

Mentor[®] Embedded Linux[®] Omni OS User's and Reference Manual

Software Version 1.0.1

© 2019 Mentor Graphics Corporation
All rights reserved.

This document contains information that is proprietary to Mentor Graphics Corporation. The original recipient of this document may duplicate this document in whole or in part for internal business purposes only, provided that this entire notice appears in all copies. In duplicating any part of this document, the recipient agrees to make every reasonable effort to prevent the unauthorized use and distribution of the proprietary information.

This document is for information and instruction purposes. Mentor Graphics reserves the right to make changes in specifications and other information contained in this publication without prior notice, and the reader should, in all cases, consult Mentor Graphics to determine whether any changes have been made.

The terms and conditions governing the sale and licensing of Mentor Graphics products are set forth in written agreements between Mentor Graphics and its customers. No representation or other affirmation of fact contained in this publication shall be deemed to be a warranty or give rise to any liability of Mentor Graphics whatsoever.

MENTOR GRAPHICS MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

MENTOR GRAPHICS SHALL NOT BE LIABLE FOR ANY INCIDENTAL, INDIRECT, SPECIAL, OR CONSEQUENTIAL DAMAGES WHATSOEVER (INCLUDING BUT NOT LIMITED TO LOST PROFITS) ARISING OUT OF OR RELATED TO THIS PUBLICATION OR THE INFORMATION CONTAINED IN IT, EVEN IF MENTOR GRAPHICS HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

U.S. GOVERNMENT LICENSE RIGHTS: The software and documentation were developed entirely at private expense and are commercial computer software and commercial computer software documentation within the meaning of the applicable acquisition regulations. Accordingly, pursuant to FAR 48 CFR 12.212 and DFARS 48 CFR 227.7202, use, duplication and disclosure by or for the U.S. Government or a U.S. Government subcontractor is subject solely to the terms and conditions set forth in the license agreement provided with the software, except for provisions which are contrary to applicable mandatory federal laws.

TRADEMARKS: The trademarks, logos and service marks ("Marks") used herein are the property of Mentor Graphics Corporation or other parties. No one is permitted to use these Marks without the prior written consent of Mentor Graphics or the owner of the Mark, as applicable. The use herein of a third-party Mark is not an attempt to indicate Mentor Graphics as a source of a product, but is intended to indicate a product from, or associated with, a particular third party. A current list of Mentor Graphics' trademarks may be viewed at: mentor.com/trademarks.

The registered trademark Linux[®] is used pursuant to a sublicense from LMI, the exclusive licensee of Linus Torvalds, owner of the mark on a world-wide basis.

End-User License Agreement with Embedded Software Supplement: You can print a copy of the EULA with Embedded Software Supplement from: mentor.com/embeddedeula.

Mentor Graphics Corporation
8005 S.W. Boeckman Road, Wilsonville, Oregon 97070-7777
Telephone: 503.685.7000
Toll-Free Telephone: 800.592.2210
Website: mentor.com
Support Center: support.mentor.com

Send Feedback on Documentation: support.mentor.com/doc_feedback_form

Table of Contents

Chapter 1

Introduction to Mentor Embedded Linux Omni OS.....	7
Key Concepts.....	8
Mentor Embedded Linux Omni OS Components	10
Introduction to Isar.....	10

Chapter 2

Target Images	13
Standard Images	13
Prepare the Host System	14
Create a Project	15
Create Standard Images	16
Create Images on a Virtual Machine	17
Add Optional Predefined Features.....	18

Chapter 3

Components Customization	21
Customize Your Own Image	21
Customize the Layers	24
Customize an OS Package.....	25

Chapter 4

Debian Packages Creation	27
Package Build Files	28
Debian Package Classes.....	29
Debian Package Build Environment	30
Post Install Using Transient Packages.....	30
Build a Debian Package From the Source	31

Chapter 5

Application Build Environment	35
Generate an ADE Image	35
Launch the IDE	36
Install the ADE on the IDE	36
Create Sample Applications.....	37
Build and Run Applications on the Target	38

Chapter 6

Supported IoT Protocols	39
Protocols and Definitions.....	39
Building Images With IoT	40

