Getting started with Xillinux for Zynq-7000 EPP v1.3

Xillybus Ltd.

www.xillybus.com

Version 3.3

1

Introduction

1.1 The Xillinux distribution

Xillinux is a complete, graphical, Ubuntu 12.04 LTS-based Linux distribution for the Zynq-7000 EPP device, intended as a platform for rapid development of mixed software / logic projects. The currently supported boards are Zedboard, MicroZed and Zybo.

Like any Linux distribution, Xillinux is a collection of software which supports roughly the same capabilities as a personal desktop computer running Linux. Unlike common Linux distributions, Xillinux also includes some of the hardware logic, in particular the VGA adapter.

With Zedboard and Zybo, the distribution is organized for a classic keyboard, mouse and monitor setting. It also allows command-line control from the USB UART port, but this feature is made available mostly for solving problems.

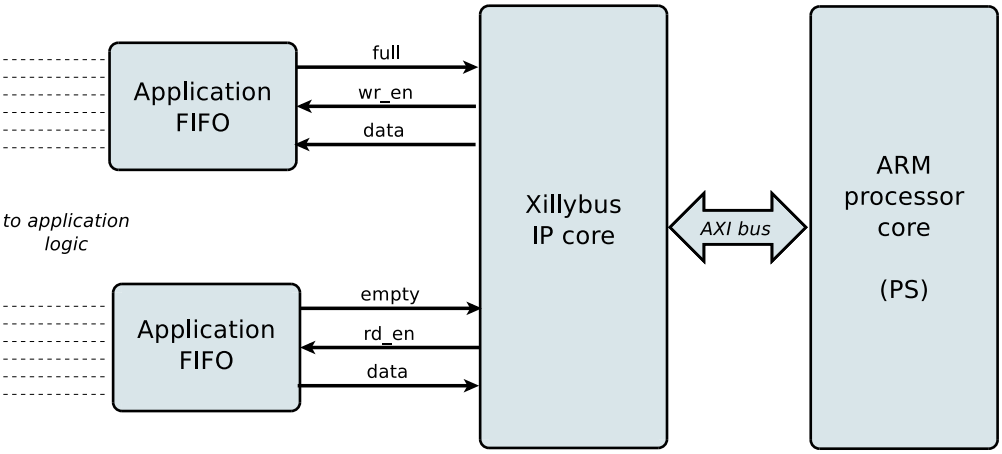
When used with MicroZed, which lacks a VGA/DVI output, only the USB UART functions as a console.

Xillinux is also a kickstart development platform for integration between the device's FPGA logic fabric and plain user space applications running on the ARM processors. With its included Xillybus IP core and driver, no more than basic programming skills and logic design capabilities are needed to complete the design of an application where FPGA logic and Linux-based software work together.

The bundled Xillybus IP cores eliminate the need to deal with the low-level internals of kernel programming and interface with the processor, by presenting a simple and yet efficient working environment to the application designers.

1.2 The Xillybus IP core

Xillybus is a straightforward, portable, intuitive, efficient DMA-based end-to-end turnkey solution for data transport between an FPGA and a host running Linux or Microsoft Windows. It's available for personal computers and embedded systems using the PCI Express bus as the underlying transport, as well as ARM-based processors, interfacing with the AMBA bus (AXI3/AXI4).



As shown above, the application logic on the FPGA only needs to interact with standard FIFOs.

For example, writing data to the lower FIFO in the diagram makes the Xillybus IP core sense that data is available for transmission in the FIFO's other end. Soon, the Xillybus reads the data from the FIFO and sends it to the host, making it readable by the userspace software. The data transport mechanism is transparent to the application logic in the FPGA, which merely interacts with the FIFO.

On its other side, the Xillybus IP core implements the data flow utilizing the AXI bus, generating DMA requests on the processor core's bus.

The application on the computer interacts with device files that behave like named pipes. The Xillybus IP core and driver stream data efficiently and intuitively between the FIFOs in the FPGAs and their respective device files on the host.

The IP core is built instantly per customer's spec, using an online web interface. It's recommended to build and download your custom IP core at http://xillybus.com/custom-ip-factory after walking through the demo bundle flow described in this guide.

The number of streams, their direction and other attributes are defined by customer to achieve an optimal balance between bandwidth performance, synchronization, and

design simplicity.

This guide explains how to rapidly set up the Xillinux distribution including a demo Xillybus IP core, which can be attached to user-supplied sources or sinks for real application scenario testing. The IP core is “demo” in the sense it’s not tailored to any specific application.

Nevertheless, the demo core allows creating a fully functional link with the host.

Replacing the demo IP core with one tailored for special applications is a quick process, and requires the replacement of one binary file and the instantiation of one single module.

More information about using the Xillybus IP core can be found in these documents:

- Getting started with Xillybus on a Linux host
- Xillybus host application programming guide for Linux
- The guide to defining a custom Xillybus IP core

2

Prerequisites

2.1 Hardware

The Xillybus for Zynq Linux distribution (Xillinux) currently supports the Zedboard, 7010 MicroZed and Zybo boards. 7020 Microzed is also supported with a minor fix: See section 5.5.

Owners of other boards may run the distribution on their own hardware, but certain changes, some of which may be nontrivial, may be necessary. More about this in section 5.2.

In order to use the board (MicroZed excluded) as a desktop computer with a monitor, keyboard and mouse, the following items are required:

- A monitor capable of displaying VESA-compliant 1024x768 @ 60Hz with an analog VGA input (i.e. virtually any PC monitor).
- An analog VGA cable for the monitor
- A USB keyboard
- A USB mouse
- A USB hub recognized by Linux 3.12.0, if the keyboard and mouse are not combined in a single USB plug

When Zybo is used, the monitor can be connected to the HDMI port, which outputs the same monitor image in DVI format. Most monitors having an HDMI input will display the image correctly when connected with a plain HDMI cable to the board. Alternatively, a passive HDMI/DVI adapter or cable can be used to connect to virtually any monitor's DVI input for correct operation.

Zedboard's HDMI output port is not supported.

A wireless keyboard/mouse combo is recommended, since it eliminates the need for a USB hub, and prevents possible physical damage to the USB port on the board, as a result of accidentally pulling the USB cables.

On the Zedboard, the connection of the keyboard and mouse is done through a Micro B to Type A female USB cable, which arrives with the board. On the other two boards, a standard USB type A female connector (like a PC's USB plug) is available for connection of peripherals.

Also required:

- A reliable SD card (for Zedboard) or MicroSD (for MicroZed / Zybo) with 2GB or more, preferably Sandisk. The card that (possibly) came with the board is not recommended, as problems have been reported using it with Xillinux.
- Recommended: A USB adapter between an (Micro)SD card and PC, for writing the image and boot file to the card. This may be unnecessary if the PC computer has a built-in slot for SD cards. The Zynq board itself can also be used as a flash writer, but this is somewhat trickier.

2.2 Downloading the distribution

The Xillinux distribution is available for download at Xillybus site's download page:

http://xillybus.com/xillinux/

The distribution consists of two parts: A raw image of the (Micro)SD card consisting of the file system to be seen by Linux at bootup, and a set of files for implementation with the Xilinx tools to populate the boot partition. More about this is section 3.

The distribution includes a demo of the Xillybus IP core for easy communication between the processor and logic fabric. The specific configuration of this demo bundle may perform relatively poorly on certain applications, as it's intended for simple tests.

Custom IP cores can be configured, automatically built and downloaded using the IP Core Factory web interface. Please visit http://xillybus.com/custom-ip-factory for using this tool.

Any downloaded bundle, including the Xillybus IP core, and the Xillinux distribution, is free for use, as long as this use reasonably matches the term "evaluation". This includes incorporating the core in end-user designs, running real-life data and field testing. There is no limitation on how the core is used, as long as the sole purpose of

this use is to evaluate its capabilities and fitness for a certain application.

2.3 Development software

Vivado is the preferred tool for compiling the logic fabric parts of the Xillinux distribution, as the alternative, ISE, is being phased out by Xilinx.

Vivado 2014.4 and later is supported. In particular 2014.4 and 2015.2 have been tested, but any version after 2014.4 should work fine as well.

A bundle for 2014.1 is also available for download, but 2014.2 and 2014.3 are *not* supported either way.

The same bundle can also be used with ISE Design Suite release 14.2 and later.

The software can be downloaded directly from Xilinx' website (http://www.xilinx.com).

Any of the design suite's editions is suitable, including

- the WebPACK Edition, which can be downloaded and used with no license fee for an unlimited time, assuming that the target device is covered. XC7Z020 and XC7Z010, which are used on the Zedboard, MicroZed and Zybo, are covered by this edition.
- the Logic Edition (or Design Edition for Vivado), which requires a purchased license (but a 30-day trial is available).
- any target-locked edition that may have been licensed specifically along with a purchased board.
- the DSP and Embedded Editions are fine as well, but have a higher license fee.

All of these editions cover the Xilinx-supplied IP cores necessary to implement Xillybus for Zynq, with no extra licensing required.

2.4 Experience with FPGA design

When targeting Zedboard, MicroZed or Zybo, no previous experience with FPGA design is necessary to have the distribution running on the platform. Targeting another board requires some knowledge with using Xilinx' tools, and possibly some basic capabilities related to the Linux kernel.

To make the most of the distribution, a good understanding of logic design techniques, as well as mastering an HDL language (Verilog or VHDL) are necessary. Neverthe-

less, the Xillybus distribution is a good starting point for learning these, as it presents a simple starter design to experiment with.

3

Building Xillinux

3.1 Overview

The Xillinux distribution is intended as a development platform, and not just a demo: A ready-for-use environment for custom logic development and integration is built during its preparation for running on hardware. This makes the preparation for the first test run somewhat timely (typically 30 minutes, most of which consist of waiting for Xilinx' tools) but significantly shortens the cycle for integrating custom logic.

To boot the Xillinux distribution from an (Micro)SD card, it must have two components:

- A FAT32 filesystem in a boot partition, consisting of boot loaders, a configuration bitstream for FPGA part (known as PL), and the binaries for booting the Linux kernel.
- An ext4 root file system mounted by Linux.

