OpenCPI
Installation Guide

OpenCPI Release: v2.4.7

Revision History

Revision	Description of Change	Date
1.0	Creation	2014-06-23
1.01	Add all FPGA and embedded system (Zed) content	2014-07-07
1.02	Update details for CentOS6 and Zed	2015-01-31
1.03	Update new simplified ZedBoard Installation	2015-02-27
1.1	Add ML605 details, change to use new/std doc template, bug fixes	2015-06-26
1.2	Add system.xml details	2016-05-18
1.3	Minor update for 2017.Q2	2017-09-07
1,4	Update for 2018.Q3, simplified CentOS and ZedBoard installation	2018-08-27
1.5	Update for 1.5	2019-04-30
1.6	Reorganize simulator chapter, add xsim information, add quick start, AV install	2019-12-31
1.7	Update for 1.7, fix some case in pathnames, add zynq release downloads for licensing	2020-06-16
1.9	Change to use ocpiadmin for install-platform and deploy-opencpi, remove zedboard specific instructions which were redundant	2020-07-19
1.10	Remove CentOS6, re-organize chapter on third-party/vendor installation for Xilinx tools	2020-12-10
1.11	Add glossary section	2021-03-15
1.12	Update e31x and zed to use xilinx19_2_aarch32, bug fixes	2021-05-28
1.13	Remove AV GUI install, replace with OpenCPI GUI install, add picoevb, adrv9361-Z7035 to supported platform tables	2021-11-1
1.14	Update GUI installation procedure to use script and command	2022-1-11
1.15	Add air-gapped installation procedure	2022-01-19
1.16	Update Xilinx installation procedure, correct typo in OpenCPI GUI installation (PyQt)	2022-02-23
1.17	Add server mode instructions, add Avnet OSP, Xilinx OSP to supported platform tables	2022-05-30
1.18	Revise Chapter 7 to be PCIe platform-generic and include concept of runtime host	2022-08-31
1.19	Add new chapter on Ethernet-based platforms, change the term “PCIe card” to “PCIe-based HDL platform” and rename chapter “Setting up PCI Express-based HDL Platforms”	2022-10-19
1.20	Remove references to Ubuntu 16.04 and add Ubuntu 20.04	2022-11-18

Table of Contents

1 Overview

This document describes how to install OpenCPI from downloaded *source code*. The basic installation is for a development "host" that supports development and execution of OpenCPI applications using only software-based components. Additional optional aspects of the installation enable development and execution of FPGA-based components and applications that combine software and FPGA execution, using FPGA simulators. FPGA simulators are considered target processors, which can execute code written for FPGAs. There is also an optional GUI IDE as an alternative to using command line tools.

This document does not describe how to install OpenCPI from packaged binary distributions; for example, from CentOS or RedHat RPM packages. Binary packages have not been produced for the current release but are planned for availability in a future patch release.

To go beyond the basic development host installation and prepare for using OpenCPI on embedded systems, some additional concepts are required:

- OpenCPI ***systems*** consist of one or more processing ***platforms***, thus enabling OpenCPI to develop for a particular embedded ***system*** involves enabling OpenCPI to target each of its ***platforms***. Many systems targeted by OpenCPI have a CPU-based platform for running software components and a different FPGA-based platform for running HDL (Hardware Description Language) components (e.g. VHDL).
- Some ***platforms*** are optional/attachable/pluggable add-ons to a system. They are not considered part of any system, so they are defined distinct from any particular system.
- OpenCPI ***projects*** are source code work areas/directories that contain code developed under OpenCPI. When a ***project*** contains code to enable a particular ***platform*** to be used by OpenCPI, it is called an ***OSP*** (OpenCPI System support Project). When the support for a platform is included in the initial download of OpenCPI, we say support is built-in to OpenCPI. If not, then the OSP that includes such support requires separate downloading.

This document assumes a basic understanding of the Linux command line (or "shell") environment. It does not require a working knowledge of OpenCPI, although if anything goes wrong with the installation, more experience with OpenCPI may be required. For other documentation on OpenCPI, see opencpi.gitlab.io/opencpi. In particular, *after* installation, OpenCPI users should visit the ***OpenCPI User Guide*** to start using it.

Below is a diagram that shows the basic installation steps to install and enable OpenCPI to be used with development hosts as well as embedded systems:

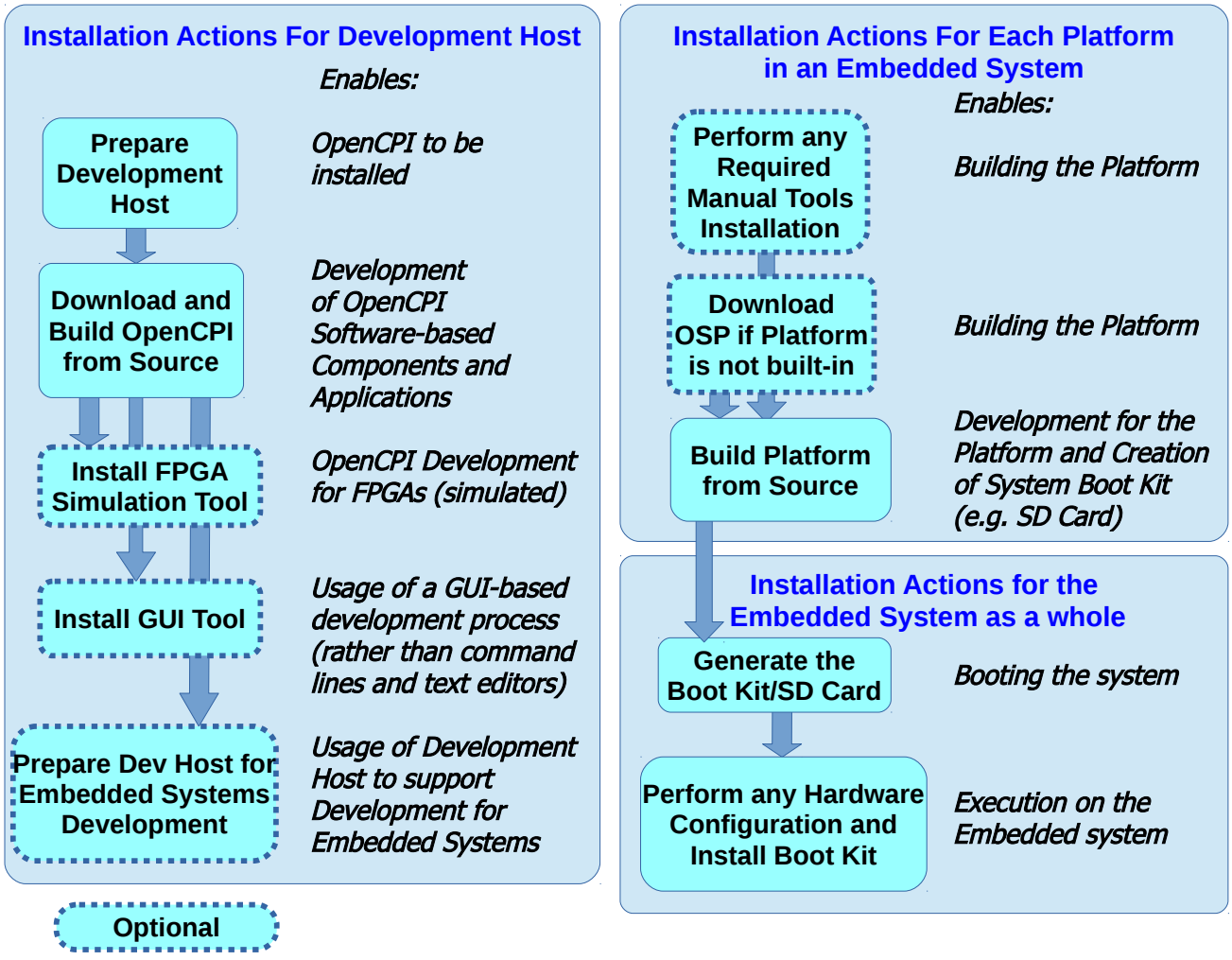


Figure 1: Installation Sequences

The most common installation platform for OpenCPI development hosts is CentOS7 Linux x86_64 (64-bit). Ubuntu18_04 and Ubuntu20_04 are also supported. Other Linux and MacOS variants and 32-bit systems have been used successfully. CentOS7 64-bit is the default, tested installation for a development host. Development hosts can either be actual physical systems or virtual machine installations.

Additional installation options exist for other typically embedded target processors such as the Xilinx® Zynq® SoC (with ARM® processor cores and FPGA resources) and various FPGAs. For embedded CPUs, OpenCPI uses cross-building on the development host using tools that can target the embedded processors and FPGAs.

This document is divided into sections according to the above diagram.

2 Quick Start: Source Code Installation for Software-only Execution

This quick start section describes the procedure to establish an OpenCPI environment for software-only (not FPGA) development and execution on development systems. If any of the following requirements are *not* met, more detailed instructions are in following sections.

The quick start is based on the most recent OpenCPI release, and requires:

- A computer running the CentOS7 Linux or other supported development host operating system that can be used as the target of OpenCPI, with the **`git`** application installed.
- Linux **`sudo`** privileges for the user installing OpenCPI on the development host system.

The download uses **`git`** and thus has full SCM history that is easily updated. Use these commands:

```
sudo yum install -y git
git clone https://gitlab.com/opencpi/opencpi.git
cd opencpi
./scripts/install-opencpi.sh
```

This leaves the git repository checked out on the master branch, which is the most recent release. During the execution of the last command, you will be prompted twice for your password.

This quick start accomplishes a software-only installation of OpenCPI, enabling an OpenCPI user to become familiar with OpenCPI and run non-FPGA tutorials. The next sections describe the process and steps in more detail with more options. If you used this quick start method and want to enable FPGA development (recommended), skip the next section and go to the Installing FPGA Simulators section.

3 Installing OpenCPI on Development Hosts

The basic installation on (Linux) development host systems enables the development and execution of OpenCPI components and applications on the development host itself (called native execution). It also establishes the host platform for the tools that enable development for other target platforms, including embedded CPUs, real or simulated FPGAs, and GPUs. The installation process takes these steps, in three phases:

Phase 1: Hardware and OS installation *(only for installing CentOS from scratch)*

1. Installing the hardware or creating a bare virtual machine.
2. Installing and configuring the operating system from CD or CD image file.
3. Updating the operating system to the latest patch level and enabling "sudo".

   At this this point we have a basic up-to-date OS installation. There are many ways to get here, but we outline a basic approach that works on CentOS7. If your system is already installed, this phase *is skipped*.

Phase 2: Download OpenCPI sources, and install prerequisite software packages.

4. Install the **`git`** software package, and use it to download the OpenCPI source distribution.
5. Install standard required packages using the package update and installation tools of the operating system distribution.
6. Install and build some prerequisite packages that need special OpenCPI-supplied installation scripts.

   At this point, we have installed all prerequisites and have a clone/copy of the OpenCPI code tree *ready to build* on *the* OS installation.

Phase 3: Build and test OpenCPI

7. Build OpenCPI's core tools, libraries, components and example applications.
8. Execute some tests to verify that the installation is working.

These steps result in a development system with tools and runtime software ready to support development and native execution (on the development system) of OpenCPI components and applications. Steps 5 through 8 are done with a single command:

```
./scripts/install-opencpi.sh
```

This installation process is based on source code that is downloaded and built on the development host.

The section ***Preparing the Hardware and OS for OpenCPI Development*** describes Phase 1 in detail. The sections ***Obtaining the OpenCPI Code Base***, ***"Installing Required Standard Software Packages for OpenCPI"*** and ***"Installing Preqrequisite Packages for OpenCPI"*** describe Phase 2. The sections ***"Building the OpenCPI Framework and its Built-in Projects"*** and ***"Testing the Software Aspects of the***

Installation" describe Phase 3 in detail. For other target systems and platforms, see the sections on FPGA simulators, embedded systems and FPGA platforms.

3.1 Preparing the Hardware and OS for OpenCPI Development

> The quick description of this OS installation section is: install *the development host OS, including* the "git" software and *enable* your user *ID* for "sudo".

This step is only necessary if you are installing the OS from scratch. If your OS is already installed and up to date, your account is already **`sudo`**-enabled, and the **`git`** command is available on your system, you can skip to the section Installing OpenCPI from Source.

To just deal with the **`sudo`** and **`git`** issues on an installed OS, skip to: Enabling Your User for **`sudo`** and Installing **`git`**.

Since a development host has no special hardware requirements, it must simply support the recommended and supported development host operating system (e.g. CentOS7 Linux 64-bit). Some development tools (especially those for FPGAs) require large memories and exploit multiple CPU cores and thus the minimum memory should be at least 8GB, with 16GB or more preferred. For test purposes, VMs with 2GB have been successful, but slow.

If the development host system will also host other embedded CPU or FPGA cards acting as OpenCPI target platforms, the appropriate slots, cooling and power supplies should be considered.

If the development host will also be the runtime host for Ethernet-attached devices (such as the Ettus N210), it is sometimes useful to use dedicated Ethernet ports for such devices. In this case, a host system with multiple Ethernet ports/interfaces should be considered. This minimizes interference between Ethernet traffic to the locally-attached platforms, and general LAN or WAN/internet traffic.

OpenCPI development is commonly hosted on laptops, server machines with card slots, and virtual machines hosted on other operating systems. One example system is a CentOS7 64-bit virtual machine running under the "Parallels" virtual machine system on Apple MacBook Pro laptops. Another is a Dell server with well-powered PCI Express slots for hosting a number of FPGA and/or GPU boards.

3.1.1 Obtaining the CD Image File for the OS Installation

The normal operating system installation starts with a CD image downloaded for the development host OS. For CentOS7, this is from **`centos.org`** (or one of its mirrors). An example of a "minimal" CentOS7 installation is described here. Many installation scenarios are possible.

For a physical system, you can burn this CD image file (a.k.a. ISO file) onto a real CD/DVD, and then boot from that physical CD/DVD. For a virtual machine, you can usually designate that the CD/DVD image file be mounted to the virtual machine as a virtual CD/DVD device.

Creating a virtual machine usually involves answering a few questions about the to-be-created VM, and then booting it from the Installation CD/DVD image file. For most VM

systems, the most import questions to answer are the amount of memory to give to the VM, and the number of cores to provide. Each VM system does it slightly differently.

Booting the development host system from the CD or CD image file proceeds the same whether it is a physical system or a VM.

> For the case of using the Parallels VM system on Macs, we select “customize settings before installation”, and set memory at *4*GB and *2* core*s/CPUs*.

The CD image for the minimal CentOS7 installation is available from mirrors at:

http://isoredirect.centos.org/centos/7/isos/x86_64

The actual file name to download is:

CentOS-7-x86_64-Minimal-<*version*>.iso

This installation is suitable for command-line only (no GUI) headless systems, and booting from the CD (image) immediately runs the installer. Other CentOS7 installation images are available with more packages preinstalled on the image.

3.1.2 Booting from and Running the Installation CD/Image

When using CentOS7 with minimal CD image boots, it immediately runs an installer that asks for things like language and root password. It is better not to create users at this point, but to simply run the installer instead. When the installed system boots, it is in command line (shell) mode. At this point, you must at least enable the network interface, if not done during the installer, using the **nmtui** tool (the default network interface is **eth0**), and then update your system using the command:

```
# yum update
```

To ensure that you are up to date, you must reboot (using the **reboot** command) after **yum update**, and then, after the reboot, run **yum update** again, and repeat this process until “No packages marked for update” is displayed.

On the command line you can add users, using the simple **adduser** command. An example would be:

```
# adduser -m -N -r -u 501 -g 20 user1
```

You only need to specify the user and group ids if you are trying to match them to an existing installation for convenient NFS mounting. Otherwise you can simply use:

```
# adduser -m user1
```

This would use default behavior to create the account and home directory.

3.1.3 Enabling Your User for “sudo” and Installing “git”

A number of scripts supplied by OpenCPI require that the user be enabled for the “sudo” command. You should add your user account to the list of accounts that are allowed to use “sudo”. On the CentOS7 minimal installation, you are already at the command line. With <RootPasswd> and <User> being replaced by your root password and your user name, do:

```
% su
Password: <RootPasswd>
# echo <User> ALL = ALL >> /etc/sudoers
# exit
%
```

The "% " and "# " are just command prompts: you don't type them. Be careful to use the two ">>" characters to append the line to the file. Of course if you are experienced with Linux, you may do this many other ways.

OpenCPI has several software prerequisites, but only one of them is required to be installed before the OpenCPI code base is installed: **`git`**. The **`git`** program is used to download a local copy of the code base, and after that, a script in the code base is used to finish the installation of OpenCPI. To obtain **`git`**, the following command should be issued at the command prompt in the terminal window:

```
% sudo yum -y install git
```

This ensures that you have the **`git`** package installed on your system so that you can use the **`git`** command. **`Git`** is the distributed revision control system used by OpenCPI. We will only use a few **`git`** features and commands for installation.

3.1.4 Summary of Installation Steps Prior to Installing OpenCPI

- Download the development host OS CD image file, and check the md5 digest.
- For physical systems, burn a CD from that file.
- Boot the physical or VM system from the CD or CD image file.
- Run the OS installer using the "Install to HardDrive" icon and answer questions. **Note**: be sure to select manual partitioning so that you can allocate most of the disk space to the root (**/**) partition.
- Boot your newly installed system and answer more questions.
- Upgrade the software repeatedly until there are no more updates available.
- Enable your user ID for the **`sudo`** command.
- Install the **`git`** configuration management software.

You now have an installed, up-to-date operating system, with a user account that is sudo-enabled and the **`git`** software configuration management package is available.

3.2 Installing OpenCPI from Source

We assume you are in a shell window, in the directory where the codebase should go, into a subdirectory, which the following command*s* will create.

The source installation procedure downloads, builds and uses the software in a directory of the user's choosing (for example, ~/opencpi). As a result, multiple versions can be downloaded and coexist, but not execute simultaneously.

OpenCPI source installations make no global changes to the user's system other than:

- Installing or updating some required standard packages using the yum install or equivalent command.
- Dynamically/temporarily loading and using the OpenCPI kernel driver in order to test it.

Both these steps require **`sudo`** privileges.

*3.2.1 Download OpenCPI using **`git`***

To download OpenCPI via cloning the entire OpenCPI code base with history, issue the following command:

```
git clone https://gitlab.com/opencpi/opencpi.git
```

The above command creates an **`opencpi`** subdirectory and populates it with the current OpenCPI code base: i.e., a “git clone” of the code base that can be easily updated in the future. Change into the created subdirectory.

```
cd opencpi
```

By default, the **`git clone`** operation downloads the “latest stable” release (called the **`master`** branch). This may or may not be what you want. After downloading, if you want a specific, perhaps earlier release, you use the **`git tag`** command to list the tagged releases available, then set the code base to the one you want, using **`git checkout`**, with the release tag (as listed by **`git tag`**) as an argument.

The OpenCPI source code releases are tagged with the following format:

```
v<major>.<minor>.<patch>
```

For example, a recent release was:

```
v1.5.0
```

Early releases of a next minor release are identified with minor releases starting with “rc” for “release candidate”, e.g.:

```
v.1.5.0-rc.0
```

List the available tagged releases with the command:

```
git tag
```

Check out the tagged release you want with the command:

```
git checkout <release-tag>
```

Whenever you check out a different tag after any building activity, or want to return this tree to its downloaded and checked-out state, you can perform a clean operation on the code base:

```
% make cleaneverything
```

Now you can install, build and test OpenCP.

Summary of Steps to Prepare the OpenCPI Code Base using git

```
% git clone https://gitlab.com/opencpi/opencpi.git
% cd opencpi
% git tag
% git checkout v1.5.0  # use tag from git tag output
```

You now have an OpenCPI source tree configured for a specific tagged version of OpenCPI.

3.2.2 Install, Build and Test OpenCPI

After performing the above steps to obtain the code base (using **`git clone`** and **`cd`** to the **`opencpi`** subdirectory), you are in the top-level directory of the source distribution. At this point, you can use one command to perform the rest of the installation, including running some tests to ensure that the installation is working. This command is:

```
./scripts/install-opencpi.sh
```

This command may take a while to complete and will require you to provide the **`sudo`** password twice for the two global actions described earlier. It is *not* recommended nor supported to perform the whole installation under sudo or as root.

If you are not present to provide the password, it may fail, but it can be rerun.

It will require internet access to download and build software it needs. Occasionally some download sites are unavailable and the command must be repeated when they are available.

The testing done by this script only executes software-based components and applications. After this command completes and succeeds, OpenCPI is ready for use on the development host for OpenCPI software components and applications. At this point you can take the next installation step by Installing an FPGA simulator.

After running the **`install-opencpi.sh`** script:

You now have an OpenCPI installation built and tested, ready to be used. This does not include FPGA tools, code, or bitstreams, or support for embedded systems, which are separate installation steps.

The last step before actually using OpenCPI is to establish the OpenCPI environment using:

```
source <where-is-opencpi>/cdk/opencpi-setup.sh -s
```

A user may wish to put this command in their **~/.bash_profile** (after any **PATH** settings) if every shell/terminal window should automatically have the OpenCPI environment established. The string **<where-is-opencpi>** is the directory where OpenCPI is installed. This is explained further in the ***OpenCPI User Guide***. Since in this case we are actually in the OpenCPI source installation directory, we can simply do:

```
source cdk/opencpi-setup.sh -s
```

3.2.3 Details of the Installation Process for Troubleshooting

The following description is for the curious or when the above **install-opencpi.sh** script fails. If it succeeds, the rest of this section is just a detailed description of what the script does. This script performs these functions, in order, with each depending on the previous ones:

1. Install standard packages required for OpenCPI development, globally on the system, from the OS's repository. This is accomplished using the underlying **install-packages.sh** script described below. Important examples of such required packages are **make** and **python**. For CentOS or Redhat Linux, this script uses the **yum install** command.
2. Build and/or install prerequisite packages for OpenCPI, in an area only used by this installation of OpenCPI (i.e. sandboxed). This will compile these packages as necessary for the development host. The **install-prerequisites.sh** script performs this function and is described below. This usually involves downloading source tarballs for the package and building them.
3. Build the OpenCPI framework (libraries and executables) itself. This step and the following one are accomplished using the **build-opencpi.sh** script described below.
4. Build the built-in projects that are part of the OpenCPI source distribution, for all *software* assets (RCC workers and ACI applications).
5. Run a number of tests to verify the installation. This step uses the **ocpitest** command.

These underlying steps can be run individually for troubleshooting purposes, but are otherwise unnecessary. The rest of this section describes them in more detail, but can be skipped if the **install-opencpi.sh** script succeeds.

3.2.3.1 Installing Required Standard Software Packages for OpenCPI

This step (#1 above) uses the underlying script:

```
scripts/install-packages.sh
```

Most OS distributions are associated with an internet-based repository of software packages, and have a way to install packages from that repository. For Redhat and CentOS Linux systems, the **yum** command is used to access software packages in the repository for that OS. Most embedded/cross-compiled OSs do not have such a repository, but might. When this **yum/rpm** type of package is installed on your system,

it is globally visible and usable and is not removed when the OpenCPI directory you created for the installation is removed. In this way, this script simply adds standard software to your system if it is not already there.

Since different OSs use different package management systems, this script uses the appropriate commands for the OS you are running on and adds the packages required for OpenCPI development.

Since installing packages on to your system typically requires **`sudo`** privileges, a password prompt usually happens when this script is run. Do not run the command using **`sudo`** directly.

3.2.3.2 Installing Prerequisite Packages for OpenCPI

This step (#2 above) uses the script:

`scripts/install-prerequisites.sh [-f]`

We use the term ***prerequisite*** to mean software required by OpenCPI framework software or its built-in projects that must be compiled specifically for OpenCPI for all software platforms whether cross-compiled or not. These are built and used specifically for OpenCPI, in directories inside the OpenCPI installation's file hierarchy and are thus not used nor visible to other software on the system.