Chapter 7	
System Update Using SWUpdate	43
Create the SWUpdate Image	43
Overview of the sw-description File	44
Update the System From a USB	45
Update the System Over the Air	47
Install and Set up the hawkBit Server.	47
Update the System Using the hawkBit Server	48
Chapter 8	
Security Modules.....	51
Introduction to SELinux	51
SELinux Commands Overview	53
Extend the Security Policy.....	56
Recommendations for Security Policy Creation	59
Appendix A	
Examples	61
Example: Creating a Build.	61
Example: Adding a Debian Package Using local.conf	61
Example: Creating a Custom Layer.....	62
Example: Customizing the development-image Recipe	62
Example: Building With a Customized Layer.....	63
Example: Including Configuration Files and Scripts.....	63
Example: Using Post-Install Scripts	64
Example: Managing sudo Users	64
Example: Setting the root Password	65
Example: Enabling or Disabling systemd Services	65
Example: Creating a Read-Only root File System.....	65
Appendix B	
Linux Commands	67
Glossary	
End-User License Agreement	
with Embedded Software Supplement	

List of Figures

Figure 1-1. Layer Composition in Isar. 11

Figure 3-1. Customizable Components 21

Figure 4-1. Debian Packaging Overview. 27

Chapter 1

Introduction to Mentor Embedded Linux Omni OS

Mentor® Embedded Linux®¹ Omni OS is a Debian-based Linux distribution. Debian is a broadly utilized, enterprise-class, open source Linux operating system (OS) that allows different levels of OS customization for multiple architectures.

Mentor Embedded Linux Omni OS and Debian have the same OS base. The majority of Mentor Embedded Linux Omni OS packages come unchanged from Debian. On top of the Debian base, Mentor Embedded Linux Omni OS provides additional components, such as Docker and SWUpdate, to help embedded platform developers assemble and configure the OS before deployment.

Mentor Embedded Linux Omni OS introduces the benefits of a customizable OS on top of a stable Linux distribution.



Key Concepts	8
Mentor Embedded Linux Omni OS Components	10
Introduction to Isar	10

1. Linux® is a registered trademark of Linus Torvalds in the U.S. and other countries.

Key Concepts

Mentor Embedded Linux Omni OS developers must be aware of the key platform concepts when customizing the OS before deployment.

Debian Package

Linux administrators refer to a set of files required to implement specific commands or features as a “package.” Debian packages contain information on how to install, upgrade, configure, and remove the software in Debian. For example, a Debian package may contain a script that runs after the software installation (postinst) or uninstallation (postrm).

There are two types of Debian packages:

- **Binary Packages** — Contain executable files, configuration files, and settings specific to a particular hardware and OS version.
- **Source Packages** — Contain source files, typically in C or C++, along with metadata. You must compile source packages before using them.

Metadata

The package metadata includes information about the software, such as a description and a list of any other packages it requires. For example, most Debian packages require loading the C library before running any software.

Development Host

A supported host OS that developers use to build Mentor Embedded Linux Omni OS packages and produce target images. The development host runs the build system.

Build System

A framework that provides a set of instructions (recipes) for building software packages and repeatably generating Debian-based root file systems. Developers can customize the framework for their own use.

Isar

The “Integrated System for Automated Root file system generation” (Isar) is the build system for Mentor Embedded Linux Omni OS. Isar enables developers to configure and assemble Debian-based packages. Mentor Embedded Linux Omni OS developers use Isar on their development host to produce target images for a specific target system. Isar uses a layering method that enables developers to produce systems that can work with a specific product, hardware design, or specific deployment type.

Recipes

A list of instructions that include the required settings and tasks to build the software. Recipe files define important information such as where to get the sources from and whether there are dependencies to execute the tasks. Isar uses the BitBake task execution tool to execute recipes. Recipe files have a *.bb* extension.

Mentor Embedded Linux Omni OS uses the Debian build metadata for all packages involving more than a few raw files. The Debian packaging process is also typically used for more complex projects that use autoconf and cmake.

Buildchroot

The environment in which Isar and Mentor Embedded Linux Omni OS can execute the build system steps.

