WIND™

AN INTEL COMPANY

# WIND RIVER®
# PULSAR™ LINUX

## SOFTWARE DEVELOPMENT GUIDE

8

**Corporate Headquarters**

Wind River
500 Wind River Way
Alameda, CA 94501-1153
U.S.A.
Toll free (U.S.A.): +1-800-545-WIND
Telephone: +1-510-748-4100
Facsimile: +1-510-749-2010

For additional contact information, see the Wind River website:

www.windriver.com

For information on how to contact Customer Support, see:

www.windriver.com/support

*Pulsar™ Linux*

*Software Development Guide, 8*

7 June 2017

# Contents

# 1

# Introduction

Wind River Pulsar Linux is an application-ready Linux distribution that has been certified on select hardware. Pulsar Linux is designed to be simple to install so you can start developing in minutes.

Pulsar Linux is a small, high-performance, secure, and manageable image that is certified and bundled with COTS platforms. It includes select packages from the traditional Wind River Linux development environment and its market-specific profiles.

### Features

- Open source Linux run-time tools.
- Support for multiple architectures.
- Ready for application development and equipped with device lifecycle management.
- Preinstalled binary image for quick prototyping.
- Access to security updates and other critical Linux updates.
- Extensible through the use of package feeds. Packages can be added and built directly on the device.
- Built-in container support for application middleware abstraction. Any application from any ecosystem can run on any device, even applications with their own middleware requirements.

### Benefits

- Immediately start developing applications within minutes from powering up the device.
- Achieve faster time to market with an application-ready platform (smaller learning curve).
- Easy-to-use and secure device update mechanism using the Wind River repository, a trusted host.
- Mitigate risk by protecting devices from security threats.
- Gain performance with an ecosystem that allows application stacks to be built once and able to run anywhere.

# 2

# Application Development Overview



Pulsar Linux is an application-ready platform so it comes with tools you can use to develop and run Linux applications on a variety of devices, from carrier grade to traditional embedded and many types in between.

The GNU toolchain is available with Pulsar Linux. Among other ancillary tools and libraries, the toolchain includes the GNU Compiler Collection (GCC) compiler system. For complete documentation on GCC, refer to the project's website at the following URL:

https://gcc.gnu.org/

### Native Software Development Tools

Application debugging tools are also available with Pulsar Linux. They are the following:

- **gdb** — The GNU Project Debugger (GDB) to debug C and C++ applications. For complete documentation on GDB, refer to the project's website at the following URL:

  http://www.gnu.org/software/gdb/

- **gdbserver** — A helper application that enables you to run GDB on a machine other than the device on which the program to debug is running.
- **strace** — A system call tracer. It provides a trace of all system calls made by a specified program.
- **tcf-agent** — A Target Connection Framework (TCF) agent to support debug target connections from Wind River Workbench, an Eclipsed-based IDE.
- **wrproxy** — A network proxy agent for Wind River Workbench.

It is recommended that you check regularly whether updates are available for install from the Wind River repository. For details, refer to the topic *Wind River Pulsar Linux System Administration Guide: Updating the Device*.

#### Helix App Cloud Integrated Development Environment

Wind River provides the Helix App Cloud, a cloud-based integrated development environment, for use with Pulsar Linux. You can use the Helix App Cloud to develop applications, debug C, C++, and Node.js with access to remote device hardware targets.

To use the Helix App Cloud, you must create an account and register your hardware device. For additional information, see Registering a Device for Use on the Wind River Helix App Cloud on page 6.

#### Wind River Software Development Kit (SDK)

Wind River provides a SDK for developing applications on a Linux host. This lets you develop independently from a live device in a cross-compiled environment and helps prevent extended downtime by keeping the development and deployment environments separate.

In addition, the SDK includes the Pulsar Linux **rpm-tool**, which helps simplify many aspects of package creation and signing.

To use the SDK, you must have a compatible Linux host with network and Internet access, and a copy of the SDK. For additional information, see Application Development with the SDK Overview on page 9.

## Native Application Development

You can develop applications directly on the device instead of working in a cross-compiling environment.

There are numerous ways in which to develop applications. The Pulsar ecosystem provides best-in-class Linux tools to assist you with this.

