Getting started with Xillinux for Cyclone V SoC (SoCKit)

Xillybus Ltd.
www.xillybus.com

Version 2.0

1 Introduction **3**
1.1 The Xillinux distribution 3
1.2 The Xillybus IP core 4

2 Prerequisites **6**
2.1 Hardware 6
2.2 Downloading the distribution 7
2.3 Development software 7
2.4 Experience with FPGA design 8

3 Building Xillinux **9**
3.1 Overview 9
3.2 Unzipping the boot image kit 10
3.3 Generating the processor's wrapper 11
3.4 Generating the raw bitstream file 12
3.5 Loading the microSD with the image 14
3.5.1 General 14
3.5.2 Loading the image (Windows) 15
3.5.3 Loading the image (Linux) 16

3.6 Copying the soc_system.rbf file into the microSD card 17

4 Performing the boot 19

4.1 Jumper and DIP switch settings 19
4.2 Attaching peripherals 19
4.3 Powering up the board 21
4.4 UART output during boot 23
4.5 To do soon after the first boot 25
4.5.1 Resize the file system 25
4.5.2 Allow remote SSH access 28
4.6 Using the desktop 28
4.7 Shutting down 29
4.8 What to do from here 29

5 Making modifications 30

5.1 Integration with custom logic 30
5.2 Using other boards 31
5.3 Custom build projects and preflow.tcl 32
5.4 Changes in the Qsys project 33

6 Troubleshooting 34

6.1 Port ”xillybus_0_conduit_...” does not exist 34
6.2 Problems with USB keyboard and mouse 34
6.3 File system mount issues 35
6.4 “startx” fails (Graphical desktop won’t start) 35

1 Introduction

IMPORTANT:

Due to relatively low public interest, Xillinux for SoCKit is phasing out: It works out of the box with no known issues, but its implementation is limited to the rather outdated Quartus II 13.0sp1 for building the FPGA bundle. This is not expected to change in the future. The overall support for Cyclone V SoC, and SoCKit in particular, is limited.

1.1 The Xillinux distribution

Xillinux is a complete, graphical, Ubuntu 12.04 LTS-based Linux distribution for the SoCKit board, intended as a platform for rapid development of mixed software / logic projects. 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.

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.

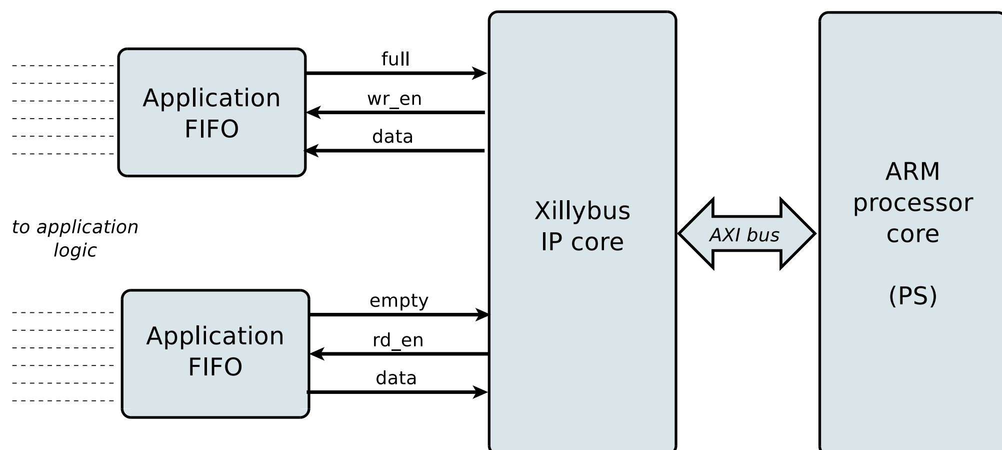
Xillinux is also a kickstart development platform for integration between the device's FPGA logic fabric and plain user space applications running on the ARM processor. With its included Xillybus IP core and driver, no more than basic skills in programming and logic design are needed to complete the 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 as well as the interface between application logic and the processor, by presenting a simple and yet efficient working environment to the application designers.

1.2 The Xillybus IP core

Xillybus is a DMA-based end-to-end solution for data transport between an FPGA and a host that runs Linux or Microsoft Windows. It offers a simple and intuitive interface to the designer of the FPGA logic as well as the programmer of the software. 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 IP core 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 host interacts with device files that behave like named pipes. The Xillybus IP core and driver transport data efficiently and intuitively between the FIFOs in the FPGAs and their related device files on the host.

The IP core is built instantly per customer's spec, using an online web application. The number of streams, their direction and other attributes are defined by customer to achieve an optimal balance between bandwidth performance, synchronization, and simplicity of design.

After going through the preparation as described in this guide, it's recommended to build and download your custom IP core at http://xillybus.com/custom-ip-factory.