The downloaded raw Xillinux image has almost everything set up already. There are just three files missing in the boot partition, one of which needs to be generated with Xilinx' tools, and two that are copied from the boot partition kit.

The various operations for preparing the (Micro)SD are detailed step by step in this section.

This flow consists of the following steps, which must be done in the order outlined below. Note that two of the steps listed below are skipped when using Vivado.

- Unzipping the boot partition kit
- ISE suite only: Generating the processor netlist

- ISE suite only: Generating Xilinx' IP cores
- Implementing the main PL (FPGA) project
- Writing the raw Xillinux image to the (Micro)SD card
- Copying three files into the (Micro)SD card's boot partition

How to target other boards is discussed in paragraph 5.2.

3.2 Unzipping the boot partition kit

Unzip the previously downloaded xillinux-eval-*board-XXX*.zip file into a working directory.

IMPORTANT:

The path to the working directory must not include white spaces. In particular, the Desktop is unsuitable, since its path includes "Documents and Settings".

The bundle consists of the following directories:

- verilog – Contains the project file for the main logic and some sources in Verilog (in the 'src' subdirectory)
- vhdl – Contains the project file for the main logic and some sources, with the user-editable source file in VHDL (in the 'src' subdirectory)
- cores – Precompiled binaries of the Xillybus IP cores
- system – Directory for generating processor-related logic
- runonce – Directory for generating general-purpose logic (CoreGen FIFO IP cores).
- bootfiles – Contains two board-specific files, to be copied to the boot partition.
- vivado-essentials – Definition files and build directories for processor-related and general-purpose logic for use by Vivado.

Note that both 'src' directories also contain the UCF file for the targeted board. This file must be edited if a board, which isn't among the three supported, is targeted.

Also note that the vhdl directory contains Verilog files, but none of them should need editing by user.

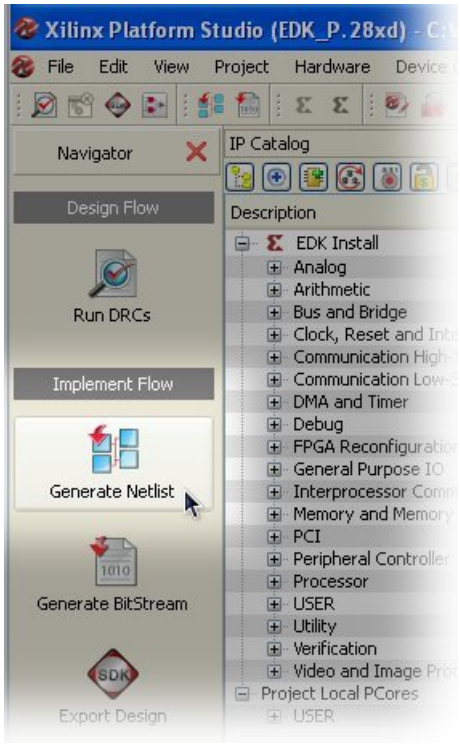
The interface with the Xillybus IP core takes place in the xillydemo.v or xillydemo.vhd files in the respective 'src' subdirectories. This is the file to edit in order to try Xillybus with your own data sources and sinks.

3.3 Generating the processor netlist (ISE only)

This part is relevant only when using the ISE suite. The Vivado flow starts in paragraph 3.5.2.

Within the boot partition kit, double-click the system.xmp file in the "system" directory. This opens the Xilinx Platform Studio (XPS).

Click "Generate Netlist" to the left (as shown in the image below).



The process takes up to 10 minutes, depending on the computer running it. Several warnings are generated, but no errors should be tolerated (which is unlikely to happen).

The console output says "XST completed" and "Done!" upon a successful completion of the process. At this point, close the XPS completely.

There is no need to repeat this process in the future.

It's possible that a Xilinx License Error dialog box followed by a Xilinx License Configuration Manager window will appear upon invocation of XPS. These should be ignored (with "OK" and "Close", respectively), since a license for XPS targeting certain Zynq devices is actually included in the WebPack license group. Some versions of XPS issue an error despite this.

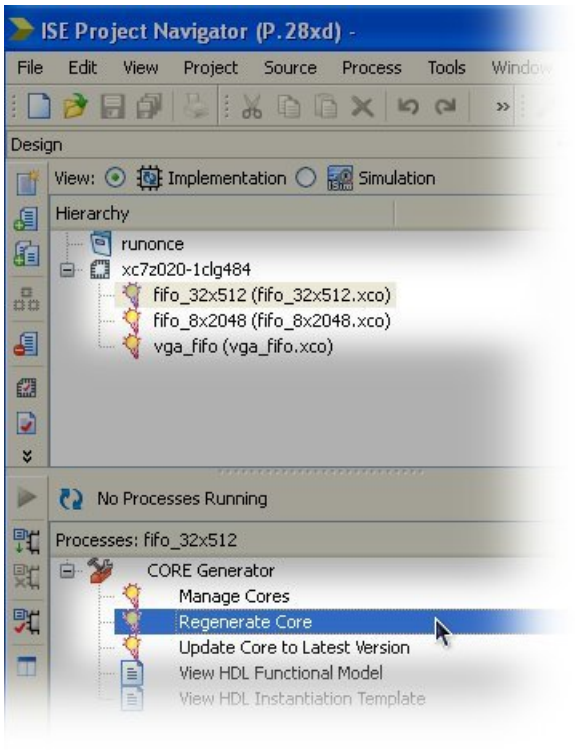
If licensing errors continue to appear, preventing the generation of a netlist, make sure a WebPack (or broader) license is installed on the machine.

3.4 Generating Xilinx IP cores (ISE only)

This part is relevant only when using the ISE suite. The Vivado flow starts in paragraph 3.5.2.

Within the boot partition kit, double-click the runonce.xise file in the "runonce" directory. This opens the Xilinx' ISE Project Navigator.

On the opened window's top left, click on fifo_32x512, then expand the "CORE Generator" line in the process window below (clicking on the "+"), and finally, double click "Regenerate Core", as shown in the image below.



The process produces a handful of warnings, but no errors, and ends with a message saying: Process “Regenerate Core” completed successfully.

Repeat this for the two other IP cores: fifo_8x2048 and vga_fifo (or fifo_8x2048 only when targeting MicroZed).

At this point, close the ISE Project Navigator completely. There is no need to ever repeat this process.

3.5 Generating the bitstream file

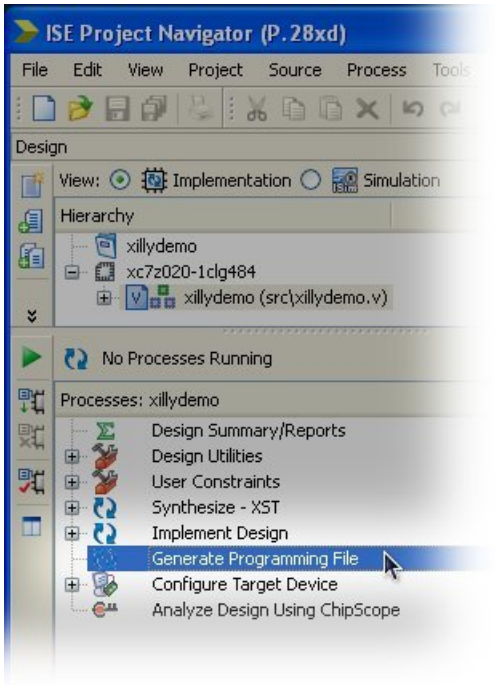
3.5.1 Using ISE

IMPORTANT:

*Please verify that the processes described in paragraphs 3.3 and 3.4 were indeed carried out, before moving on to generating the bitstream. In particular, please recall that the "Regenerate Core" process was run **three times** in paragraph 3.4, once for each core. There is no risk to get a faulty bitstream though, since the implementation will fail with an error if the preparations weren't completed.*

Depending on your preference, double-click the 'xillydemo.xise' file in either the 'verilog' or 'vhdl' subdirectory. Both directories are in the boot partition kit. In the absence of preference, pick the 'verilog' subdirectory.

The Project Navigator will launch and open the project with the correct settings. Just click "Generate Programming File".



The procedure will produce several warnings (FPGA implementations always do) but should not present any errors. The process should end with the output "Process

Generate Programming File completed successfully". The result can be found as xillydemo.bit in the 'verilog' or 'vhdl' directory (whatever was chosen) along with several other files.

If an error occurs, please check that the steps of paragraphs 3.3 and 3.4 were carried out fully, in particular that the latter was carried out for all three IP cores. Most notably, if the bottom line at the console is "Process Translate failed", with a previous error message complaining about not being able to resolve some logical block, the proper completion of the the two previous steps should be verified.

Close the ISE Project Navigator completely when this task is completed successfully.

3.5.2 Using Vivado

Vivado's outline of intermediate files is relatively complex and difficult to control. In order to keep the file structure in the bundle compact, a script in Tcl is supplied for creating the Vivado project. This script creates a new subdirectory, "vivado", and populates this directory with files as necessary.

The project relies on the files in the src/ subdirectory (no copies of these files are made). The processor, its interconnect and peripherals, as well as the FIFOs used by the logic are defined in vivado-essentials/, which is also populated with intermediate files by Vivado as the project is implemented.

The project can be implemented based upon Verilog or VHDL. If VHDL is chosen, vhdl/src/xillydemo.vhd must be edited to **remove** the following three lines in the beginning (Xillydemo's entity port list):

```
    PS_CLK : IN std_logic;
    PS_PORB : IN std_logic;
    PS_SRSTB : IN std_logic;
```

also, **uncomment** the following lines in the architecture definition (remove the "--" comment marks):

```
-- signal PS_CLK :  std_logic;
-- signal PS_PORB :  std_logic;
-- signal PS_SRSTB :  std_logic;
```

There is no need to make changes in any Verilog source file.