Source code for sample applications are available in the Wind River repository for your reference. To refer to them, install the **gcc-example** package. The package group includes the following applications:

- **hello-Linux** — A simple C application.
- **hello-cxx-Linux** — A simple C++ application.
- **mthread** — An application that creates multiple pthreads (POSIX threads).
- **clientserver** — A project that creates two applications, a socket client and a socket server.

For additional information on developing applications on your Pulsar Linux device, see *Wind River Pulsar Linux Tutorial: Native Application Development: Tutorial Overview*.

# Building Source RPMs

Build and install an application from a source RPM (SRPM) package.

Before building an SRPM package, consider the burden it may place on you. You will have to maintain it and build it every time updates such as security updates become available.

Many SRPM packages do build and run successfully when built for Pulsar Linux. However, the chance of success in building an SRPM package largely depends on how close the original environment is to Pulsar Linux. There are a number of reasons for possible failures such as incompatible library versions, kernel differences, missing libraries, and so forth that might affect the build and run-time behavior.

SRPM packages, in general, have a slightly different directory model, package feature division (and, thus, different dependency trees), and different default users. In this case, it may be better to find a respective package in the Yocto Project or Debian/Ubuntu communities.

### Prerequisites

Your device has been recently updated. For details, refer to *Wind River Pulsar Linux System Administration Guide: Updating the Device*.

### Procedure

1. Download the desired SRPM package.

   The following example downloads the SRPM for Lynx, a text-based Web browser. It also downloads the CentOS GNU Privacy Guard (GPG) public key as it was available on the Web:

   ```
   # wget http://vault.centos.org/7.1.1503/os/Source/\
   SPackages/lynx-2.8.8-0.3.dev15.el7.src.rpm
   # wget http://mirror.centos.org/centos/RPM-GPG-KEY-CentOS-7
   # rpm --import RPM-GPG-KEY-CentOS-7
   ```

2. If applicable, verify the authenticity of the package for security purposes.

   The following example uses the CentOS GPG public key to verify the SRPM package has not been compromised with computer viruses or other security threats:

   ```
   # gpg --quiet --with-fingerprint ./RPM-GPG-KEY-CentOS-7
   ```

3. Install the dependency packages for the downloaded SRPM package.

   The following example installs the dependency packages for the Lynx package from the Wind River repository:

   ```
   # smart builddep lynx-2.8.8-0.3.dev15.el7.src.rpm --yes
   ```

   During installation, the package manager will ask you to confirm the install request. By using the **--yes** option, you confirm the install request in advance so the installation can run continuously until the operation finishes.

4. Build the RPM package from the downloaded SRPM package.

The following example builds the Lynx SRPM package:

```
# rpmbuild --rebuild lynx-2.8.8-0.3.dev15.el7.src.rpm
```

5. Install the RPM package built from the downloaded SRPM package.

   The following example installs the Lynx text-based Web browser on the device. Your run of the command may vary slightly for your specific hardware:

   ```
   # smart install /usr/src/rpm/RPMS/ArchitectureOfDevice/
   lynx-2.8.8-0.3.dev15.ArchitectureOfDevice.rpm --yes
   ```

   The *ArchitectureOfDevice* string refers to the architecture of the hardware device.

   To confirm the install request in advance so the installation runs continuously until the operation finishes, use the**--yes** option.

6. Verify the application package is installed.

   The following example verifies the **lynx** package is installed:

   ```
   # smart query lynx
   # smart info lynx
   ```

7. Run the application package.

   The following example launches the **lynx** browser with the Wind River URL:

   ```
   # lynx www.windriver.com
   ```

## Registering a Device for Use on the Wind River Helix App Cloud

To use the Helix App Cloud with Pulsar Linux, you must create an account and register your hardware device.

### Prerequisites

- You have a supported device with Pulsar Linux installed.
- You have a Wind River customer support account.
- Your device is running with console access.
- You have another device with Internet access and a Web browser for accessing the Wind River Helix App Cloud website.

The steps in this procedure will prepare you for application development with the Helix App Cloud.

### Procedure

1. Verify or create a Helix App Cloud account.

   In a Web browser, navigate to https://app.cloud.windriver.com/.