This guide explains how to rapidly set up the Xillinux distribution, with a Xillybus IP core included. This IP core can be attached to user-supplied data sources and data sinks, for real application scenario testing. It's not a demonstration kit, but a fully functional starter design, which can perform useful tasks as is.

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

For those who are curious, a brief explanation on how Xillybus IP core is implemented can be found in Appendix A of Xillybus host application programming guide for Linux.

2 Prerequisites

2.1 Hardware

The Xillybus for Sockit Linux distribution (Xillinux) currently supports only the SoCKit board.

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.

The following pieces of equipment are also 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 3.8.0, if the keyboard and mouse are not combined in a single USB plug
- A USB cable to the SoCKit board card, type A receptacle (female) to USB Micro B plug. This cable is **not included** when purchasing a SoCKit board.
- A reliable microSD card with 4 GB or more, most preferably a card manufactured by Sandisk.

  Other brands are not recommended, as problems have been reported using them with Xillinux.

- A USB adapter between a microSD 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.

  Note that an SD card to USB adapter can be used instead of a MicroSD to USB adapter, in conjunction with a SD to microSD adapter, as the difference between SD and microSD is just the physical form factor.

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.

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 microSD card consisting of the file system to be seen by Linux at bootup, and a set of files for implementation with the Intel FPGA tools to produce a first-stage boot image. 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 application. 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 IP core in end-user designs, running real-life data and field testing. There is no limitation on how the IP 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

Only Quartus II 13.0sp1 (Web or Subscription Edition) may be used to build Xillinux. Attempting to build the bundle with any other revision will fail with an error. As the ARM processor is new to Intel FPGA devices, the supporting software tools are changing quite rapidly, so the only way to ensure a reliable outcome is using exactly the same revision with which Xillinux has been properly tested.

As Xillinux for Cyclone V SoC is phasing out, there are unfortunately no plans to lift this restriction in the future.

The 64-bit version of Quartus II is preferred, in particular on Windows machines, since the tools allocate more than 2GB of RAM, which may fail with 32-bit Quartus II. If a 32-bit Windows XP is used, adding the `/3gb` flag to boot.ini allows the tools to run successfully. See http://msdn.microsoft.com/en-us/library/windows/hardware/gg487508.aspx.

32-bit Windows 7 and up can be modified with the `increaseuserva` parameter, see:

http://msdn.microsoft.com/en-us/library/windows/hardware/ff542202(v=vs.85).aspx.

The Soc EDS package is **not necessary** for building Xillinux. The compilation of user space programs and/or kernel modules can be done on the board's processor directly.

This software can be downloaded directly from Intel FPGA's website (https://www.intel.com).

2.4 Experience with FPGA design

When working with the SoCKit board, no previous experience with FPGA design is necessary to have the distribution running on the platform. Working with another board requires some knowledge with using Intel FPGA's tools, and possibly some basic skills 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. Nevertheless, 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. Accordingly, the preparation time for the first test run is a bit long (typically 30 minutes, most of which consist of waiting for Xilinx' tools). However this long preparation shortens the cycle for integrating custom logic.

To perform boot of the Xillinux distribution from a microSD card, it must have three components:

- An initial boot image environment, residing in a special partition, consisting of the U-boot loader
- A small FAT filesystem, with files containing the configuration bitstream for the FPGA part, the Linux kernel binary and its device tree.
- A root file system mounted by Linux.

All of this, except the FPGA's bitstream, is already included in the microSD image for Xillinux.

The various operations for preparing the microSD are detailed step by step in this section. Most of the time is spent on preparing the FPGA bitstream.

This procedure assumes a SoCKit board is used. It consists of the following steps, which must be done in the order outlined below:

- Unzipping the boot image kit

- Generating the processor's wrapper and bus infrastructure
- Implementing the main FPGA project and converting the bitstream to the correct format.
- Writing the raw Xillinux image to the microSD card
- Copying the bitstream file into the microSD card

How to work with other boards is discussed in paragraph 5.2.

3.2 Unzipping the boot image kit

Unzip the previously downloaded xillinux-eval-sockit-*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 files. The file in VHDL to edit is in the 'src' subdirectory
- core – Precompiled binaries of the Xillybus IP cores
- soc_system – ARM processor wrapper and bus infrastructure
- instantiation templates – Contains the instantiation templates in Verilog and VHDL

Note that both 'verilog' and 'vhdl' directories also contain the QSF file for the SoCKit board. This file must be edited if another board is used.

