotan$ ls ../../platforms
atlys  lx9  nexys2-1200  nexys2-500  nexys3  s3board  xylo-1
chris@wotan$ make PLATFORM=atlys
```

4.4.4 Specifying the JTAG Chain

The Altera tools will only generate SVF files for a single-device JTAG chain, but the Xilinx tools support JTAG chains with multiple devices. The JTAG chain and the actual device to program is specified in the **platforms/*/platform.batch** files. All devices except the one being programmed need BSDL descriptions so that the programming algorithm can bypass them.

A future release of FPGALink will simplify this, and provide support for multi-device chains with Altera FPGAs.

4.4.5 Location Constraints

For boards based on Xilinx FPGAs, the location constraints are specified in the **platforms/*/platform.ucf** files. For boards based on Altera FPGAs, the location constraints are specified in the **platforms/*/platform.qsf** files.

4.4.6 The xsvf2csvf Utility

The CSVF format is very similar to Xilinx’s XSUF, except that it has a few simplifications and the JTAG bitmaps have been reversed, making it better suited

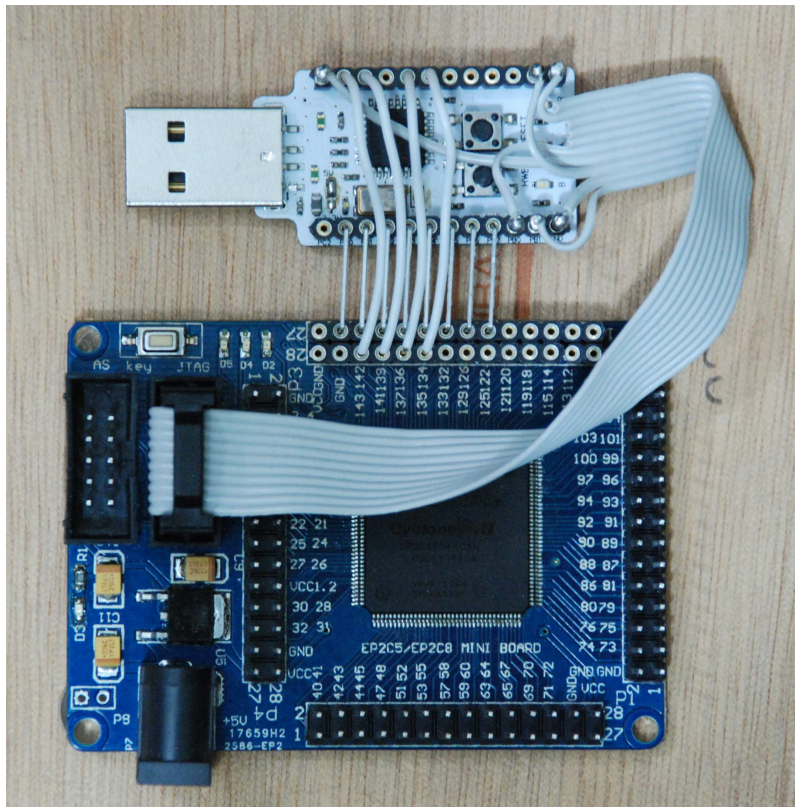
for playback on small microcontrollers. It also has a simple run-length compression scheme making the programming files smaller. You can generate a CSVF file from an XSVF or SVF file using the **xsvf2csvf** utility.

A Custom Boards

Most of the FPGA boards supported by FPGALink are ready-made, but in some cases it's useful to be able to wire an existing microcontroller board to an existing FPGA board, or even design a composite board from scratch.

A.1 Minimus/EP2C5

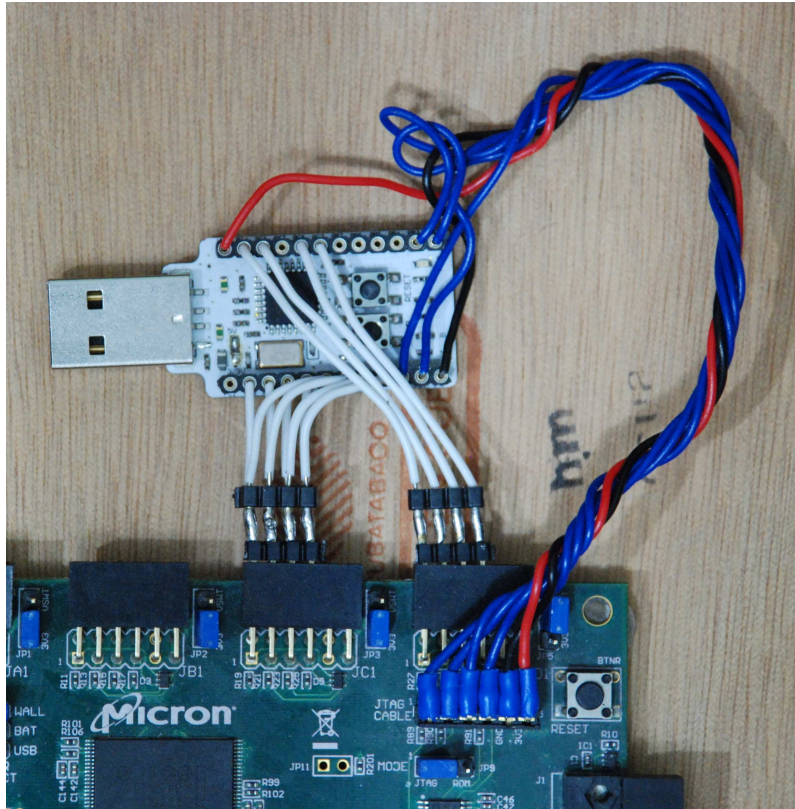
By far the cheapest hardware solution for FPGALink-based applications, this is an EP2C5 Mini Board attached to a Minimus AVR USB board:



To make one you will need an EP2C5 Mini Board and a Minimus AVR USB board. The only modification necessary is to the Minimus board to allow it to draw power from the EP2C5 board. To do this you must sever the USB +5V line, which is the fourth one down on the USB connector in the picture above. Since the Minimus's connector is a Type A plug, the result is physically quite awkward if you don't have a Type A extension cable. Another option is to remove the plug altogether and replace it with a cable with a Type A plug on one end and the other end soldered to the Minimus. Thus, it becomes straightforward to connect the USB +5V power to the P8 connector in the bottom left of the EP2C5 board, to make the whole thing draw its power from USB, eliminating the need for an external +5V supply.

A.2 Minimus/Nexys2

Although the Digilent Nexys2 has a built-in FX2LP USB interface, in some circumstances it might be necessary to have two channels into the FPGA, either both to the same PC, or to different PCs. Here's how to wire a Minimus AVR USB board to a couple of the expansion ports of a Nexys2:



The JTAG connections are **TMS** to **PB0**, **TCK** to **PB1**, **TDI** to **PB2** and **TD0** to **PB3**. The CommFPGA connections should be fairly clear from the photo: the eight data lines go from the Minimus's Port D to the top and bottom rows of the Nexys2's JC1 connector, and the four control lines go to the top row of JD1. As before, the Minimus needs to draw power from the Nexys2, so you will need to sever the USB +5V line, which is the fourth one down on the USB connector in the picture above.