> *When this script is used for a cross-compiled platform, it first ensures that the prerequisites are also installed for the development system you are running on. It also checks whether the targeted software platform has any platform-specific prerequisites (such as a cross-compiler not otherwise needed), and builds/installs those before installing the generic prerequisites required for all software platforms.*

This script checks for a timestamp indicating that all the prerequisites for a platform have been built/installed and does not do it again unless the force option (**`-f`**) is specified. It also recognizes when the download for each prerequisite has been done before and does not re-download software, even if the force option (**`-f`**) causes it to be built/installed again.

Prerequisites are usually downloaded from the internet based on URLs for each prerequisite. If your organization does not allow this (or is not connected to the internet), and has a staging server or file share for vetted downloads, the following environment variables can be used to redirect the download process to an alternative local server or file share instead of the server indicated by the internet URL associated with the prerequisite.

`OCPI_PREREQUISITES_LOCAL_SERVER` replaces the host part of the URLs, and assumes that all the prerequisite downloads will be found at that server.

`OCPI_PREREQUISITES_LOCAL_PATHNAME` replaces the host part of the URL with a local directory path.

If you set both, they both will be tried.

3.2.3.3 Building the OpenCPI Framework and its Built-in Projects

This step (#3 and #4 above) uses the script:

```
scripts/build-opencpi.sh
```

This building script will build:

- The core software infrastructure libraries, and utility command executables.
- The OpenCPI Linux kernel/device driver
- The software components in libraries in the built-in projects.
- Some example applications in the built-in projects.

This script first builds the OpenCPI framework executables, libraries and drivers, and then builds the software aspects of the built-in projects for the targeted software platform. It does not do any HDL (FPGA) building in the built-in projects since that depends on what FPGA tools may be available, and is thus done separately after this development host OpenCPI installation.

3.2.3.4 Testing the Software Aspects of the Installation

This step (#5 above) uses the script:

```
scripts/test-opencpi.sh
```

A variety of tests are run. One is to test loading the kernel driver, which requires **`sudo`** privileges and typically prompts for a password. The last message displayed by the test script should be like this:

```
All tests passed: driver os datatype load-drivers container assets
swig python av ocpidev core inactive
```

3.3 Installing FPGA Simulation Platforms

For OpenCPI to be used as a heterogeneous development framework for FPGAs, it is necessary to install one or more FPGA simulators. This is highly recommended and is a requirement for most of the tutorials for learning OpenCPI. OpenCPI considers an FPGA simulator as "just another FPGA platform". This is done separate from any setup necessary for actual FPGA hardware platforms. Setting up an FPGA simulator for use as an OpenCPI platform usually consists of:

1. Installing the simulation software according to the FPGA tool vendor's instructions, which in some cases requires a purchased license.
2. Perhaps setting some simulator-specific environment variables in the **`<where-opencpi-is-installed>/user-env.sh`** file (normally unnecessary).
3. Building the built-in projects for the simulation platform using the script (this can take 10-15 minutes):

   `ocpiadmin install platform <simulation-platform>`

OpenCPI currently supports the following third-party FPGA simulators:

- **`xsim`**, which comes with the Xilinx Vivado® Design Suite HLx Edition tool set. The Xilinx Vivado WebPACK section in this guide provides instructions on how to download and install the free, unlicensed variant of this simulator and its tool set.
- **`modelsim`**, offered by Mentor, a Siemens business. The Modelsim section in this guide provides more information about how to set up this simulator.
- **`isim`**, which comes with the Xilinx ISE® Design Suite tool set. The Isim section in this guide provides more information about how to set up this simulator.

Installing the free (WebPACK) version of **`xsim`** is the recommended first step for all installations. The **`modelsim`** simulator is considerably faster, but also is expensive.

3.4 Installing the OpenCPI GUI (Optional)

OpenCPI provides a GUI development tool called ***OpenCPI GUI*** that you can optionally install from source and use as a GUI alternative to many of the OpenCPI command-based development tools (mostly one called **`ocpidev`**). To install the OpenCPI GUI:

- Install the pre-requisite GUI support toolkit PyQt
- Download the OpenCPI GUI source code base with **`git`**
- Install the OpenCPI GUI from its downloaded location

3.4.1 Installing PyQt

OpenCPI GUI requires the PyQt GUI toolkit package. To install this package on CentOS7 development hosts, issue the command:

```
sudo yum install python36-qt5.x86_64
```

To install this package on Ubuntu development hosts, issue the command:

```
sudo apt install python3-pyqt5
```

Note: in a future release, the PyQt package will be automatically installed when OpenCPI is installed and this step will no longer be necessary.

*3.4.2 Downloading the OpenCPI GUI Source using **`git`***

To download the OpenCPI GUI via cloning the entire code base with history, issue the command:

```
git clone https://gitlab.com/opencpi/ie-gui.git
```

This command creates an **`ie-gui`** subdirectory in the directory where you issue it and populates this subdirectory with the current OpenCPI GUI code base (a “git clone” of the code base that can easily be updated in the future).

3.4.3 Installing the OpenCPI GUI

To install the OpenCPI GUI, change directory to the **`ie-gui`** subdirectory and then run the following command:

```
./install-gui.sh
```

When the installation completes, the OpenCPI GUI can be started with the command:

```
ocpigui[options]
```

See the **`ocpigui(1)`** man page for usage information about the command. See the ***OpenCPI GUI User Guide*** for information about OpenCPI GUI configuration and operation.

Note: before invoking **`ocpigui`**, be sure to establish the OpenCPI environment in the shell/terminal window being used to start the GUI. The command to do so is:

```
source <where-is-opencpi>/cdk/opencpi-setup.sh -s
```

where **`<where-is-opencpi>`** is the directory where OpenCPI is installed.

3.5 Preparing a Development Host to Support Embedded Systems

There are several optional setup tasks on development hosts to prepare using attached embedded systems. The two "attachments" are serial consoles and NFS mounts. Most embedded systems boot from SD cards, and the development host typically needs a drive for writing such cards.

3.5.1 Attaching Serial Port Consoles of Embedded Systems to Development Hosts

When embedded systems have serial consoles, it is common and helpful to attach them to the development systems so that console input and output are in a window on the development system. In the common case where this connection is actually a USB cable that is emulating a serial port, this requires that the development host be prepared to recognize these USB connections as serial ports.

On most systems this recognition is automatic, so when the cable is plugged in, there is a Linux/UNIX **`tty`** device dynamically created in the **`/dev`** directory. Thus a terminal emulator application can be directed to connect this tty device and its window will be a console for the embedded system. Two problems arise from this situation:

- The name of the tty device is arbitrary and unpredictable, and if there is more than one attached embedded system, it is hard to know which is which.
- The permissions of the device are commonly set to read-only.

3.5.1.1 Using udev Rules Files on Linux

Linux has files that help with the auto-recognition process (called ***udev rules files***), which can fix the naming and permission problems as long as there is only one USB-based serial cable with the vendor and part of that embedded system. Each supported OpenCPI embedded system with a USB serial console port and cable provides one of these files that can be symbolically linked into the system directory where such files live, namely:

```
/etc/udev/rules.d
```

So for OpenCPI hardware platform **`<pf>`**, you would issue this command to tell the system how to connect it to the development system:

```
sudo ln -s $OCPI_CDK_DIR/<pf>/udev_rules/*.rules /etc/udev/rules.d
```

Of course this modifies the development host system outside of this OpenCPI installation and could potentially conflict with other installations or software packages. These **`udev`** rules files that are specific to a given platform usually result in the **`tty`** device with the name:

```
/dev/tty<pf>_0
```

that can be provided to the terminal emulator application.

The **`udev`** rules files are typically examined by the system at boot time, and if the symbolic links created above point to file systems that are not mounted early in the boot

process, you may need to tell the system to re-examine the **udev** rules files later. This is done using the command:

```
sudo udevadm control --reload
```

3.5.1.2 Using Console tty Ports without udev Rules

If there is no **udev** rules file for a platform or there are multiple platforms of the same type with console serial cables, the **tty** device can be determined manually by looking in the **/dev** directory and seeing which devices appear when the cable is plugged in. Then to fix the permission problem you need to issue the following command every time the cable is plugged in or the embedded system is reset or the development host is reset:

```
sudo chmod 666 /dev/<new-tty-device-appearing-when-cable-plugged-in>
```

3.5.1.3 Using a Terminal Emulator Program to Access the Console Port

The most common terminal emulator application on Linux systems is **screen**, where the command to run that application with a window connected to that serial port would be:

```
screen /dev/<tty-device-name> <baudrate>
```

Where **<baudrate>** is the speed of the serial port.

For Emacs users, you can use the serial terminal emulator built in to Emacs, by simply doing: **M-x serial-term** in a window, and providing the **/dev/tty<xxx>** name and baud rate when prompted. There are two modes in this window, ***char*** and ***line***. The default is ***char*** mode, where every character typed is immediately sent without any interpretation at all. This means no Emacs keyboard commands work in that Emacs window and you have to use the mouse to switch to another Emacs window. The ***line*** mode is more line buffered, like Emacs shell mode, and all line editing and other Emacs commands work fine. But the display can get confused in this line mode so you have sometimes switch back and forth: **control-c control-j** switches to ***line*** mode, **control-c control-k** switches to ***char*** mode, and **control-c control-c** sends a control C in ***char*** mode.

With a good terminal emulator connection, hit return a few times in the serial console window to see the console prompt, or, if the system was previously logged in and running, you might already see the prompt.

Sometimes if the screen or terminal emulator is confused, or if, after hitting "return", the prompt stays on the same line, try typing the command "clear", to clear the state of the terminal emulator.

3.5.2 Enabling the Development Host to be an NFS Server for the Embedded System

It is frequently useful to be able to mount and access the file systems on the development system where the OpenCPI codebase is built and cross-built. This\ requires that the development system be enabled as a file server, and any associated firewall issues are addressed between the embedded system and the development

host. This configuration typically uses **nfs**, with the embedded system acting as an **nfs** client and the development system acting as the **nfs** server.

The NFS server needs to be enabled on the development host in order to run the embedded systems in Network Mode. If Network Mode is not being used, it is not necessary. Directions on how to do this for CentOS7 are described here. Many development systems are already enabled as NFS servers, but not all.

From the host, allow NFS past SELinux[1]:

```
% sudo setsebool -P nfs_export_all_rw 1
% sudo setsebool -P use_nfs_home_dirs 1
```

From the host, allow NFS past the firewall:

```
% sudo firewall-cmd --permanent --zone=public --add-service=nfs
% sudo firewall-cmd --permanent --zone=public --add-port=2049/udp
% sudo firewall-cmd --permanent --zone=public --add-service=mountd
% sudo firewall-cmd --permanent --zone=public --add-service=rpc-bind
% sudo firewall-cmd --reload
```

Define the export(s) by creating a new file in the **/etc/exports.d** directory,

```
/etc/exports.d/user_ocpi.exports
```

It must have the extension **.exports**, otherwise, it will be ignored.

Add the following line to that file and replace **XX.XX.XX.XX/MM** with a valid netmask for the DHCP range that the SDR will be set to for your network, e.g. **192.168.0.0/16**. This should be as "tight" as possible for security reasons. **Do *not* share out your top-level directory! This would allow theft of your private ssh keys, etc!**

```
<opencpi> XX.XX.XX.XX/MM(rw,sync,no_root_squash,crossmnt)
```

Where **<opencpi>** is your OpenCPI installation (git repo). Any other projects or files outside that installation would require additional, separate exported mount points.

If the file system that you are mounting is XFS, then each mount needs to have a unique **fsid** defined. Instead, use:

```
<opencpi> XX.XX.XX.XX/MM(rw,sync,no_root_squash,crossmnt,fsid=33)
```

After creating this file, run these commands to have the changes take effect:

```
% sudo systemctl enable rpcbind
% sudo systemctl enable nfs-server
% sudo systemctl enable nfs-lock
% sudo systemctl enable nfs-idmap
% sudo systemctl restart rpcbind
% sudo systemctl restart nfs-server
% sudo systemctl restart nfs-lock
% sudo systemctl restart nfs-idmap
```

1 You can use **getsebool** to see if these values are already set before attempting to set them. Some security tools may interpret the change attempt as a system attack.

Some of the **`enable`** commands may fail based on your system configuration, but usually does not cause any problems.

3.5.3 Using SD Card Reader/Writer Devices

SD card reader/writer devices are required when the embedded systems use such cards and it is required to make changes to them. Reading an SD card is sometimes necessary to "patch" it to add new files while leaving some of its contents and formatting untouched. Most such devices are pluggable USB devices.

The first step when using an SD card is to discover its device name, which is usually something like **`/dev/sdb`** or **`/dev/mmcblk0`**. After inserting a card, the following command usually has output that identifies the device name:

```
dmesg | tail -n 15
```

To make a backup copy of a card, into a file called **`backup.image`**, use this command:

```
dd if=<device-name> of=backup.image
```

To write the image file back onto an SD card, use:

```
dd of=<device-name> if=backup.image
```

To actually use or change the contents of file systems on an SD card, it must be mounted such that its partitions are accessible as file systems. In most cases the file systems on an SD card are automatically mounted upon insertion, and the mount point (root directory of the partition's file system) can be discovered using the **`df -h`** command that lists mounted file systems and their capacities.

If the desired partition is not automatically mounted, you must 1) discover the device's name (as above), 2) the partition's name, 3) possibly create a mount point, and 4) mount it. Search the internet for "how to mount an SD card" for many approaches to this on many systems.

Information about preparing SD cards for enabling embedded systems for OpenCPI is in the SD Card Preparation section.

4 Supported Systems and Platforms

This section provides a series of tables that contain information about the development systems, FPGA simulators, embedded systems and platforms supported by OpenCPI. It is intended as a guide to the supported software and hardware configurations and their requirements and dependencies.

Below is a list of supported development system types, with the corresponding OpenCPI ***software platform*** and URLs for related information and download. All supported development platforms are in the **`ocpi.core`** project built in to OpenCPI. Other unsupported or not-yet-supported development system types can be in external projects. The default supported development platform is **`centos7`**.

Table of Supported Development Systems (Hosts)

Description	OpenCPI Software Platform	Limitations	Vendor Links
CentOS7 Linux on x86-64 bit	**`centos7`**		CentOS7 general information link CentOS7 download link
Rocky 8 Linux on x86-64 bit	**`rocky8`**		Rocky 8 general information link Rocky 8 download link
Ubuntu 18.04 Linux on x86-64 bit	**`ubuntu18_04`**		Ubuntu 18.04 LTS general information link Ubuntu 18.04 LTS download link
Ubuntu 20.04 Linux on x86-64 bit	**`ubuntu20_04`**		Ubuntu 20.04 LTS general information link Ubuntu 20.04 LTS download link
MacOS Catalina 10.15	**`macos10_15`**	No FPGA tools or drivers	MacOS Catalina general information link MacOS Catalina download link

Below is the list of supported FPGA simulators and the corresponding OpenCPI ***"hardware" (HDL) platforms***. When the associated tools are installed on a development system, the development system then includes these platforms in addition to the software platform. These simulators enable FPGA development without actual FPGA hardware. The default and recommended FPGA simulator is **`xsim`**, which is part of the free (WebPACK) version of the Xilinx Vivado tools package.

Table of Supported FPGA Simulators

Description	OpenCPI Platform	Vendor Links
Xilinx Vivado XSIM 2017.1-2021.2	**`xsim`**	Vivado® Simulator (XSIM) is part of all Vivado HL Editions, including the free WebPack edition. Vivado Simulator general information link Vivado Simulator download link
Xilinx ISE ISim 14.7	**`isim`**	ISE® Simulator (ISim) is part of the older Xilinx ISE tool set. ISE/ISim general information link ISE/ISim download link
Siemens/Mentor Modelsim 10.4c-10.6c	**`modelsim`**	Modelsim® is powerful, fast and expensive. Modelsim general information link No download link available before purchase.

On the next page is a table of supported embedded systems, which includes the platforms each system consists of. Embedded systems generally have a software processor (CPU) using the "primary" OpenCPI software platform, along with other, usually hardware/FPGA, platforms. A physical processor (CPU) or processing device (FPGA) may be supported by a variety of OpenCPI platforms (versions or OSes).

Table of Supported Embedded Systems

Description	OSP Repository	Primary Software Platform(s)	Other Platforms in the system	Vendor Links
Epiq Solutions Matchstiq-Z1 radio	Part of OpenCPI	`xilinx13_3`	`matchstiq_z1`	epiqsolutions.com/matchstiq
Avnet® ZedBoard	Part of OpenCPI	`xilinx19_2_aarch32`	Using Vivado: `zed` Using ISE: `zed_ise`	zedboard.org
Ettus E310 USRP™	gitlab.com/opencpi/osp/ocpi.osp.e3xx	`xilinx19_2_aarch32`	`e31x`	www.ettus.com/all-products/e310
Analog Devices™ ADALM-PLUTO	gitlab.com/opencpi/osp/ocpi.osp.pluto	`adi_plutosdr0_32`	`plutosdr`	wiki.analog.com/university/tools/pluto
Analog Devices ADRV9361-Z7035	gitlab.com/opencpi/osp/ocpi.osp.analog	`xilinx19_2_aarch32`	`adrv9361`	www.analog.com/en/design-center/evaluation-hardware-and-software/evaluation-boards-kits/adrv9361-z7035
Avnet MicroZed™	gitlab.com/opencpi/osp/ocpi.osp.avnet	`xilinx19_2_aarch32`	`microzed_10_cc` `microzed_20_cc`	microzed.org
Avnet PicoZed™	gitlab.com/opencpi/osp/ocpi.osp.avnet	`xilinx19_2_aarch32`	`picozed_10_cc` `picozed_15_cc` `picozed_20_cc` `picozed_30_cc`	picozed.org
Xilinx® Zynq UltraScale+™ ZCU102	gitlab.com/opencpi/osp/ocpi.osp.xilinx	`xilinx19_2_aarch64`	`zcu102`	www.xilinx.com/products/boards-and-kits/ek-u1-zcu102-g.html
Xilinx Zynq UltraScale+ ZCU111	gitlab.com/opencpi/osp/ocpi.osp.xilinx	`xilinx19_2_aarch64`	`zcu111`	www.xilinx.com/products/boards-and-kits/ek-u1-zcu111-g.html

The following table lists all supported platforms (software and hardware) in the current release. Every system running OpenCPI contains one or more platforms that support

different components of an application running on different platforms in the system. Some platforms may be optionally attached/inserted/plugged into a system, such as PCI-Express plug-in cards with FPGA or CPUs on them.

Each entry in the table specifies the platform name, the OpenCPI project that supports it, and any vendor tool installations that the platform requires. The chapter “Installing Third-party/Vendor Tools” in the ***OpenCPI Installation Guide*** provides download and installation information for most of these vendor tools.

Table of Supported Platforms

OpenCPI Platform Name	Description	OpenCPI Project/Repo	Dependencies required before building/installing the platform
Development Host Platforms			
centos7	CentOS7 on x86_64 CPUs	**ocpi.core** built-in	A CentOS7 installation
rocky8	Rocky 8 on x86_64 CPUs	**ocpi.core** built-in	A Rocky 8 installation
ubuntu18_04	Ubuntu 18.04 on x86_64 CPUs	**ocpi.core** built-in	An Ubuntu 18.04 installation
ubuntu20_04	Ubuntu 20.04 on x86_64 CPUs	**ocpi.core** built-in	An Ubuntu 20.04 installation
macos10_15	MacOS Catalina	**ocpi.core** built-in	A MacOS Catalina installation
Embedded Software Platforms			
xilinx13_3	Xilinx Linux 14.7 from 2013 Q3	**ocpi.core** built-in	ISE® EDK 14.7 or Vivado® SDK 2013.4 Xilinx Linux tag: **xilinx-v14.7**
xilinx13_4	Xilinx Linux 14.7 from 2013 Q4	**ocpi.core** built-in	ISE® EDK 14.7 or Vivado® SDK 2013.4 Xilinx Linux tag: **xilinx-v2013.4**
xilinx19_2_ aarch32	Xilinx Linux from 2019Q2 (2019.2) For Zynq-7000	**ocpi.core** built-in	Xilinx Vitis™ SDK 2019.2 Xilinx Binary Zynq Release 2019.2 Xilinx Linux git clone Xilinx Linux tag: **xilinx_v2019.2**
xilinx19_2_ aarch64	Xilinx Linux from 2019Q2 (2019.2) For Zynq-Ultra	**ocpi.core** built-in	Xilinx Vitis SDK 2019.2 Xilinx Binary Release 2019.2 Xilinx Linux git clone Xilinx Linux tag: **xilinx_v2019.2**
xilinx21_2_ aarch32	Xilinx Linux from 2021Q2 (2021.2) For Zynq-7000	**ocpi.core** built-in	Xilinx Vitis SDK 2021.2 Xilinx Binary Zynq Release 2021.2 Xilinx Linux git clone Xilinx Linux tag: **xilinx_v2021.2**
xilinx21_2_ aarch64	Xilinx Linux from 2021Q2 (2021.2) For Zynq-Ultra	**ocpi.core** built-in	Xilinx Vitis SDK 2021.2 Xilinx Binary Release 2021.2 Xilinx Linux git clone Xilinx Linux tag: **xilinx_v2021.2**
adi_plutosdr0 _32	Analog Devices Pluto Linux 0.32	**ocpi.osp. plutosdr**	Vivado SDK 2019.2

OpenCPI Platform Name	Description	OpenCPI Project/Repo	Dependencies required before building/installing the platform
FPGA/HDL Platforms			
`zed`	ZedBoard Zynq FPGA w/Vivado	**`ocpi.platform`** built-in	Vivado WebPACK
`zed_ise`	ZedBoard Zynq FPGA w/ISE	**`ocpi.platform`** built-in	ISE 14.7 WebPACK
`zed_ether`	ZedBoard Zynq FPGA/PL w/dual 1 GbE	**`ocpi.platform`** built-in	Vivado WebPACK
`matchstiq_z1`	Epic Solutions Zynq FPGA/PL	**`ocpi.assets`** built-in	Vivado WebPACK
`e31x`	Ettus E31x Zynq FPGA/PL	**`ocpi.osp.e3xx`**	Vivado WebPACK
`ml605`	Xilinx PCI-Express card w/Virtex6	**`ocpi.assets`** built-in	ISE 14.7
`alst4`, `alst4x`	Altera PCI-Express card w/Stratix4 gx230, gx530	**`ocpi.assets`** built-in	Quartus12.1+ gx230: User Guide, Ref Manual gx530: User Guide, Ref Manual
`plutosdr`	Analog Devices ADALM-PLUTO	**`ocpi.osp.plutosdr`**	Vivado WebPACK
`zcu104`	Xilinx Zynq-UltraScale+ Dev Bd	**`ocpi.platform`** built-in	Vivado WebPACK
`zcu106`	Xilinx Zynq-UltraScale+ Dev Bd	**`ocpi.platform`** built-in	Vivado WebPACK
`picoevb`	RHS Research PicoEVB	**`ocpi.platform`** built-in	Vivado WebPACK
`adrv9361`	Analog ADRV9361 Zynq FPGA/PL	**`ocpi.osp.analog`**	Vivado licensed
`microzed_10_cc` **`microzed_20_cc`**	Avnet MicroZed Zynq FPGA/PL	**`ocpi.osp.avnet`**	Vivado WebPACK
`picozed_10_cc` **`picozed_15_cc`** **`picozed_20_cc`** **`picozed_30_cc`**	Avnet PicoZed Zynq FPGA/PL	**`ocpi.osp.avnet`**	Vivado WebPACK
`zcu102`	Xilinx Zynq UltraScale+ MPSoC	**`ocpi.osp.xilinx`**	Vivado licensed
`zcu111`	Xilinx Zynq UltraScale+ RFSoC	**`ocpi.osp.xilinx`**	Vivado licensed

OpenCPI Platform Name	Description	OpenCPI Project/Repo	Dependencies required before building/installing the platform
`x310`	Ettus X310 Xilinx Kintex-7 FPGA w/dual 1/10 GbE	`ocpi.osp.ettus`	Vivado licensed

Some of the platforms listed in the table above require system-specific setup procedures to be performed in addition to the general procedures described in the ***OpenCPI Installation Guide***. The following table lists these systems and provides links to the corresponding setup documents for these systems.