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

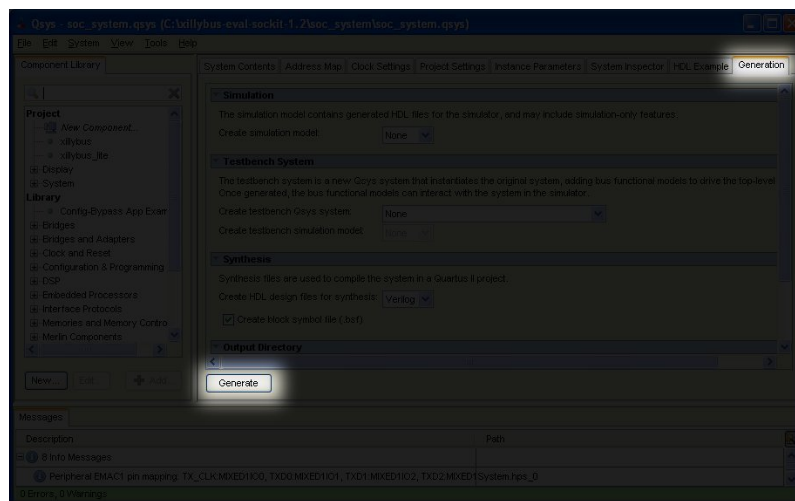
The interface with the Xillybus IP core takes place in the xillydemo.v or xillydemo.vhd files in the respective 'src' subdirectories. This is the file to edit in order to try Xillybus with your own data sources and sinks.

3.3 Generating the processor's wrapper

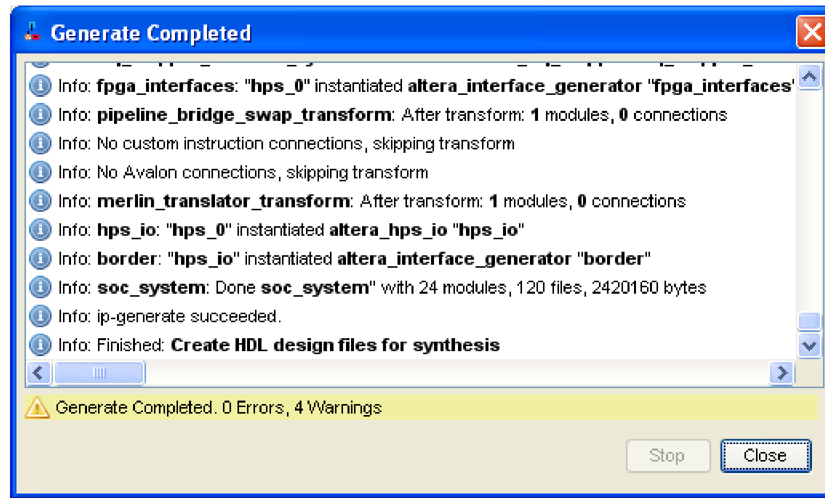
Open Quartus II, possibly by double-clicking the 'xillydemo.qpf' file in either the 'verilog' or 'vhdl' subdirectory in the boot image kit.

Within Quartus, open the Qsys project by picking "File > Open..." and navigate upwards one directory, enter the soc_system directory, and choose soc_system.qsys. The Qsys tool launches and opens the processor wrapper project.

Pick the "Generation" tab near the top of the window, and click "Generate" (near the bottom)



The process takes a couple of minutes or so, with the progress window ending up like this:



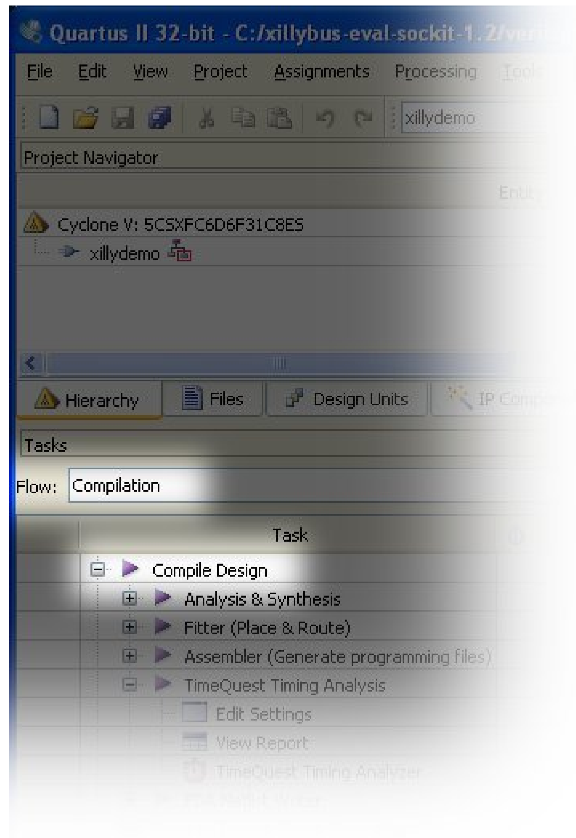
Close the progress windows as well as the Qsys window. There is no need to repeat this process in the future.

3.4 Generating the raw bitstream file

This step is taken after completing the generation of the processor's wrappers, as described in paragraph 3.3.