Start **Vivado 2014.4 and later** (or possibly 2014.1, using a legacy bundle). With no project open, Pick Tools > Run Tcl Script... and choose **xillydemo-vivado.tcl** in the verilog/ or vhdl/ subdirectory, depending on your preference. A sequence of events

takes place for less than a minute. The success of the project's deployment can be verified by choosing the "Tcl Console" tab at Vivado's window's bottom, and verify that it says

```
INFO: Project created: xillydemo
```

If this is not the bottom line of the Tcl console, something went wrong, most likely because the wrong revision of Vivado is used.

Critical warnings will appear during this stage, but no errors. However if the project has already been generated (i.e. the script has been run already), attempting to run the script again will result in the following error:

```
ERROR: [Common 17-53] User Exception: Project already exists on disk,
       please use '-force' option to overwrite:
```

After the project has been created, run an implementation: Click "Generate Bitstream" on the Flow Navigator bar to the left.

A popup window asking if it's OK to launch synthesis and implementation is likely to appear – pick "Yes".

Vivado runs a sequence of processes. This takes normally takes a few minutes. Several warnings are issued, some of which may be classified critical. There should be no errors.

A popup window, informing that the bitstream generation was completed successfully will appear, giving choices of what to do next. Any option is fine, including picking "Cancel".

The bitstream file, xillydemo.bit, can be found at vivado/xillydemo.runs/impl_1/

The implementation is never expected to fail. There are however a few error conditions worth mentioning:

- The Placer fails on “IO placement is infeasible” on a VHDL design. If this happens on an VHDL implementation, please make sure that xillydemo.vhd was edited as required above.
- write_bitstream fails with DRC errors complaining about PS_CLK, PS_PORB and PS_SRSTB being unspecified, unrouted and unconstrained, then again – please make sure that xillydemo.vhd was edited as required above.
- Error saying “Timing constraints weren’t met”. This can happen when custom logic has been integrated, causing the tools to fail meeting the timing requirements. This means that the design is syntactically correct, but needs corrections to make certain paths fast enough with respect to given clock rates and/or I/O requirements. The process of correcting the design for better timing is often referred to as *timing closure*.

  A timing constraint failure is commonly announced as a critical warning, allowing the user to produce a bitstream file with which the FPGA’s behavior is not guaranteed. To prevent the generation of such a bitstream, a timing failure is promoted to the level of an error by virtue of a small Tcl script, “showstopper.tcl”, which is automatically executed at the end of a route run. To turn this safety measure off, click “Project Settings” under “Project Manager” in the Flow Navigator. Choose the “Implementation” button, and scroll down to the settings for “route_design”. Then remove showstopper.tcl from tcl.post.
- Any other error is most likely a result of changes made by the user, and should be handled per case.

3.6 Loading the (Micro)SD with the image

3.6.1 General

The purpose of this task is to write the downloaded (Micro)SD card image file to the (micro)SD device. The image was downloaded as a file named xillinux.img.gz (or similar), and is a gzip-compressed image of the (Micro)SD card, which is ready for booting, except for three files, which are added later.

This image should be uncompressed and then written to the (Micro)SD card’s first sector and on. There are several ways and tools to accomplish this. A few methods are suggested next.

The image contains a partition table, a partly populated FAT file system for placing initial boot files, and the Linux root file system of ext4 type. The second partition is

ignored by almost all Windows computers, so the (Micro)SD card may appear to be very small in capacity (16 MB or so).

Writing a full disk image is not an operation intended for plain computer users, and therefore requires special software on Windows computers and extra care on Linux. The following paragraphs explain how to do this on either operating system.

If there's no USB adapter for the (Micro)SD card (or a dedicated slot on a computer), the board itself can be used to write the image, as described in paragraph 3.6.4.

IMPORTANT:

Writing an image to the (Micro)SD irrecoverably deletes any previous content it may contain. It's warmly recommended to make a copy of its existing content, possibly with the same tools used to write the image.

3.6.2 Loading the image (Windows)

On Windows, a special application is needed to copy the image, such as the USB Image Tool. This tool is suitable when a USB adapter is used to access the (Micro)SD card.

Some computers (laptops in particular) have an (Micro)SD slot built in, and may need to use another tool, e.g. Win32 Disk Imager. This may also be the case when running Windows 7.

Both tools are available free for downloading from various sites on the web. The following walkthrough assumes using the USB Image Tool.

For a Graphical interface, run "USB Image Tool.exe". When the main window shows up, plug in the USB adapter, select the device icon which appears at the top left. Make sure that you're in "Device Mode" (as opposed to "Volume Mode") on the top left drop down menu. Click Restore and set the file type to "Compressed (gzip) image files". Select the downloaded image file (xillinux.img.gz). The whole process should take about 4-5 minutes. When finished, unmount the device ("safely remove hardware") and unplug it.

On some machines, the GUI will fail to run with an error saying the software initialization failed. In that case, the command line alternative can be used, or a Microsoft .NET framework component needs to be installed.

Alternatively, this can be done from the command line (which is a quick alternative if the attempt to run GUI fails). This is done in two stages. First, obtain the device's

number. On a DOS Window, change directory to where the application was unzipped to and go (typical session follows):

```
C:\usbimage>usbitcmd l

USB Image Tool 1.57
COPYRIGHT 2006-2010 Alexander Beug
http://www.alexpage.de

  Device   | Friendly Name             | Volume Name | Volume Path | Size
 ------------------------------------------------------------------------------
      2448 | USB Mass  Storage Device  |             | E:\         |   2014 MB
```

(That was the letter “l” as in “list” after ”usbitcmd”, not the figure “one”)

Now, when we have the device number, we can actually do the writing (”restore”):

```
C:\usbimage>usbitcmd r 2448 \path\to\xillinux.img.gz /d /g

USB Image Tool 1.57
COPYRIGHT 2006-2010 Alexander Beug
http://www.alexpage.de

Restoring backup to "USB Mass  Storage Device USB Device" (E:\)...ok
```

Again, this should take about 4-5 minutes. And of course, change the number 2448 to whatever device number you got in the first stage, and replace `\path\to` with the path to where the (Micro)SD card’s image is stored on your computer.

3.6.3 Loading the image (Linux)

IMPORTANT:

Raw copying to a device is a dangerous task: A trivial human error (typically choosing the wrong destination disk) can result in irrecoverable loss of the data of an entire hard disk. The distance between the desired operation and a complete disaster is one character typed wrong. Think before pressing Enter, and consider doing this in Windows if you’re not used to Linux.

As just mentioned, it’s important to detect the correct device as the (Micro)SD card.

This is best done by plugging in the USB connector, and looking for something like this is the main log file (/var/log/messages or /var/log/syslog):

```
Sep  5 10:30:59 kernel: sd 1:0:0:0: [sdc] 7813120 512-byte logical blocks
Sep  5 10:30:59 kernel: sd 1:0:0:0: [sdc] Write Protect is off
Sep  5 10:30:59 kernel: sd 1:0:0:0: [sdc] Assuming drive cache: write through
Sep  5 10:30:59 kernel: sd 1:0:0:0: [sdc] Assuming drive cache: write through
Sep  5 10:30:59 kernel:  sdc: sdc1
Sep  5 10:30:59 kernel: sd 1:0:0:0: [sdc] Assuming drive cache: write through
Sep  5 10:30:59 kernel: sd 1:0:0:0: [sdc] Attached SCSI removable disk
Sep  5 10:31:00 kernel: sd 1:0:0:0: Attached scsi generic sg0 type 0
```

The output may vary slightly, but the point here is to see what name the kernel gave the new disk. “sdc” in the example above.

Uncompress the image file:

```
# gunzip xillinux.img.gz
```

Copying the image to the (Micro)SD card is simply:

```
# dd if=xillinux.img of=/dev/sdc bs=512
```

You should point at the disk you found to be the flash drive, of course.

IMPORTANT:

/dev/sdc is given as an example. Don’t use this device unless it happens to match the device recognized on your computer.

And verify

```
# cmp xillinux.img /dev/sdc
cmp: EOF on xillinux.img
```

Note the response: The fact that EOF was reached on the image file means that everything else compared correctly, and that the flash has more space than actually used. If cmp says nothing (which would normally be considered good) it actually means something is wrong. Most likely, a regular file “/dev/sdc” was generated rather than writing to the device.

3.6.4 Using the Zynq board for loading the image

Paragraph 3.6.3 above described how to load the image with a Linux machine and a USB adapter. The Zynq board itself can be used, running the sample Linux system that came with the board. Basically, the same instructions can be followed, using /dev/mmcblk0 as the target device (instead of /dev/sdc).

This works fine when booting from the QSPI flash, as well as with the sample Linux system on an SD card (if available), since it runs completely from RAM, and doesn't use the SD card after booting has finished (this is not the case with Xillinux, of course). So if an SD card was used for booting, it's possible to pull it out, and insert another for writing the image to.

How to give the Zynq board access to the (Micro)SD image and boot partition files is a matter of preference and Linux knowledge. There are several ways to do this over the network, but the simplest way is to write these files to a USB disk-on-key, and connect it to the USB OTG port. Mount the disk-on-key with

```
> mkdir /mnt/usb
> mount /dev/sda1 /mnt/usb
```

The files on the disk-on-key can then be read at /mnt/usb/.

On Zedboard, make sure that the JP2 jumper is installed, so that the USB port is fed with 5V power supply.

3.7 Copying the files into the boot partition

This final stage places the necessary files for booting:

- Copy boot.bin and devicetree.dtb from the boot partition kit's bootfiles/ subdirectory, into the (Micro)SD card's boot partition (the first partition).
- Copy xillydemo.bit that was generated in section 3.5 (from the verilog/ or vhdl/ subdirectory, whichever was chosen).

Before copying these files: If the (Micro)SD image was just written to the card, unplug the USB adapter and then connect it back to the computer. If the Zynq board was used for writing the raw image, pull the (Micro)SD card from its slot, and reinsert it.

This is necessary to make sure that the computer is up to date with the (Micro)SD card's partition table.

On Linux systems, it may be necessary to manually mount the first partition (e.g /dev/sdb1). Most computers will do this automatically.

For example, when the Zynq board itself is used for this purpose, type

```
> mkdir /mnt/sd
> mount /dev/mmcblk0p1 /mnt/sd
```

and copy the files to /mnt/sd/.