Table of Platforms with System-Specific Setup Requirements

OpenCPI Platform Name	OpenCPI System-Specific Setup Guide
4.1.1 ***`alst4, alst4x`***	*4.1.2* ***Alst4 Getting Started Guide***
4.1.3 ***`e31x`***	*4.1.4* ***Ettus E31x Getting Started Guide***
4.1.5 ***`matchstiq_z1`***	*4.1.6* ***Matchstiq Z1 Getting Started Guide***
`adrv9361`	***Analog Devices ADRV9361-Z7035 Getting Started Guide***
`alst4, alst4x`	***Alst4 Getting Started Guide***
`e31x`	***Ettus E31x Getting Started Guide***
`matchstiq_z1`	***Matchstiq Z1 Getting Started Guide***
`microzed_10_cc` `microzed_20_cc`	***Avnet MicroZed 7Z010 Getting Started Guide*** ***Avnet MicroZed 7Z020 Getting Started Guide***
`ml605`	***ML605 Getting Started Guide***
`picoevb`	***RHS Research PicoEVB Getting Started Guide***

`picozed_10_cc`
`picozed_15_cc`
`picozed_20_cc`
`picozed_30_cc`
	Avnet PicoZed 7Z010 Getting Started Guide
	Avnet PicoZed 7Z015 Getting Started Guide
	Avnet PicoZed 7Z020 Getting Started Guide
	Avnet PicoZed 7Z030 Getting Started Guide

`plutosdr`	***PlutoSDR Getting Started Guide***
`x310`	***Ettus X310 Getting Started Guide***
`zcu102`	***Xilinx ZCU102 Getting Started Guide***
`zcu111`	***Xilinx ZCU111 Getting Started Guide***
`zed, zed_ise`	***ZedBoard Getting Started Guide***

OpenCPI Platform Name	OpenCPI System-Specific Setup Guide
`zed_ether`	*ZedBoard Ethernet Getting Started Guide*

5 Enabling OpenCPI Development for Embedded Systems

OpenCPI uses the term ***embedded systems*** for processors and systems that will execute OpenCPI components and applications, but are not used to build or compile OpenCPI or components (i.e. *not* development hosts). The primary example is the Avnet (Digilent) ZedBoard™, which has a Xilinx Zynq SoC chip that contains two ARM CPU cores for software and an FPGA section for "gateware". Embedded systems always have a processor to run the OpenCPI software runtime and usually have FPGAs, too.

An OpenCPI ***system*** is a collection of processing elements that can be used together. We call each available processor and its surrounding directly-connected hardware a ***platform***. Platforms come in different types; the primary ones are ***software*** platforms which usually run C++ and Linux, and ***FPGA*** platforms which run VHDL or Verilog code.

For the purposes of installation of OpenOPI for an embedded ***system***, each ***platform*** is considered separately. If a system is supported by OpenCPI, it means that each of its platforms is supported by OpenCPI. Once you know which platforms are present in the system, you perform the installation tasks related to each platform, and then any final installation tasks for the system as a whole (which is usually minor).

To use the ZedBoard as an example, its dual-ARM-core CPU can run many flavors and versions of Linux, and each is considered a different software ***platform***. So when we set up a ZedBoard, we first select one of several possible software platforms. The default one is called **`xilinx19_2_aarch32`** and is based on the Xilinx distribution of Linux released in February 2019. Similarly, an FPGA platform might have different tool sets and versions that are used to build executable binaries for it, and each would be considered an FPGA ***platform***. The default FPGA platform for the ZedBoard is called **`zed`**, and is defined to use the Xilinx Vivado tool set for building binaries for it. There is an alternative FPGA platform for the ZedBoard, called **`zed_ise`**, which is defined to use the older Xilinx ISE tool set.

So, installation for the ZedBoard system consists of steps for the software platform (e.g. **`xilinx19_2_aarch32`**), steps for the FPGA platform (e.g. **`zed_ise`**), and finally steps for the ZedBoard system as a whole.

5.1 Installation Steps for Platforms

The support for every OpenCPI platform is contained in an OpenCPI ***project***. The support for some OpenCPI platforms, like the **`zcu104`**, is contained in the built-in projects (in this case, **`ocpi.platform`**). The support for other OpenCPI platforms, like the **`e31x`**, is contained in projects called ***OSPs*** (***OpenCPI System support Projects***; in this case, **`ocpi.osp.3xx`**). The Table of Supported Platforms lists the project that each platform requires.

In a project, a platform resides in its own directory **`<name>`** (which is usually the lowercase version of the name used by the platform's vendor) under a **`platforms/`** subdirectory in the project's **`hdl/`** or **`rcc/`** directory. For example:

```
../projects/platform/hdl/platforms/zcu104
../projects/core/rcc/platforms/xilinx19_2_aarch32
../projects/osps/ocpi.osp.e3xx/hdl/platforms/e31x
```

The **`README`** file that is usually supplied in a platform's directory provides more specific information about it.

Installing (enabling development for) OpenCPI platforms requires three steps:

1. Manually install the required tools if they are needed (as stated in the Table of Supported Platforms) and have not been installed already for another platform. Instructions for installing these required tools are found in the section Installing Third-party/Vendor Tools.
2. Download the OpenCPI project repository (if not built-in) that contains support for the platform (per the Table of Supported Platforms) and which has not already been downloaded for another platform.
3. Build the required platform support from the built-in projects and the platform's OSP (if any) for the platform.

Step 1 can enable cross compilers for software platforms, or simulators and synthesis tools for FPGA platforms.

Steps 2 and 3 are performed by running the OpenCPI command:

```
ocpiadmin install platform <platform> [<options>]
```

This command downloads the OSP (use command options to specify its location; see the **`ocpiadmin(1)`** man page for details) if the platform support isn't in the built-in projects and builds the platform. *This command must be used on* both software and FPGA platforms.

The command places downloaded projects in the **`projects/osps`** subdirectory of the OpenCPI installation directory. For hardware (FPGA) platforms, assembly builds are suppressed except for a single test assembly to verify the installation and any assemblies in the OSP itself.

5.2 Installation Steps for Systems after their Platforms are Installed

There are three modes of operation for OpenCPI on embedded systems and the setup tasks depend on which modes will be used. The modes are:

Server Mode: the embedded system acts as an SSH server which does not rely on any local/embedded media or NFS client/server configuration. It is loaded by the development system pushing files to it via SSH which are only stored in non-volatile memory. Applications are launched on the development hosts, and the embedded system acts as a server for execution of OpenCPI components on its platforms ("containers" at runtime).

Network Mode: a development system hosts the OpenCPI installation as an NFS server to the embedded system as an NFS client. This configuration provides quick and dynamic access to all of OpenCPI, and presumably any applications, components and bitstreams. Changes and rebuilds on the development host are instantly visible to the embedded system with no need to change or reload files on the embedded system (e.g. SD card). The downside is the complexity of setting up NFS servers, mounts, firewalls etc.

Standalone Mode: the embedded system relies totally on the OpenCPI files on the media embedded or plugged into it, but has no reliance on the network. Any changes during development must be copied to the media in the embedded system, which then usually requires rebooting.

Server mode requires no OpenCPI files on non-volatile media, and no NFS software or configurations, but does depend on the presence of a network connection to a development system to load and launch applications. The bootable SD card in the system (the Linux kernel that boots) must provide an SSH server and nothing else. Some embedded systems still require an OpenCPI-enhanced boot environment on the SD card if the default boot configuration does not provide for SSH networking. This mode is best and simplest for automated testing of applications and components on the embedded system in a lab environment.

Network mode requires a network connection, the development host configured as an NFS server (with firewalls configured appropriately), and the embedded system configured as an NFS client, with OpenCPI loaded on the its non-volatile media.

Standalone mode requires no network connection and relies on OpenCPI files being loaded onto non-volatile media (e.g. the bootable SD card). Thus the media must be loaded onto the development system and then physically transferred to (plugged into) the embedded system. No NFS software or configurations are required, but any changes involve plugging and unplugging the non-volatile media, updating it on the development host, and then physically transferring onto the embedded system.

All three modes have implications on how SD cards are prepared by OpenCPI for the embedded systems. In some cases, the SD card that comes from the manufacturer of the embedded system is usable unmodified, for server mode. The OpenCPI tools that create SD cards currently create contents that support all modes.

5.2.1 Preparing the SD Card Contents

The SD card contents for OpenCPI is a combination of files required by the software platform and the hardware platform. Essentially the software platform of the system establishes a baseline of SD card content, and then the SD card is further specialized for the hardware platform. For example, the **xilinx19_2_aarch_32** software platform (Linux kernel, compiler, libraries), is usable for the Avnet ZedBoard system and the Ettus E310 system. So when that platform is used for either of these systems, the SD card contents are first established by that platform. Then the contents are further specialized for the actual hardware platform (**zed**, **e31x**).

A single command is used to create the SD card contents for a given embedded system, and the command is supplied with the two platforms that make up the system.

```
ocpiadmin deploy platform <rcc-platform> <hdl-platform>
```

The command is run after the dependencies for both OpenCPI platforms are installed and the OpenCPI platforms are installed and built with the **ocpiadmin install platform <platform>** command. It results in a directory **cdk/<*hdl*-platform>/sdcard-<*rcc*-platform>**, whose contents can be copied to an SD card. In the example of the Ettus E310, a system using the **xilinx19_2_aarch32** software platform and the **e31x** hardware platform, the SD card contents would be in the directory:

```
cdk/e31x/sdcard-xilinx19_2_aarch32
```

In general, when creating a new SD card for an embedded system, you start with the card that came from the manufacturer, make a raw copy of it, mount it, remove files, and perform the copy operation below. This preserves whatever formatting and partitioning that is correct for the system. Sometimes it is necessary to format the card in some way specific to that system (e.g. if the cards are not the same size). Any variations of this method will be explained in the documentation or README or Getting Started Guide for that system. Making a raw copy is mentioned in the section on preparing a development host for reading and writing SD cards here.

The SD card contents created by OpenCPI using the **ocpiadmin deploy platform** command above includes the files necessary for any of the modes described above (network, standalone, server). Standalone mode requires the most files.

Some customization of setup scripts on the SD card may be required for standalone or network modes, as described in the SD Card Startup Scripts section.

5.2.2 Writing the SD Card

To write the prepared contents onto the physical SD card, it must be mounted as a file system on the development system. Some SD cards are partitioned into multiple partitions, but the prepared OpenCPI SD card contents are intended to be copied to the *boot* partition. Thus you must determine the mount point of the boot partition (or only partition) of the SD card. Use the **df -h** command to find where the SD card (in particular, its boot partition) is mounted. A common mount point is **/media/BOOT**.

If the boot partition of the SD card is not mounted (in unusual cases where it is not automatically mounted upon insertion), you must mount it. See the SD Card Reader/Writer section for this.

To actually write the SD card, you would do, e.g. with the SD card mounted as **/media/xyz**:

```
% rm -r -f /media/xyz/*
% cp -RLp cdk/e31x/sdcard-xilinx19_2_aarch32/* /media/xyz
% umount /media/xyz
```

At this point, the SD card can be removed and plugged into the embedded system (while powered off).

5.2.3 SD Card OpenCPI Startup Script Setup

For network and standalone modes, a startup script is used to configure the environment of the embedded system. The OpenCPI framework provides a default script for each mode. The default scripts are to be copied and modified per the user's requirements. The use of **<mount>** below indicates where the boot partition of the SD card is mounted on the development system.

Note: Server mode does *not* require an OpenCPI startup script.

5.2.3.1 Network Mode Script Setup

1. Make a copy of the default script for editing/customizing.

```
$ cp /run/media/<mount>/opencpi/default_mynetsetup.sh \
     /run/media/<mount>/opencpi/mynetsetup.sh
```

2. Edit the copy in **mynetsetup.sh**

- In **mynetsetup.sh**, uncomment the following lines which are necessary for mounting the **core** and **assets** built-in projects. Note the $1 will be filled in with the network address of the development host.

```
mkdir -p /mnt/ocpi_core
mount -t nfs -o udp,nolock,soft,intr \
      $1:/home/user/ocpi_projects/core /mnt/ocpi_core
mkdir -p /mnt/ocpi_assets
mount -t nfs -o udp,nolock,soft,intr \
      $1:/home/user/ocpi_projects/assets /mnt/ocpi_assets
```

3. Change the pathnames **/home/user/*ocpi_projects*/core** and **/home/user/*ocpi_projects*/assets** to reflect the paths to the **core** and **assets** projects on the development host, e.g. resulting in:

```
mkdir -p /mnt/ocpi_core
mount -t nfs -o udp,nolock,soft,intr \
      $1:/home/johndoe/opencpi/projects/core /mnt/ocpi_core
mkdir -p /mnt/ocpi_assets
mount -t nfs -o udp,nolock,soft,intr \
      $1:/home/johndoe/opencpi/projects/assets /mnt/ocpi_assets
```

5.2.3.2 Standalone Mode Script Setup

In this mode, all OpenCPI artifacts that are required to run any application on the system are copied onto the SD card. Once the artifacts have been created, they must be copied to the SD card. In general, any required **`.so`** (RCC workers), **`.bit.gz`** (HDL assemblies), and application XMLs or executables must be copied to the SD card.

1. Make a copy of the default script for editing

```
cp /run/media/<mount>/opencpi/default_mysetup.sh \
   /run/media/<mount>/opencpi/mysetup.sh
```

2. Unlike network mode, there are no required modifications to this script, but some customizations could be made.
3. Copy any additional artifacts to SD card's **`opencpi/artifacts/`** directory.

5.2.3.3 System Time Setup

If Linux system time on the embedded system is not required to be accurate, this step may be skipped. For either network or standalone mode, the following settings in **`my[net]setup.sh`** may require modification:

- Identify the system that is to be used as a time server, where the default is **`time.nist.gov`** and is set in **`opencpi/ntp.conf`** on the SD card. A valid time server must support NTP.
- Identify the current timezone description, where the default is **`EST5EDT,M3.2.0,M11.1.0`**. Change this if required for the local timezone. See **`man tzset`** on the development host for more information.

If a time server is not required, or cannot connect to a time server, the user is required to manually set the time at system start-up. Use the **`date`** command on the embedded system to manually set the Linux system time. Use **`date --help`** on the embedded system to display command usage information. Use **`man date`** on the development host if additional information is needed.

5.2.4 Establishing a Serial Console Connection

The section Preparing the Development Host for Serial Console describes how to set up serial consoles from the development host. This step depends on two pieces of information about the embedded system: the baud rate of the serial port and the **`udev`** rules file located at:

```
$OCPI_CDK_DIR/<pf>/udev_rules/*.rules
```

This file is created during installation of the hardware platform.

5.2.5 Configuring the Runtime Environment on the Embedded System

This section shows how to verify the runtime environment before running any applications.

Unless in a standalone mode without a network connection, make sure the network connection (e.g. Ethernet cable and/or USB dongle) is connected to a network configured for DHCP for network connectivity.

Ensure there is a console serial cable (typically a micro-USB to USB-A cable) connected between the embedded system's serial port and development host.

Apply power to the embedded system.

Use a serial terminal application to establish a serial connection. For example, to use the **`screen`** serial terminal application, use this command on the development host:

```
sudo screen /dev/ttyUSB0 115200
```

This command assumes the USB connection is on **`/dev/ttyUSB0`** and the baud rate is **`115200`**; both these parameters can be different. If there is a **`udev`** rules file installed for the embedded system, the **`tty`** device name may be the actual name of the embedded system with a **`_0`** suffix and **`sudo`** may not be required.

After successfully booting the Linux OS on the embedded system, login to the system. Commonly-used (default) credentials are "root" for the user name and "root" for the password, but that depends on the embedded system.

5.2.5.1 Server Mode

After a successful login, we now establish the embedded system as an OpenCPI container server to permit applications to execute on the system over SSH when run from the development host. This setup takes advantage of the OpenCPI remote container feature described in detail in the ***OpenCPI Application Development Guide***.

To perform the setup:

- Use the OpenCPI environment variable OCPI_SERVER_ADDRESSES to set the network address (XX.XX.XX.XX) and optionally the TCP port of the embedded system for subsequent commands.
- Use the **`ocpiremote(1)`** command to load the OpenCPI server software onto the embedded system and start the server as follows:

  For the **`ocpiremote load`** operation, use the **`--rcc-platform`** and **`--hdl-platform`** options as necessary to specify the embedded system's RCC and HDL platforms; the default RCC platform used by the command is **`xilinx19_2_aarch32`** and the default HDL platform is **`zed`**. Use the **`--password`** option as necessary to specify the server's login password; the default password used by the command is **`root`**.

  For the **`ocpiremote start`** operation, use the **`--bitstream`** option to load a test OpenCPI bitstream over whatever has already been loaded.

Here is a sequence of steps done on a development host to set up an OpenCPI container server on an Ettus Research E310 USRP embedded system. In this example, the TCP port being used at the server is **`12345`**, and the explicitly specified

network address of the server is **10.0.0.86**. The RCC platform used for the system is **xilinx19_2_aarch32**, so it does not need to be specified. The password used is **root**, so it, too, does not need to be specified.

```
# set the server address and port for both ocpiremote and ocpirun
export OCPI_SERVER_ADDRESSES=10.0.0.86:12345
ocpiremote load --hdl-platform e31x
ocpiremote start --bitstream
```

After loading and starting the server, use the **ocpiremote status** command to make sure it's running. For the example above, use:

```
ocpiremote status
```

You should see output similar to the following:

```
Executing remote configuration command: status
Server is running with port: 12345 and pid: 26224
```

At this point, the embedded system is ready to execute applications run from the development host.

5.2.5.2 Network Mode

After a successful login, we now establish the OpenCPI environment for this login shell as well as subsequent SSH logins via the network.

Each time the system is booted, the OpenCPI environment must be established. By sourcing the **mynetsetup.sh** script, the embedded system's environment is configured for OpenCPI, and NFS directories are mounted for network mode.[2] The user must provide two arguments to the script: 1) the network address (**XX.XX.XX.XX**) of the development system and 2) the pathname (**/path/to/share**) to mount on the development system to the script as its only argument:

```
source /mnt/card/opencpi/mynetsetup.sh XX.XX.XX.XX /path/to/share
```

where XX.XX.XX.XX is the IP address of the NFS host (i.e. the development host, e.g. **192.168.1.10**). A successful run should output the following:

```
An IP address was detected.
Setting the time from time server: time.nist.gov
My IP address is: XX.XX.XX.XX, and my hostname is: zynq
Running login script. OCPI_CDK_DIR is now /mnt/net/cdk.
Executing /home/root/.profile
No reserved DMA memory found on the linux boot command line.
The mdev config has no OpenCPI rules. We will add them to /etc/mdev.conf
NET: Registered protocol family 12
Driver loaded successfully.
OpenCPI ready for zynq.
Discovering available containers...
Available containers:
 #  Model Platform            OS        OS-Version  Arch     Name
```

2 This script calls the **zynq_setup.sh** script, which should not be modifiable by the user.

```
 0  hdl   e31x                                                  PL:0
 1  rcc   xilinx19_2_aarch32  linux     x19_2       arm       rcc0
```

The platforms (**e31x** and **xilinx19_2_aarch32** in this example) will correspond to the platforms of the actual embedded system.

Note: If the output includes:

```
Attempting to set the time from time server
Alarm clock
```

it indicates that **ntp** was unable to set time using servers in **ntp.conf.** For more information, see the section System Time Setup.

5.2.5.3 Standalone Mode

WARNING: Applications (including XML-only ones) fail if there is not an IP address assigned to the platform, even when in standalone mode. When the Ethernet port is not connected to a network configured with DHCP, a temporary IP address must be set, e.g.

```
ifconfig eth0 192.168.244.244
```

Each time the system is booted, the OpenCPI environment must be set up. By sourcing the **mysetup.sh** script, the embedded system's environment is configured for OpenCPI. There are no arguments for this script.

```
source /mnt/card/opencpi/mysetup.sh
```

A successful run should output the following:

```
Attempting to set the time from time server: time.nist.gov
Setting the time from time server: time.nist.gov
Running login script. OCPI_CDK_DIR is now /mnt/card/opencpi.
Executing /home/root/.profile
No reserved DMA memory found on the linux boot command line.
The mdev config has no OpenCPI rules. We will add them to /etc/mdev.conf
NET: Registered protocol family 12
Driver loaded successfully.
OpenCPI ready for zynq.
Discovering available containers...
Available containers:
 #  Model Platform             OS     OS-Version  Arch     Name
 0  hdl   e31x                                             PL:0
 1  rcc   xilinx19_2_aarch32   linux  x19_2       arm       rcc0
```

The platforms (**e31x** and **xilinx19_2_aarch32**) in this example will correspond to the platforms of the actual embedded system.

5.2.6 Running a Test Application

This section provides confirmation of a successful installation by running an application on the embedded system.

5.2.6.1 Running an Application in Server Mode

Once the server network address and port have been established and the OpenCPI server software has been loaded and started on the target embedded system, no further

setup is required. Use the **ocpirun(1)** command to run the test application as follows:

```
ocpirun -v -t 1 -d -m bias=hdl bias.xml
```

The output should be similar to:

```
Available containers are:  0: PL:0 [model: hdl os:  platform: e31x], 1:
rcc0 [model: rcc os: linux platform: xilinx19_2_aarch32]
Actual deployment is:
  Instance  0 file_read (spec ocpi.core.file_read) on rcc container 1:
rcc0, using file_read in
/mnt/ocpi_core/artifacts/ocpi.core.file_read.rcc.0.xilinx19_2_aarch32.so
dated Fri Jan 22 18:18:26 2021
  Instance  1 bias (spec ocpi.core.bias) on hdl container 0: PL:0, using
bias_vhdl/a/bias_vhdl in
/mnt/ocpi_assets/artifacts/ocpi.assets.testbias_e31x_base.hdl.0.e31x.gz
dated Mon Jan 25 11:59:53 2021
  Instance  2 file_write (spec ocpi.core.file_write) on rcc container 1:
rcc0, using file_write in
/mnt/ocpi_core/artifacts/ocpi.core.file_write.rcc.0.xilinx19_2_aarch32.so
dated Fri Jan 22 18:18:31 2021
Application XML parsed and deployments (containers and artifacts) chosen
Application established: containers, workers, connections all created
Dump of all initial property values:
Property  0: file_read.fileName = "test.input" (cached)
Property  1: file_read.messagesInFile = "false" (cached)
Property  2: file_read.opcode = "0" (cached)
Property  3: file_read.messageSize = "16"
Property  4: file_read.granularity = "4" (cached)
Property  5: file_read.repeat = "<unreadable>"
Property  6: file_read.bytesRead = "0"
Property  7: file_read.messagesWritten = "0"
Property  8: file_read.suppressEOF = "false"
Property  9: file_read.badMessage = "false"
Property 10: file_read.ocpi_debug = "false" (parameter)
Property 11: file_read.ocpi_endian = "little" (parameter)
Property 12: bias.biasValue = "16909060" (cached)
Property 13: bias.ocpi_debug = "false" (parameter)
Property 14: bias.ocpi_endian = "little" (parameter)
Property 15: bias.test64 = "0"
Property 16: file_write.fileName = "test.output" (cached)
Property 17: file_write.messagesInFile = "false" (cached)
Property 18: file_write.bytesWritten = "0"
Property 19: file_write.messagesWritten = "0"
Property 20: file_write.stopOnEOF = "true" (cached)
Property 21: file_write.ocpi_debug = "false" (parameter)
Property 22: file_write.ocpi_endian = "little" (parameter)
Application started/running
Waiting up to 1 seconds for application to finish
Application finished
Dump of all final property values:
Property  3: file_read.messageSize = "16"
Property  5: file_read.repeat = "<unreadable>"
Property  6: file_read.bytesRead = "4000"
Property  7: file_read.messagesWritten = "251"
Property  8: file_read.suppressEOF = "false"
```

```
Property  9: file_read.badMessage = "false"
Property 15: bias.test64 = "0"
Property 18: file_write.bytesWritten = "4000"
Property 19: file_write.messagesWritten = "250"
```

Run the following command to view the input:

```
hexdump test.input | less
```

The output should look like the following:

```
0000000 0000 0000 0001 0000 0002 0000 0003 0000
0000010 0004 0000 0005 0000 0006 0000 0007 0000
0000020 0008 0000 0009 0000 000a 0000 000b 0000
0000030 000c 0000 000d 0000 000e 0000 000f 0000
...
0000130 004c 0000 004d 0000 004e 0000 004f 0000
0000140 0050 0000 0051 0000 0052 0000 0053 0000
0000150 0054 0000 0055 0000 0056 0000 0057 0000
0000160 0058 0000 0059 0000 005a 0000 005b 0000
```

Run the following command to view the output:

```
hexdump test.output | less
```

The output should look like the following:

```
0000000 0304 0102 0305 0102 0306 0102 0307 0102
0000010 0308 0102 0309 0102 030a 0102 030b 0102
0000020 030c 0102 030d 0102 030e 0102 030f 0102
0000030 0310 0102 0311 0102 0312 0102 0313 0102
...
0000130 0350 0102 0351 0102 0352 0102 0353 0102
0000140 0354 0102 0355 0102 0356 0102 0357 0102
0000150 0358 0102 0359 0102 035a 0102 035b 0102
0000160 035c 0102 035d 0102 035e 0102 035f 0102
```

5.2.6.2 Running an Application in Network Mode

The default setup script sets the **OCPI_LIBRARY_PATH** variable to include the RCC artifacts that are required to execute the application, but it must be updated to include to the assembly bitstream that was built on the development host. After running the **mynetsetup.sh** script, run the following commands to update the **OCPI_LIBRARY_PATH** variable and execute the application:

```
export \
  OCPI_LIBRARY_PATH=/mnt/ocpi_assets/artifacts:/mnt/ocpi_core/artifacts
cd /mnt/ocpi_assets/applications
ocpirun -v -t 1 -d -m bias=hdl bias.xml
```

The viewing commands and output should be similar to the output shown above in Running in Server Mode.