There are two types of buildchroot environments:

- **Target Buildchroot** — Uses native compilers and libraries that match the target system. When using native compilers, the build system installs development tools and uses QEMU to execute foreign binaries such as the GNU Compiler Collection (GCC).
- **Host Buildchroot** — Uses cross compilation on the development host. When using cross compilation, the build system installs the cross compilers.

Target System

The hardware on which the application and Mentor Embedded Linux Omni OS will run. BitBake recipes refer to the target system as “`${MACHINE}`”. Refer to the *Mentor Embedded Linux Omni OS Release Highlights and Release Notes* document for more information about the supported target systems.

Target Image

A file system image assembled from user-selected binary packages and meant for running on the target system. During the image creation, developers configure the image by defining the target system on which to install the image. The image installation depends on the target system. Some targets support direct installation from the target flash memory (NAND or eMMC), while other targets support installation from portable storage devices.

Related Topics

[Mentor Embedded Linux Omni OS Installation Instructions](#)

[Target Images](#)

[Introduction to Isar](#)

Mentor Embedded Linux Omni OS Components

Mentor Embedded Linux Omni OS provides an LTS kernel and prebuilt Debian packages. It also provides development tools for creating applications, customizing OS images, and generating a file system.

- **LTS Kernel** — The LTS kernel is a stable operating system containing the core functionality.
- **Prebuilt Debian Packages** — The installer includes images for required functionality, such as BitBake for customizing the OS and Isar for generating a file system.
- **Development Tools** — The Sourcery™ CodeBench software is bundled with Mentor Embedded Linux Omni OS. It provides an Eclipse-based integrated development environment (IDE). Sourcery CodeBench additionally provides various debugging and analysis tools to create an application development environment (ADE).

Related Topics

[Application Build Environment](#)

Introduction to Isar

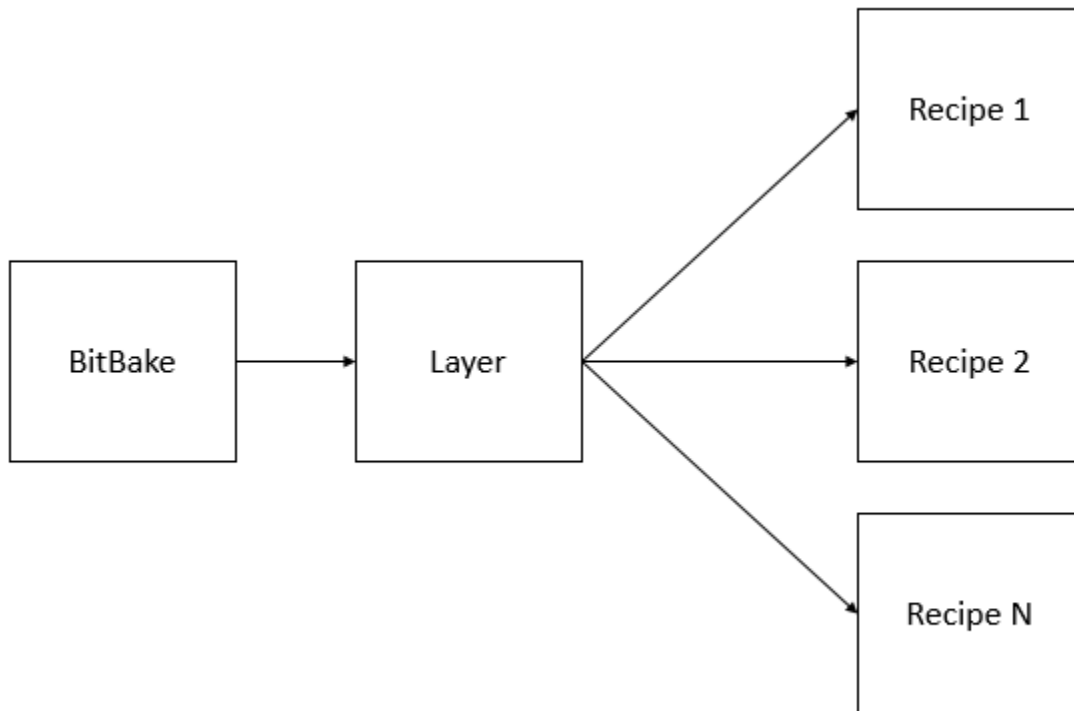
The “Integrated System for Automated Root file system generation” (Isar) enables you to build Debian-based products and optimize the built system for systems operations. Isar uses the BitBake task execution system to automate the process of creating embedded systems.