2. Click **Sign in with Wind River Account** and enter your account credentials.

**NOTE:**

Your user name is the email address associated with your Wind River customer support account.

3. Register your Pulsar Linux hardware device.

   Run the following command on your Pulsar Linux device to begin the registration process and create a registration key.

   ```
   # registerTarget -n targetName
   ```

   In this example:

   - the **-n** option indicates a new registration
   - the *targetName* is a name that you specify to recognize your device

   Once the command completes, it provides information concerning the registration status, for example:

   ```
   # registerTarget -n MyComputeStick-v2
   Created target [ computestick ] on server [ app.cloud.windriver.com ]
   Device registration status: pending
   Device registration expires in 19.9666666666667 (mins)
   Device Registration Key: 20d98
   ```

   At the end of the output, the system provides a device registration key. In this example, the key is **20d98**. Record this key, as it is required to register your device for use on the Helix App Cloud.

4. Register your device on the Helix App Cloud website.
   a) Log in to your account at https://app.cloud.windriver.com/.
   b) Click **New Device** in the upper right-hand corner.
   c) When prompted, click **Register an existing device using its Registration ID**.
   d) Enter the Registration ID (key) obtained in *step 2*, and a device name, then click **Register Device**.

      Once the registration completes, a page will display with your device information and development options. You are now ready to use the Helix App Cloud.

5. Optionally create a new application project by clicking **Create New Project**.

**NOTE:** Information on developing applications with the Helix App Cloud is available on the website.

# Installed and Available Software Packages

Each Wind River Pulsar Linux device has a default list of packages based on the device containers and available packages to customize your device based on your requirements.

### Installed Packages

As a minimum, all devices include the required **cube-dom0** and **cube-essential** containers. It also includes a baseline container that the device boots into. The baseline container is device-specific. Refer to your device quick start document for additional information specific to your device.

Wind River provides the base package lists for each container on the Knowledge Library at https://knowledge.windriver.com/en-us. Select **Operating Systems** > **Pulsar Linux 8** > **Release and Product Information**, then select **Browse All Downloads** to access the package lists.

Note that these are the base configuration of packages for the device. If your Pulsar Linux device includes additional software features, you will need to run the following command from the container to view all installed packages on the device:

```
# rpm -qa --queryformat '%{NAME}, %{VERSION}, \
%{LICENSE}, %{DESCRIPTION} \t' | tr -d '\n' | tr '\t' '\n'
```

Once the command completes, it will display four columns of package information, including the package name, version, license type, and description. Since this list can be quite long, you can run the following command to redirect the output to a **pulsar-linux-package-list.csv** file:

```
# rpm -qa --queryformat '%{NAME}, %{VERSION}, %{LICENSE}, \
%{DESCRIPTION} \t' | tr -d '\n' | tr '\t' '\n' > pulsar-linux-package-list.csv
```

To obtain package information on a different container, such as **cube-dom0**, switch to the container and re-run the command. For additional information, see the *Wind River Pulsar Linux System Administration Guide: Switching Containers*.

### Available Packages

Depending on your development needs, you may require additional packages for your Pulsar Linux device file system. Wind River provides additional packages that you can add using the **smart** package manager on your device.

To obtain a list of installed packages, run the following command from any container:

```
# smart search all
```

Once the command completes, it will display the RPM package names and a short description. Run the following command to redirect the output to a **pulsar-linux-available-packages** text file:

```
# smart search all > pulsar-linux-available-packages
```

# 3

# Application Development with the SDK Overview



Wind River provides a software development kit (SDK) for developing applications and creating packages that you can install on your Pulsar Linux device.

This SDK is based on the Yocto Project SDK, but also includes the **rpm-tool** and **ima-tool** tools, which provide the ability to package an application with proper IMA and GPG signatures as required by the device's security implementation.

In addition, it also has the ability to assist you in creating an RPM repository, and make the RPM packages in that repository available to your Pulsar Linux device. For existing RPM repositories, the SDK includes the **repo-tool** to sign RPM packages in bulk with a private GPG key.