Depending on your preference, double-click the 'xillydemo.qpf' file in either the 'verilog' or 'vhdl' subdirectory. Quartus II will launch and open the project with the correct settings. If you just finished with Qsys, chances are that you already have Quartus II open. In that case, just verify that your preferred language was chosen (pick Verilog if you don't care).

Make sure that the "flow" is set to "Compilation" and click "Compile Design" to create the FPGA programming file, as shown in this image.



The process produces around 100 warnings, but should end with a dialog box informing that the "Full Compilation was successful". Critical warnings are produced, but no errors should be tolerated. Also, it's always mandatory to verify, after a compilation, that **no** warning said "Timing requirements not met" (332148). This is in particular true later on, when running a compilation of the design after making your own changes in the Verilog/VHDL sources.

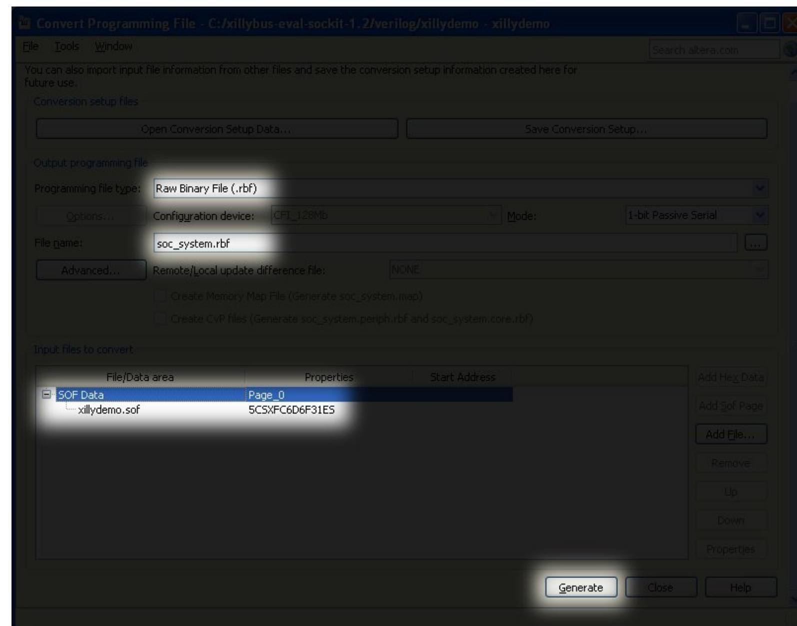
Finally, convert the obtained programming file to the necessary format. In Quartus II, pick File > Convert Programming Files... and pick Raw Binary File (.rbf) as the Programming file type. Just below, set the "File name" to soc_system.rbf.

In the "Input files to convert" area, click "SOF Data" and then "Add File..." to the right. Choose xillydemo.sof.

Then click "Generate" at the bottom right. Close the window after successfully generating the file. The soc_system.rbf should be copied to the MicroSD card, as explained in paragraph 3.6.

IMPORTANT:

Do not enable compression when generating the .rbf file (stick to the default). soc_system.rbf should be 6-7 MBytes.



3.5 Loading the microSD with the image

3.5.1 General

The purpose of this task is to write the downloaded microSD card image file to the device. The image was downloaded as a file named xillinux-1.1-sockit.img.gz (or similar), and is a gzip-compressed image of the microSD card (the soc_system.rbf file needs to be added to it though).

This image should be uncompressed and then written to the microSD 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 the boot image, a raw boot partition and the Linux root file system of ext4 type. Only FAT partitions are detected by almost all Windows computers, so the microSD card will appear to be very small in capacity (47 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 with Linux. The following paragraphs explain how to do this on either operating system.

IMPORTANT:

Writing an image to the microSD 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.

3.5.2 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 microSD card.

Some computers (laptops in particular) have an 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-1.1-sockit.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
```

(Note that the character after ”usbitcmd” is the letter “l” and not the number “1”)

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

```
C:\usbimage>usbitcmd r 2448 \path\to\xillinux-1.1-sockit.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 `\path\to` replaced with the path to where the microSD card's image is stored on your computer.

3.5.3 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 all data in the computer's hard disk. 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 microSD 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-1.1-sockit.img.gz
```

Copying the image to the microSD card is simply:

```
# dd if=xillinux-1.1-sockit.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-1.1-sockit.img /dev/sdc
cmp: EOF on xillinux-1.1-sockit.img
```

Note the response: The fact that EOF was reached on the image file means that everything else compared correctly, and that the flash drive 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 Copying the soc_system.rbf file into the microSD card

Unplug the USB adapter and then connect it back to the computer. This is necessary to make sure that the computer is up to date with the microSD card’s partition table. If necessary, mount the first partition (e.g /dev/sdb1). Most computers will do this automatically.

Then copy the soc_system.rbf (which was generated in paragraph 3.4) to the FAT file system on the microSD card. On Windows systems, this is the only "disk" the system will display. On Linux systems, it's the first (and smaller) partition. Either way, the correct destination is easily recognized by its existing content, which is only two files: 'socfpga.dtb' and 'uImage'.

When done, unmount the microSD card properly, and unplug it from the computer, e.g.

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

If the SoCKit board is already running Xillinux, it's possible to update the soc_system.rbf from the board itself. To mount the FAT filesystem on Xillinux, go

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

and access the file system at /mnt/sd/. Note that if there's a problem with the new soc_system.rbf itself (e.g. it's compressed) or something else is done improperly, the board may not perform boot next time. So by all means, an alternative means for accessing the microSD card must be kept handy.

4 Performing the boot

4.1 Jumper and DIP switch settings

For the board to perform boot from the microSD card, there is probably no need to make any changes with the jumpers, but it's best to make sure that they match the setting shown in the first image on the next page.

Only jumpers J15-J19 are relevant for the boot of the system. The other two jumpers are related to the LCD backlight and HSMC interface voltage level.

The DIP switches (at the board's back side) will usually need to be changed as shown in the second image on the next page.

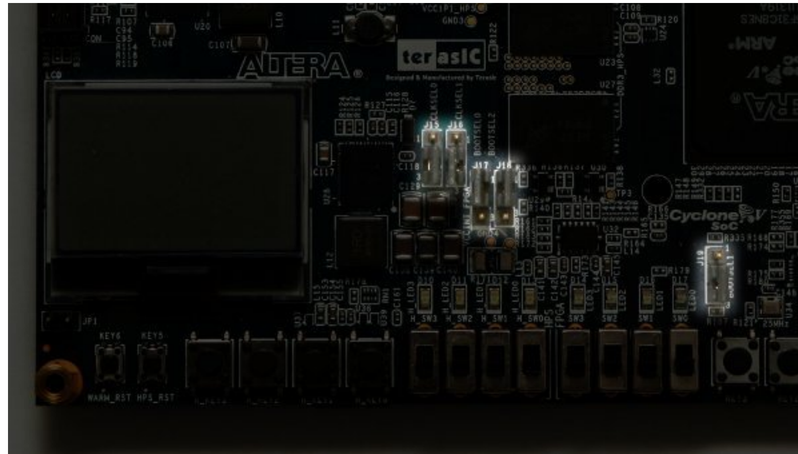
4.2 Attaching peripherals

IMPORTANT:

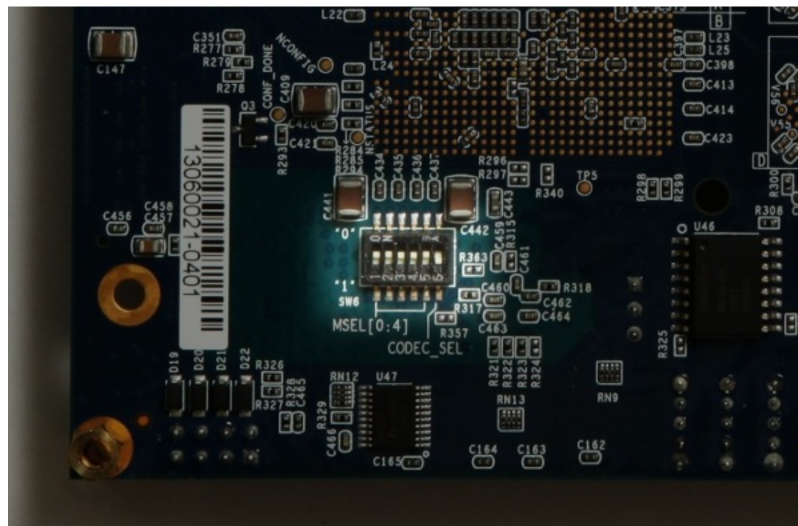
If the USB port is required for use (e.g. attaching a mouse and keyboard), some peripheral must be connected to the OTG USB port when Linux performs boot (even a USB hub with nothing connected to it). This is required for Linux to determine the roles between the processor and what is connected to it, as both can be host on an OTG port.

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

- 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.



Front side of the board, jumper settings highlighted.



Back side of the board, DIP switch highlighted. All switches, except the rightmost, are pushed upwards.

- A mouse and keyboard to the USB OTG connector, through a USB female cable (not included in the SocKit hardware kit). The system will perform boot in the absence of these, and there is no problem connecting and disconnecting the keyboard and mouse as the system runs, as long as some peripheral (or just a USB hub) was connected when Linux performed its boot sequence. The system detects and works with whatever keyboard and mouse it has connected at any given moment.
- 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. Some of the boot messages are sent there, and a shell prompt is issued on this interface when the boot completes.

  This is useful when either a PC monitor or a keyboard is missing or don't work properly.

  Note that if this port is connected to a computer, some terminal software must run on this computer. Otherwise, the boot process may stop while Linux waits for its boot messages to be read. It's fine to perform boot with this port unconnected.

4.3 Powering up the board

This paragraph describes what to expect when powering up the system. In the descriptions below, the board is viewed so that the red power button is at the upper left, and the row of eight LEDs at the bottom, as shown in the images on the previous page.

IMPORTANT:

Xillinux' root file system permanently resides on the microSD card, and is written to while the system is up. The Linux system should therefore be shut down properly before powering off the board to keep the system stable, just like any PC computer, even though a proper recovery is usually observed on sudden power drops.

Plug the microSD card into the SoCKit board, and power it on by pressing the red button. The following sequence is expected:

- The following LEDs turn on immediately:

  - The green LED next to the power button
  - All eight LEDs at the bottom of the board, with weak light
  - The backlight LED of the LCD screen (unless a jumper has been installed to disable this light)

  A few brief flashes are also expected slightly to the right of the power button. These are the UART LEDs. They will continue to flash if a computer is connected to the UART USB port.

- Less than a second after powering on, the four leftmost LEDs turn off, as a sign that the initial bootloader has initialized the processor. If this doesn't happen, the likely causes are either the DIP switches or jumpers are wrong (see section 4.1) or the microSD isn't installed properly, is faulty or improperly loaded with the Xillinux image.
- About 14 seconds after powering on, the other four (rightmost) LEDs go off as well, and the rightmost LED begins flashing at 1 Hz (approximately). A "Xillybus" screensaver appears on the VGA screen for less than a second, and is replaced with a blank screen, with two Linux Penguins logos at the upper left. If this doesn't happen, verify that the soc_system.rbf file is in place and is uncompressed (5-6 MBytes).
- About 26 seconds after powering on, boot text appears on the VGA screen.
- A login prompt should appear no later than 35 seconds after powering on. The system auto-logins as root, presenting a greeting message and a shell prompt. A similar shell prompt is also presented at the USB UART link, mostly for troubleshooting.

If nothing happens after the four leftmost LEDs turn off, it's probably an issue with the soc_system.rbf file. It may be helpful to look at the UART's output, as described in paragraph 4.4.

A short video clip showing the board's powerup can be viewed at

http://youtu.be/mTDaAn4IX3I. Note that the UART's LEDs show little activity in the clip, since there is nothing connected to the UART USB port there.

Type "startx" at shell prompt to launch a Gnome graphical desktop. The desktop takes some 15-30 seconds to initialize.

Notes:

- The root user's password is set to nothing, so logging in as root, if ever needed, doesn't require a password.
- 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 screen 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.

4.4 UART output during boot

If the boot process fails, the UART's output may supply hints on what went wrong. The terminal application on the computer should be configured for 57600 baud, 8 bits, 1 stop bit, no parity and no flow control ("57600 8N1").

This is what is typically seen:

```
U-Boot SPL 2012.10 (Dec 30 2013 - 18:03:34)
SDRAM: Initializing MMR registers
SDRAM: Calibrating PHY
SEQ.C: Preparing to start memory calibration
SEQ.C: CALIBRATION PASSED
DESIGNWARE SD/MMC: 0


U-Boot 2012.10 (Dec 30 2013 - 18:03:46)

CPU   : Altera SOCFPGA Platform
BOARD : Altera SOCFPGA Cyclone 5 Board
DRAM:  1 GiB
MMC:   DESIGNWARE SD/MMC: 0
*** Warning - bad CRC, using default environment

In:    serial
Out:   serial
Err:   serial
```

```
Net:   mii0
Warning: failed to set MAC address
```

The last line counts down a few seconds, and then goes on automatically with reading three files. The byte counts, as well as the displayed kernel version may vary.

If the bootloader reports an error reading any of these files, that's probably the reason for failing to perform boot. The "bad CRC" error indicates that U-boot failed to find a saved set of environment variables, and therefore uses the default values, which is fine.

```
reading uImage

3328400 bytes read
reading socfpga.dtb

15576 bytes read
reading soc_system.rbf

7007184 bytes read
## Booting kernel from Legacy Image at 00007fc0 ...
   Image Name:   Linux-3.8.0-xillinux-1.1
   Image Type:   ARM Linux Kernel Image (uncompressed)
   Data Size:    3328336 Bytes = 3.2 MiB
   Load Address: 00008000
   Entry Point:  00008000
## Flattened Device Tree blob at 00000100
   Booting using the fdt blob at 0x00000100
   XIP Kernel Image ... OK
OK
   Loading Device Tree to 0fff8000, end 0fffecd7 ... OK

Starting kernel ...
```

At this point, the boot loader hands over control to the Linux kernel. Only the beginning of the kernel's boot messages are given below. After the kernel finishes the boot sequence, a shell prompt is given.

If nothing appears after "Starting kernel..." please verify that a blue LED is flashing on the board, which is an indication that the FPGA part was loaded successfully. A failure to load the soc_system.rbf file is the most likely reason for this.