On Windows systems, plugging in the (micro)SD card will reveal a single “disk”, with a single file, uImage. This is the correct destination to copy the files to.

When done, unmount the (Micro)SD card properly, and unplug it from the computer, e.g.

```
> umount /mnt/sd
```

or “remove the disk safely” on Windows.

3.8 The files in the boot partition

Before attempting to boot, please verify that the boot partition is populated as follows.

In order to boot, four files need to exist in the (micro)SD card’s first partition (the boot partition):

- uImage – The Linux kernel binary. This is the only file in the boot partition after writing the Xillinux (Micro)SD image to the card. The kernel is board-independent.
- boot.bin – The initial bootloader. This file contains the initial processor initializations and the U-boot utility, and is significantly different from board to board.
- devicetree.dtb – The Device Tree Blob file, which contains hardware information for the Linux kernel.
- xillydemo.bit – The PL (FPGA) programming file, which was generated in section 3.5

4

Booting up

4.1 Jumper settings

For the board to boot from the (Micro)SD card, modifications in the jumper settings need to be made. The settings are detailed for each of the boards below.

4.1.1 Zedboard

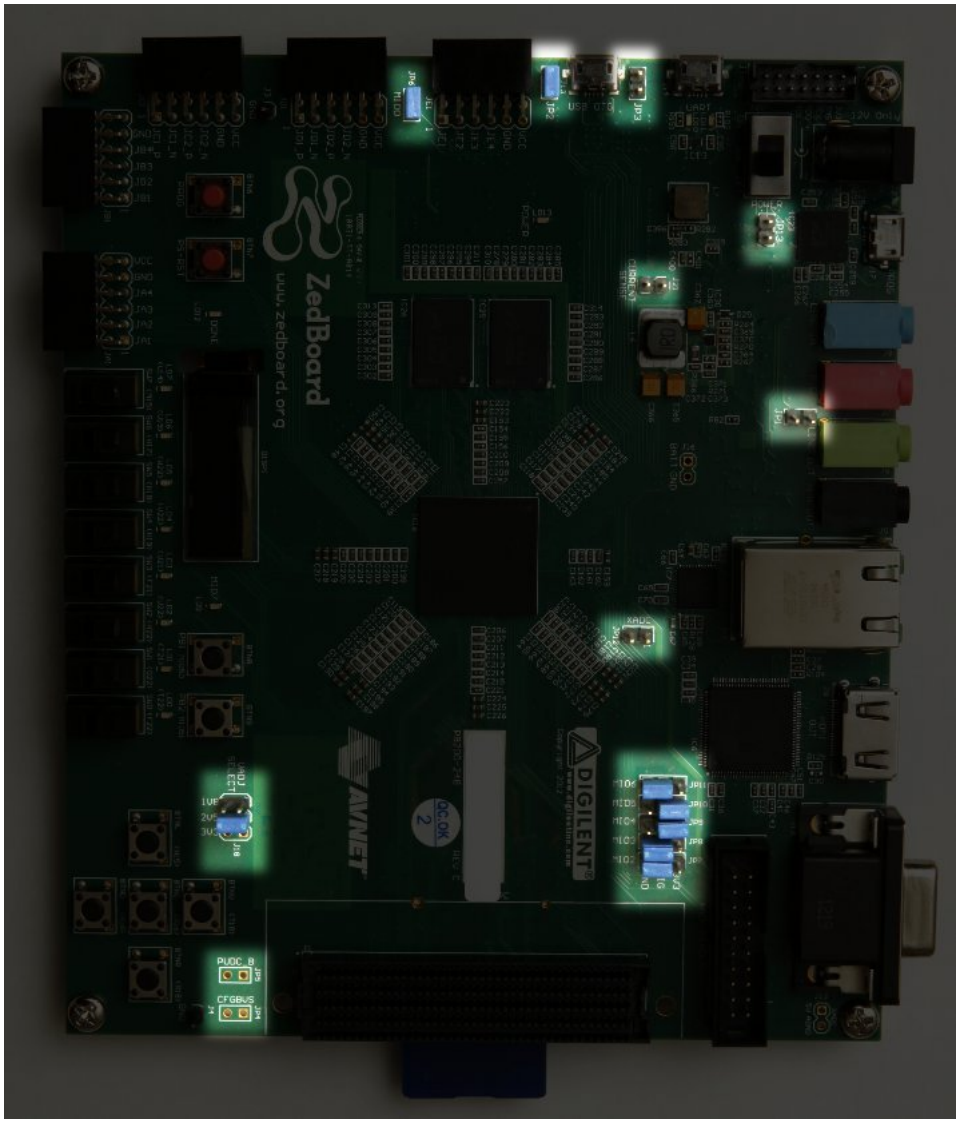
The correct setting is depicted in the image on the following page.

Typically, the following jumper changes are necessary (but your board may be set differently to begin with):

- Install a jumper for JP2 to supply 5V to USB device.
- JP10 and JP9 moved from GND to 3V3 position, the three others in that row are left connected to GND.
- Install a jumper for JP6 (required for CES silicon, see page 34 of the Zedboard's Hardware Guide).

IMPORTANT:

The required setting differs from the one detailed in the Zedboard Hardware User Guide in that JP2 is jumpered, so that the USB devices attached to the board (USB keyboard and mouse) receive their 5V power supply.



Jumper settings highlighted on the Zedboard

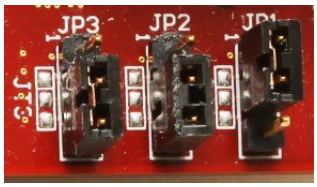
4.1.2 MicroZed

The proper jumper setting for booting Xillinux from the MicroSD card is as follows:

- JP1: 1-2 (GND)
- JP2: 2-3 (VCC)
- JP3: 2-3 (VCC)

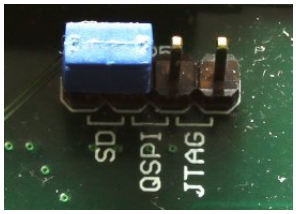
Given a board with the default setting, only JP2 needs to be moved.

The correct jumper setting is depicted here:



4.1.3 Zybo

The boot mode is selected by a jumper near to the VGA connector, which should be set on the two pins marked with “SD”, as shown on this image:



The other jumpers are set depending on the desired operating mode. For example, the power supply jumper can be set to take power from an external 5V source, or from the UART USB jack – both are fine for booting Xillinux.

4.2 Attaching peripherals

The following general-purpose hardware can be attached the board:

- Zedboard / Zybo: A computer monitor to the analog VGA connector. Since Xillinux produces a VESA-compliant 1024x768 @ 60Hz signal through the analog VGA plug, it’s almost certain that any computer monitor will suffice.

- Zybo only: A computer monitor with a DVI input, connected to the board's HDMI plug through an HDMI/DVI adapter or cable. Alternatively, if the monitor has an HDMI input, there's a good chance that connecting the monitor to the board with a plain HDMI cable will result in correct operation.
- Zedboard/Zybo: A mouse and keyboard to the USB (OTG) connector. On Zybo, there's a PC-like USB plug for this. On Zedboard, this goes through the USB female cable that came with the board (which is also the shorter one). The system will boot in the absence of these, and there is no problem connecting and disconnecting the keyboard and mouse as the system runs – the system detects and works with whatever keyboard and mouse it has connected at any given moment.

  Note that on Zedboard, JP2 must be installed for this USB port to function.
- The Ethernet port is optional for common network tasks. The Linux machine configures the network automatically if the attached network has a DHCP server.
- The UART USB port is optionally connected to a PC, but is redundant in most cases for Zedboard and Zybo. Some boot messages are sent there, and a shell prompt is issued on this interface when the boot completes.

  This is useful for debugging boot failures, or when either a PC monitor or a keyboard is missing or don't work properly.

  Linux' support for some of the UART to USB converters is somewhat incomplete. It's recommended to use Tera Term under Windows for this purpose. This is relevant in particular with MicroZed and Zybo.

4.3 Powering up the board

4.3.1 Initial diagnostics

Boot failures are rare when the build instructions above have been followed. Among the common reasons are:

- Incorrect jumper settings (see paragraph 4.1).
- Using a non-Sandisk (Micro)SD card. Even if the card seems to work fine, scattered data corruptions are easily overlooked, but result in errors that appear to have a completely different reason.
- Incomplete or faulty writing of the (Micro)SD image into the card

- Old and inadequate environment variables loaded by U-boot from the board's QSPI flash. See paragraph 4.3.3.
- Deviations from the instructions, usually attempting to tweak the system on the first attempt to build it.

The correct UART setting is 115200 baud, 8 data bits, 1 stop bits, and no flow control. Some text should appear no later than 4 seconds after the board is powered on. The only requirement for this to happen is that the boot.bin file is found in the first (FAT32) partition on the (Micro)SD card.

If nothing happens after powering on the board

- Verify that the correct boot.bin has been copied from a boot partition kit that matches the board's type. The kit's file name indicates which board it should be used with.
- Verify that the UART to computer link works properly, possibly with the sample Linux on the QSPI flash or on the SD card that (possibly) arrived with the board. Note that Linux, as a host for the UART terminal application, may not work properly with some UART/USB converters, so trying Tera Term under Windows may be the only option.

If U-boot emits messages on the console, but the boot process fails, it may be helpful to compare with the transcript in paragraph 4.3.5. The rest of this section may also contain relevant information for understanding what's wrong.

4.3.2 When boot completes

At the end of the boot process, a shell prompt is given on the UART console, with no need to log in manually. Even so, the root user's password is set to nothing, so logging in as root, if ever needed, doesn't require a password.

IMPORTANT:

Unlike the Linux sample on the (Micro)SD card that came with the Zedboard, Xillinux' root file system permanently resides on the (Micro)SD card, and is written to while the system is up. The Linux system should be shut down properly before powering off the board to keep the system stable, just like any PC computer, even though a graceful recovery is usually observed on sudden power drops.