Isar Components

Isar consists of BitBake recipes that implement the logic of the main build system. Each BitBake recipe represents some instructions to perform a specific task. Recipes usually describe the common tasks, such as build or clean.

Multiple recipes can be grouped into one layer that serves a specific purpose. Each layer must have its own configuration file that controls the layer components. A layer with a configuration file that serves a specific purpose is called metadata that you can customize for your own use.

Figure 1-1. Layer Composition in Isar



Isar supports the following operations:

- Install the Debian base system and upstream packages.
- Build and install product software packages.
- Create ready-to-use customizable firmware images.

Isar provides a stable Debian distribution. This has many advantages:

- You do not need to create a new Linux distribution.
- The OS builds more quickly because there is less to compile.
- It is easier to customize the OS because you are starting with a known base.
- It is easier to add parts with the upstream Debian packages.
- It is easier to troubleshoot because there is a large support community.

For more information about Isar, visit the following link:

<https://github.com/ilbers/isar>

Chapter 2

Target Images

Developers create the target image of the OS that is installed on embedded devices. Mentor Embedded Linux Omni OS comes with hundreds of binary packages built for major CPU architectures. They may be used as a base for your target system by following the steps described in this chapter.

Creating the image is a three-step process:

1. Create a project directory for your work. This directory will contain configuration files, scripts for setting environment variables, any custom layers, and the results of building images.
2. Build the standard target images to check the project. You may also want to run the image on your target.
3. (Optional) Customize the target image, such as by adding support for a particular protocol.

Directions for creating variants of the target image for different applications are provided in “[Components Customization](#)” on page 21.

Standard Images	13
Prepare the Host System	14
Create a Project	15
Create Standard Images	16
Create Images on a Virtual Machine	17
Add Optional Predefined Features	18

Standard Images

Mentor Embedded Linux Omni OS supports standard OS images suitable for different application requirements. You can create one or more images to install on the target system.

Supported Images

- **Core Image** — Provides default system settings, components, and user access. Suitable for production or restricted systems where the OS developers select specific OS features for the end systems that users can access.

- **Development Image** — Provides advanced setup and diagnostic tools. Suitable for OS developers who want to customize the OS or for open systems that interact with other systems or external environment.
- **Service Stick Image** — Provides a ready-to-use image that enables you to install the core or development OS on the target system. Suitable for commissioning, maintaining, and servicing devices.

Prepare the Host System

You can run a script to download the required software packages to your host system. The software packages enable you to create the Mentor Embedded Linux Omni OS images that you can install on the target system.

Prerequisites

- Mentor Embedded Linux Omni OS source files are downloaded and installed on your host system. Refer to the [Mentor Embedded Linux Omni OS Installation Instructions](#) for more information.
- You have sudo access on the host system.

Procedure

Open a terminal and do *one* of the following, depending on your host system operating system:


- **Debian** — Run the *setup-debian* script to install the required Debian packages.

```
sudo <install-dir>/omni/mel-core/scripts/setup-debian
```

- **Ubuntu** — Run the *setup-ubuntu* script to install the required Ubuntu packages.

```
sudo <install-dir>/omni/mel-core/scripts/setup-ubuntu
```

Tip

 To create an environment variable for *<install_dir>*, add a line to your *.bashrc* file similar to the following example, but with your directory. The definition lets you use *\$OMNI_HOME* instead of typing the full directory path for *<install_dir>* each time.

```
export OMNI_HOME=$HOME/mgc/embedded/omni
```

Results

The script installs the required packages for running the tool and enables passwordless sudo operations.

Create a Project

Before building images, you must set up a project. This step is performed by running a script. The script also creates required configuration files and sets project-specific environment variables.

A project is a collection of build configurations. Build configurations define the environment for a specific target machine. Often a project contains only one build configuration, but it is possible for it to have many.