Use the **rpm-tool** to:

- Package all binaries/libraries into an RPM package
- Sign the RPM with a GPG key for use on security-enabled devices
- Generate a repository to maintain the RPM package created with the tool

Use the **ima-tool** to:

- Sign an application binary so it will run on security-enabled devices
- Create a shell script that will install the signed application to your device

Use the **repo-tool** in an existing RPM repository to:

- Sign or re-sign RPM packages with a GPG key for use on security-enabled devices using the **rpm** command *addsign* or *resign* modes for package signing.
- Prepare an existing directory for use as an RPM repository

- Perform bulk updates to sign packages in the repository with a single command

In Pulsar Linux documentation, application development will focus on using the **rpm-tool** to create packages for use on your device.

For specific information on using the SDK for basic development, see the Yocto Project documentation at: http://www.yoctoproject.org/docs/2.1/sdk-manual/sdk-manual.html#sdk-appendix-obtain.

### SDK Development Workflow

To begin developing with the SDK, you must:

1. Install the SDK on your development host
2. Build and generate the application outputs to a directory.
3. Create or import the IMA key for package signing with the IMA signature
4. Create or import the GPG key for RPM signing.
5. Install the **realpath** package if it is not already present on the host. The following example is for an Ubuntu Linux host:

   ```
   $ apt-get install realpath
   ```

   The **realpath** application helps resolve symbolic links and store absolute paths in the file system for applications.
6. Run the **rpm-tool** on a supported Linux host system to create an installable RPM package.
7. Deploy the package to the RPM repository.
8. Install the package on the Pulsar Linux device.

## Installing the SDK

Before you can use the SDK, you must install it and source the environment.

Wind River provides a SDK for Pulsar Linux based on the Yocto Project SDK. For additional information on using the Yocto Project SDK, see http://www.yoctoproject.org/docs/2.1/sdk-manual/sdk-manual.html#sdk-appendix-obtain.

### Prerequisites

You are using a recommended Linux host or some other Linux host with equivalent configuration. For details, refer to the *Wind River Pulsar Linux Release Notes: System Requirements*.

### Procedure

1. Download the Pulsar Linux SDK at https://distro.windriver.com/public_feeds/.

   Select your device, then navigate to the product releases. The SDK file is named after the product version and device image, for example:

   **pulsar*version-deviceName-containerName*-sdk.zip**

2. Extract the contents of the compressed SDK file to a directory on your Linux host system.

```
$ unzip pulsarversion-deviceName-containerName-sdk.zip -d /opt/pulsarSDK
```

3. Provide executable rights to the SDK installer script.

```
$ chmod a+x sdkDir/wrlinux-small-version-glibc-arch-cpu-containerName-sdk.sh
```

   where the variables for *version*, *arch*, *cpu*, and *containerName* represent the text that is appropriate for your device. You may need to list the contents of the directory (**ls** command) to determine the correct file name to use.

   The following is an example for an IA-based device:

```
$ chmod a+x sdkDir/wrlinux-small-8.0.0.6-glibc-x86_64-intel_corei7_64-cube-gw-sdk.sh
```

4. Install the SDK.

   In the ***sdkDir***, run the shell script (***.sh**) that is appropriate for your device. The following example demonstrates how to install the SDK for a 64-bit IA-based device:

```
$ ./sdkDir/wrlinux-small-8.0.0.6-glibc-x86_64-intel_corei7_64-cube-gw-sdk.sh

Pulsar Linux SDK installer version 8.0-intel-corei7-64
======================================================
Enter target directory for SDK (default: /opt/windriver/wrlinux-small/8.0-intel-
corei7-64): /opt/windriver/wrlinux-small/8.0-intel-corei7-64
You are about to install the SDK to "/opt/windriver/wrlinux-small/8.0-intel-
corei7-64". Proceed[Y/n]?Y
Extracting
SDK.....................................................................................
.....................................................................................
....done
Setting it up...done
SDK has been successfully set up and is ready to be used.
Each time you wish to use the SDK in a new shell session, you need to source the
environment setup script e.g.
 $ . /buildarea1/pulsaruser/sdk/environment-setup-core2-64-wrs-linux
 $ . /buildarea1/pulsaruser/sdk/environment-setup-x86-wrsmllib32-linux
```

**NOTE:** This output demonstrates the results from installing an Intel Architecture-based SDK. For other architectures, the output will be different.