5.2.6.3 Running an Application in Standalone Mode

The default setup script sets the **OCPI_LIBRARY_PATH** variable to include the artifacts that are required to execute the application. Specifically, all three of the artifacts that are located on the SD card are available at:

```
/mnt/card/opencpi/<rcc-platform>/artifacts
```

After sourcing **mysetup.sh**, run the application with these commands:

```
cd /mnt/card/opencpi/xml
ocpirun -v -t 1 -d -m bias=hdl bias.xml
```

The viewing commands and output should be similar to the output shown above in Running in Server Mode.

5.2.7 Updating Files on the Embedded System at Runtime

Currently OpenCPI requires SSH access to embedded systems, thus the easiest way to update files on the embedded system is to use the **scp** command on the development host. When the embedded system is running in standalone mode (no network mounts), the applications or artifacts can be updated this way after being edited or rebuilt on the development host.

6 Installing Third-party/Vendor Tools

OpenCPI uses a collection of third-party tools to enable it to target embedded software and FPGA/HDL platforms. Each platform supported by OpenCPI depends on one or more of these third-party tools. The Table of Supported Platforms in this guide lists the platforms and their tool dependencies. This section describes how to install the tools that are required for the particular platform(s) you plan to use. Installing these tools enables the platform to be built, as shown in the figure that depicts the installation sequence for each platform in an embedded system.

As of the current release, all the tools described in this section require manual installation steps using browsers and vendor-provided installation GUIs. This means that the development system must support typical GUI tools using X-Windows, and it is assumed that the FireFox browser is used for any of the web-based steps.

Also, some of these tools have prerequisite software packages that are automatically installed when OpenCPI is installed per the Install, Build and Test OpenCPI section. Thus these instructions for third-party tools assume that the basic OpenCPI installation for the development host is already done.

6.1 Installing Xilinx Tools

Here is the recommended procedure for installing Xilinx tools. Most of these steps require internet access and a Xilinx web site account:

1. Create a directory to act as a common top-level directory for all the Xilinx tools to be installed.
2. Install the Xilinx Vivado WebPACK. This installation is explicitly for the most recent version of the tool supported by OpenCPI. After you perform this step, you will have:

- The Xilinx FPGA simulator **`xsim`**, which allows you to follow the OpenCPI tutorials and learn about using OpenCPI for heterogeneous development on FPGAs without needing any FPGA hardware.
- The Xilinx unified installer “downloader", which you can use to download installers for licensed versions of all Xilinx parts if needed (see step 4).
- The current Xilinx SDK, which you can use if your target Xilinx-related software platform(s) require it. See step 3).

3. For each Xilinx-related software platform you plan to use, install the Xilinx tools required for that platform. These are:

- Xilinx SDK for cross-compilation on embedded platforms
- Xilinx Linux binary release and Xilinx Linux kernel and U-boot source code for preparing bootable SD cards for Zynq-7000/UltraScale platforms

  These tools are specific to both the SDK version and the ARM CPU architecture used by the platform. Install the versions that correspond to your target software platform(s), as listed in the “Dependencies required...” column of the Table of Supported Platforms.

4. If Xilinx Vivado WebPACK doesn’t support your Xilinx parts, install the Xilinx Vivado licensed tools that do support them. You will need to get your licenses from Xilinx and you should uninstall/remove the Vivado WebPACK installation first.
5. If Vivado does not support your Xilinx parts because they are too old (for example, the Virtex-6 FPGA), install the ISE Design Suite 14.7 tool set.

After you have finished this procedure:

- All the Xilinx tools required for your particular platform(s) are installed and are located in one place: the directory you created in step 1.
- You can run the **`ocpiadmin`** tool to install a Xilinx-based platform without needing network access. Be sure the account that runs the tool has write permission for any directories created under the top-level Xilinx installation directory.

The remainder of this section provides instructions for performing each step.

6.1.1 Xilinx Tools Installation Directory

Before you begin to install the Xilinx tools, decide where you want to install them. It is recommended that you install all the Xilinx tools in the same top-level directory to keep them all in one place. The Xilinx tool sets require a very large amount of disk space for installation (for example, WebPACK installation requires about 36GB) and the Xilinx installer needs write access to the directory you choose. You do one of three alternatives, with the first being recommended:

1. Use the default directory offered by the Xilinx installer. The default directory is **/tools/Xilinx** for Vivado 2019.2 or later and **/opt/Xilinx** for versions older than 2019.2. Create this directory outside of the installer so that the installer doesn't need to run with **root** or **sudo** permission. For example:
   ```
   sudo mkdir /tools/Xilinx
   sudo chmod 777 /tools/Xilinx
   ```
2. Use a different directory from the default – for example, **~/MyXilinxTools** – and then create a symbolic link from the default directory to that directory. E.g.:
   ```
   sudo ln -s ~/MyXilinxTools /tools/Xilinx
   ```
3. Use a different directory from the default and then set the **OCPI_XILINX_DIR** environment variable to that directory by editing/uncommenting the **OCPI_XILINX_DIR** line in the **user-env.sh** file. E.g. if you will install the tools under **~/MyXilinxTools**, the line in **user-env.sh** should be:
   ```
   export OCPI_XILINX_DIR=~/MyXilinxTools
   ```

Once you've set up your top-level Xilinx tools installation directory, you can proceed to install the tools.

6.1.2 Xilinx Vivado WebPACK for Simulation and for Small, Recent FPGA Parts

The Xilinx Vivado tool set is offered in several different variants; of these, the Vivado Design Suite HL WebPACK™ Edition is free and is the easiest one to install because it doesn't require a license. Unfortunately, it is still a large and slow installation.

In addition to simulation, the WebPACK Edition supports compilation for a limited set of Xilinx hardware FPGA parts; however, the Xilinx parts it does support are those found in most small systems, including the Avnet (Digilent) ZedBoard, the Epiq Matchstiq-Z1 and the Ettus E310. Consult this link for vendor information on supported parts for WebPACK Edition. If you need to use OpenCPI platforms with Xilinx parts that WebPACK Edition does not support, you must buy licenses and perform a licensed installation of at least the Vivado Design Edition.

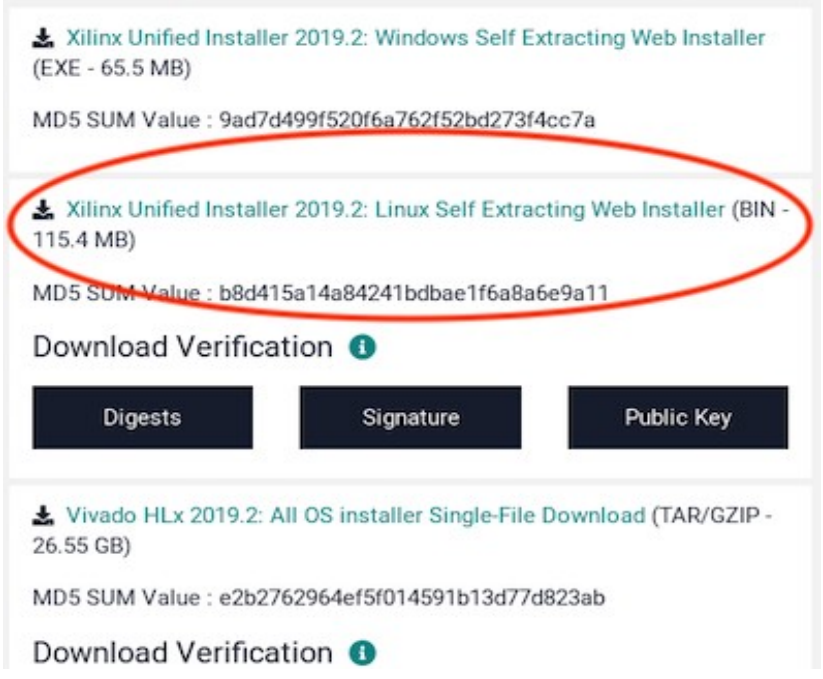
The rest of this section describes how to install Vivado WebPACK with vendor defaults on a CentOS7 development host. This is the basic installation required to get the **xsim** simulator up and running to support FPGA development without FPGA hardware. It is correct for the Vivado 2019.2 release.

6.1.2.1 Download the Xilinx Unified Installer Binary

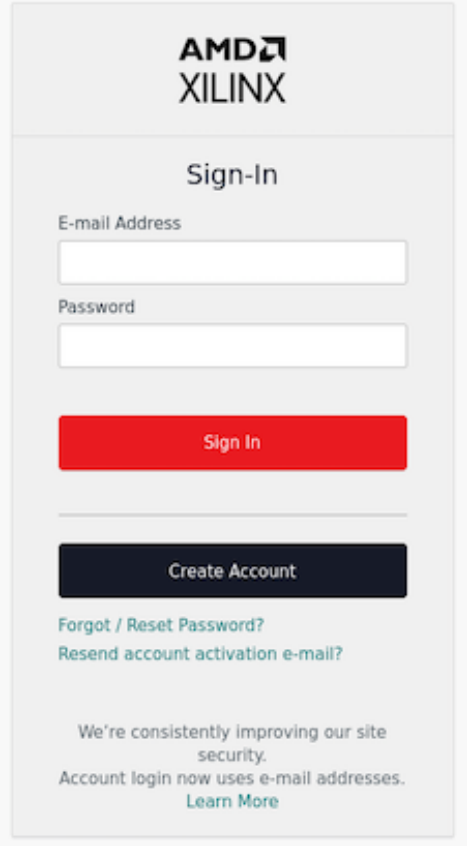
In a browser on your development host, go to https://www.xilinx.com/support/download.html.

In **Version** on the left, click on **Vivado Archive** and then click **2019.2**.

Scroll down to the downloads menu region Vivado Design Suite – HLx Editions – 2019.2 and then select **Xilinx Unified Installer 2019.2: Linux Self Extracting Web Installer**.

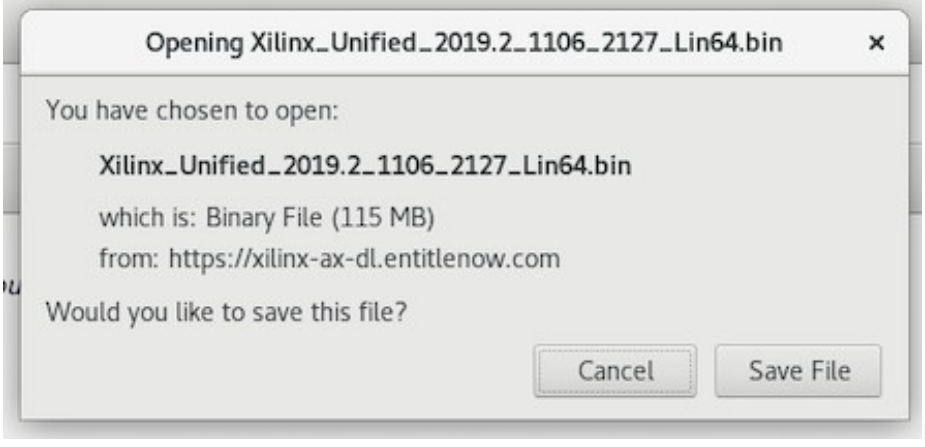


In the login dialog, enter your email address and password or click **Create Account** to create one.



Fill out the required information in the Xilinx Download Center Name and Address Verification page and then click **Download**.

In the dialog that pops up, click **Save File**.



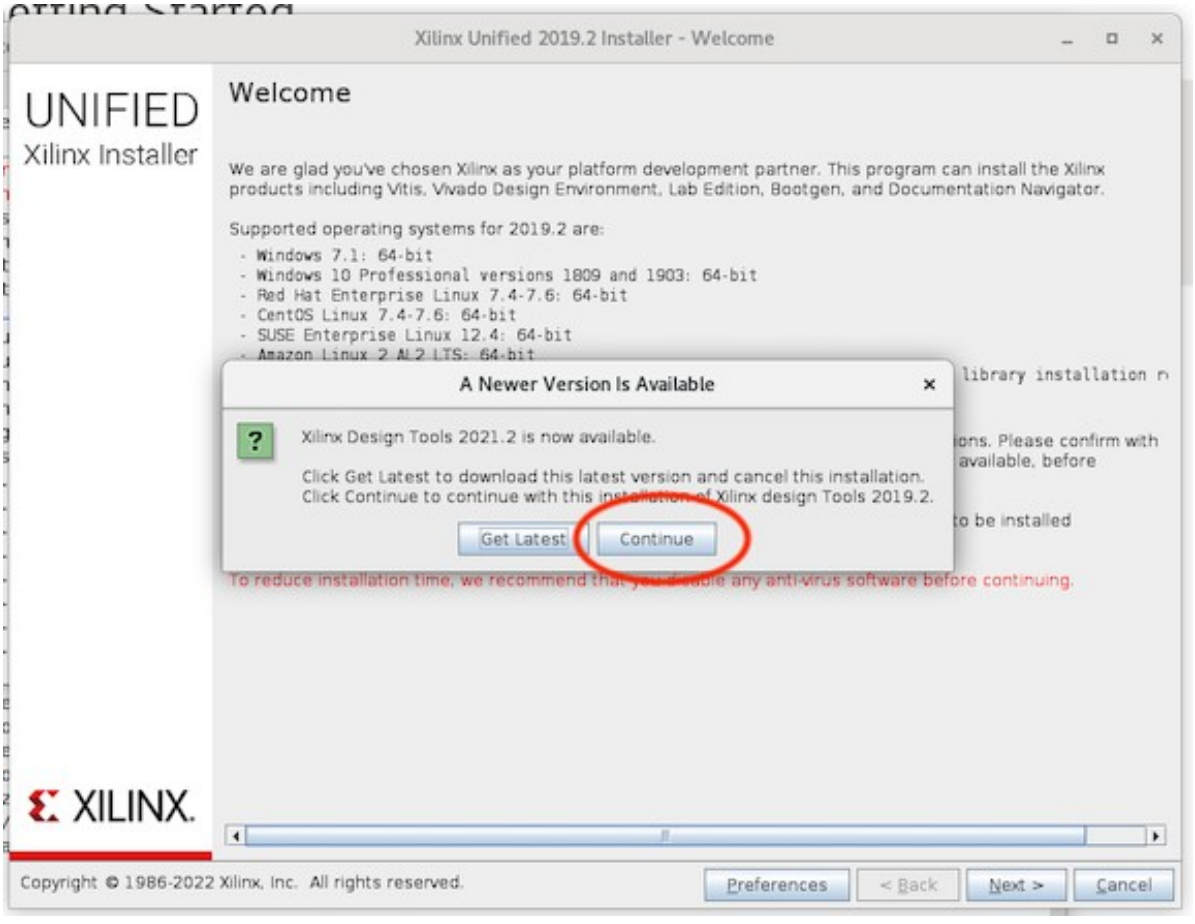
The file is saved to the **`~/Downloads`** directory on your CentOS7 development host.

6.1.2.2 Run the Unified Installer to Download the Vitis/Vivado Design Suite Installer

In a terminal window on your CentOS7 development host, navigate to the ~/**`Downloads`** directory. Run the unified installer binary with the **`bash`** command. For example:

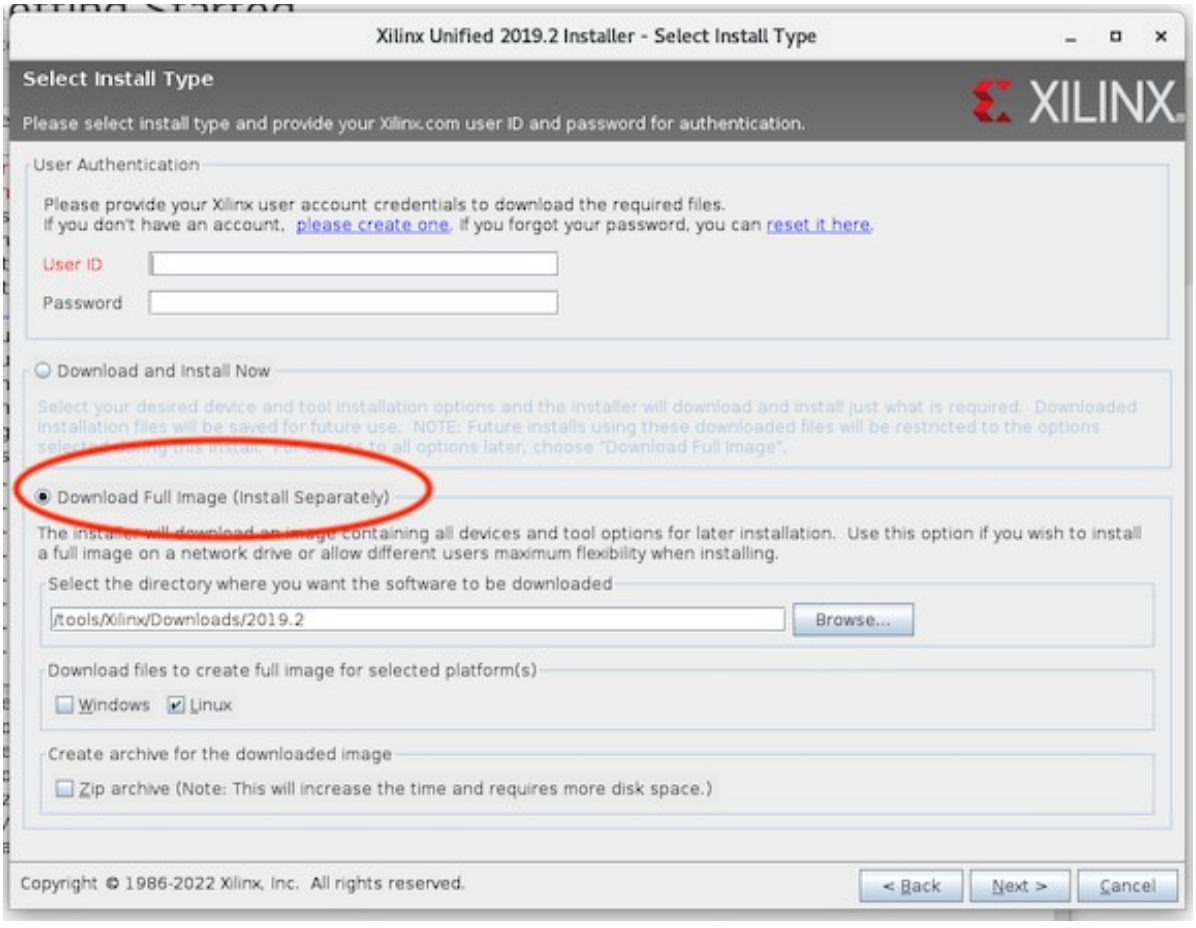
```
bash Xilinx_Unified_2019.2_1106_2127_Lin64.bin
```

The installer starts and displays a dialog informing you that there is a newer version of the software available. Click **Continue** to close the dialog, then click **Next**.



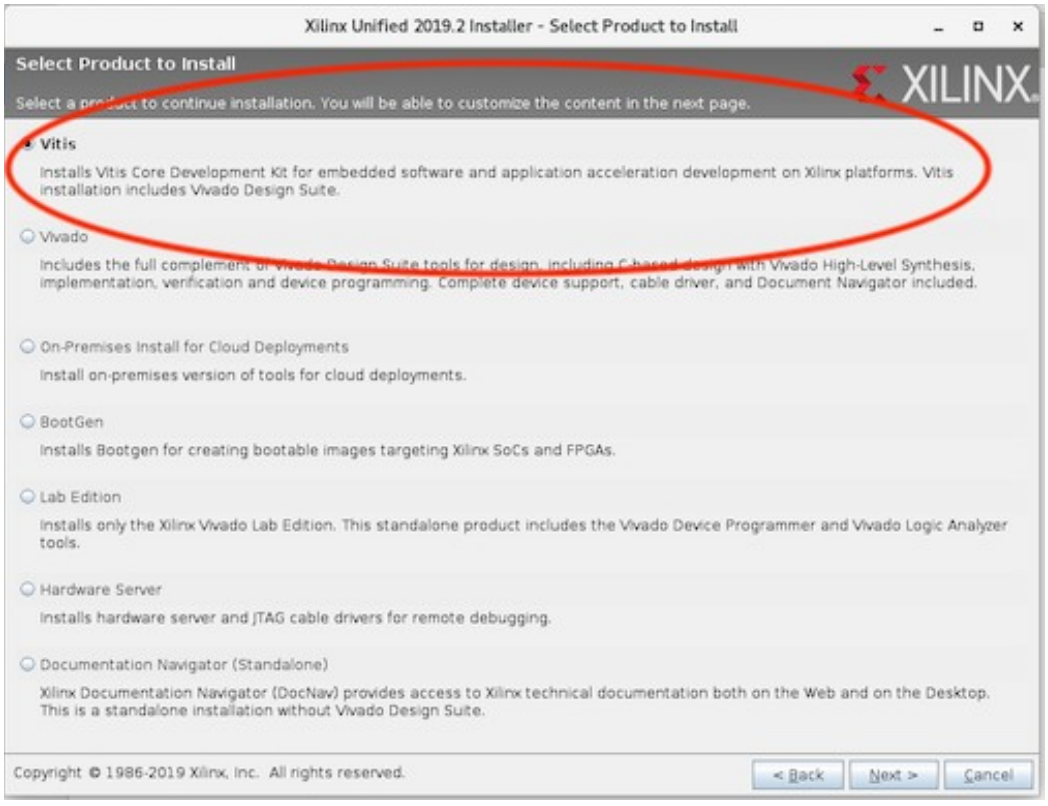
In the Select Install Type dialog:

- Select **Download Full Image (install separately)**.
- Replace the default directory provided in the dialog with a temporary directory of your choice that has sufficient disk space to accommodate the installation files and sufficient permissions to permit the binary installer to write to this location. We will refer to this directory as the ***downloads*** directory. It is temporary and can be removed at the end of the installation. It is not the final installation location.
- Do not choose to create a ZIP archive.