Prerequisites

- [Prepare the Host System.](#)

Procedure

1. Open a terminal and change to a directory outside the Mentor Embedded Linux Omni OS installation directory.

Do not create project directories inside the installation directory or modify the contents of the installation directory in any way. Any changes are likely to be lost if the installer is run again, such as for updating the software.

```
mkdir new_project
cd new_project
```

2. Run the *setup-environment* script to create a build configuration.


In the example below, *config1* is the directory name for the build configuration, and “<target-name>” should be replaced with your board support package. Available choices are listed in *mel-omni/bsps/*.

```
. $OMNI_HOME/mel-core/setup-environment -b config1 <target-name>
```

This creates a build configuration directory named *config1* that contains a local copy of the *setup-environment* script, and changes your current directory to *config1*. It also adds required environment variables to the Linux shell.

If you do not specify “-b <name>,” the script creates a project named *build*.

Note

 If you resume work later from a new terminal window, change to the build configuration directory and run its *setup-environment* script without any arguments. This quickly sets the environment variables to the correct values for the project.

Results

The *setup-environment* script generates required configuration files and adds the environment variables to your current shell.

- The project directory contains a build configuration directory.

- The build configuration directory contains a subdirectory named *conf* and two important files, *local.conf* and *bblayers.conf*. (These are used when customizing the image.)
- To list the BitBake environment variables, run this command:

```
bitbake -e
```

- The BitBake environment variables are defined for the project. To check, run this command at the shell:

```
env | grep BB
```

Create Standard Images

You must build the image before installing Mentor Embedded Linux Omni OS on the target system.

Procedure

1. On the host system, open a terminal and change the directory to your build configuration directory.

```
cd new_project/config1
```

2. Run the local *setup-environment* script.

```
. setup-environment
```

3. Create the images.

You can create just one image or all three images in sequence.

```
bitbake core-image
bitbake development-image
bitbake service-stick-image
```

The images are written to the *tmp/deploy/images/<target-name>* directory.

Note



If you get an error message regarding *do_cache_config*, some of the post-installation packages are missing. Use the following command to install them, and repeat Step 3.

```
sudo $OMNI_HOME/mel-core/scripts/setup-<linux_distro>
```

Results

The required images have been built. To deploy on a target system, the core or development images need to be added to the service stick image.

Refer to the target's Getting Started guide for information on how to install the images on the target.

Related Topics

[Customize Your Own Image](#)

Create Images on a Virtual Machine


Mentor Embedded Linux Omni OS provides templates to create a virtual machine using third-party software. You can then use the virtual machine to create the target images.

Prerequisites

- Mentor Embedded Linux Omni OS installed on a host system that is not a virtual machine.
- The latest Packer version installed on your host system.

<https://www.packer.io/>

Note

 By default, the Debian distribution contains the Packer package, but it may not be the latest Packer version. Make sure you have the latest Packer version before creating the image.

- The latest Vagrant version installed on your host system.

<https://www.vagrantup.com/>

- The latest VMware® Workstation Pro version.

<https://www.vmware.com/>

Procedure

1. Change to the *omni-vm* directory and stage the release files in it.

```
cd $OMNI_HOME/omni-vm
./prepare-omni-vm-data
```

2. Make sure there is a minimum of 20 MB available space.

If the partition holding */tmp* does not have at least 20 MB available, set the environment variable *\$TMPDIR* to a directory with enough space, as in the following lines. These create a *tmp* directory under *omni-vm*.

```
export TMPDIR=$PWD/tmp
mkdir -p $TMPDIR
```

3. Build *industrial-vmware-debian9.box* using the Packer utility.

```
./<packer-download_dir>/packer_<version>_linux_amd64/packer \
build --only=vmware-iso debian9.json
```

This generates a Vagrant box encapsulating the VMware virtual machine. You can find it under `$OMNI_HOME/omni-vm/builds`.

4. Extract the virtual machine from the Vagrant box.

```
cd builds
vagrant box add --name=omni-vm industrial-vmware-debian9.box
```

You can find the extracted `packer-debian-9-amd64.vmx` virtual machine file under `$HOME/.vagrant.d/boxes/omni-vm/0/vmware_desktop`.