Notes for Zedboard and Zybo:

- Type “startx” at command prompt to launch a Gnome graphical desktop. The desktop takes some 15-30 seconds to initialize. If nothing appears to happen, monitoring the activity meter on the OLED display helps telling if something is going on (Zybo doesn’t have an OLED, though).
- The Xillybus logo screensaver with a white background is present on the screen from the moment the logic fabric is loaded until the Linux kernel launches. It will also show when the operating system puts the driver in “blank” mode, which is a normal condition when the system is idle, or when the X-Windows system attempts to manipulate the graphic mode.
- A Xillybus screensaver on **blue** background, or random blue stripes on the screen indicate that the graphics interface suffers from data starvation. This is never expected to happen, and should be reported, unless an obvious reason is known.

4.3.3 U-boot environment variables

Xillinux relies on U-boot for loading xillydemo.bit, the kernel image and the device tree during the boot process. This utility offers a large variety of boot configurations, and has a simple command-line interface which allows experimentation and modification of the settings.

U-boot’s shell is reached by typing any character on the UART console immediately after U-boot starts:

```
U-Boot 2013.07 (Mar 15 2014 - 22:59:21)

Memory: ECC disabled
DRAM:  512 MiB
MMC:   zynq_sdhci: 0
SF: Detected S25FL129P_64K/S25FL128S_64K with page size 64 KiB, total 16 MiB
*** Warning - bad CRC, using default environment

In:    serial
Out:   serial
Err:   serial
Net:   Gem.e000b000
Hit any key to stop autoboot:  1
```

U-boot always attempts to retrieve saved environment variables from the QSPI flash. The “bad CRC” warning indicates that no valid data was found, so U-boot reverted to its hardcoded default environment, which is correct for booting Xillinux from the (Micro)SD card.

IMPORTANT:

Note that even when the system boots from a (Micro)SD card, the environment variables are taken from the QSPI flash on the board itself. If the QSPI flash contains environment variables matching another boot scenario, U-boot may fail booting, relying on inadequate variables.

When no key is pressed for one second, U-boot continues booting according to its environment variables (those loaded from the QSPI flash, or the hardcoded default settings). More precisely, it executes the content of the “bootcmd” variable, which says “run $modeboot” by default. “modeboot”, in turn, is set dynamically by U-boot, depending on where U-boot was loaded from, so it says “sdboot” on a regular Xillinux boot. The “sdboot” variable contains a series of commands for booting Xillinux.

U-boot’s command-line shell has a “help” command, which lists all commands and their meaning. Some useful commands are

- help *command* – show help on *command*
- env print – print all environment variables’ current values
- env set – set a certain environment variable’s value
- env default -a – set all environment variables to their hardcoded defaults.
- saveenv – save the current environment variables to QSPI flash (*not* to MicroSD/SD card).

In particular, the two last commands in the list are important when U-boot fails to boot. If the “bad CRC, using default environment” warning is **not** issued by U-boot, it’s relying on stored variables. In order to use the default variables (which are correct for Xillinux), go

```
xillinux-uboot> env default -a
## Resetting to default environment
xillinux-uboot> saveenv
Saving Environment to SPI Flash...
```

```
SF: Detected S25FL129P_64K/S25FL128S_64K with page size 64 KiB, total 16 MiB
Erasing SPI flash...Writing to SPI flash...done
```

If there were any desirable changes in the stored environment variables, they are erased as well, of course.

4.3.4 Setting a custom Ethernet MAC address

Linux relies on the Ethernet MAC address that it finds on the hardware, which is set by U-boot, depending on a certain environment variable. Since the environment variables are stored on the QSPI flash, the MAC address is persistently bound to the hardware, so even if a specific (Micro)SD card is used on different boards, each board retains its own MAC address.

For example, on the U-boot's shell using the USB UART console:

```
xillinux-uboot> set ethaddr 00:11:22:33:44:55
xillinux-uboot> saveenv
Saving Environment to SPI Flash...
Erasing SPI flash...Writing to SPI flash...done
```

and later on, after recycling the board's power and letting Linux boot automatically:

```
root@localhost:~# ifconfig
eth1      Link encap:Ethernet  HWaddr 00:11:22:33:44:55
          inet addr:10.1.1.164  Bcast:10.1.1.255  Mask:255.255.255.0
          inet6 addr: fe80::211:22ff:fe33:4455/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:16 errors:0 dropped:0 overruns:0 frame:0
          TX packets:50 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:2720 (2.7 KB)  TX bytes:9230 (9.2 KB)
          Interrupt:54 Base address:0xb000
```

(The IP address was given by the DHCP server in the case above)

4.3.5 Sample boot transcript

For reference, a typical UART transcript during boot is given below. The example is shown for MicroZed, but the differences between the boards are minor. If booting fails, an error message will probably indicate what stage went wrong, and possibly why.

```
U-Boot 2013.07 (Mar 15 2014 - 22:59:21)

Memory: ECC disabled
DRAM:  512 MiB
MMC:   zynq_sdhci: 0
SF: Detected S25FL129P_64K/S25FL128S_64K with page size 64 KiB, total 16 MiB
*** Warning - bad CRC, using default environment

In:    serial
Out:   serial
Err:   serial
Net:   Gem.e000b000
Hit any key to stop autoboot:  1  0
Device: zynq_sdhci
Manufacturer ID: 3
OEM: 5344
Name: SU08G
Tran Speed: 50000000
Rd Block Len: 512
SD version 3.0
High Capacity: Yes
Capacity: 7.4 GiB
Bus Width: 4-bit
Booting Xillinux...
reading xillydemo.bit
2083851 bytes read in 157 ms (12.7 MiB/s)
  design filename = "xillydemo.ncd;HW_TIMEOUT=FALSE;UserID=0xFFFFFFFF"
  part number = "7z010clg400"
  date = "2014/03/11"
  time = "12:10:22"
  bytes in bitstream = 2083740
zynq_load: Align buffer at 10006f to 100080(swap 1)
reading uImage
3499448 bytes read in 258 ms (12.9 MiB/s)
reading devicetree.dtb
7702 bytes read in 15 ms (501 KiB/s)
## Booting kernel from Legacy Image at 03000000 ...
   Image Name:   Linux-3.12.0-xillinux-1.3
   Image Type:   ARM Linux Kernel Image (uncompressed)
   Data Size:    3499384 Bytes = 3.3 MiB
   Load Address: 00008000
   Entry Point:  00008000
   Verifying Checksum ... OK
## Flattened Device Tree blob at 02a00000
   Booting using the fdt blob at 0x2a00000
   Loading Kernel Image ... OK
   Loading Device Tree to 1fb4e000, end 1fb52e15 ... OK

Starting kernel ...
```

At this point, the kernel boots. This generates a flood of messages. Only the beginning is listed here.

```
Uncompressing Linux... done, booting the kernel.
[    0.000000] Booting Linux on physical CPU 0x0
[    0.000000] Initializing cgroup subsys cpuset
[    0.000000] Initializing cgroup subsys cpuacct
[    0.000000] Linux version 3.12.0-xillinux-1.3 (eli@ocho.localdomain) (gcc version 4.5.1 (Sourcery G++ Lite 2010.09-62) )
               #1 SMP PREEMPT Thu Mar 13 18:39:32 IST 2014
[    0.000000] CPU: ARMv7 Processor [413fc090] revision 0 (ARMv7), cr=18c5387d
[    0.000000] CPU: PIPT / VIPT nonaliasing data cache, VIPT aliasing instruction cache
[    0.000000] Machine: Xilinx Zynq Platform, model: Xilinx Zynq
[    0.000000] bootconsole [earlycon0] enabled
[    0.000000] Memory policy: Data cache writealloc
[    0.000000] PERCPU: Embedded 7 pages/cpu @c0efb000 s7936 r8192 d12544 u32768
[    0.000000] Built 1 zonelists in Zone order, mobility grouping on.  Total pages: 260624
[    0.000000] Kernel command line: console=ttyPS0,115200n8 consoleblank=0 root=/dev/mmcblk0p2 rw rootwait earlyprintk
```

and it goes on and on. After no more than 10 seconds, a boot prompt appears as follows. There's possibly a few seconds' pause before this final piece of output:

```
Ubuntu 12.04 LTS localhost.localdomain ttyPS0

localhost login: root (automatic login)

Last login: Thu Jan  1 00:00:06 UTC 1970 on tty1
Welcome to the Xillinux distribution for Zynq-7000 EPP.

You may communicate data with standard FPGA FIFOs in the logic fabric by
writing to or reading from the /dev/xillybus_* device files. Additional
pipe files of that sort can be set up by configuring and downloading a
custom IP core from Xillybus' web site (at the IP Core Factory).

For more information: http://www.xillybus.com.

root@localhost:~#
```

4.4 To do soon after the first boot

4.4.1 Resize the file system

The root file system image is kept small so that writing it to the device is as fast as possible. On the other hand, there is no reason not to use the (Micro)SD card's full capacity.

IMPORTANT:

There's a significant risk of erasing the entire (Micro)SD card's content while attempting to resize the file system. It's therefore recommended to do this as early as possible, while the cost of such a mishap is merely to repeat the (Micro)SD card initialization (writing the image and populating the boot partition)

The starting point is typically as follows:

```
# df -h
Filesystem      Size  Used Avail Use% Mounted on
/dev/root       1.7G  1.4G  173M  93% /
devtmpfs        243M  4.0K  243M   1% /dev
none             49M  664K   48M   2% /run
none            5.0M     0  5.0M   0% /run/lock
none            243M   76K  243M   1% /run/shm
```

So the root filesystem is 1.7 GB, with 173 MB free.

The first stage is to repartition the (Micro)SD card. At shell prompt, type:

```
# fdisk /dev/mmcblk0
```

and then type as following (also see a session transcript below):

- d [ENTER] – Delete partition
- 2 [ENTER] – Choose partition number 2
- n [ENTER] – Create a new partition
- Press [ENTER] 4 times to accept the defaults: A primary partition, number 2, starting at the lowest possible sector and ending on the highest possible one.
- w [ENTER] – Save and quit.

If something goes wrong in the middle of this sequence, just press CTRL-C (or q [ENTER]) to quit fdisk without saving the changes. Nothing changes on the (Micro)SD card until the last step.