Xilinx Unified 2019.2 Installer - Select Install Type
Select Install Type
Please select install type and provide your Xilinx.com user ID and password for authentication.
XILINX
User Authentication
Please provide your Xilinx user account credentials to download the required files.
If you don't have an account, please create one. If you forgot your password, you can reset it here.
User ID
Password
Download and Install Now
Select your desired device and tool installation options and the installer will download and install just what is required. Downloaded installation files will be saved for future use. NOTE: Future installs using these downloaded files will be restricted to the options
Download Full Image (Install Separately)
The installer will download an image containing all devices and tool options for later installation. Use this option if you wish to install a full image on a network drive or allow different users maximum flexibility when installing.
Select the directory where you want the software to be downloaded
/tools/Xilinx/Downloads/2019.2
Browse...
Download files to create full image for selected platform(s)
Windows
Linux
Create archive for the downloaded image
Zip archive (Note: This will increase the time and requires more disk space.)
Copyright © 1986-2022 Xilinx, Inc. All rights reserved.
< Back
Next >
Cancel

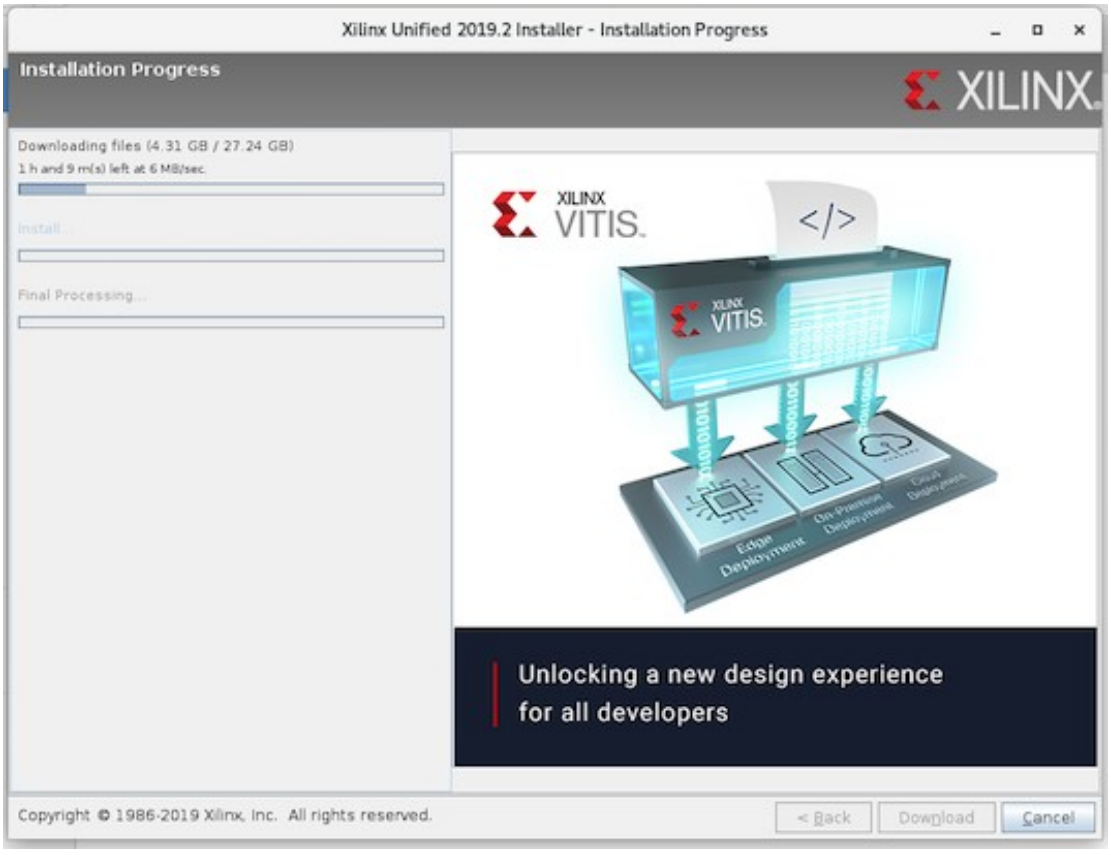
Click **Next**.

In the Select Product to Install dialog, **Vitis** is pre-selected. This product provides everything we need, including the WebPACK edition of Vivado and the current SDK (we *do not* want the Vivado package). Leave this selection as **Vitis** and then click **Next**.

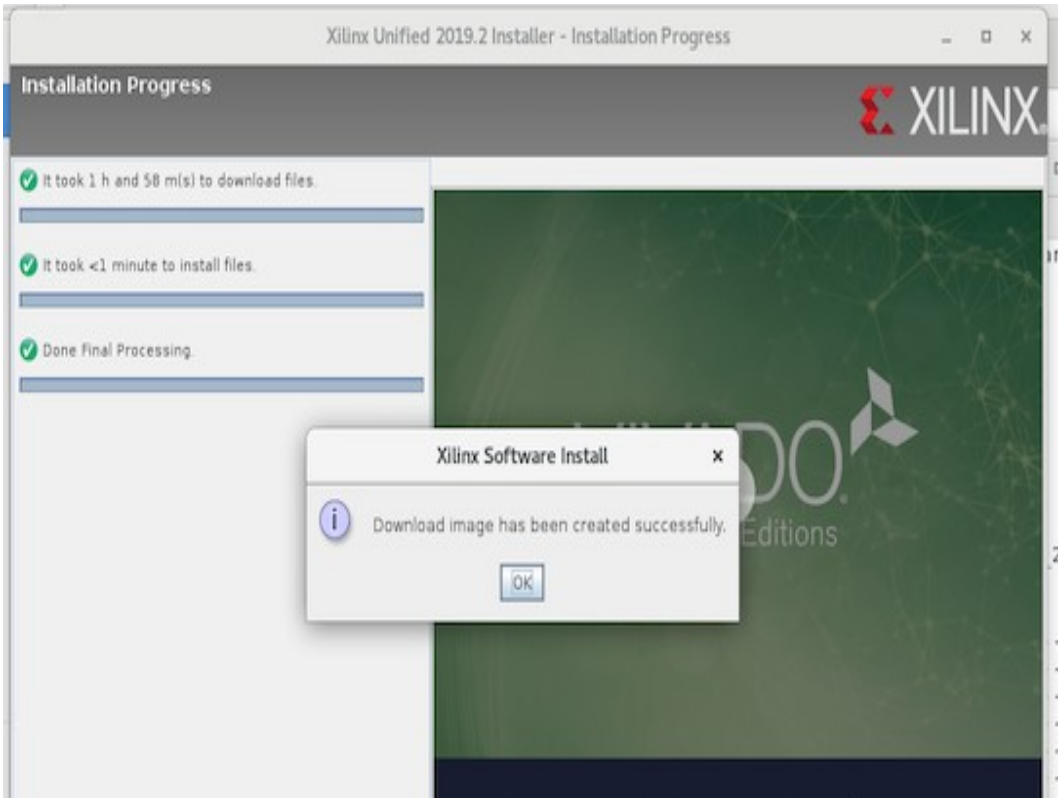


The unified installer binary displays a Download Summary that allows you to verify the directory where you're downloading, the size and disk space required (27.24GB) and the platform (Linux). Click **Download**.

The Download Center displays an **Installation Progress** dialog.



When the download completes, click **OK** in the pop-up dialog to exit. (A notification that the unified installer binary has completed also pops up.)



Now you have downloaded the main Vitis/Vivado Design Suite ***installer*** (called Vitis IDE in some windows) into the ***downloads*** directory, which you can use to install the Vivado WebPACK Edition and the current SDK. This installer can also be used later to install other Xilinx tools, including licensed versions for all Xilinx parts. Don't delete it until all such installations are done.

Note that the initial "unified installer" that you *downloaded* was really just used as a "downloading tool", to download the (big) Vitis/Vivado installer, which, when it runs, also displays the title "unified installer". So at this point you have downloaded a small "unified installer" download tool, and run it to download the big Vitis/Vivado "unified installer".

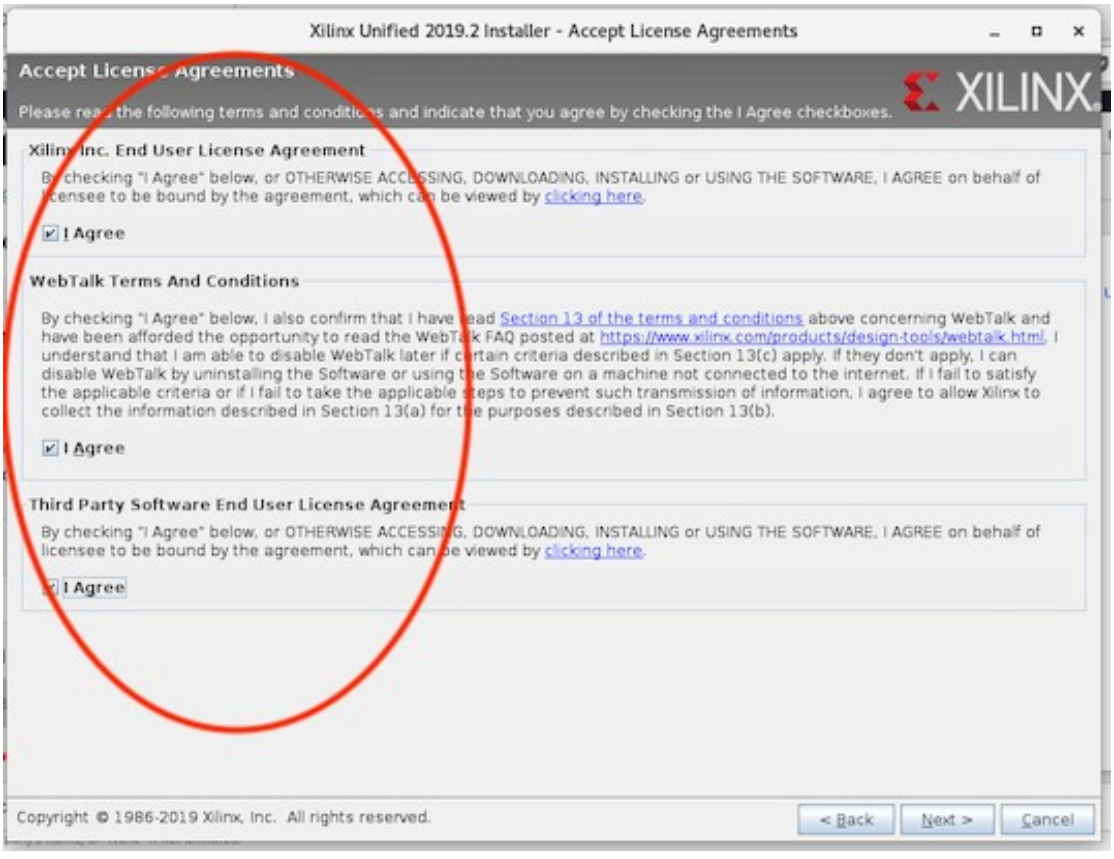
6.1.2.3 Run the Vitis/Vivado Design Suite Installer

In a terminal/shell window, navigate to the directory where you downloaded the installation files (the ***downloads*** directory) and then run the following command:

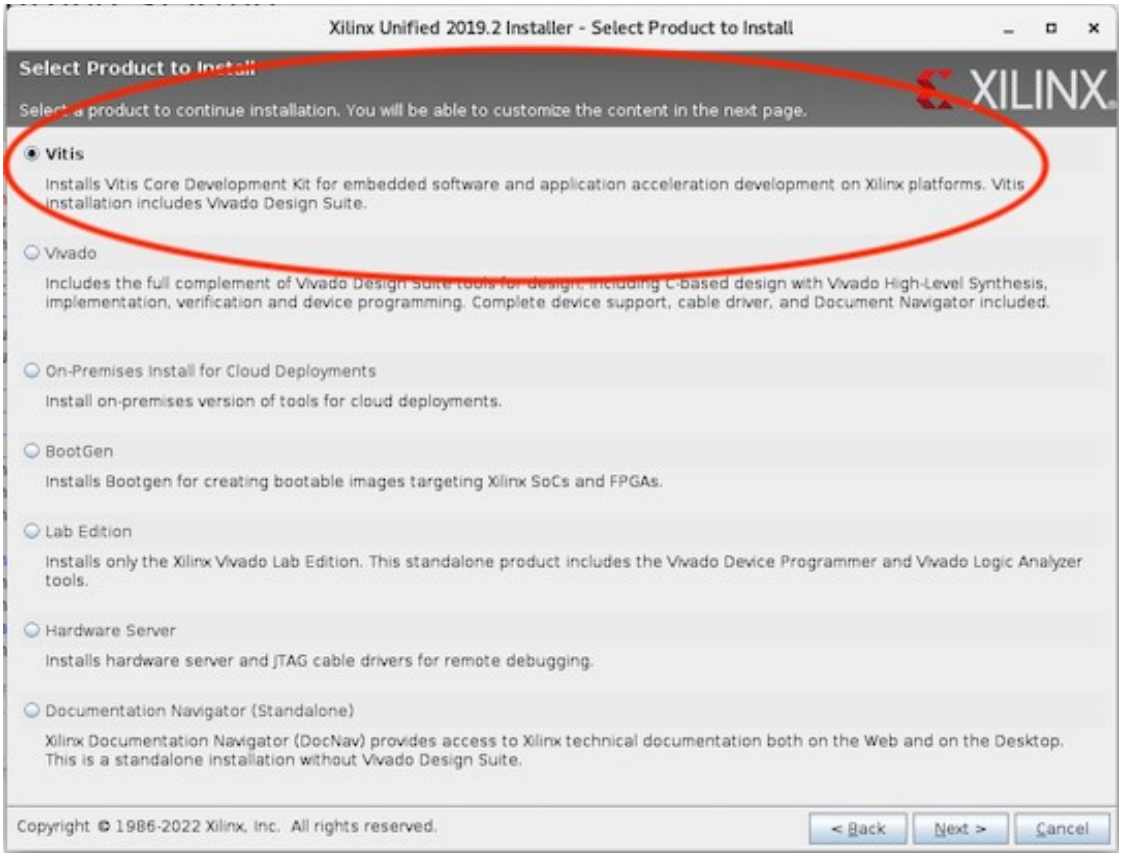
```
./xsetup
```

The installer starts and briefly shows a window titled "Vitis IDE", and then a new window comes up with the title "Xilinx Unified Installer", which is the same title as the initial "downloader" program used above. Be aware that it can take as much as 20 minutes for you to see the first dialog.

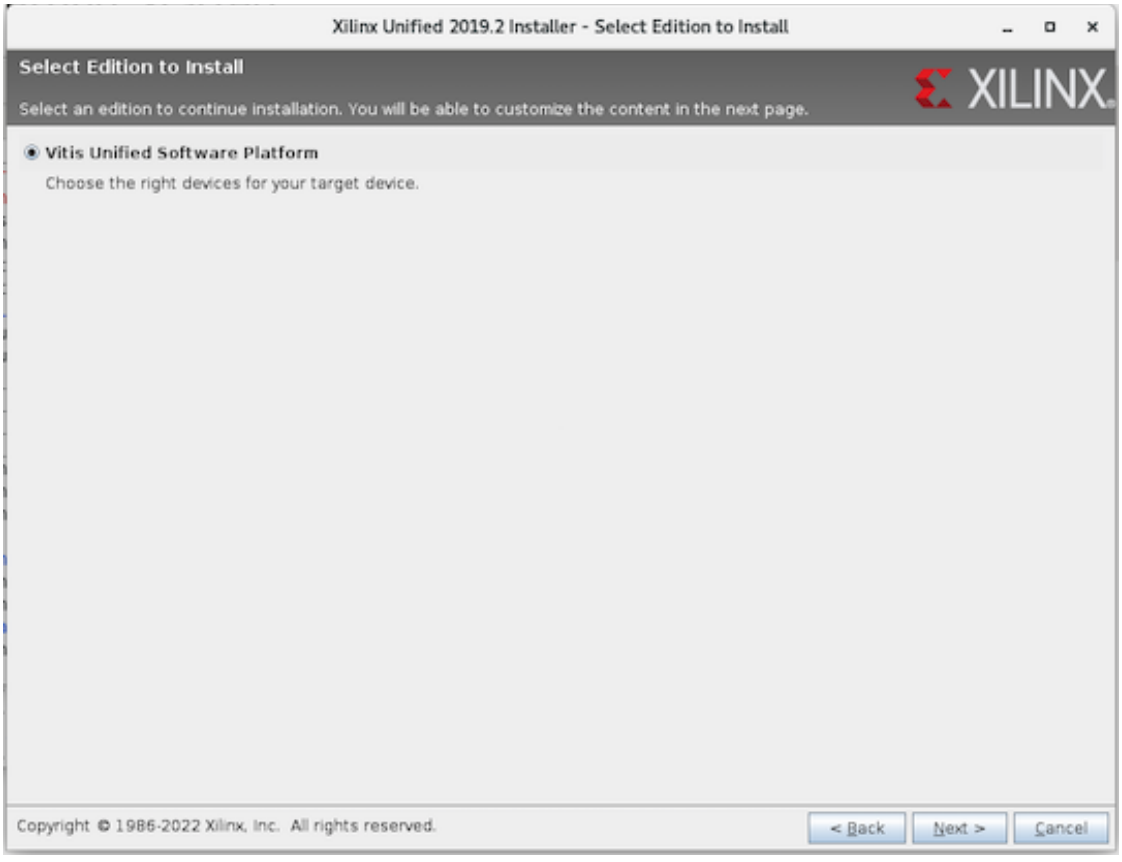
The first dialog is the license terms. Agree with the license terms.



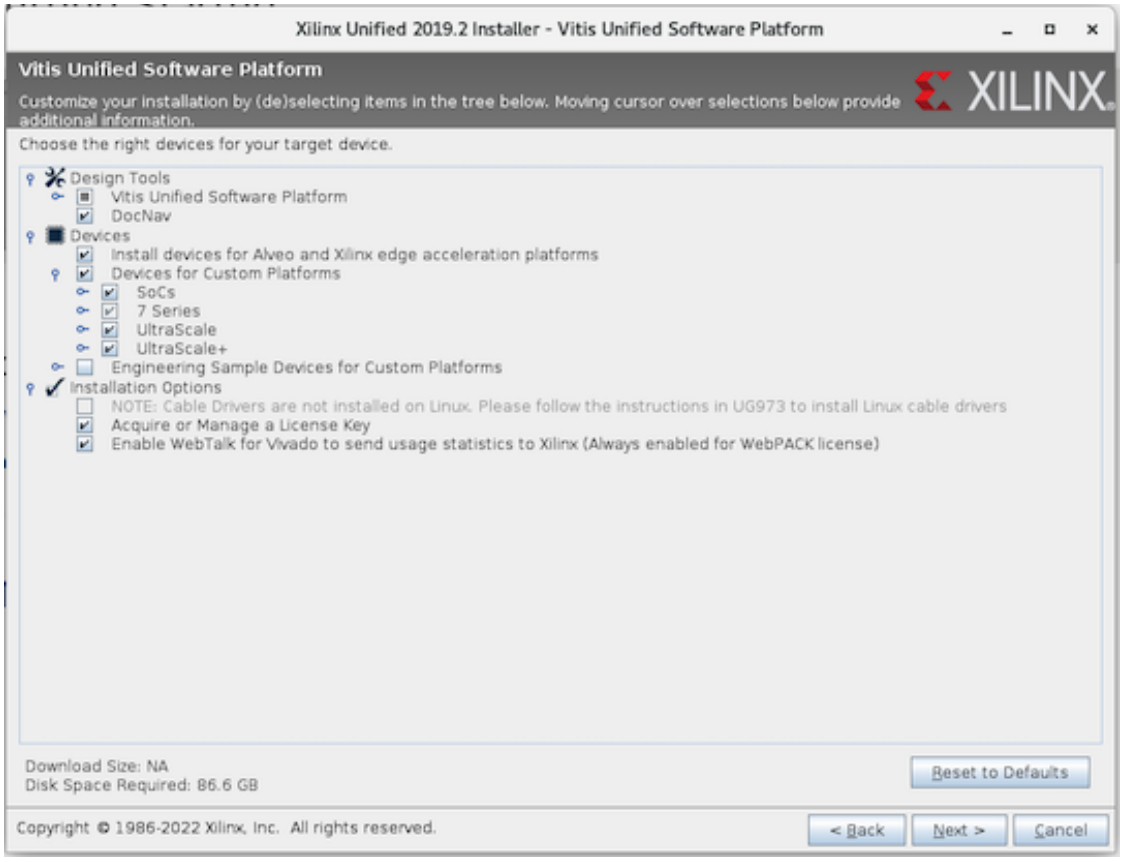
In the Select Product to Install dialog, **Vitis** is already selected. This is the product we want to install (*we **do not** want the Vivado option*), so leave this selection as it is and click **Next**.



In the Select Edition to Install dialog, **Vitis Unified Software Platform** is already selected. Leave this selection as it is and click **Next**.

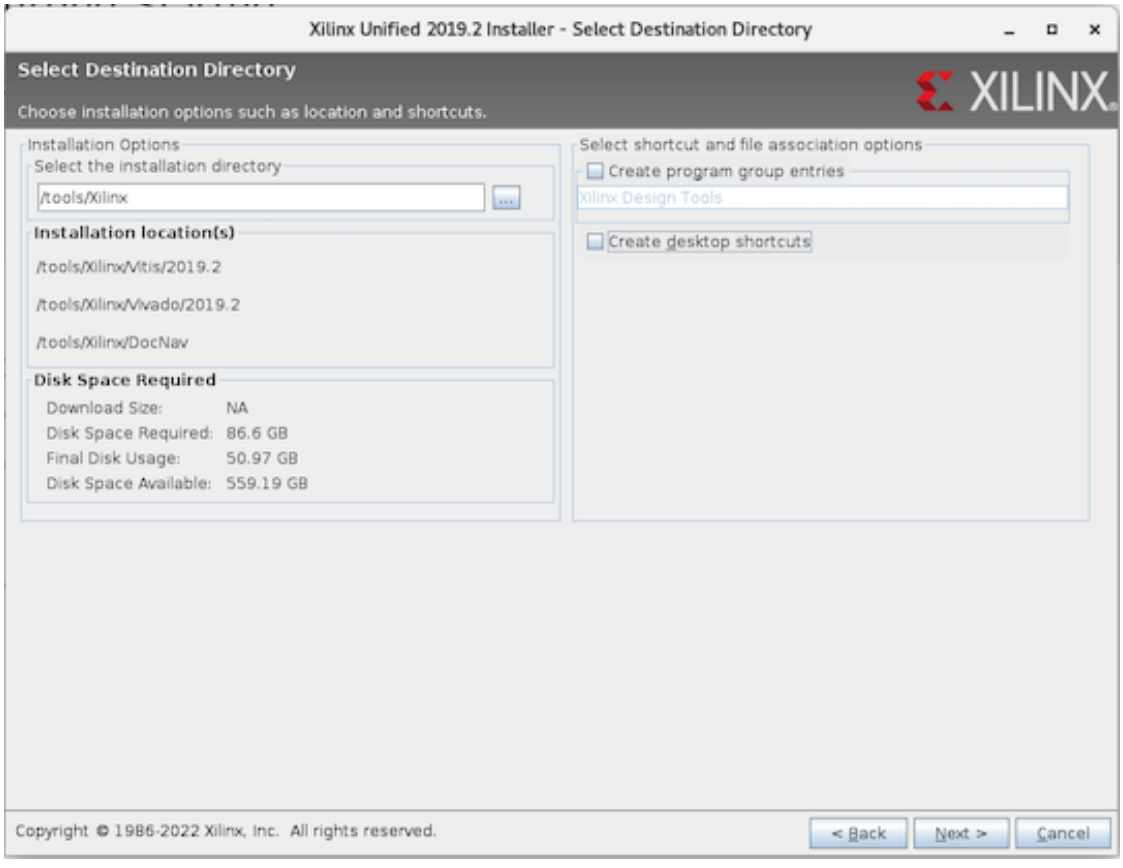


In the Vitis Unified Software Platform dialog, leave everything as it is and then click **Next** to accept the selected defaults.



In the Select Destination Directory dialog:

- Specify the Xilinx tools directory you created before you started the Xilinx tools installation process. Ensure you have read/write permissions and that the destination has sufficient disk space for the installed software. If you have chosen to use the default directory, make sure that your choice is the same as what's specified in this dialog.
- Uncheck **Create program group entries**.
- Uncheck **Create desktop shortcuts**.
- Click **Next**.



The Installation Summary dialog appears. Click **Install** to start the installation.