   Once installed, the location will be referred to as the ***sdkDir***.

5. Set up the SDK environment.

   You must perform this step each time you open a new terminal for SDK development purposes.

```
$ source sdkDir/environment-setup-arch-wrs-linux
```

   Once complete, you will find the **rpm-tool** in a subdirectory of the **sysroots** directory, for example:

```
$ which ima-tool rpm-tool
sdkDir/sysroots/x86_64-wrlinuxsdk-linux/usr/bin/ima-tool
sdkDir/sysroots/x86_64-wrlinuxsdk-linux/usr/bin/rpm-tool
```

# Creating a Sample Hello World Application

Create a sample application for use with the Pulsar Linux SDK.

In this procedure, you will create a sample Hello World application with source code and Makefile.

### Prerequisites

- You are using a recommended Linux host or some other Linux host with equivalent configuration. For details, refer to the *Wind River Pulsar Linux Release Notes: System Requirements*.
- You have the Pulsar Linux SDK installed on your Linux host system. For details, see Installing the SDK on page 10.

### Procedure

1. Create a working directory for your application on your Linux host system and navigate to it.

   This location will be referred to as the ***appDir***.

2. Set up the **hello.c** source file with **vi**.
   a) Create the **hello.c** file.

   ```
   $ vi hello.c
   ```

   b) Enter or copy the following text and save the file.

   ```
   #include <stdio.h>

   int main(void)
   {
           printf("Hello World!\n");
           return 0;
   }
   ```

3. Set up the **Makefile** with vi.
   a) Create the **Makefile** file.

   ```
   $ vi Makefile
   ```

b) Enter or copy the following text and save the file.

```
 # Generated Makefile for "hello-Linux"
# No selected build spec
PROJ_DIR=$(shell pwd)
CFLAGS?=-c

AINCLUDES= SOURCES=hello.c

OBJECTS=$(SOURCES:.c=.o)

hello_Linux=hello_Linux

all: $(hello_Linux) install

$(hello_Linux):
        $(CC) $(CFLAGS) $(CINCLUDES) -c -o hello.o hello.c
        $(CC) $(LFLAGS) $(LINCLUDES) -o $(hello_Linux) hello.o

install :
        mkdir -p  $(PROJ_DIR)/install/usr/bin
        cp $(hello_Linux) $(PROJ_DIR)/install/usr/bin
        chmod 755 $(PROJ_DIR)/install/usr/bin/$(hello_Linux)

.PHONY: clean

clean :
        @rm -f *.o
        @rm -f $(hello_Linux)
        @rm -rf $(PROJ_DIR)/install
        @echo Directory Cleaned!

clean-wb :
        @for i in `ls -d */*/Debug 2> /dev/null`; do \
          d=`dirname $$i`; d=`dirname $$d`; rm -rf $$d; \
        done;

        @for i in `ls -d */*/NonDebug 2> /dev/null`; do \
          d=`dirname $$i`; d=`dirname $$d`; rm -rf $$d; \
        done;
```

4. Set up the SDK environment.

```
$ source sdkDir/environment-setup-arch-wrs-linux
```

5. Build the application.

```
$ make
```

Once the command completes, the project directory will include the **hello_Linux** binary and an **install** directory, for example:

```
$ ls
hello.c  hello_Linux  hello.o  install  Makefile
```

6. Install the application binary to the *appDir*/**install/usr/bin** on the Linux host system.

This step ensures the application binary is located in the directory structure required for using the **rpm-tool**.

| Options | Description |
|---|---|
| **With Install Rule** | Use this command if your application Makefile has an install rule, such as the one included with the Hello World Makefile.<br>`$ make install`<br>If the system returns **install is up to date** then the application is already installed from the previous step. |
| **Without Install rule** | Manually copy the binaries to the installation directory, typically ***appDir*/install/usr/bin**. |

This directory will be used to create an RPM package for the application. For additional information, see Creating an RPM Package and Populating the RPM Repository on page 14.