A typical session looks as follows. Note that the sector numbers may vary.

```
root@localhost:~# fdisk /dev/mmcblk0

Command (m for help): d
Partition number (1-4): 2

Command (m for help): n
Partition type:
   p   primary (1 primary, 0 extended, 3 free)
   e   extended
Select (default p):
Using default response p
Partition number (1-4, default 2):
Using default value 2
First sector (32130-15523839, default 32130):
Using default value 32130
Last sector, +sectors or +size{K,M,G} (32130-15523839, default 15523839):
Using default value 15523839

Command (m for help): w
The partition table has been altered!

Calling ioctl() to re-read partition table.

WARNING: Re-reading the partition table failed with error 16:
         Device or resource busy.
The kernel still uses the old table. The new table will be used at
the next reboot or after you run partprobe(8) or kpartx(8)
Syncing disks.
```

If the default first sector displayed on your system is different from the one above, pick your system's default, and not the one shown here.

The only place in this sequence, where it might make sense to divert from fdisk's defaults, is the last sector, in order to make a file system smaller than the maximum possible (but there's no need to do this).

As the warning at the bottom says, Linux' view of the partition table can't be updated, because the root partition is in use. So a reboot is due:

```
# shutdown -r now
```

The system should boot up just like before, but the boot may fail at any stage if something has been done incorrectly during the repartitioning.

The file system has not been resized yet; it has only been given room to resize. So at shell prompt, type:

```
# resize2fs /dev/mmcblk0p2
```

to which the following response is expected:

```
resize2fs 1.42 (29-Nov-2011)
Filesystem at /dev/mmcblk0p2 is mounted on /; on-line resizing required
old_desc_blocks = 1, new_desc_blocks = 1
The filesystem on /dev/mmcblk0p2 is now 1936463 blocks long.
```

The block count depends on the size of the partition, so it may vary.

As the utility says, the resizing takes place on a file system that is actively used. This is safe as long as power isn't lost in the middle.

The result is effective immediately: There is no need to reboot.

A typical session using an 8 GB (Micro)SD card:

```
# df -h
Filesystem      Size  Used Avail Use% Mounted on
/dev/root       7.4G  1.6G  5.5G  23% /
devtmpfs        243M  4.0K  243M   1% /dev
none             49M  664K   48M   2% /run
none            5.0M     0  5.0M   0% /run/lock
none            243M   68K  243M   1% /run/shm
```

Note that the sizes given by the "df -h" utility are with 1 GiB = 2^{30} bytes, which is 7.3% larger than a "commercial" Gigabyte of 10^9 bytes. That's why an 8 GB card appears as 7.4 GiB above.

4.4.2 Allow remote SSH access

To install an ssh server on the board, connect the board to the Internet and type

```
# apt-get install ssh-server
```

at shell prompt. Please note that the root password is none by default, and ssh rightfully refuses to login someone without a password.

To rectify this, set the root password with

```
# passwd root
```

at shell prompt.

After installing the SSH server, it's recommended to add the following line at the bottom of /etc/ssh/sshd_config:

```
UseDNS no
```

This will turn off the rather meaningless reverse DNS check made by the SSH server, which causes a delay when attempting to start a session.

4.4.3 Compiling locale definitions

In certain situations, application software requires knowledge on how to display characters depending on some locale setting. The most common reason is when connecting with ssh to the board, because ssh copies the client's locale settings to the server's environment when setting up a shell session.

This leads to warning messages like

```
bash: warning: setlocale: LC_CTYPE: cannot change locale (en_US.UTF-8)
```

When such error message appears, it's quite obvious which locale is missing. To resolve this before an error occurs, check which locales are required:

```
# locale
LANG=en_US.UTF-8
LANGUAGE=
LC_CTYPE="en_US.UTF-8"
LC_NUMERIC="en_US.UTF-8"
LC_TIME="en_US.UTF-8"
LC_COLLATE="en_US.UTF-8"
LC_MONETARY="en_US.UTF-8"
LC_MESSAGES="en_US.UTF-8"
LC_PAPER="en_US.UTF-8"
LC_NAME="en_US.UTF-8"
LC_ADDRESS="en_US.UTF-8"
LC_TELEPHONE="en_US.UTF-8"
LC_MEASUREMENT="en_US.UTF-8"
LC_IDENTIFICATION="en_US.UTF-8"
LC_ALL=
```

Compare with the available locales:

```
# locale -a
C
C.UTF-8
POSIX
```

In this example, it's quite obvious which locale is missing, so let's add it:

```
# locale-gen en_US.UTF-8
Generating locales...
  en_US.UTF-8... done
```

Note that the necessary locale depends on the computer from which the ssh connection is made. Users from different places in the world need to install different locales on their boards, to achieve smooth ssh sessions. The shell on the UART port is based upon POSIX locale, which is included by default.

4.5 Using the desktop

The Xillinux desktop (on Zedboard and Zybo) is just like any Ubuntu desktop. Due to the (Micro)SD card's relatively low data bandwidth, applications will load somewhat slowly, but the desktop itself is fairly responsive.

To run applications in the desktop environment, click the top-left Ubuntu icon on the desktop ("Dash home") and type the name of the desired application, e.g. "terminal" for a shell prompt terminal window, or "edit" for the gedit text editor.

Additional packages can be installed with "apt-get" like with any Ubuntu distribution.

Upgrading the entire Ubuntu operating system with apt is not possible, and will fail leaving the system unbootable.

4.6 Shutting down / rebooting

To power down the system, pick the top-right icon on the desktop (if available), and click "Shut Down...". Alternatively, type

```
# halt
```

at shell prompt. When a textual message saying "System Halted" appears on the UART console, it's safe to power the board off. An alternative sign for a complete

shutdown is that the cursor on the VGA/DVI text console (when applicable) stops blinking.

For a reboot, which includes reconfiguring the FPGA (PL) part, pick the reboot option on the desktop menu, or type

```
# shutdown -r now
```

Note that this doesn't necessarily reset the external hardware components, e.g. the sound chip.

4.7 Taking it from here

The Zynq board has now become a computer running Linux for all purposes. The basic steps for interaction with the logic fabric through the Xillybus IP core can be found in Getting started with Xillybus on a Linux host. Note that the driver for Xillybus is already installed in the Xillinux distribution, so the part in the guide dealing with installation can be skipped.

Paragraph 5.1 refers to integrating application-specific logic with the Linux operating system.

Note that Xillinux includes the gcc compiler and GNU make, so host applications can be compiled natively on the board's processors. Additional packages may be added to the distribution with apt-get as well.

5

Making modifications

5.1 Integration with custom logic

The Xillinux distribution is set up for easy integration with application logic. The front end for connecting data sources and sinks is the xillydemo.v or xillydemo.vhd file (depending on the preferred language). All other HDL files in the boot partition kit can be ignored for the purpose of using the Xillybus IP core as a transport of data between the Linux host and the logic fabric.

Additional HDL files with custom logic designs may be added to the project presented in paragraph 3.5, and then rebuilt the same way it was done the first place. To boot the system with the updated logic, copy the new xillydemo.bit into the (Micro)SD's card's boot partition, overwriting the existing one. Note that it's possible to use the Zynq board itself for copying xillydemo.bit into the boot partition, as shown in paragraph 3.7.

There is no need to repeat the other steps of the initial distribution deployment, so the development cycle for logic is fairly quick and simple.

Programming the PL part through JTAG is not supported.

When attaching the Xillybus IP core to custom application logic, it is warmly recommended to interact with the Xillybus IP core only through FIFOs, and not attempt to mimic a FIFO's behavior with logic, at least not in the first stage.

An exception for this is when connecting memories or register arrays to Xillybus, in which case the schema shown in the xillydemo module should be followed.

In the xillydemo module, FIFOs are used to loop back data arriving from the host back to it. Both of the FIFOs' sides are connected to the Xillybus IP core, which makes the core function as its own data source and sink.

In a more useful setting, only one of the FIFO's ends is connected to the Xillybus IP core, and the other end to an application data source or sink.

The FIFOs used in the xillydemo module accept only one common clock for both sides, as both sides are driven Xillybus' main clock. In a real-life application, it may be desirable to replace them with FIFOs having separate clocks for reading and writing, allowing data sources and sinks to be driven by a clock other than the bus clock. By doing this, the FIFOs serve not just as mediators, but also for proper clock domain crossing.

Note that the Xillybus IP core expects a *plain* FIFO interface, (as opposed to First Word Fall Through) for FPGA to host streams.

The following documents are related to integrating custom logic:

- The API for logic design: Xillybus FPGA designer's guide
- Basic concepts with the Linux host: Getting started with Xillybus on a Linux host
- Programming applications: Xillybus host application programming guide for Linux
- Requesting a custom Xillybus IP core: It's easiest to follow the instructions at the IP Core Factory, but there's also The guide to defining a custom Xillybus IP core

5.2 Using other boards

Before attempting to run Xillinux on a board other than Zedboard, MicroZed or Zybo, certain modifications may be necessary. This is a partial list of issues to pay attention to.

- A purchased board should have a reference processor XML file (for use as ps7_system_prj.xml). This file contains the processor's settings, including de-facto use of the MIO pins and the electrical parameters of the DDR pins. The recommended practice is adopting the reference file, at least a starting point.
- If a reference XML file is adopted, the FPGA CLK1 (FCLK_CLK1) must be set to 100 MHz, regardless of what the reference XML states.
- If changes are made manually, attention should be paid to the processor core's MIO assignments: The ARM core has 54 I/O pins which are routed to physical pins on the chip with a fixed placement. The ARM core is configured in the Xilinx Platform Studio to assign specific roles to these pins (e.g. USB interface, Ethernet etc.), which must match what these pins are wired to on the board.