When the download completes, click **OK** in the pop-up dialog to exit. You have now installed the Vivado WebPACK Edition tools and the Xilinx **`xsim`** simulator. This installation procedure has also automatically installed the current SDK for cross compiling various Xilinx SoC platforms.

There many advanced installation scenarios for Xilinx Vivado. This one is sufficient for running **`xsim`**, and also for targeting OpenCPI platforms with small Xilinx FPGA parts.

6.1.2.4 Prepare and Build the Simulator as an OpenCPI Platform

To complete the installation of **xsim** for OpenCPI so that you have a base for learning about OpenCPI development on FPGAs without requiring FPGA hardware, you must run the following **ocpiadmin** command:

```
ocpiadmin install platform xsim
```

This command builds the OpenCPI built-in projects for using **xsim**. It may take 10-15 minutes to complete.

6.1.3 Xilinx Current SDK Tools for Cross-compilation for Newer Xilinx-based Software Platforms

The current SDK for cross compiling various Xilinx SoC platforms is part of the Vitis/Vivado Design Suite and is automatically installed when performing the Xilinx Vivado WebPACK installation. No additional installation steps are necessary.

6.1.4 Xilinx Binary Releases for Zynq-7000 and Zynq-UltraScale Systems

To prepare bootable SD cards for Xilinx Zynq-7000 and Zynq-UltraScale software platforms, the Xilinx-provided binary Linux release packages are required. Recent Xilinx releases require these downloads to be done using a browser to confirm the user's account, physical location and license acceptance. These releases are called (by Xilinx) “Linux Prebuilt Images”, and contain a set of built, binary assets for booting Linux.

If a platform mentions that it requires a “Xilinx Binary Release” in the “Dependencies required..” column in the Table of Supported Platforms, then the following download procedure should be performed. We will use the **xilinx19_2_aarch32** platform as an example.

The “Dependencies required” column for the **xilinx19_2_aarch32** platform mentions that it requires “Xilinx Binary Zynq Release 2019.2”. So the Xilinx release identifier is **2019.2**. Here are the steps to make this binary release available to OpenCPI:

1. If there is not already one existing, create a directory under the top-level Xilinx tools installation directory you created before starting the Xilinx tools installation process (usually **/tools/Xilinx** or **/opt/Xilinx**), called **ZynqReleases/*<xilinx-release>***. E.g. in this example:

   ```
   mkdir -p /tools/Xilinx/ZynqReleases/2019.2
   ```

2. Go to the Xilinx Wiki page at wiki.xilinx.com. Select the ***Linux Prebuilt Images*** page and then select the particular release, e.g. ***2019.2 Release***. On this page for the release, navigate/scroll to the ***Downloads*** section, which has a list of downloadable files for various hardware platforms.
3. Download all these files into the directory you have created above. You may avoid files that correspond to systems you will never use, but it is usually best to download them all at once.

Once these steps are taken, the binary release is available to OpenCPI tools; in particular, it will be used when the **`ocpiadmin install platform <platform>`** command is run for the software platform *after the Xilinx tools installation procedure is complete*. Note that the **`ZynqReleases/<xilinx-release>/`** directory must have write permission for the account that runs this **`ocpiadmin`** command.

6.1.5 Xilinx Linux Kernel and U-Boot Source Code Repositories

To prepare bootable SD cards for Xilinx Zynq-7000 and Zynq-UltraScale software platforms, the Xilinx source code repositories for the Linux kernel and U-boot are required. Among other things, these repositories allow the OpenCPI loadable kernel module to be built against the correct Linux kernel — the one that is in the binary release mentioned in the previous section.

These Xilinx repositories reside on **`github.com`** and are easily downloaded ("cloned" in this case). Since these repositories are fully open source software, no browser-based licensing dialog is required. If a platform mentions that it requires a "Xilinx Linux git clone" in the "Dependencies required.." column in the Table of Supported Platforms, then the following download procedure should be performed. We will use the **`xilinx19_2_aarch32`** platform as an example.

The "Dependencies required..." column for the **`xilinx19_2_aarch32`** platform mentions that it requires "Xilinx Linux git clone" and that the "Xilinx Linux Tag" is **`xilinx_v2019.2`**. This tag is not used in the download procedure, but documents the version of the Xilinx Linux kernel that will be used when the OpenCPI kernel module is built. Here are the steps to make these repositories available to OpenCPI:

1. If there is not already one existing, create a directory under the top-level Xilinx tools installation directory that you created before you started the Xilinx tools installation process (usually **`/tools/Xilinx`** or **`/opt/Xilinx`**), called **`git`**. E.g. in this example:

   ```
   mkdir -p /tools/Xilinx/git
   ```

2. Change into that directory and then download (clone) the two Xilinx source code repositories there using these commands:

   ```
   cd /tools/Xilinx/git
   git clone https://github.com/Xilinx/linux-xlnx.git
   git clone https://github.com/Xilinx/u-boot-xlnx.git
   ```

Once these steps are taken, the Xilinx Linux git clone is available to OpenCPI tools; in particular, it will be used when the **`ocpiadmin install platform <platform>`** command is run for the software platform *after the Xilinx tools installation procedure is complete*. Note that the **`git/`** directory must have write permission (recursively) for the account that will run this **`ocpiadmin`** command.

6.1.6 Older Xilinx SDK Tools for Cross-compilation for Older Xilinx-based Software Platforms

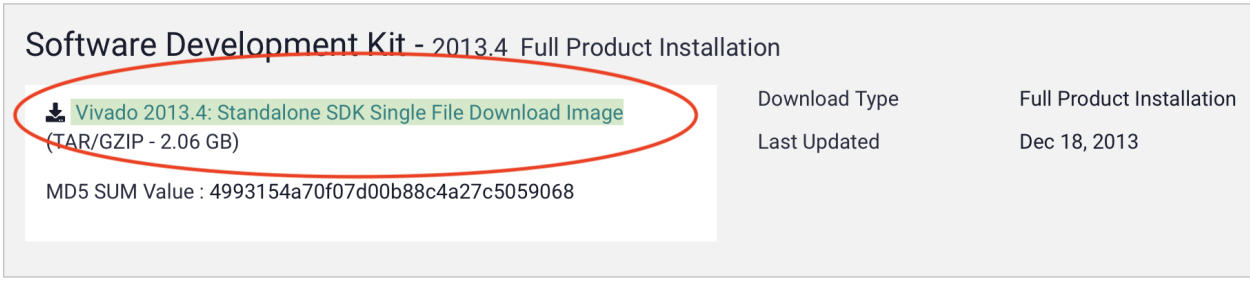
These instructions require a Xilinx web site account, and the login sequence is the same as for the Vivado WebPACK installation sequence described in the section: Download the Xilinx Unified Installer Binary. The login dialogs are displayed when you click the download as shown below.

These installation instructions assume you have already established the top-level directory for Xilinx tools and have installed the Xilinx Vivado WebPACK.

The OpenCPI platforms that need these tool sets specify their versions explicitly, so installing one of these older versions of the Vivado SDK does not force using the old SDK when no specific version is specified.

6.1.6.1 Xilinx SDK 2013.4 Installation

In a browser on your development host, go to https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/vivado-design-tools/archive.html. Select the **2013.4** choice, and choose the **Vivado 2013.4: Standalone SDK Single File Download Image**.



After the download completes successfully, there should be a file in your **~/Downloads** directory named:

```
Xilinx_SDK_2013.4_1210_1.tar
```

Unpack the file, enter the directory you created by unpacking it, and run the installer by entering these commands:

```
cd ~/Downloads
tar -xf Xilinx_SDK_2013.4_1210_1.tar
cd Xilinx_SDK_2013.4_1210_1
./xsetup
```

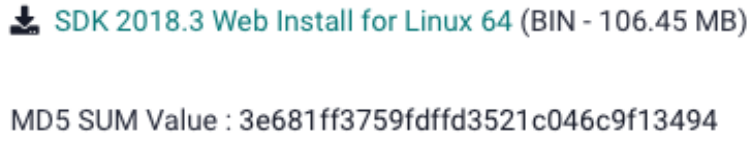
This runs the SDK installer. Select **Next>** on the welcome window, check the **Agree** box and **Next>** on the license windows, **Next>** on the **Select Products to Install** window since there is only one choice and it is enabled. Select **Next>** on the **Select Installation Options** window, leaving **Install Cable Drivers** unchecked. In the **Select Destination Directory** window, specify the top-level Xilinx tools installation directory that you created before starting the Xilinx tools installation process. Select **Next>** and then **Next>** again in the **Installation** window to complete the installation.

6.1.6.2 Xilinx SDK 2018.3 Installation

In a browser on your development host, go to:

https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/vitis/archive-sdk.html

Click the **2018.3** choice, and then choose **SDK 2018.3 Web Install for Linux 64**:



After the download completes successfully, there should be a file in your **`~/Downloads`** directory named:

```
Xilinx_SDK_2018.3_1207_2324_Lin64.bin
```

This is the SDK downloader binary that will download the installer for the SDK. Navigate to your **`~/Downloads`** directory and run the command:

```
bash Xilinx_SDK_2018.3_1207_2324_Lin64.bin
```

The SDK downloader binary starts. A pop-up dialog alerts you that there is a newer version of the SDK available. To download the 2018.3 version, click **Continue** and then click **Next**. In the Select Install Type dialog, enter your Xilinx user name and password, Select **Download Full Image (install separately), r**eplace the default directory provided in the dialog with a temporary directory of your choice that has sufficient disk space to accommodate the installation files and sufficient permissions to permit the binary installer to write to this location, and don't create a ZIP archive. Click **Download**. When the download completes, click **OK** in the pop-up dialog to exit.

In a terminal/shell window, navigate to the directory where you downloaded the SDK installer and then run it with the following command:

```
./xsetup
```

The installer starts. Step through the installer dialogs: accept the license agreement, select the SDK as the installation target (it should already be selected in the dialog), and specify the top-level Xilinx tools installation directory that you created before starting the Xilinx tools installation process as the destination directory.

6.1.7 Xilinx Vivado Licensed Tools for Larger Recent FPGA Parts

The Vivado WebPACK installation described in the Xilinx Vivado WebPACK section is insufficient for some OpenCPI FPGA platforms due to lack of support for the Xilinx parts on those platforms. Consult this link for vendor information on supported parts for WebPACK Edition.

When installing Vivado licensed tools, some different installation steps are required, which are described in the next sections.

6.1.7.1 Choose Vivado HL Design Edition During Installation

When running the Xilinx unified installer, in the dialog **Select Edition to Install**, choose the **Vivado HL Design Edition** rather than the **Vivado HL WebPACK**. Note that these two editions cannot coexist in the same directory tree, so if you have installed the WebPACK Edition, and then discover you need the Design Edition, it is best to uninstall the WebPack Edition before installing the Design Edition.

6.1.7.2 Obtain a License for the Tools and Specify its Location to OpenCPI

Licensing of these tools needs to be established per Xilinx instructions and the license file must be made known to OpenCPI. By default, OpenCPI looks for the file named:

```
Xilinx-License.lic
```

in the top-level Xilinx tools installation directory `/tools/Xilinx` or `/opt/Xilinx`. If you need a different location, and you don't want to use a symbolic link in the default location, you must set the variable `OCPI_XILINX_LICENSE_FILE` in the `user-env.sh` file to that different location.

6.1.8 Xilinx ISE 14.7 Tools for Older FPGA Parts and the isim Simulator

These instructions require a Xilinx web site account, and the login sequence is the same as for the Vivado WebPACK installation sequence in Download Xilinx Installer. The login screens come up when you click the download as shown below.

This installation assumes you have already created the top-level directory for Xilinx tools.

6.1.8.1 Download the ISE Full Installer for Linux

In a browser on your development host, go to https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/vivado-design-tools/archive-ise.html. Then select the **14.7** choice (*not* 14.7 Windows 10), and choose the **ISE Design Suite: Full Installer for Linux**.

6.1.8.2 Run the ISE Full Installer for Linux

After the download completes successfully, there should be a file in your `~/Downloads` directory named:

```
Xilinx_ISE_DS_Lin_14.7_1015_1.tar
```

Unpack the file, enter the directory created by unpacking it, and run the installer, by entering these commands:

```
cd ~/Downloads
tar -xf Xilinx_ISE_DS_Lin_14.7_1015_1.tar
cd Xilinx_ISE_DS_Lin_14.7_1015_1
./xsetup
```

This runs the **ISE 14.7 Installer**. Select **Next>** on the **Welcome** window, check the **Agree** boxes and **Next>** on the license windows.

The **Select Products to Install** window allows the selection of one product to install. For OpenCPI, the relevant choices are:

ISE WebPACK which supports small and old devices, specified at https://www.xilinx.com/publications/matrix/Software_matrix.pdf. It is useful with OpenCPI for smaller devices that are not supported by the Vivado WebPACK. Licenses are free, but required.

ISE Design Suite Logic Edition is usable with OpenCPI for any older device not supported by Vivado. It is licensed.

ISE Design Suite Embedded/DSP/System Editions have extra features not used by OpenCPI.

6.2 Installing Modelsim

Modelsim is a third-party simulator offered by Mentor, a Siemens business. Although Intel/Altera has previously provided a stripped down version of Modelsim, that version was not usable with OpenCPI since it did not support mixing VHDL and Verilog languages, which OpenCPI requires. Newer versions of Intel/Altera version of Modelsim may have lifted this restriction, but there is no support yet in OpenCPI for it in any case. The normal, purchased version of Modelsim is fully supported by OpenCPI.

To use Modelsim with OpenCPI, you need the version that runs on the Linux OS running on your development host, and also that supports both VHDL and Verilog mixed in the same design. CentOS7 is a supported development host for Modelsim.

6.2.1.1 Setting OpenCPI Environment Variables for Modelsim

To use Modelsim, you must set the following environment variables in the **`user-env.sh`** file.

- Set **`OCPI_MODELSIM_DIR`** to the Modelsim installation directory. This directory is expected to contain the **`linuxpe`** subdirectory and the **`modelsim.ini`** file.
- Set **`OCPI_MODELSIM_LICENSE_FILE`** to the pathname of the license file or the network address of the license server in the form: **`<port>@<server.ip.addr>`**

6.2.1.2 Compiling Xilinx Vivado Simulation Libraries for Modelsim

To compile Xilinx Vivado simulation libraries for Modelsim, execute the following commands:

```
vivado -mode tcl <<EOF
compile_simlib -simulator modelsim -32bit -directory XX\
               -simulator_exec_path $OCPI_MODELSIM_DIR/linuxpe
EOF
```

7 Setting up PCI Express-based HDL Platforms

Any host system that has PCI Express (PCIe) slots (most, but not all of these are x86 PCs) can host FPGA cards where the FPGA is attached directly to the PCI Express fabric. When supported by OpenCPI, these FPGA cards can be installed and used as OpenCPI ***HDL platforms*** that can host OpenCPI components built for these FPGAs.

Most PCIe-based HDL platform installations are similar:

- Install OpenCPI on the PCIe host appropriately depending on whether it will be used as a development host or a runtime only host.
- Enable the OpenCPI development environment for the FPGA card.
- Enable the OpenCPI execution environment for the FPGA card.

Some FPGA cards have their own installation steps in addition to these generic ones or they have card-specific requirements that pertain to their installation. This information is contained in the ***OpenCPI getting started guide*** for the card/platform. The next sections describe PCIe-based HDL platform installation and setup for OpenCPI as they apply to all PCI Express-based FPGA cards. When following this generic procedure, be sure to refer to the platform/card's OpenCPI getting started guide, if one exists, for platform/card-specific installation details to be applied to the steps described here.

7.1 Installing OpenCPI on the PCIe Host

OpenCPI can be installed on a PCIe host such that it functions as an OpenCPI ***development host*** or as an OpenCPI ***runtime host***. All OpenCPI-supported PCI Express-based FPGA cards require that the OpenCPI installation on the PCIe host, whether development or runtime, is complete and functional.

A PCIe host established as an OpenCPI ***development host*** has the full complement of OpenCPI tools and runtime software ready to support development and native execution (on the development system) of OpenCPI components and applications. To establish an OpenCPI development host installation, follow the instructions in the section "Installing OpenCPI on Development Hosts" in this guide. Note that since OpenCPI development-mode software is a superset of OpenCPI runtime-mode software, all development hosts are implicitly also runtime hosts.

A PCIe host established as an OpenCPI ***runtime host*** has the OpenCPI runtime software, which has been built on another development host and loaded onto the target PCIe host. After installation, the runtime PCIe host supports native execution of OpenCPI components and applications but has no OpenCPI development support: there is no **`ocpidev`** tool and there is no support for OpenCPI projects. To establish a PCIe host as an OpenCPI runtime host, follow the instructions in "Enabling OpenCPI Development for Embedded Systems" on a *separate OpenCPI development host system* to create, load, and start the bootable image for the OpenCPI runtime software on the target PCIe host. Network mode is most often deployed on PCIe runtime-only systems, but using the other OpenCPI modes of operation is also possible. After installation is complete, the PCIe host functions much like any other OpenCPI embedded system.

The PCIe system to which the FPGA card is to be connected is usually set up as both development host and runtime host. However, in some cases it may make sense to establish a different PCIe host as a runtime host and connect the FPGA card to this system instead of the development host system.

7.2 Enabling the OpenCPI Development Environment for the PCIe FPGA Card

To enable the OpenCPI development environment for the PCI Express-based FPGA card, follow the steps described in the section "Installation Steps for Platforms" in this guide on the development host:

- Install any dependencies required by the OpenCPI HDL platform for the FPGA card that have not already been installed for another platform, as defined in the "Table of Supported Platforms" in this guide.

Some FPGA cards have their own installation steps in addition to these generic ones or they have card-specific requirements that pertain to their installation. This information is contained in the ***OpenCPI getting started guide*** for the card/platform. The next sections describe PCIe card installation and setup for OpenCPI as they apply to all PCI Express-based FPGA cards. When following this generic procedure, be sure to refer to the card's OpenCPI getting started guide, if one exists, for card-specific installation details to be applied to the steps described here.

7.2.1 Installing OpenCPI on the PCIe Host

OpenCPI can be installed on a PCIe host such that it functions as an OpenCPI ***development host*** or as an OpenCPI ***runtime host***. All OpenCPI-supported PCI Express-based FPGA cards require that the OpenCPI installation on the PCIe host, whether development or runtime, is complete and functional.

A PCIe host established as an OpenCPI ***development host*** has the full complement of OpenCPI tools and runtime software ready to support development and native execution (on the development system) of OpenCPI components and applications. To establish an OpenCPI development host installation, follow the instructions in the section "Installing OpenCPI on Development Hosts" in this guide. Note that since OpenCPI development-mode software is a superset of OpenCPI runtime-mode software, all development hosts are implicitly also runtime hosts.

A PCIe host established as an OpenCPI ***runtime host*** has the OpenCPI runtime software, which has been built on another development host and loaded onto the target PCIe host. After installation, the runtime PCIe host supports native execution of OpenCPI components and applications but has no OpenCPI development support: there is no **`ocpidev`** tool and there is no support for OpenCPI projects. To establish a PCIe host as an OpenCPI runtime host, follow the instructions in "Enabling OpenCPI Development for Embedded Systems" on a *separate OpenCPI development host system* to create, load, and start the bootable image for the OpenCPI runtime software on the target PCIe host. Network mode is most often deployed on PCIe runtime-only systems, but using the other OpenCPI modes of operation is also possible. After installation is complete, the PCIe host functions much like any other OpenCPI embedded system.

The PCIe system to which the FPGA card is to be connected is usually set up as both development host and runtime host. However, in some cases it may make sense to

establish a different PCIe host as a runtime host and connect the FPGA card to this system instead of the development host system.

7.2.2 Enabling the OpenCPI Development Environment for the PCIe FPGA Card

To enable the OpenCPI development environment for the PCI Express-based FPGA card, follow the steps described in the section "Installation Steps for Platforms" in this guide on the development host:

- Install and build the OpenCPI HDL platform for the FPGA card with the **`ocpiadmin`** utility.

When following this procedure, be sure to consult the OpenCPI getting started guide for the specific card/platform (if one exists) for any details pertaining to these steps and/or any additional steps that need to be performed on the development host.

7.3 Enabling the OpenCPI Execution Environment for the PCIe FPGA Card

To enable the OpenCPI execution environment for the PCI Express-based FPGA card, follow the steps described in this section on the runtime host:

- Ensure there is sufficient power and cooling for the slots that will be used by the card.

When following this procedure, be sure to consult the OpenCPI getting started guide for the specific card/platform (if one exists) for any details pertaining to these steps and/or any additional steps that need to be performed on the development host.

7.3.1 Enabling the OpenCPI Execution Environment for the PCIe FPGA Card

To enable the OpenCPI execution environment for the PCI Express-based FPGA card, follow the steps described in this section on the runtime host:

- Configure any required jumpers and/or switches on the card.
- Plug in the FPGA card and power up the system.
- Enable bitstream loading and (usually) JTAG access.
- Load an OpenCPI bitstream into the power-up flash memory on the card.
- Reboot the system.
- Check and load the OpenCPI Linux kernel device driver.
- Test OpenCPI's ability to see the card.
- Verify the installation.

When following this procedure, be sure to consult the OpenCPI getting started guide for the specific card/platform (if one exists) for any details pertaining to these steps and/or any additional steps that need to be performed on the runtime host.

We use the term ***bitstream*** in this section for the file containing an FPGA configuration to be loaded into the FPGA. Bitstream files are created by the OpenCPI development process when the target platform is an “HDL” platform, which usually means an FPGA.

7.3.2 Ensure Sufficient Power and Cooling for the Card

PCI Express-based FPGA cards have a range of cooling and power requirements, and some even require that the chassis and box they are plugged into be left open for access to connectors etc.: they are “lab cards” that remain fully exposed. Others are typical PCI Express cards that simply plug into a slot, and the box can be closed. Frequently there are LEDs and other indicators, switches or displays that are useful to see when the card is fully exposed.

Some cards require extra power supply cables to supply more power than is available through the backplane connector (i.e. the PCIe slot). Some of these have their own “power blobs” that connect directly to AC outlets, while others have power cables that attach to the power harness in the box that usually supplies power to hard drives. For

the latter case, you may need an adapter cable for the power harness in the system box.

It is out of scope here to provide guidance on power and cooling issues, but ignoring the issues can frequently result in unreliable or broken hardware. Usually the hardware manuals of these cards provide sufficient guidance.

This step is complete when you have decided on how and where (which slot) the board should be plugged in, how it will receive sufficient cooling, and how its power supply requirements (and cables) will be satisfied.

7.3.3 Configure any Required Jumpers and/or Switches on the Card

Nearly all cards have hard-wired jumpers and switches that configure how the board should operate. For the purpose of OpenCPI, the most common options relate to how the board powers up and how the PCI Express interface behaves. For most cases, you should configure the board so that it boots a bitstream from a part of flash memory that can be written with a new bitstream, usually via JTAG. Even during active bitstream development and loading via JTAG, it is required to have a baseline OpenCPI bitstream loaded in flash memory that is automatically loaded on power up and reset.

Some PCI Express-based FPGA cards have an option to disconnect the PCI Express interface or to become the “root” of the PCI Express fabric. Neither of these options should be selected. The PCI Express interface should be a normal peripheral endpoint (both master and slave) on the fabric.

Some boards only re-load the bitstream from flash memory on power cycling, while others can/will also do it on system (PCIe fabric) reset. OpenCPI supports both configurations.

7.3.4 Plug in the Card and Power up the System

After the cards are plugged in and the system is powered up, there is usually some board-specific LEDs that show that the power is good, the PCI Express bus is alive, and perhaps a factory-default bitstream has been successfully loaded from flash. See the getting started guides for individual cards/platforms for details.

7.3.5 Enable Bitstream Loading and JTAG Access

As supported by OpenCPI, bitstreams can be loaded in one of 4 ways:

- On power-up and/or PCI Express fabric reset, from on-board flash memory.
- On command, from on-board flash memory.
- On command, from JTAG.
- On command from some other processor-accessible interface.

It is a roadmap item to support reloading a (part of) a bitstream via the PCI-Express interface itself.