5. Double-click the `packer-debian-9-amd64.vmx` file to launch the virtual machine in the VMware workstation.

Note



By default, the created virtual machine enables automatic login. The default user and password are “industrial”. If you add other user accounts, you must set up passwordless sudo in order to use Isar to build target images.

6. Prepare the project build and create the images on the virtual machine.

```
cd ~/Projects/<version>
. ./mel-core/setup-environment <target-name>
bitbake core-image
bitbake development-image
bitbake service-stick-image
```

Results

You have a Vagrant box that contains the Mentor Embedded Linux Omni OS and standard target images.

Add Optional Predefined Features

For your convenience, Mentor Embedded Linux Omni OS defines some common features that can be included in an image by setting variables in the build configuration’s `conf/local.conf` file.

Restrictions and Limitations

- This procedure does not support the service stick image.
- This method applies only to the predefined features listed in the table. To add other packages, see “[Customize Your Own Image](#)” on page 21.

Table 2-1. Supported Predefined Features

Feature	Description
alsa	Advanced Linux® Sound Architecture
amqp	Advanced Message Queuing Protocol
antivirus	ClamAV antivirus

Table 2-1. Supported Predefined Features (cont.)

Feature	Description
apt	Software installation and removal tool
coap	Constrained Application Protocol
docker	Docker runtime
firewall	UFW Network firewall (included by default)
flatpak	Support for Flatpak applications
hardening	Various OS hardening measures (included by default) <ul style="list-style-type: none"> • Check for password quality • Periodic scan for insecure files • Restrict su to administrative groups • PAM namespaces to isolate <i>/tmp</i> and <i>/var/tmp</i> for non-root users
kubernetes	Kubernetes container orchestration engine
lttng	LTTng flight recording capabilities (included by default for the development image)
lvm-support	Logical Volume Manager support
monitor	Monitoring tools for files with invalid groups or owners
mqtt	Message Queuing Telemetry Transport
network manager	NetworkManager
perf	Performance analysis tools
preempt_rt	Switch to the PREEMPT_RT kernel if available
pwquality	Passwords quality configuration tool
qt5	QT5 libraries
selinux	SELinux module (enables kernel parameters for SELinux and enables <i>.autorelabel</i>)
splash	Splash screen when target host starts
ssh-server	SSH (secure shell) server (included by default)
swupdate	SWUpdate framework
systemd-debug	Enable systemd debugging messages
tools-debug	Debugging tools (gdb)
tools-profile	Profiling tools such as ttng, valgrind, and perf
xmpp	Extensible Messaging and Presence Protocol

Procedure

1. Verify the features already available in the image.

The development image includes features not enabled by default in the core image. Different BSPs also support different packages. Additionally, the features may have already been customized.

To check which features are included for a given image, enter the following commands in the build directory:

```
bitbake -e <image_type> | grep ^IMAGE_FEATURES=  
bitbake -e <image_type> | grep ^BSP_FEATURES=
```

where *<image_type>* is *core-image* or *development-image*. The command returns a list similar to this:

```
IMAGE_FEATURES=" apt          antivirus      firewall  
hardening      monitor      pwquality    ssh-server  
"
```

2. Change directory to the build's *conf* directory, and open *local.conf* in a text editor.
3. To add features for all images, use `IMAGE_FEATURES_append = " <feature> "` where *<feature>* is one of the keywords in the table.

For example, to add LVM support, add the following line.

```
IMAGE_FEATURES_append = " lvm-support "
```

Note



The whitespace around the feature name must be included.

4. To add or remove features for only the development image, use `DEVELOPMENT_FEATURES_append` or `DEVELOPMENT_FEATURES_remove`, as in the following example.

```
DEVELOPMENT_FEATURES_append = " systemd-debug "  
DEVELOPMENT_FEATURES_remove = " splash "
```

This adds enables systemd debugging output to the development image, and disables the Linux boot splash screen.