```
Booting Linux on physical CPU 0x0
```

```
Initializing cgroup subsys cpuset
Linux version 3.8.0-00116-gffe44a8 (eli@ocho.localdomain) (gcc version 4.6.3 (S3
CPU: ARMv7 Processor [413fc090] revision 0 (ARMv7), cr=10c5387d
CPU: PIPT / VIPT nonaliasing data cache, VIPT aliasing instruction cache
Machine: Altera SOCFPGA, model: Altera SOCFPGA Cyclone V
Memory policy: ECC disabled, Data cache writealloc
PERCPU: Embedded 8 pages/cpu @80e77000 s10880 r8192 d13696 u32768
Built 1 zonelists in Zone order, mobility grouping on.  Total pages: 260096
Kernel command line: console=ttyS0,57600 root=/dev/mmcblk0p3 rw rootwait
```

4.5 To do soon after the first boot

4.5.1 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 microSD card's full capacity.

IMPORTANT:

There's a significant risk of erasing the entire microSD 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 microSD card initialization (writing the image and soc_system.rbf)

The starting point is typically as follows:

```
# df -h
Filesystem      Size  Used Avail Use% Mounted on
/dev/root       2.3G  1.9G  304M  87% /
devtmpfs        505M  4.0K  505M   1% /dev
none            101M  736K  101M   1% /run
none            5.0M     0  5.0M   0% /run/lock
none            505M     0  505M   0% /run/shm
```

So the root filesystem is 2.3 GB, with 304 MB free.

The first stage is to repartition the microSD card. At shell prompt, type:

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

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

- d [ENTER] – Delete partition
- 3 [ENTER] – Choose partition number 3
- n [ENTER] – Create a new partition
- Press [ENTER] 4 times to accept the defaults: primary partition, number 3, 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 microSD 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): 3

Command (m for help): n
Partition type:
   p   primary (2 primary, 0 extended, 2 free)
   e   extended
Select (default p):
Using default response p
Partition number (1-4, default 3):
Using default value 3
First sector (112455-15523839, default 112455):
Using default value 112455
Last sector, +sectors or +size{K,M,G} (112455-15523839, 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 for the 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.

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 -h now
```

After there's a message saying "System halted." on the UART console (or the cursor stops blinking on the VGA screen), power the board off and on again. The system should perform a boot 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/mmcblk0p3
```

to which the following response is expected:

```
resize2fs 1.42 (29-Nov-2011)
Filesystem at /dev/mmcblk0p3 is mounted on /; on-line resizing required
old_desc_blocks = 1, new_desc_blocks = 1
The filesystem on /dev/mmcblk0p3 is now 1926423 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 microSD card:

```
# df -h
```

```
Filesystem      Size  Used Avail Use% Mounted on
/dev/root       7.1G  1.9G  5.0G  28% /
devtmpfs        505M  4.0K  505M   1% /dev
none            101M  736K  101M   1% /run
none            5.0M     0  5.0M   0% /run/lock
none            505M     0  505M   0% /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 Gigabyte of 10^9 bytes. That's why an 8 GB card appears as 7.1 GiB above.

4.5.2 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 the following command at shell prompt:

```
# passwd
```

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.

4.6 Using the desktop

The Xillinux desktop is just like any Ubuntu desktop. Due to the microSD 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.

4.7 Shutting down

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

```
# shutdown -h now
```

at shell prompt.

When a textual message saying "System Halted" appears on the UART console, it's safe to power the board off. Alternatively, when the cursor stops blinking on the textual console on the VGA screen, that's also a sign that Linux has shut down.

4.8 What to do from here

The SoCKit 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 directly 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

IMPORTANT:

If the FPGA design is built with a QSF file other than the one supplied with the demo bundle, please refer to paragraph 5.3 for more information

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 image 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 handled in paragraph 3.4, and then rebuilt the same way it was done the first place. For a boot of the system with the updated logic, soc_system.rbf needs to be regenerated as described in paragraph 3.4 and written to the microSD card as described in paragraph 3.6. 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.

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 method 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 scenario, 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 streams from the FPGA to host.

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: The guide to defining a custom Xillybus IP core

5.2 Using other boards

Before attempting to run Xillinux on a board other than the SoCKit board, certain modifications may be necessary.

Among others, these involve replacing pins and clocks:

- The attributes of the clock PLL defined in xillybus.v must be set up to match the free-running clock that is connected to clk_bot1, if it's not 50 MHz (and the SDC file updated accordingly).
- The VGA output needs to be matched to the intended board.
- The HPS' multiplexed pins: The ARM core has I/O pins which are routed to physical pins on the chip with a fixed placement. The ARM core is configured in Qsys 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.

The latter issue is important, not only because crucial hardware functions may fail, but illegal physical signal conditions on the board may result in damage to the hardware (even though actual damage is very rare).

Inconsistencies in HPS pin assignments are rectified in three steps:

- Correcting these assignments in Qsys.
- Building the entire project (including FPGA part) and generating an updated boot loader based upon the new handoff files (it's the SPL part of U-boot that sets up the relevant registers on the processor).
- Updating the DTS file to match the device assignments, and performing a compilation of this file, producing the DTB file to perform boot of the system with.

5.3 Custom build projects and preflow.tcl

If the sources of the demo bundle are adopted into another Quartus II project, the common practices for transferring information between QSF files apply. Extra care must be taken to handle a non-standard script, preflow.tcl, which resides in the demo bundle's soc_system subdirectory.

The script's main purpose is to rewire the toplevel module of the SoC infrastructure which was generated by Qsys, in order to bypass buggy AXI bus logic that implements interconnect. Without this rewiring, the data transport over the AXI bus may appear to work properly, but sporadic data corruptions are observed.

The script is executed automatically by Quartus II before compilation takes place. This is a result of the following line in xillydemo.qsf:

```
set_global_assignment -name PRE_FLOW_SCRIPT_FILE \
                  quartus_sh:../soc_system/preflow.tcl
```

In order to make sure that a Xillybus-based FPGA design doesn't accidentally use infrastructure that hasn't been rewired, preflow.tcl also modifies the names of some top-level ports, making them incompatible with the original. The said ports are those named xillybus_0_conduit_*.

It's easy to tell if the module has been modified by the script: The first line of the modified file, soc_system/soc_system/synthesis/soc_system.v contains

```
// soc_system.v (mangled by preflow.tcl)
```

if the file is indeed rewired.

To include the SoC infrastucture in a custom project, adopt the entire SoC module tree by copying the files from an already fully built demo bundle, and add the following line to the QSF file

```
set_global_assignment -name QIP_FILE path/to/soc_system.qip
```

5.4 Changes in the Qsys project

Due to the automatic modification of one of the Verilog files that is generated by Qsys, it's not recommended to modify the Qsys project. There is no guarantee that a modified Qsys project will be fixed correctly by the script, in particular if the project's topology is altered.

Changes to attributes of the elements are probably fine, e.g. modifying the processor's pin multiplexing. Note however that if changes are made to the HPS processor's attributes, they take effect through changes in C files that are included in the preloader, and not through the logic.

6 Troubleshooting

6.1 Port ”xillybus_0_conduit_...” does not exist

During the compilation of the Xillydemo project with Quartus II, error messages like the following may appear:

```
Error (12002): Port "xillybus_0_conduit_M_AXI_ARADDR" does not exist in macrofunction "u0" File: xillybus.v Line: 442
Error (12002): Port "xillybus_0_conduit_M_AXI_ARBURST" does not exist in macrofunction "u0" File: xillybus.v Line: 442
Error (12002): Port "xillybus_0_conduit_M_AXI_ARCACHE" does not exist in macrofunction "u0" File: xillybus.v Line: 442
Error (12002): Port "xillybus_0_conduit_M_AXI_ARID" does not exist in macrofunction "u0" File: xillybus.v Line: 442
Error (12002): Port "xillybus_0_conduit_M_AXI_ARLEN" does not exist in macrofunction "u0" File: xillybus.v Line: 442
Error (12002): Port "xillybus_0_conduit_M_AXI_ARLOCK" does not exist in macrofunction "u0" File: xillybus.v Line: 442
Error (12002): Port "xillybus_0_conduit_M_AXI_ARPROT" does not exist in macrofunction "u0" File: xillybus.v Line: 442
Error (12002): Port "xillybus_0_conduit_M_AXI_ARREADY" does not exist in macrofunction "u0" File: xillybus.v Line: 442
Error (12002): Port "xillybus_0_conduit_M_AXI_ARSIZE" does not exist in macrofunction "u0" File: xillybus.v Line: 442
...
```

The failure to find the ports probably indicates that a script that was supposed to run before compilation didn't run. This is likely a result of a faulty change in the QSF file or an improper adoption of the sources into a custom Quartus II project. For more details, see paragraph 5.3.

6.2 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:

- Was the device connected when Linux performed boot? If not, reboot Linux with the mouse and/or keyboard connected.
- Are you using the correct USB plug? It should be the one marked “HPS USB”, in the middle.

- If a USB hub is used, attempt to connect only a keyboard or mouse directly to the USB cable that goes to the SoCKit board's OTG port.

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. Please refer to SoCKit board's documentation regarding how to set up a UART link.

6.3 File system mount issues

Experience shows that if a proper microSD 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 performs mount read-only, the most likely cause is a low quality microSD card. It's quite typical for such storage to function properly for a while, after which random error messages begin to appear. If /var/log/syslog contains messages such as this one, the (Micro)SD card is most likely the reason:

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

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

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

Even though not directly related, this problem is reported quite frequently when the microSD card is not made by 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 microSD card that generates read errors.

The obvious solution is using a Sandisk microSD card.