If JTAG is required (which it normally is), a cable (and possibly a “JTAG pod”) must be connected to the board, and to a (usually) USB port on the PCIe host.

When more than one card is in a system, and thus multiple JTAG cables are attached to multiple USB ports, the OpenCPI system must know which JTAG cable is connected to which card. This is done by making this association in the OpenCPI system configuration file in **`$OCPI_ROOT_DIR/system.xml`**. An example is:

```
<opencpi>
  <container>
    <hdl>
      <device name="0000:05:00.0" esn="000013C1E1F401"/>
    </hdl>
  </container>
  ...(other xml)
</opencpi>
```

This example says: the OpenCPI HDL (FPGA) device found at PCIe slot named "0000:05:00.0" should be associated with the USB-based JTAG pod with the electronic serial number (ESN) of **`000013C1E1F401.`** The way to find the serial number is vendor-specific, but the OpenCPI script **`probeJtag`** will scan for JTAG cables for both Xilinx and Intel/Altera. Unfortunately, not every JTAG pod has such a serial number, and thus more than one of them cannot be used in a system. The **`lspci`** Linux utility can be used to list PCI Express slots and information about the cards that are plugged into them.

This editing of the **`system.xml`** file is only necessary when there is more than one such JTAG cable in a system.

7.3.6 Load an OpenCPI Bitstream into the Power-up Flash Memory on the Card

OpenCPI FPGA PCI-Express cards will not be discovered by OpenCPI unless they are currently running a bitstream that was created by OpenCPI. Thus part of installation is to write a bitstream to the card's flash memory. As mentioned in the configuration step above, the board should be configured to boot from this flash on power up or PCI Express reset. A good candidate for loading is the "testbias" bitstream that is built for all supported platforms. An example of loading this default bitstream on a runtime host is:

```
loadFlash alst4 \
$OCPI_CDK_DIR/alst4/ocpi.assets.testbias_alst4_base.hdl.0.alst4.bitz \
01c4b5f5
```

This command loads the flash memory of an **`alst4`** PCI Express card (the Altera Stratix4 development board, OpenCPI platform name **`alst4`**) from the OpenCPI default bitstream for the **`alst4`**, using the JTAG pod whose serial number is 01c4b5f5.

There is always also a vendor-specific method for those that are experienced users of that vendor's tools, but the actual file to load would then not be the **`.bitz`** file but rather a vendor-specific file probably built as part of OpenCPI's build flow.

It is a roadmap item to specify the existence of a PCI Express card in the system configuration file, and avoid the flash bitstream requirement. This automatic mode is currently supported only on Xilinx Zynq-based embedded processors.

7.3.7 Reboot the System

After the initial flash bitstream is installed on the card, power cycle the runtime host. At this point, the FPGA card is installed but it has not yet been verified that OpenCPI can recognize it.

7.3.8 Check and Load the OpenCPI Linux Kernel Device Driver

The OpenCPI Linux kernel device driver is used to communicate between CPUs and FPGAs. This driver is automatically loaded during runtime PCIe host setup via server, network, or standalone script execution. It is not automatically loaded, however, during development host setup. When the PCIe host has been set up as both development host and runtime host, the driver must be explicitly loaded to enable OpenCPI communication between the host and the installed PCI Express FPGA card(s).

To load the driver, use the command:

```
ocpidriver load
```

Note that this command requires **`root`** privileges/**`sudo`**. See the **`ocpidriver`** man page for command usage details.

You can manually load the Linux kernel device driver with **`ocpidriver load`** when necessary, or have it loaded at system boot time by running the **`ocpidriver load`** command in the startup script that is appropriate for the operating system in use. The driver *must* be loaded before OpenCPI applications can be run on the PCI Express FPGA card.

To check whether the Linux kernel driver is loaded, issue the command:

```
ocpidriver status
```

If the command reports that the driver is not currently loaded, issue the **`ocpidriver load`** command as described above. You should receive a message that the driver was successfully loaded.

7.3.9 Test OpenCPI's Ability to See the Card

After the HDL platform for the FPGA card is installed/built on the development host and the OpenCPI Linux kernel device driver is loaded on the runtime host, you can use the OpenCPI utility **`ocpihdl`** on the runtime host to check that OpenCPI can locate the FPGA card. The following command will search for recognizable cards and print out their information:

```
ocpihdl search
```

This command should print out the cards it could find, along with the PCI Express address for each. If you have more than one of the same card, in different slots, you will see them both, although other than the slot, there is no other identifying information.

It is a roadmap item to display the electronic serial number of the card/chip itself, although not all boards or chips have such unique identification.

7.3.10 Verify the Installation

To confirm a successful installation, run the test application on the runtime host as described in "Running a Test Application" in the section "Installation Steps for Systems After Their Platforms Are Installed", using the **`--platform`** (**`-P`**) instance option to **`ocpirun(1)`** to assign the test application "bias" worker to the HDL platform of the FPGA card.

8 Connecting HDL Platforms to an Ethernet Network

Each HDL (FPGA) platform connected to an Ethernet network must have its own unique MAC address. However, some hardware platforms supported by OpenCPI either have no built-in MAC address or their built-in MAC address is not unique to the board. In this case, you must assign a unique MAC address to each platform of this type that you plan to connect to your Ethernet network by editing the OpenCPI SD card startup script **`mynetsetup.sh`** or **`mysetup.sh`** that you create when you prepare the SD card contents for the embedded system. Open the appropriate SD card startup script with a text editor, uncomment the following lines, and then change the Ethernet address to be unique. For example:

```
ifconfig eth0 down
ifconfig eth0 hw ether 00:0a:35:00:01:24
ifconfig eth0 up
udhcpc
```

where these lines are uncommented and **`00:0a:35:00:01:24`** is the unique MAC address for the board.

9 Performing an Air-gapped Installation

These steps apply to CentOS7 systems only.

If you are starting from scratch, you may want to download the “CentOS7 everything” iso from one of the available mirrors onto a separate bootable medium. Otherwise, you should only need to download required files onto a disk or fat32-formatted USB.

Note: when copying/pasting text from this document, be sure to remove any line breaks from the copied text.

9.1 Download OpenCPI

Download a clone of the desired OpenCPI release (see the instructions for doing so in this guide) and save it on your storage device.

9.2 Download CentOS Repositories

On a network-accessible system, you'll want to install **createrepo** and **yum-utils**:

```
sudo yum install createrepo
sudo yum install yum-utils
```

The **yum** tool **createrepo** is quite important as it will allow you to bundle several **.rpm** files together into a **repomd** repository.

Now we want to create directories for a local repository. Run the following commands:

```
sudo mkdir –p /var/local/repos/
{base,centosplus,extras,updates,epel}

sudo reposync -g -l -d -m --repoid=base --newest-only
--download-metadata --download_path=/var/local/repos/
sudo reposync -g -l -d -m --repoid=centosplus --newest-only
--download-metadata --download_path=/var/local/repos/
sudo reposync -g -l -d -m --repoid=extras --newest-only
--download-metadata --download_path=/var/local/repos/
sudo reposync -g -l -d -m --repoid=updates --newest-only
--download-metadata --download_path=/var/local/repos/
sudo reposync -g -l -d -m --repoid=epel --newest-only
--download-metadata --download_path=/var/local/repos/
```

If you look inside these directories, you'll see a **Packages/** directory full of **.rpm** files.

Next, we need to turn each directory into a repository with the commands:

```
sudo createrepo /var/local/base
sudo createrepo /var/local/centosplus
sudo createrepo /var/local/extras
sudo createrepo /var/local/updates
sudo createrepo /var/local/epel
```

Looking into your local repositories, you'll see a **repodata/** directory in each one with a **repomd.xml** file as well as some compressed files.

9.2.1 Copy Repository Files

Since you're on a CentOS7 system, you can copy over the **/etc/yum.repos.d/.repo** files needed. You will need to modify them, as they are likely set to use online mirrors.

You'll need to copy over the **CentOS-Base.repo** and **epel.repo** files.

9.3 Edit Repository Files

Edit your **`.repo`** files as shown below to use local files rather than trying to look for mirrors online. You can use a placeholder path for your repository location for now.

For **`CentOS-Base.repo`**:

```
[base]
name=CentOS-$releasever - Base
#mirrorlist=http://mirrorlist.centos.org/?release=$releasever&arch=
$basearch&repo=os
baseurl=file:///<your_repo_location>
enabled=1
gpgcheck=0
#gpgkey=file:///etc/pki/rpm-gpg/RPM-GPG-KEY-CentOS-5


#released updates
[updates]
name=CentOS-$releasever - Updates
```

#mirrorlist=http://mirrorlist.centos.org/?release=$releasever&arch=

$basearch&repo=updates

```
baseurl=file:///<placeholder_path>/updates
```

enabled=1

gpgcheck=0

#gpgkey=file:///etc/pki/rpm-gpg/RPM-GPG-KEY-CentOS-5

#additional packages that may be useful

[extras]

name=CentOS-$releasever - Extras

#mirrorlist=http://mirrorlist.centos.org/?release=$releasever&arch=

$basearch&repo=extras

```
baseurl=file:///<placeholder_path>/extras
enabled=1
gpgcheck=0
#gpgkey=file:///etc/pki/rpm-gpg/RPM-GPG-KEY-CentOS-5
```

```
#additional packages that extend functionality of existing packages
[centosplus]
name=CentOS-$releasever - Plus
#mirrorlist=http://mirrorlist.centos.org/?release=$releasever&arch=
$basearch&repo=centosplus
baseurl=file:///<your_repo_location>/centosplus
enabled=1
gpgcheck=0
#gpgkey=file:///etc/pki/rpm-gpg/RPM-GPG-KEY-CentOS-5
```

For `epel.repo`:

```
[epel]
name=Extra Packages for Enterprise Linux 7 - $basearch
# It is much more secure to use the metalink, but if you wish to use
a local mirror
# place its address here.
baseurl=file:///<your_repo_location>/epel
#metalink=https://mirrors.fedoraproject.org/metalink?repo=epel-
7&arch=$basearch&infra=$infra&content=$contentdir
enabled=1
gpgcheck=0
#gpgkey=file:///etc/pki/rpm-gpg/RPM-GPG-KEY-EPEL-7

[epel-debuginfo]
name=Extra Packages for Enterprise Linux 7 - $basearch - Debug
# It is much more secure to use the metalink, but if you wish to use
a local mirror
# place its address here.
#baseurl=http://download.example/pub/epel/7/$basearch/debug
metalink=https://mirrors.fedoraproject.org/metalink?repo=epel-debug-
7&arch=$basearch&infra=$infra&content=$contentdir
failovermethod=priority
enabled=0
gpgkey=file:///etc/pki/rpm-gpg/RPM-GPG-KEY-EPEL-7
gpgcheck=1

[epel-source]
name=Extra Packages for Enterprise Linux 7 - $basearch - Source
# It is much more secure to use the metalink, but if you wish to use
a local mirror
# place it's address here.
#baseurl=http://download.example/pub/epel/7/source/tree/
```

```
metalink=https://mirrors.fedoraproject.org/metalink?repo=epel-
source-7&arch=$basearch&infra=$infra&content=$contentdir
failovermethod=priority
enabled=0
gpgkey=file:///etc/pki/rpm-gpg/RPM-GPG-KEY-EPEL-7
gpgcheck=1
```

After you've edited these files, copy over the entire **`/var/local/`** directory and the repo files into your storage device. It is the first of things you'll want to copy.

9.4 Download Python Packages

Here is the list of required Python packages:

```
python3-3.6.8-18.el7.x86_64
python3-devel-3.6.8-18.el7.x86_64
python36-jinja2-2.11.1-1.el7.noarch
python36-numpy-1.12.1-3.el7.x86_64
python36-scipy-0.18.1-2.el7.x86_64
python3-tkinter-3.6.8-18.el7.x86_64
python3-pip-9.0.3-8.el7.noarch
cmake3-3.17.5-1.el7.x86_64
fakeroot-1.26-4.el7.x86_64
ocl-icd-2.2.12-1.el7.x86_64
python36-scons-3.1.2-1.el7.noarch
dtc-1.4.6-1.el7.x86_64
1:openssl-devel-1.0.2k-24.el7_9.x86_64
perl-ExtUtils-MakeMaker-6.68-3.el7.noarch
chrpath-0.16-0.el7.x86_64
diffstat-1.57-4.el7.x86_64
texinfo-5.1-5.el7.x86_64
python36-PyYAML-3.13-1.el7.x86_64
libjpeg-turbo-devel-1.2.90-8.el7.x86_64
matplotlib-3.3.2
```

However, it is simpler to download just **`numpy`** and **`matplotlib`**, as the rest of the packages are likely already in other repositories. Gather them into a file called **`requirements.txt`** in this format:

```
Flask==0.12
requests>=2.7.0
scikit-learn==0.19.1
numpy==1.14.3
pandas==0.22.0
matplotlib==3.3.2
```

Then download them with the command:

```
pip3 download -r requirements.txt
```

Copy these contents into your storage device.

9.5 Download Prerequisites

The following table lists the needed prerequisites along with where to get them.

OpenCPI Prerequisites

Prerequisite	Location
ad9361	https://codeload.github.com/analogdevicesinc/no-OS/zip/8c52aa42e74841e3975a6f33cc5303ed2a670124
asciidoc3	https://asciidoc3.org/asciidoc3-3.2.3.tar.gz
gmp	https://ftp.gnu.org/gnu/gmp/gmp-6.2.0.tar.xz
gpsd	https://download-mirror.savannah.gnu.org/releases/gpsd/gpsd-3.18.1.tar.gz
gtest	wget https://github.com/google/googletest/archive/release-1.8.1.tar.gz
inode64	http://www.mjr19.org.uk/sw/inode64.c
iperf	https://downloads.es.net/pub/iperf/iperf-3.9.tar.gz
liquid	wget http://liquidsdr.org/downloads/liquid-dsp-1.3.1.tar.gz
lzma	https://cfhcable.dl.sourceforge.net/project/lzmautils/xz-5.2.5.tar.gz
myhostname	https://gitlab.com/opencpi/opencpi/-/blob/develop/build/prerequisites/myhostname/myhostname.c
mykonos	https://github.com/analogdevicesinc/no-OS/tarball/master
patchelf	http://nixos.org/releases/patchelf/patchelf-0.9/patchelf-0.9.tar.gz
tar	https://ftp.gnu.org/gnu/tar/tar-1.32.tar.gz
xvcd	https://codeload.github.com/RHSResearchLLC/xvcd/legacy.tar.gz/d42b07f70cffd9e53f41c33b3960e1474cfbfc04
yaml_cpp	http://sources.buildroot.net/yaml-cpp/yaml-cpp-0.6.3.tar.gz

9.5.1 Using the Cache Method

There is another “cheat” method for prerequisites where you direct the OpenCPI installation to use the cache rather than point it to a local directory. You can also use a combination of both methods. This method is also the more reliable one at the moment as some of the **git** files required may be difficult to configure.

On a network-connected system, clone the OpenCPI code base (see the instructions in this guide) and then run the following commands in your **opencpi** directory:

```
./scripts/install-packages.sh
./scripts/install-prerequisites
```

Copy the contents of **opencpi/prerequisites** and **opencpi/prerequisites-build** to where you have the other packages.

Caution: For any of the packages, in particular **`ad9361`**, be careful that when copying contents and using the cache method that the files match the **`chmod rwx`** permissions in the network connected system.

Now you have all the external items needed to install with **`opencpi-setup.sh`**.

9.6 Set up the Air-gapped System

On your air-gapped system, ensure that you have disabled secure boot in the BIOS. This affects **`ocpidriver`**, which is tested at installation.

You can now move the packages into your air-gapped system:

- Move your local repository into **`/var/local/`**
- Move your Python files into **`/var/python3.6/`**
- Move your prerequisites into **`~/prerequisites/`**

If there are existing repo files in **`/etc/yum.repos.d/`**, you can move them into a temporary directory (**`/etc/yum.repos.d/tmp/`**) and then move the two repo files into **`/etc/yum.repos.d/`**. Don't forget to modify the paths in your repo files to reflect where you placed the repositories on your system.

Finally, move your OpenCPI clone somewhere into your system and unpack it if necessary.

9.7 Set up and Install python3.6 Packages

Make sure you have **pip3** installed. We recommend installing your packages before running the OpenCPI installation scripts.

From a shell/terminal window, run the following command.

```
pip3 install --no-index --find-links /path/to/download/dir/ -r
requirements.txt
```

9.8 Set Environment Variables

As explained in **`opencpi/tools/scripts/setup-prerequisite.sh`**, there are environment variables to set to where the script should look for the prerequisites and whether to use local, cache, or internet.

In a shell/terminal window, navigate to your **`opencpi`** directory and run the following commands:

```
export OCPI_PREREQUISITES_LOCAL_PATHNAME=<path-to-prerequisites>
export OCPI_PREREQUISITES_DOWNLOAD_PATH=local:cache
```

9.9 Run OpenCPI Installation Script

If you haven't had any problems up to this point, in the shell/terminal window, navigate to your **`opencpi`** directory and run the following command:

```
./scripts/install-opencpi.sh
```

10 Glossary of Terms

This glossary provides definitions for OpenCPI-specific and industry-wide terms and acronyms used in OpenCPI documentation.

10.1 OpenCPI Terminology

This section provides definitions for terms that are specific to OpenCPI.

ACI

See ***Application Control Interface***.

adapter worker

An ***adapter worker*** is the worker used when two connected HDL workers are not connectable in some way due to different interface choices in the OWD. Adapter workers are normally inserted automatically as needed, e.g. between a worker that has a 16-bit bus and one with a 32-bit bus, or HDL workers in different clock domains. These workers are considered part of the OpenCPI framework and not created by users. See also ***worker***.

application

[noun] In the context of component-based development, an ***application*** is a composition or assembly of connected components that, as a whole, perform some useful function. See also ***component***.

[adjective] The term ***application*** can also be used to distinguish functions or code from infrastructure to support the execution of a component-based application; e.g., an HDL device worker vs. an application worker.

Application Control Interface (ACI)

The ***Application Control Interface (ACI)*** is an application launch and control API for executing XML-based OpenCPI applications within a C++ or Python program. See the chapter on the ACI in the ***OpenCPI Application Development Guide*** for more information.

application worker

An ***application worker*** is the implementation of a component used in an application, generally portable and hardware independent.

argument

See ***operation argument***.

artifact

An ***artifact*** is a file that contains executable code for one or more workers, built for a specific platform. An artifact is a binary file that results from building some assets. See the overview in the ***OpenCPI Application Development Guide*** for more information.

asset

An ***asset*** is an object that is developed for OpenCPI, including applications, components, workers, protocols, platforms, primitives and other asset types. An asset is developed in a project and is defined by an XML file.

authoring model

An ***authoring model*** is a method for creating component implementations in a specific language using a specific API between the worker and its execution environment. An authoring model represents a particular way to write the source code and XML for a worker and is usually associated with a class of processors and a set of related languages. Existing models include RCC, HDL and OCL. See the chapter on authoring models in the ***OpenCPI Component Development Guide*** for more information.

back pressure

Back pressure is the resistance or force opposing the desired flow of data through an application. Back pressure within an OpenCPI system is a common occurrence that happens when worker output is temporarily not possible due to processing or communication congestion from whatever the output is connected to. Back pressure can be the result of resource-loading issues or passing data between containers.

build configuration

A ***build configuration*** is a set of parameter property (compile-time) values to use when building a worker. See the chapter on worker build configuration XML files in the ***OpenCPI Component Development Guide*** for more information.

CDK

See ***Component Development Kit***

component

A ***component*** is the interface "contract" specified by an OpenCPI Component Specification (OCS) and implemented by a worker. A component performs a well-defined function regardless of implementation. A component has ports and properties. See the chapter on component specifications in the ***OpenCPI Component Development Guide*** for more information.

Component Development Kit (CDK)

The OpenCPI ***Component Development Kit (CDK)*** is the set of OpenCPI tools, scripts, documents, and libraries used for developing components, workers and other assets in projects.

component library

A ***component library*** is a collection of component specifications, workers and test suites that can be built, exported, and installed to support applications. See the chapter on component libraries in the ***OpenCPI Component Development Guide*** for more information.

component specification

See ***OpenCPI Component Specification (OCS)***.

component unit test suite

A ***component unit test suite*** is a collection of test cases in a component library for testing all the workers in the library that implement the same spec (OCS) across all available platforms. The workers that are tested can be written to different authoring models or languages or simply be alternative source code implementations of the same spec. The OpenCPI unit test framework manages multiple dimensions of worker testing, with automation to minimize test design and preparation efforts. See the chapter on worker unit testing in the ***OpenCPI Component Development Guide*** for more information.

configuration property

A ***configuration property*** is a writeable and/or readable value specified in the OCS or OWD that enables control software to control and monitor a worker. Configuration properties (usually abbreviated to ***properties***) are logically the knobs and meters of the worker's "control panel". Each worker may have its own, possibly unique, set of configuration properties which can include hardware resources such registers, memory, and state. Properties can be specified as compile time or runtime. See the chapter on property syntax and ranges in the ***OpenCPI Component Development Guide*** for more information. See also ***configuration property accessibility***.

configuration property accessibility

Configuration property accessibility is the set of declarations within an OCS or OWD that indicate when it is valid to read from or write to a configuration property. The various accessibility attributes (defined in the ***OpenCPI Component Development Guide***) establish the rules in relation to the worker's lifecycle and may declare the property as fixed at build time (see ***parameter***).

container

A ***container*** is the OpenCPI infrastructure element that "contains," manages, and executes a set of workers. Logically, the container "surrounds" the workers, mediating all interactions between the workers and the rest of the system. A container typically provides the OpenCPI runtime environment for a particular processor in the system. See the section on the RCC worker interface in the ***OpenCPI RCC Development Guide*** for more information on RCC containers. See the section on HDL container XML files in the ***OpenCPI HDL Development Guide*** for more information on HDL containers.

control-application

A ***control-application*** is the conventional application (e.g. "main program") that constructs and runs component-based applications. See the chapter on the ACI in the ***OpenCPI Application Development Guide*** for more information.

control interface

The ***control interface*** is the interface as seen by HDL worker code that an HDL container uses to provide (at a minimum) a control clock and associated reset into the worker, convey life cycle control operations like ***initialize***, ***start*** and ***stop***, and access the worker's configuration properties as specified in the OCS and OWD. See the section on the control interface in the ***OpenCPI HDL Development Guide*** for more information.

control operations

Control operations are a fixed set of operations that every worker may have. These operations implement a common control model that allows all workers to be managed without having to customize the management infrastructure software for each worker, while configuration properties are used to specialize components. The most commonly used control operations are "start" and "stop". See the section on lifecycle control operations in the ***OpenCPI Component Development Guide*** for more information.

control plane

In OpenCPI, the ***control plane*** is the control and configuration infrastructure for runtime lifecycle control and configuration of worker instances throughout the system at runtime. See the control plane introduction in the ***OpenCPI Component Development Guide*** for more information.

control software

Control software is the software that launches and controls OpenCPI applications, either the standard OpenCPI utility **`ocpirun`** or custom C++ or Python programs that perform the same function embedded inside them using the Application Control Interface application launch and control API. See the control plane introduction in the ***OpenCPI Component Development Guide*** for more information. See also ***control-application***.

Control software generally launches, configures and controls an application and the runtime workers that make up the application. A proxy, meanwhile, is a worker within an application that can control and configure other workers. See the ***OpenCPI RCC Development Guide*** for more information. See also ***device proxy worker***.

core project

The OpenCPI ***core project*** is a built-in OpenCPI project (package ID **`ocpi.core`**) that contains the minimum set of workers and infrastructure VHDL for OpenCPI framework operation on software and FPGA simulators.

data interface

A ***data interface*** is the set of ports defined in a worker's OCS that convey data, message boundaries, opcodes and EOF into and out of the worker and which implement flow control.

data plane

In OpenCPI, the ***data plane*** is a data-passing infrastructure that allow workers of all types to consume/produce data from/to other workers in an application regardless of the container on which the workers are executing in (or the processor on which they are executing). See the data plane introduction in the ***OpenCPI Component Development Guide*** for more information.

device proxy worker

A ***device proxy worker*** is a software worker (RCC/C++) that is specifically paired with one or more HDL device workers in order to translate a higher-level control interface for a class of devices into the lower-level actions required on a specific device. See the section on controlling slave workers from proxies in the ***OpenCPI RCC Development Guide*** and the section on device support for FPGA platforms in the ***OpenCPI Platform Development Guide*** for more information.

device worker

See ***HDL device worker***.