- Note that if changes are made in the processor's configuration (i.e. in the XML file), the boot.bin must be rebuilt, based upon an FSBL (First Stage Boot Loader) that is derived from the new XML file, and a U-boot binary. The changes made in XPS (or Vivado's IP integrator) take effect through the initialization routine that is part of the FSBL. This routine writes to registers in the ARM processor, with values that reflect the settings made in XPS (or IP integrator), and exported to SDK. Note that the parameters of the processor in the Vivado project may not be accurate, so the FSBL should be generated based upon the XPS project. To set up the sources for U-boot, please refer to the README file in /usr/src/xillinux/uboot-patches/.
- It may also be necessary to make changes in devicetree.dtb, in order to reflect the new setting. The sources of the existing DTB (in DTS format) can be found in the /boot directory.
- The VGA/DVI outputs (if applicable) need to be matched to the target board. This is done by editing the xillybus.v file in the src/ subdirectory. Note that the signals arriving from the "system" module are 8 bits wide, and the truncation to 4 bits takes place in xillybus.v. Hence it's fairly easy to connect these signal to any DVI/VGA encoder chip.

5.3 Changing the system's clock frequencies

The ARM processor's core supplies four clocks for use by the logic fabric, commonly referred to as FCLK_CLKn. It's important to note, that their frequencies are set by the FSBL (First Stage Boot Loader), before U-boot is loaded.

Accordingly, even though the clocks' frequencies are set in the Xilinx Platform Studio, these frequencies are effective only for propagating timing constraints and initialization by bare-metal applications compiled on the SDK.

If the hardware application requires different clock frequencies, the following series of actions is suggested:

- Update the clock frequencies in XPS
- Rebuild the netlist (this is necessary for updating the timing constraints in the .ncf files)
- Export the project to SDK, and create an FSBL application project based upon this. Vivado is not suitable for this, since the processor's parameters in the project may not be accurate.

- Generate a release binary of the FSBL.
- Generate a U-boot binary, by compiling it from its sources. In order to reconstruct Xillinux' U-boot utility, see /usr/src/xillinux/uboot-patches/.
- Generate a boot.bin file from the FSBL and U-boot binaries.
- Replace boot.bin in the boot partition with the new one.

Please refer to Xilinx' guides for the details of how to perform each step.

5.4 Taking over GPIO I/O pins for PL logic

On the Zedboard and Zybo boards, many physical I/O pins are connected to the ARM processor's (PS) GPIO ports, which allows controlling and monitoring these pins directly from Linux. It's however often desired to connect these physical pins to FPGA (PL) logic instead.

The technique for using Zynq's PL pins for I/O is exactly the same as any Xilinx FPGA: The signals are exposed in the toplevel module (xillydemo.v or xillydemo.vhd) as inputs, outputs or inouts. The assignment of physical pins to these signals takes place in xillydemo.ucf.

Since the pins are originally taken by GPIO signals, these are effectively evicted by the PL signals. For example, the following line in the UCF file

```
NET PS_GPIO[55] LOC=V18 | IOSTANDARD=LVCMOS33;
```

can be replaced with

```
NET my_output LOC=V18 | IOSTANDARD=LVCMOS33;
```

if we want my_output to appear on pin V18.

But doing this replacement causes PS_GPIO[55] to lack a pin assignment. Even though there is a chance that Xilinx' tools will place this port automatically during implementation, it's recommended, and usually mandatory, to assign any evicted PS_GPIO with an I/O pin. The alternative is to eliminate the signal, as explained below.

The reason for the sensitivity of these PS_GPIO signals is that XPS includes logic elements (BUFIO) in the netlist it generates, that must reside in an I/O pin site. This is

why these signals can't be left dangling during the implementation of the programming file – each of these elements must be placed.

There are two solutions for these evicted PS_GPIO signals:

- The easy way: Finding unused pins on the device, and assign these pins to the evicted PS_GPIO signals. Even though it's not a very clean solution (GPIO pins are connected to just something on the board), it's practically harmless, because GPIOs are inputs by default. The electrical condition on these pins remains, unless the GPIO gets driven by software accidentally (which isn't likely). For example, on Zedboard, the FMC connector often supplies many unused pins.
- The harder way: The number of PS_GPIO is reduced, so there are less BUFIOs that must be placed. This may be necessary on Zybo, which doesn't have many vacant pins.

In what follows, the second solution is discussed. For example, let's assume that PS_GPIO[55:47] were removed from the UCF file, for the sake of replacing their pins with signals from the PL. Note that if the pins from lower PS_GPIO indexes were needed, the evicted PS_GPIO signals should take over the pins of those with the highest indexes, and the latter are then eliminated. There is no possibility to eliminate a certain range of PS_GPIO indexes, only reduce the maximal index.

The width of PS_GPIO should be reduced in xillydemo.v/vhd to reflect those that have pin assignments in the UCF file.

This is not enough however. Attempting to build the project now will fail in the mapper stage with

```
ERROR:Place:866 - Not enough valid sites to place the following IOBs:
   IO Standard: Name = LVCMOS18, VREF = NR, VCCO = 1.80, TERM = NONE, DIR =
   BIDIR, DRIVE_STR = 12
    processing_system7_0_GPIO<48>
    processing_system7_0_GPIO<49>
    processing_system7_0_GPIO<50>
    processing_system7_0_GPIO<54>
    processing_system7_0_GPIO<53>
    processing_system7_0_GPIO<52>
    processing_system7_0_GPIO<51>
    processing_system7_0_GPIO<55>
```

To resolve this, the XPS project needs to be adapted and rebuilt, so that the width of its GPIO signal matches:

In XPS' main window (the large one to the top right), choose the "Zynq" tab for the graphical representation of the processing system. Click on the I/O Peripherals block to the left to open the Zynq PS MIO Configurations window. Expand the GPIO Peripheral entry (click on the lowest "+" symbol to the left). The check box next to "EMIO GPIO (Width)" should already be enabled. Reduce the width from 56 to the desired one (e.g. 48 following the example above).

Then choose the "Ports" tab (to the right of the "Zynq" tab) and expand "External Ports". Find processing_system_7_0_GPIO and reduce its range to match the chosen width. In the example, it's reduced from [55:0] to [47:0].

Pick Project > Clean all generated files on the top menu, and generate the netlist again, by clicking on "Generate Netlist" to the left (like when the project was built for the first time).

If the range of processing_system_7_0_GPIO matches the width assigned to EMIO GPIO, the netlist generation will finish successfully, Otherwise, the following error is issued by XPS when attempting to build a netlist:

```
ERROR:EDK:4073 - INSTANCE: processing_system7_0, PORT: GPIO_I, CONNECTOR:
   processing_system7_0_GPIO_I - 56 bit-width connector assigned to 48 bit-width
   port - /tmp/xillinux-eval-zybo-1.3/system/system.mhs line 116
```

And finally, open /verilog/src/system.v or /vhdl/src/system.v (depending on where xillydemo.bit is built) and change and modify the line saying

```
  inout [55:0] processing_system7_0_GPIO;
```

So that the [55:0] range matches the one set for this signal ([47:0] in the example). It's also possible to change the width of PS_GPIO in xillybus.v to match the same width, but this has no significance, except for silencing warnings.

5.5 Targeting 7020 MicroZed

The boot partition kit available for MicroZed targets 7010 MicroZed boards by default. It's however possible to target 7020 MicroZed using Vivado, after making a minor change in the xillydemo-vivado.tcl file used to create the Vivado project (that is, verilog/xillydemo-vivado.tcl or vhdl/xillydemo-vivado.tcl in the bundle, depending on the language chosen).

Just after unzipping the kit (and before using it in Vivado), the file should be edited, changing the line saying

```
set thepart "xc7z010clg400-1"
```

(around line 11) to

```
set thepart "xc7z020clg400-1"
```

The rest of the build process is exactly the same.

6

Linux notes

6.1 General

This section contains Linux-related topics that are specific to Xillinux.

A wider view on Xillybus and Linux can be found in these documents:

- Getting started with Xillybus on a Linux host
- Xillybus host application programming guide for Linux

6.2 Compiling the Linux kernel

Due to its size, the full Linux kernel is not included in the Xillinux distribution. It can be re-established fairly easily by acquiring the sources on which it is based, and applying a set of patches.

The sources can be downloaded with

```
$ git clone https://github.com/Digilent/linux-Digilent-Dev.git
```

Xillinux' kernel is based upon the commit tagged xilinx-v2013.4, but if more recent commits are in place, it's possibly beneficial to go for a later version.

The adjustments for Xillinux are given in /usr/src/xillinux/kernel-patches as git patches. They will apply cleanly on the Linux version represented by the commit tag mentioned above, and may work well with other versions as well.

Note that the first patch in the kernel-patches directory adds a .config file, and is there for convenience, even though it's not common practice to add the configuration file with a patch.

The same configuration file, as well as the device tree used, can be found in the /boot directory.

Once the kernel sources are set up, standard kernel compilation practices apply. Please refer to the guide relevant to your working environment. Cross compilation of the kernel is recommended to save time.

6.3 Compiling kernel modules

The Xillinux distribution comes with the running kernel's compilation headers. This is not enough to compile the kernel itself, but allows kernel modules to be compiled natively on the platform.

The standard way to compile a "private" (out-of-tree) kernel module is using a Makefile which invokes the kernel's own build environment, after setting up an environment variable, telling it to compile a specific module.

The minimal Makefile for a *native* compilation (performed by the Zynq processor itself) of a kernel module consisting of a single source file, sample.c, is as follows:

```
ifneq ($(KERNELRELEASE),)
obj-m := sample.o
else
TARGET := $(shell uname -r)
PWD := $(shell pwd)
KDIR := /lib/modules/$(TARGET)/build

default:
        @echo $(TARGET) > module.target
        $(MAKE) -C $(KDIR) SUBDIRS=$(PWD) modules
endif
```

Note that the name of the module, "sample", is mentioned only in the "obj-m" line. This is the only thing that differs from Makefile to Makefile.

Using this Makefile typically results in compilation session as follows (run on the board itself; this is not a cross compilation):

```
# make
make -C /lib/modules/3.12.0-xillinux-1.3/build SUBDIRS=/root/sample modules
make[1]: Entering directory `/usr/src/kernels/3.12.0-xillinux-1.3'
  CC [M]  /root/sample/sample.o
```

```
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /root/sample/sample.mod.o
  LD [M]  /root/sample/sample.ko
make[1]: Leaving directory '/usr/src/kernels/3.12.0-xillinux-1.3'
```

6.4 Sound support

6.4.1 General

The Zedboard and Zybo boards support sound recording and playback by virtue of Analog Devices' ADAU1761 and SSM2603 chipsets, respectively. These are connected to the Zynq device's logic fabric (PL) pins only.