## Creating an RPM Package and Populating the RPM Repository

Once you create an application using the SDK, you can use the **rpm-tool** to create a package of the application and add it to an RPM repository to install on your Pulsar Linux device.

This procedure is for creating unsigned RPM packages. When you create a package with this procedure, it will only work on Pulsar Linux devices without security features.

### Prerequisites

- You have an application created using the SDK. This procedure uses the Hello World application as an example. For details, see Creating a Sample Hello World Application on page 12.
- You have a Linux host system with the SDK installed. For details, see Installing the SDK on page 10.

### Procedure

1. Set up the SDK work environment.

   Perform this step on the Linux development host. You must perform this step each time you open a new terminal for SDK development purposes.

   ```
   $ source /sdkDir/environment-setup-arch-wrs-linux
   ```

2. Create a repository directory to maintain the packages.

   You will use this directory as the output directory for your unsigned packages.

   ```
   $ mkdir -p /tmp/rpm-repo
   ```

   The **/tmp/rpm-repo** directory is used as an example. If you have an existing package repository, you can specify that in the next step.

3. Create an RPM package.

| Options | Description |
|---|---|
| **Without .spec file** | Use this option if you do not have a **.spec** file. In this example, you will use the **rpm-tool** to specify the required **.spec** file options. For additional information, see The rpm-tool Command Option Reference on page 15.<br>`$ rpm-tool -p hello-world -v 1.0 -r r0 -i /application-buildDir/hello-Linux/install -o /tmp/rpm-repo` |
| **With .spec file** | `$ rpm-tool -s hello-world.spec -i /application-buildDir/hello-Linux/install -o /tmp/rpm-repo` |

Once the command completes, a new, unsigned RPM package will be added to the RPM repository specified with the **-o** option. This file will use the following naming syntax:

***appName-version-release-arch*.rpm**

With the Hello World application, the file name will be **hello-world-1.0-r0.*arch*.rpm**, where ***arch*** refers to the device architecture.

To use the new package, you must install it on your device. For details, see the *Wind River Pulsar Linux Security Features Guide: Deploying an RPM Repository to the Device*.

## The rpm-tool Command Option Reference

The **rpm-tool** utility provides several command-line options to create and sign RPM packages.

The **rpm-tool** is included with the Pulsar Linux SDK, and is always run with options. For additional information, see Application Development with the SDK Overview on page 9.

| Option | Description |
|---|---|
| **-p** | The package name. If a **.spec** file is named using the **-s** option, specifying the package name with **-p** is optional. |
| **-i** | The application installation directory. |
| **-o** | The RPM repository directory where the generated RPM package will be created. |
| **-n** | The name of the GPG key. |
| **-v** | The application version. If a **.spec** file is named using the **-s** option, the application version in the **.spec** file is used. |

| Option | Description |
|---|---|
| **-r** | The application release version, which is used for RPM version management. If a **.spec** file is named using the **-s** option, the application release version in the **.spec** file is used. |
| **-g** | The GPG private key file. If your GPG key is not yet imported, this argument is required. If the key is already imported to the SDK keyring, this argument is optional. |
| **-s** | A custom **.spec** file used to generate the RPM package. If this option is omitted, **rpm-tool** will automatically generate a **.spec** file. |

## The repo-tool Command Option Reference

The **repo-tool** utility provides several command-line options to prepare an RPM repository and sign RPM packages.

The **repo-tool** is included with the Pulsar Linux SDK, and is always run with options. For additional information, see Application Development with the SDK Overview on page 9.

| Option | Description |
|---|---|
| **-d** | Set +x |
| **-m** | The RPM package signing method. This defaults to the *resign* method, but will also use the *addsign* method when a new GPG key is used to sign a package. |
| **-p** | The The GPG private key passphrase. |
| **-r** | The name and path of the RPM repository directory where the packages reside. |
| **-s** | The SDK directory (***sdkDir***) where the SDK is installed. |
| **-g** | The GPG private key file. If your GPG key is not yet imported, this argument is required. If the key is already imported to the SDK keyring, this argument is optional. |
| **-n** | The GPG key name used to sign RPM packages. |

# 4

# Inserting a Loadable Kernel Module

Build and insert a loadable kernel module (LKM) to the running Pulsar Linux kernel.