Digital Radio Controller (DRC)

The ***Digital Radio Controller (DRC)*** is a utility component in the OpenCPI built-in core project that is used when an application needs to use radio hardware in the system to control it and to stream sample data to and from it. The “digital radio” functionality in a system usually has antennas for transmitting and receiving RF signals and channels which convert the RF signals to and from baseband digital samples that are produced and consumed by the application. See the section on utility components for applications in the ***OpenCPI Application Development Guide*** for more information.

HDL assembly

An ***HDL assembly*** is a fixed composition of connected HDL workers that are built into a complete FPGA bitstream that can be executed on an FPGA to implement some or all of the components of an OpenCPI application. The HDL code is automatically generated from the HDL assembly’s OHAD. See the chapter on preparing HDL assemblies for use by applications in the ***OpenCPI Application Development Guide*** and the ***OpenCPI HDL Development Guide*** for more information.

HDL authoring model

The ***HDL authoring model*** is the authoring model used by VHDL-language and less-supported Verilog-language workers that execute on FPGAs. See the ***OpenCPI HDL Development Guide*** for information about using this authoring model. See also ***Hardware Description Language (HDL)***.

HDL build hierarchy

The ***HDL build hierarchy*** is the structure in which FPGA bitstreams are created from other assets. See the section about the HDL build hierarchy in the ***OpenCPI HDL Development Guide*** for more information.

HDL build process

The ***HDL build process*** is the process of building HDL assets for different target devices and platforms. See the chapter on building HDL assets in the ***OpenCPI HDL Development Guide*** for more information.

HDL card

An ***HDL card*** is hardware that contains devices and plugs into a slot on an HDL platform. Devices are either directly attached to the pins on an HDL platform or attached to cards that plug into compatible slots on the platform. Devices on a card are considered to be part of the card, which can be plugged into a certain type of slot on any platform, rather than part of the platform itself. See the sections on device support for FPGA platforms and defining cards that contain devices that plug in to platform slots in the ***OpenCPI Platform Development Guide*** for more information.

HDL data interface

An ***HDL data interface*** is the set of ports defined in an HDL worker's OCS that convey data, message boundaries, opcodes and EOF into and out of the HDL worker and which implement flow control. Worker data ports can be implemented as stream or message interfaces. Stream interfaces are FIFO-like with extra qualifying bits along with the data indicating message boundaries, byte enables and EOF. Message interfaces are based on addressable message buffers. See the section on HDL worker data interfaces for OCS data ports in the ***OpenCPI HDL Development Guide*** for more information.

HDL device emulator worker

An ***HDL device emulator worker*** is a special type of HDL device worker that acts like a device for test purposes. A device emulator worker provides a mirror image of an HDL device worker's external signals so that it can can emulate the device in simulation. See the section on testing device workers with emulators in the ***OpenCPI Platform Development Guide*** for more information.

HDL device worker

An ***HDL device worker*** is a specific type of HDL worker designed to support a specific external device attached to an FPGA such as an ADC, flash memory, or I/O device. HDL device workers are typically developed as part of enabling an HDL platform. See the chapter on device support for FPGA platforms in the ***OpenCPI Platform Development Guide*** for more information.

HDL interface

(1) For HDL workers, an ***HDL interface*** is the set of input and output port signals that correspond to a high-level OpenCPI port as defined in the OCS and OWD for the HDL worker. An HDL worker has a control interface (for the implicit control port), data interfaces (for the explicit data ports defined in the OCS), and service interfaces (for service ports as defined in the HDL worker's OWD).

(2) For all worker types, an ***HDL interface*** is the implicit control port.

HDL platform

An ***HDL platform*** is an OpenCPI platform based on an FPGA that is enabled to host OpenCPI HDL workers. An HDL simulator is also considered to be an HDL platform. See the chapter on enabling FPGA platforms in the ***OpenCPI Platform Development Guide*** for more information.

HDL platform configuration

An ***HDL platform configuration*** is a pre-built (usually pre-synthesized) assembly of HDL device workers that represents a particular reusable configuration of device support modules for a given HDL platform. The HDL code is automatically generated from a brief description in XML. See the section on enabling execution for FPGA platforms in the ***OpenCPI Platform Development Guide*** for more information.

HDL platform worker

An ***HDL platform worker*** is a specific type of HDL worker that enables an HDL platform for use with OpenCPI and provides the infrastructure for implementing control/data interfaces to devices and interconnects external to an FPGA or simulator, such as PCIe or clocks. See the section on enabling execution for FPGA platforms in the ***OpenCPI Platform Development Guide*** for more information.

HDL primitive

An ***HDL primitive*** is an HDL asset that is lower level than a worker and can be used (and reused) as a building block for HDL workers. An HDL primitive is either a library or a core. See the chapter on HDL primitives in the ***OpenCPI HDL Development Guide*** for more information.

HDL primitive core

An ***HDL primitive core*** is a low-level module that can be built and/or synthesized from source code, or imported as pre-synthesized and possibly encrypted from third parties, or generated by tools like Xilinx CoreGen or Intel/Altera MegaWizard. An HDL worker declares which primitive cores it requires (and instantiates). See the chapter on HDL primitives in the ***OpenCPI HDL Development Guide*** for more information.

HDL primitive library

An ***HDL primitive library*** is a collection of low-level modules compiled from source code that can be referenced in HDL worker code. An HDL worker declares the HDL primitive libraries from which it draws modules. See the chapter on HDL primitives in the ***OpenCPI HDL Development Guide*** for more information.

HDL slot

An ***HDL slot*** is an integral part of an HDL platform that enables an HDL card to be plugged in so that its attached devices are accessible to the platform. An HDL platform has defined slot types; HDL cards that are designed for the same slot type can be plugged in to the defined slots on the platform. See the sections on device support for FPGA platforms and defining cards that contain devices that plug in to platform slots in the ***OpenCPI Platform Development Guide*** for more information.

HDL subdevice worker

An OpenCPI ***HDL subdevice worker*** is a special type of HDL application worker that supports an HDL device worker defined in another library. See the section on subdevice workers in the ***OpenCPI Platform Development Guide*** for more information.

HDL worker

An ***HDL worker*** is an HDL implementation of a component specification with the source code (for example, VHDL) written according to the HDL authoring model. An HDL worker is usually a hardware-independent, portable application worker but can alternatively be an HDL device worker that controls a specific piece of hardware attached to an FPGA. See the chapter on the HDL worker in the ***OpenCPI HDL Development Guide*** for more information.

lifecycle state model

The OpenCPI ***lifecycle state model*** specifies the control states each worker may be in and the control operations which generally change the state a worker is in, effecting a state transition. See the section on the lifecycle state model in the ***OpenCPI Component Development Guide*** for more information.

OAS

See ***OpenCPI Application Specification***.

OCS

See ***OpenCPI Component Specification***.

OHAD

See ***OpenCPI HDL Assembly Description***.

OHPD

See ***OpenCPI HDL Platform Description***.

opcode

See ***operation code***.

OpenCL (OCL) authoring model

The ***OpenCL (OCL) authoring model*** is the authoring model used by Open Computing Language (OpenCL) (C subset/superset)-language workers usually executing on graphics processors. See the ***OpenCPI OCL Development Guide*** for more information. This OpenCPI authoring model is currently experimental.

OpenCPI Application Specification (OAS)

An ***OpenCPI Application Specification (OAS)*** is an XML document that describes the collection of components along with their interconnections and configuration properties that defines an OpenCPI application. See the chapter on OpenCPI application specification XML documents in the ***OpenCPI Application Development Guide*** for more information.

OpenCPI Component Specification (OCS)

An ***OpenCPI Component Specification (OCS)*** is an XML file that describes both configuration properties and zero or more data ports (referring to protocol specifications) of a component, establishing interface requirements for multiple implementations (workers) in any authoring model. Also referred to as a *component spec* or *spec file*. See the chapter on component specifications in the ***OpenCPI Component Development Guide*** for more information.

OpenCPI HDL Assembly Description (OHAD)

An ***OpenCPI HDL Assembly Description (OHAD)*** is an XML file that describes an HDL assembly. See the chapter on HDL assemblies for creating and executing FPGA bitstreams in the ***OpenCPI HDL Development Guide*** for more information.

OpenCPI HDL Platform Description (OHPD)

An ***OpenCPI HDL Platform Description (OHPD)*** is an XML file that describes an HDL platform. An OHPD contains the same information as the OWD for the HDL platform worker and also describes the devices (controlled by HDL device workers) that are attached to the HDL platform and available for use. See the section on enabling execution for FPGA platforms in the ***OpenCPI Platform Development Guide*** for more information.

OpenCPI Protocol Specification (OPS)

An ***OpenCPI Protocol Specification (OPS)*** is an XML file that describes the allowable data messages (operation codes) and payloads (operation arguments) that may flow between the ports of components. See the chapter on protocol specifications in the ***OpenCPI Component Development Guide*** for more information.

OpenCPI System support Project (OSP)

An ***OpenCPI System support Project (OSP)*** is an OpenCPI project that contains OpenCPI assets whose purpose is to enable and test a particular system (of platforms) to be used by OpenCPI. OSPs fulfill what is generally meant by the more generic industry term: Board Support Package. An OSP may contain assets to support multiple related systems. See also ***Board Support Package (BSP)***.

OpenCPI Worker Description (OWD)

An ***OpenCPI Worker Description (OWD)*** is an XML file that describes the worker and references the component specification it is implementing. See the chapter on worker descriptions in OWD files in the ***OpenCPI Component Development Guide*** for more information.

operation argument

An ***operation argument*** is one of the data values in the payload data defined for a particular operation (message type) within a protocol specification whose type information is determined by the protocol XML.

operation (within a protocol)

An ***operation*** is a message type encapsulating zero or more operation arguments within an OpenCPI protocol specification.

operation code (opcode)

An ***operation code*** (***opcode***) is an ordinal that indicates which of the possible operations in a protocol is present.

OPS

See ***OpenCPI Protocol Specification***.

OSP

See ***OpenCPI System support Project***.

OWD

See ***OpenCPI Worker Description***.

package ID

A ***package ID*** is the globally-unique identifier of an OpenCPI asset. A project's package ID is used when it is depended on by other projects. A component's package ID is used to reference it in applications or workers. While all assets have package IDs (either explicitly specified or inferred from the directory structure), only certain assets are currently identified by their package IDs. See the section on package IDs in the ***OpenCPI Component Development Guide*** for more information.

parameter

A ***parameter*** is an immutable configuration property that is set at build time, allowing software compilers and hardware compilers to optimize accordingly. See the sections on properties that are build-time parameters, property accessibility attributes, and the parameter attribute of property elements and SpecProperty elements in the ***OpenCPI Component Development Guide*** for more information.

platform

An OpenCPI ***platform*** is a particular type of processing hardware and/or software that can host a container for executing OpenCPI workers based on artifacts. Platforms may be based on CPUs, GPUs or FPGAs. See the chapter on OpenCPI systems and platforms in the ***OpenCPI Platform Development Guide*** for more information.

platform worker

See ***HDL platform worker***.

port

An OpenCPI ***port*** is an interface of a component that allows it to communicate with other components using a protocol. Ports are unidirectional: input or output, consumer or producer. In OpenCPI, a port is a high-level data flow interface in and out of all types of workers. In the VHDL and Verilog languages, however, a "port" refers to the individual signals (of any type) that are the inputs and outputs of an entity (VHDL) or module (Verilog). See the chapter on component specifications in the ***OpenCPI Component Development Guide*** for more information.

port readiness

Port readiness indicates whether an input port has data to be consumed or an output port has capacity to produce data (e.g. no back pressure). Input ports are ready when there is message data present that has not yet been consumed by the worker. Output ports are ready when buffers are available into which they may place new data.

project

An OpenCPI ***project*** is a work area (and directory) in which to develop OpenCPI components, libraries, applications, and other platform- and device-oriented assets. See the chapter on developing OpenCPI assets in projects in the ***OpenCPI Component Development Guide*** for more information.

project registry

An OpenCPI ***project registry*** is a directory that contains references to projects in a development environment so they can refer to (and depend on) each other. Development activity takes place in the context of a project registry that specifies available projects to use. See the section on the project registry in the ***OpenCPI Component Development Guide*** for more information.

property

See ***configuration property***.

protocol specification

See ***OpenCPI Protocol Specification (OPS)***.

protocol summary

A ***protocol summary*** is the set of summary attributes, whether inferred from the messages specified for the protocol, or specified directly as attributes of the protocol, and indicates the basic behavior of a port using a protocol. A protocol summary can also be present when messages are specified, and can override the attributes inferred from the message specifications. See also ***Component Development Kit (CDK)***.

RCC, RCC authoring model

See ***Resource-Constrained C (RCC) authoring model***.

Resource-Constrained C (RCC) authoring model

The ***Resource-Constrained C (RCC) authoring model*** is the authoring model used by C or C++ language workers that execute on General-Purpose Processors (GPPs). The "Resource Constrained" prefix indicates that the environment may be constrained to use a limited set of library calls; see the ***OpenCPI RCC Development Guide*** for more information.

registry

See ***project registry***.

RCC worker

An ***RCC worker*** is an RCC implementation of an OpenCPI component specification with the source code (for example, C++ or Python) written according to the RCC authoring model. An RCC worker can act as a device proxy worker. See the ***OpenCPI RCC Development Guide*** for more information.

run condition

A ***run condition*** is the specification by an RCC worker as to when it should execute, based on a combination of port readiness and/or some amount of time having passed. The commonly-used default run condition is when all ports are ready, with no consideration of time passing.

run method

A ***run method*** is a non-blocking software method that is executed when a worker's run condition is satisfied, as determined by its container.

spec file

Spec file (and *component spec*) is shorthand notation for an OpenCPI Component Specification file.

SpecProperty

A ***SpecProperty*** is an XML element that adds a worker-specific attribute to a configuration property already defined in the component spec. See the section on worker descriptions in OWD XML files in the ***OpenCPI Component Development Guide*** for more information.

system

In OpenCPI, a ***system*** is a collection of platforms usually in a box or on a system bus or fabric.

target

An OpenCPI ***target*** is the entity for which an asset should be built (compiled, synthesized, place-and-routed, etc.) In OpenCPI, build targets are usually platforms (particular products or particular operating system releases and architectures). When a set of platforms shares a common processor architecture family, it is *sometimes* possible to build for the "family" and the results of that build can be used for all the platforms. See the section on RCC compilation and linking options in the ***OpenCPI RCC Development Guide*** and the section on HDL build targets in the ***OpenCPI HDL Development Guide*** for more information.

worker

An OpenCPI ***worker*** is a specific implementation (and possibly a runtime instance) of a component specification with the source code written according to an authoring model. See the introductory chapter on workers in the ***OpenCPI Component Development Guide*** for more information.

worker property

A ***worker property*** is a configuration property related to a particular implementation (design) of a worker; that is, one that is not necessarily common across a set of implementations of the same high-level component specification (OCS). A worker property is additional to the properties defined by the component specification being implemented. See the section on how a worker access its properties in the ***OpenCPI RCC Development Guide*** and the sections on property access and property data types in the ***OpenCPI HDL Development Guide*** for more information.

unit test

See ***component unit test suite***.

Zero-Length Message (ZLM)

A ***Zero-Length Message (ZLM)*** is a data payload with no operation arguments present when a protocol specification specifies such an operation code with no data fields.

10.2 Industry Terminology

This section provides definitions for industry-wide terms relating to OpenCPI.

Advanced eXtensible Interface (AXI)

> ***Advanced eXtensible Interface (AXI)*** is an industry-standard bus used by ARM processors.

Advanced RISC Machine (ARM)

> ***Advanced RISC Machine (ARM)*** is a widely-used processor architecture originally based on a 32-bit reduced instruction set (RISC) computer.

ARM

> See ***Advanced RISC Machine***.

AXI

> See ***Advanced eXtensible Interface***.

Board Support Package (BSP)

> A ***Board Support Package (BSP)*** is the layer of software in an embedded system that contains hardware-specific drivers and other routines that allow a particular operating system (usually a real-time operating system) to function in a particular hardware environment integrated with the operating system itself. An OpenCPI System support Project (OSP) performs the function of a BSP in OpenCPI.

Central Processing Unit (CPU)

> A ***Central Processing Unit (CPU)*** is the electronic circuitry that executes instructions comprising a computer program. A CPU performs basic arithmetic, logic, controlling, and input/output (I/O) operations specified by the instructions in the program, in contrast with external components such as main memory and I/O circuitry and specialized processors such as Graphics Processing Units (GPUs).

CPU

> See ***Central Processing Unit***.

Digital Signal Processor (DSP)

> A ***Digital Signal Processor (DSP)*** is a specialized microprocessor chip with an architecture that is optimized for the operational needs of digital signal processing.

digital signal processing

> ***Digital signal processing*** (also abbreviated to “DSP”) is the use of digital processing by General-Purpose Processors (GPPs) or Digital Signal Processors (DSPs) to perform a wide variety of signal processing operations. The digital signals processed in this way are a sequence of numbers that represent samples of a continuous variable in a domain such as time, space, or frequency.

DSP

See ***Digital Signal Processor***. This acronym is sometimes also used more generically for digital signal processing as a class of computational algorithms.

eXtensible Markup Language (XML)

eXtensible Markup Language (XML) is a standardized markup language that defines a set of rules for encoding documents in a format which is both human- and machine-readable.

Field-Programmable Gate Array (FPGA)

A ***Field-Programmable Gate Array (FPGA)*** is an integrated circuit that is designed to be configured by a customer or a designer after manufacturing. The FPGA configuration is generally specified using a hardware description language (HDL), similar to that used for an Application-specific Integrated Circuit (ASIC).

FPGA

See ***Field-Programmable Gate Array***.

FPGA bitstream

In the context of FPGA development, an ***FPGA bitstream*** is a single, standalone artifact, resulting from building an HDL assembly, that is ready for loading onto an actual, physical FPGA.

framework

A ***framework*** is a development and runtime tool set for a particular class of software, firmware, or gateware development. OpenCPI is a framework.

gateware

Gateware is source code written in an HDL for an FPGA. Gateware is like software because it is fully programmable, but it compiles to fully parallel logic, which allows it to to compute efficiently like hardware. Gateware solutions achieve performance and flexibility by running on FPGAs.

General-Purpose Processor (GPP)

A ***General-Purpose Processor (GPP)*** is a processor designed for general-purpose computers such as PCs or workstations and for which computation speed is the primary concern. See also ***Central Processing Unit (CPU)***.

GPP

See ***General-Purpose Processor***.

GPU

See ***Graphics Processing Unit***.

Graphics Processing Unit (GPU)

A ***Graphics Processing Unit (GPU)*** is a chip or electronic circuit capable of rendering graphics for display on an electronic device. In the last decade, GPUs have also been used for more general-purpose computing when algorithms can exploit the same highly parallel architectures.

Hardware Description Language (HDL)

Hardware Description Language (HDL) is a specialized language used to program the structure design and operation of digital logic circuits. In OpenCPI, it is an authoring model using the VHDL language and is targeted at FPGAs. HDL workers should be developed according to the HDL authoring model described in the ***OpenCPI HDL Development Guide***.

HDL

See ***Hardware Description Language***.

Integrated Synthesis Environment (ISE®) Design Suite

The Xilinx ***Integrated Synthesis Environment (ISE) Design Suite*** is a discontinued software tool for synthesis and analysis of HDL designs, which primarily targets development of embedded firmware for Xilinx FPGA and Complex Programmable Logic Device (CPLD) integrated circuit (IC) product families. Use of the last released edition continues for in-system programming of legacy hardware designs containing older FPGAs and CPLDs otherwise orphaned by the replacement design tool, Vivado® Design Suite.

ISE® Simulator (Isim)

The ***ISE Simulator (ISim)*** is the HDL simulator provided with the Xilinx ISE® Design Suite. In OpenCPI, this simulator is called the **`isim`** HDL platform.

isim

See ***Integrated Synthesis Environment (ISE®) Simulator (ISim)***.

OCL, OpenCL

See ***Open Computing Language***.

Open Computing Language (OCL, OpenCL)

The ***Open Computing Language (OCL, OpenCL)*** is a language and runtime for writing programs that, subject to the availability of appropriate tools, may execute on different types of processors, e.g. Central Processing Units (CPUs), Graphics Processing Units (GPUs), Digital Signal Processors (DSPs), Field-Programmable Gate Arrays (FPGAs) and other processors or hardware accelerators. OpenCL is an open standard maintained by the non-profit technology consortium Khronos Group.

OSS

See ***Open Source Software***.

Open Source Software (OSS)

Open Source Software (OSS) is a type of computer software in which source code is released under a license in which the copyright holder grants users the rights to use, study, change, and distribute the software to anyone and for any purpose. Open Source Software may be developed in a collaborative public manner.

PCI

See ***Peripheral Component Interconnect***.

PCIe

See ***Peripheral Component Interconnect Express***.

Peripheral Component Interconnect (PCI)

Peripheral Component Interconnect (PCI) is a local computer bus for attaching hardware devices in a computer and is part of the PCI Local Bus standard. The PCI bus supports the functions found on a processor bus but in a standardized format that is independent of any given processor's native bus. Devices connected to the PCI bus appear to a bus master to be connected directly to its own bus and are assigned addresses in the processor's address space. PCI is a parallel bus, synchronous to a single bus clock. Attached devices can take either the form of an integrated circuit fitted onto the motherboard (called a planar device in the PCI specification) or an expansion card that fits into a slot.

Peripheral Component Interconnect Express (PCIe)

Peripheral Component Interconnect Express (PCIe) is a high-speed serial computer expansion bus standard that is designed to replace the older PCI, PCI-X and AGP bus standards. Improvements over the older standards include higher maximum system bus throughput, lower I/O pin count and smaller physical footprint, better performance scaling for bus devices, a more detailed error detection and reporting mechanism and native hot-swap functionality. More recent revisions of the PCIe standard provide hardware support for I/O virtualization.

RPM

See ***RPM Package Manager***.

RPM Package Manager (RPM)

The ***RPM Package Manager (RPM)*** is a free and open-source package management system used in some Linux distributions. The name “RPM” refers to `.rpm` file format and to the Package Manager program command itself.

System on a Chip (SoC)

A ***system on a chip (SoC)*** is a single integrated circuit (IC, or “chip”) that integrates all or most components of a computer or other electronic system. SoC is a complete electronic substrate system that may contain analog, digital, mixed-signal or radio frequency functions. Its components usually include a Graphics Processing Unit (GPU), a Central Processing Unit (CPU) that may be multi-core, and system memory (RAM). SoCs are in contrast to the common traditional motherboard-based PC architecture, which separates components based on function and connects them through a central interfacing circuit board. SoCs used with OpenCPI typically also contain FPGAs.

Verilog

Verilog is a hardware description language (HDL) used to model electronic systems. Verilog is standardized as IEEE 1364.

VHSIC Hardware Description Language (VHDL)

VHDL is a hardware description language used in electronic design automation to describe digital and mixed-signal systems such as FPGAs and integrated circuits (ICs). VHDL can also be used as a general-purpose parallel programming language.

Vivado® Design Suite (Vivado, Xilinx Vivado)

The Xilinx ***Vivado Design Suite*** is a software suite for synthesis and analysis of HDL designs. Vivado is an integrated design environment (IDE) with system-to-IC level tools built on a shared scalable data model and a common debug environment. Vivado supersedes Xilinx ISE with additional features for system on a chip development and high-level synthesis. Vivado WebPACK Edition is a free version of Vivado that provides designers with a limited version of the Vivado Design Suite environment.

Vivado® Simulator

Vivado Simulator is an HDL event-driven simulator that Xilinx provides with Vivado Design Suite and WebPACK Edition. In OpenCPI, this simulator is called the **`xsim`** HDL platform.

XML

See ***eXtensible Markup Language***.

Xsim

See ***Vivado® Simulator***.