Except for its obvious functionality, the sound support package also demonstrates how the Xillybus IP core can be used to transport data, as well as to program a chip through SMBus / I^2C.

Xillinux supports sound natively by interfacing dedicated Xillybus streams with the most common toolkit for sound in Linux today, Pulseaudio. As a result, virtually any application requiring a sound card will properly use the board's sound chip as the system's default input and output.

It's also possible to turn the Pulseaudio daemon off, and work directly with the Xillybus streams (/dev/xillybus_audio). This presents a simpler programming interface of just opening a device file, at the cost of losing the native functionality of other applications.

Unlike the common approach for Linux sound cards, there is no dedicated kernel driver (e.g. ALSA) for the sound interface, since the Xillybus host driver handles the data transport anyhow. This has no significance, even not to programs expecting to work with /dev/dsp, since Pulseaudio has the capability of faking this interface with the "padsp" utility.

6.4.2 Usage details

By default, sound is played back to the headphones jack (black). On Zedboard, the same output goes to Line Out (green) as well. For recording, only the microphone input (pink) is used, but this can be changed, as described next.

6.4.3 Related boot scripts

The upstart script /etc/init/xillinux-sound.conf executes /usr/local/bin/xillinux-sound when the system boots up. The latter script identifies which board it runs on, and waits for the Xillybus IP core to load, if necessary.

It then runs /usr/local/bin/zybo_sound_setup.pl or /usr/local/bin/zedboard_sound_setup.pl, depending on which board was identified. When the script finished, the Pulseaudio daemon is launched as root. This is not recommended practice on a multi-user computer, but since Xillinux runs with root as the default user, there is no point in protecting against escalation of permissions.

This command should be disabled if direct access to /dev/xillybus_audio is needed.

zybo_sound_setup.pl and zedboard_sound_setup.pl are Perl scripts, which set up the audio chip's registers for proper operation. They are fairly straightforward, even for non-Perl programmers, using the /dev/xillybus_smbus device file to initiate transactions on the chip's I^2C bus.

zedboard_sound_setup.pl can be edited to achieve a different setup of the audio chip. In particular, if the Line In input is the desired source for recording, the lines

```
write_i2c(0x400a, 0x0b, 0x08);
write_i2c(0x400c, 0x0b, 0x08);
```

should be replaced with

```
write_i2c(0x400a, 0x01, 0x05);
write_i2c(0x400c, 0x01, 0x05);
```

In a similar manner, zybo_sound_setup.pl can be edited, replacing

```
write_i2c(0x04, 0x14);
```

with

```
write_i2c(0x04, 0x10);
```

to use Line In for recording.

6.4.4 Accessing /dev/xillybus_audio directly

/dev/xillybus_audio can be written to directly for playback, or read from for recording. The sample format is 32 bit per audio sample, divided into two 16-bit signed integers

in little Endian format. The most significant word corresponds to the left channel.

The sampling rate is fixed at 48000 Hz.

A Windows WAV file with this sampling rate is likely to play correctly if written directly to the /dev/xillybus_audio (the header will be played back as well for about 1 ms) e.g. with

```
# cat song.wav > /dev/xillybus_audio
```

If the response is

```
-bash: /dev/xillybus_audio: Device or resource busy
```

another process is having the device file open for write, possibly the Pulseaudio daemon. There is no problem having the device file opened for read by one process and for write by another.

6.4.5 Pulseaudio details

Pulseaudio interacts with the /dev/xillybus_audio device file through a couple of dedicated Pulseaudio modules, module-file-sink and module-file-source. Their sources can be found in Xillinux' file system at /usr/src/xillinux/pulseaudio/.

These are slight modifications of standard Pulseaudio modules for using UNIX pipes as data sinks and sources (module-pipe-sink and module-pipe-sink).

The modules are automatically loaded when Pulseaudio starts, by virtue of the two following lines in /etc/pulse/default.pa:

```
load-module module-file-sink file=/dev/xillybus_audio rate=48000
load-module module-file-source file=/dev/xillybus_audio rate=48000
```

These modules are selected as the system's sound interface automatically, as there are no other alternatives.

6.5 The OLED utility (Zedboard only)

By default, Xillinux starts an activity meter utility on boot, displaying approximate CPU usage percentage and an indication of the I/O rate on the SD flash disk.

The CPU percentage is based upon /proc/stat, where all time not spent idle is considered used CPU time.

An estimation of the SDIO traffic is made based upon the rate at which interrupts are sent to the respective driver. There is no known figure for a full utilization of this resource. Rather, the utility displays 100% for an interrupt rate that appears to be maximal according to measurements.

To display the graphical output on the board's OLED, the bitmap is sent to /dev/zed_oled, which is created by Digilent's driver. It's worth to mention that this driver sends SPI data to the OLED device with a bit-banging mechanism, toggling the clock and data in software. Hence there is a slight CPU consumption sending 512 bytes this way several times per second, but the impact to the overall system performance is minimal.

To change the parameters of this utility, edit /etc/init/zedboard_oled.conf or delete this file to disable its launch at system boot altogether.

This file boils down to the following command

```
/usr/local/bin/zedboard_oled /proc/irq/$irqnum/spurious 4 800
```

The application, zedboard_oled, takes three arguments:

- The /proc file to monitor for SDIO-related interrupts. $irqnum is the IRQ number of the mmc0 device.
- The rate at which the OLED display is updated, in times per second.
- What is considered to be 100% SDIO interrupt rate, in interrupts per second. The current figure was found by trial and error.

7

Troubleshooting

7.1 Implementation errors

Slight differences between releases of Xilinx' tools sometimes result in failures to run the implementation chain for creating a bitfile.

If the problem isn't solved fairly quickly, please seek assistance through the email address given at the company's web site. Please attach the output log of the process that failed, in particular around the first error reported by the tool. Also, if custom changes were made in the design, please detail these changes. Also please state which version of the ISE / Vivado tools was used.

If this error is issued by Vivado on an implementation for VHDL,

```
ERROR: [Place 30-58] IO placement is infeasible. Number of unplaced terminals (3) is greater than number of available sites (2).
The following Groups of I/O terminals have not sufficient capacity:
Bank: 35:
 The following table lists all user constrained IO terminals
 Please analyze any user constraints (PACKAGE_PIN, LOC, IOSTANDARD ) which may cause a feasible placement to be impossible.
 The following table uses the following notations:
 c1 - is IOStandard compatible with bank? 1 - compatible, 0 is not
 c2 - is IO VREF compatible with INTERNAL_VREF bank? 1 - compatible, 0 is not
 c3 - is IO with DriveStrength compatible with bank? 1 - compatible, 0 is not
+--------+------------+----+----+----+-------------------------------------------------------+
| BankId | IOStandard | c1 | c2 | c3 | Terminal Name                                         |
+--------+------------+----+----+----+-------------------------------------------------------+
|   35   |LVCMOS18    |  1 |  1 |  1 |   PS_CLK                                              |
|   35   |LVCMOS18    |  1 |  1 |  1 |   PS_PORB                                             |
|   35   |LVCMOS18    |  1 |  1 |  1 |   PS_SRSTB                                            |
+--------+------------+----+----+----+-------------------------------------------------------+
```

please edit xillydemo.vhd as mentioned in paragraph 3.5.2.

Another possible error resulting from the same problem:

```
ERROR: [Drc 23-20] Rule violation (NSTD-1) Unspecified I/O Standard
        ...
        Problem ports: PS_CLK, PS_PORB, PS_SRSTB.
```

```
ERROR: [Drc 23-20] Rule violation (RTSTAT-1) Unrouted net ...
ERROR: [Drc 23-20] Rule violation (UCIO-1) Unconstrained Logical Port ...
```

7.2 Problems with USB keyboard and mouse

Almost all USB keyboards and mice meet a standard specification for compatible behavior, so it's unlikely to face problems with devices that aren't recognized. The first things to check if something goes wrong are:

- Zedboard only: Are you using the correct USB plug? It should be the one marked "USB OTG", farther away from the power switch.
- Zedboard only: Is there any 5V supply to the devices? Is the JP2 jumper installed? With the Zedboard powered on, connect an optical USB mouse, and verify that the LED goes on.
- If a USB hub is used, attempt to connect only a keyboard or mouse directly to the board.

Helpful information may be present in the general system log file, /var/log/syslog. Viewing its content with "less /var/log/syslog" can be helpful at times. Even better, typing "tail -f /var/log/syslog" will dump new messages to the console as they arrive. This is useful in particular, as events on the USB bus are always noted in this log, including a detailed description on what was detected and how the event was handled.

Note that a shell prompt is also accessible through the USB UART, so the log can be viewed with a serial terminal if connecting a keyboard fails.

7.3 File system mount issues

Experience shows that if a proper (Micro)SD card is used, and the system is shut down properly before powering off the board, there are no issues at all with the permanent storage.

Powering off the board without unmounting the root file system is unlikely to cause permanent inconsistencies in the file system itself, since the ext4 file system repairs itself with the journal on the next mount. There is however an accumulating damage in the operating system's functionality, since files that were opened for write when the power went off may be left with false content or deleted altogether. This holds true for any computer being powered off suddenly.

If the root file system fails to mount (resulting in a kernel panic during boot) or mounts read-only, the most likely cause is a low quality (Micro)SD card. It's quite typical for such storage to function properly for a while, after which random error messages surface. If /var/log/syslog contains messages such as this one,

```
EXT4-fs (mmcblk0p2): warning: mounting fs with errors, running ec2fsck
   is recommended
```

the (Micro)SD card is most likely the reason.

To avoid these problems, please insist on a Sandisk device.

7.4 "startx" fails (Graphical desktop won't start)

Even though not directly related, this problem is reported quite frequently when the (Micro)SD card is a non-Sandisk. The graphical software reads a large amount of data from the card when starting up, and is therefore likely to be the notable victim of an (Micro)SD card that generates read errors.

The obvious solution is using a Sandisk (Micro)SD card.