A loadable kernel module is a compiled object file that can be inserted (loaded) into kernel space. When inserted, the kernel module becomes another part of the running kernel. Kernel modules can provide a wide variety of functionality though, in practice, they are typically device drivers, file system drivers, or system calls.

Loadable kernel modules enable you to dynamically add functionality only when it is needed. This helps reduce the kernel image size and lower your RAM usage as kernel modules are not built in to the kernel. Also, they are a great development tool for device drivers since a system reboot is not necessary.

**Procedure**

1. Install the Linux headers:

   ```
   # smart update
   # smart install kernel-devsrc --yes
   ```

   During installation, the package manager will ask you to confirm the install request. By using the **--yes** option, you confirm the install request in advance so the installation can run continuously until the operation finishes.

2. Generate the helper scripts:

   ```
   # cd /usr/src/kernel
   # make scripts
   ```

   The helper scripts are part of the Yocto Project's environment and enable you to directly build kernel modules on the Pulsar Linux system.

3. Create a build directory for the kernel module and navigate into it.

   For example:

   ```
   # mkdir /root/my_lkm
   # cd /root/my_lkm
   ```

4. In a text editor (for example, the **vi** editor), write source code for the kernel module in the C programming language.

5. In a text editor, write a makefile for which to build the kernel module.

6. Build the kernel module from the **Makefile**:

   ```
   # make
   ```

   After the kernel module successfully builds, it resides in the build directory with a file extension of **.ko**.

7. Insert the kernel module by using the **insmod** command as follows:

   ```
   insmod nameOfKernelModule.ko
   ```

   The **insmod** command inserts the kernel module to the running Pulsar Linux kernel.

8. Verify successful insertion of the kernel module:

   ```
   # lsmod
   ```

   The **lsmod** command displays a list of kernel modules that are currently loaded in the kernel.

### Example

An example to demonstrate the building and inserting of a loadable kernel module to a running Pulsar Linux kernel.

```
# cat HelloWorld.c
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Jane Doe");
MODULE_DESCRIPTION("A HelloWorld kernel module.");

static int __init HelloWorld_init(void)
{
    printk(KERN_INFO "Hello beautiful world!\n");
    return 0;
}

static void __exit HelloWorld_cleanup(void)
{
    printk(KERN_INFO "Cleaning up the HelloWorld kernel module.\n");
}

module_init(HelloWorld_init);
module_exit(HelloWorld_cleanup);
# cat Makefile
obj-m += HelloWorld.o

all:
    make -C /usr/src/kernel SUBDIRS=`pwd` modules

clean:
    make -C /usr/src/kernel SUBDIRS=`pwd` clean
# make
make -C /usr/src/kernel SUBDIRS=`pwd` modules
make[1]: Entering directory '/usr/src/kernel'
  CC [M]  /root/my_lkm/HelloWorld.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /root/my_lkm/HelloWorld.mod.o
  LD [M]  /root/my_lkm/HelloWorld.ko
make[1]: Leaving directory '/usr/src/kernel'
# ls
HelloWorld.c   HelloWorld.mod.c  HelloWorld.o  Module.symvers
HelloWorld.ko  HelloWorld.mod.o  Makefile      modules.order
# insmod HelloWorld.ko
# lsmod
Module                  Size  Used by
HelloWorld               673  0
bridge                 93640  0
stp                     1431  1 bridge
llc                     3604  2 stp,bridge
zynq_edac               3295  0
edac_core              44119  1 zynq_edac
openvswitch            62919  0
gre                     3731  1 openvswitch
# dmesg | tail -1
[ 7738.035457] Hello beautiful world!
```

# 5

# Removing a Loadable Kernel Module

If you no longer need the added functionality provided by the kernel module, you can remove it from the Pulsar Linux kernel.

You cannot remove a kernel module that is currently being used by another program.

### Prerequisites

To complete this procedure, the loadable module must currently be inserted into the running kernel. For more information, see Inserting a Loadable Kernel Module on page 17.

### Procedure

Run the **rmmod** command as follows.

```
# rmmod nameOfKernelModule.ko
```