5. Save the *local.conf* file.

Results

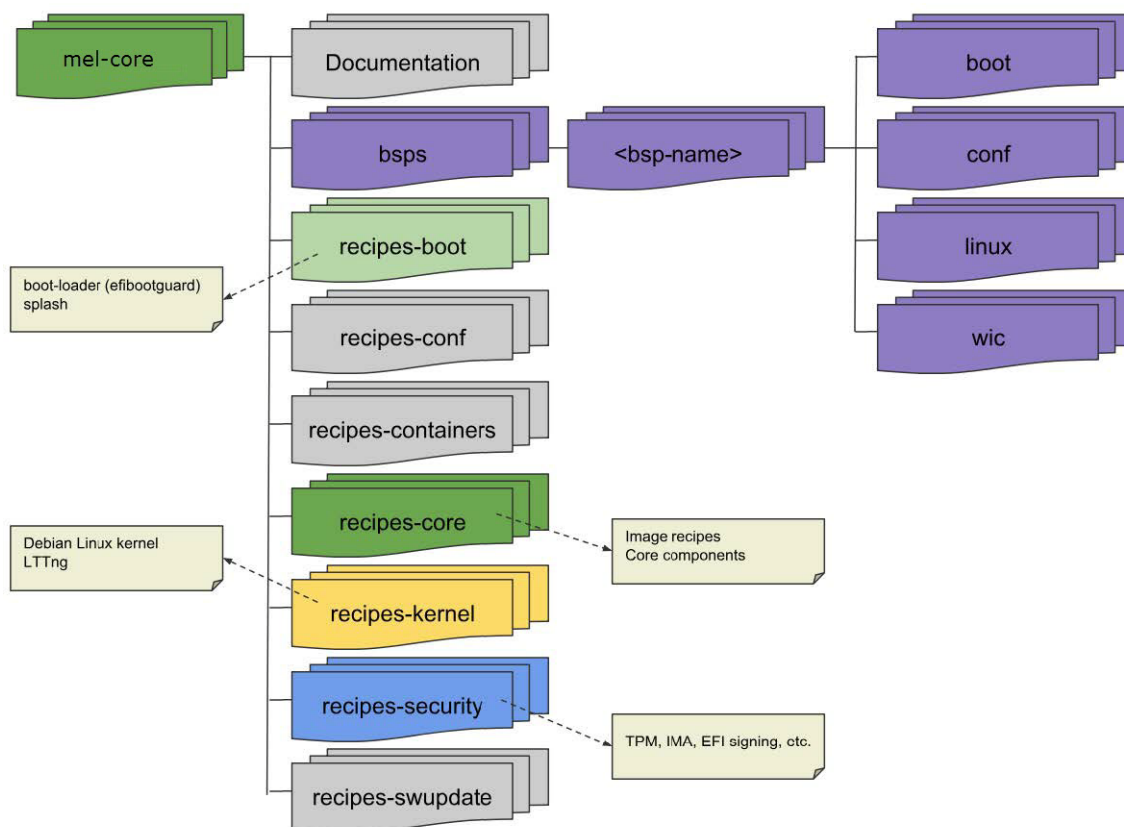
The next time you build the target images, they will use the features you have specified.

Chapter 3

Components Customization

Mentor Embedded Linux Omni OS supports customizing your OS image. This is done through the use of metadata layers. You can modify existing OS features or add new ones to fit the specific needs of the embedded devices.

Figure 3-1. Customizable Components



Customize Your Own Image.....	21
Customize the Layers	24
Customize an OS Package.....	25

Customize Your Own Image

Mentor Embedded Linux Omni OS enables you to customize the standard images by adding other packages to the standard core and development images. It is not recommended to customize the service stick image.

The customization is done with BitBake layers, which group changes to the target image based on a specific condition, such as target hardware or different versions of your application. The changes are usually referred to as “recipes.” Packages are added to recipes using the keywords `IMAGE_PREINSTALL_append` and `IMAGE_INSTALL_append`.

Prerequisites

- You have built the standard images.

Procedure

1. In your host system, go to the appropriate layer directory and create a *recipes-core/images* directory for your layer.

```
mkdir -p recipes-core/images
```

A layer directory is one that contains a *conf* directory and a *layer.conf* file. For this procedure, you can use either a build configuration directory or a custom layer directory. For instructions on creating a custom layer directory, see “[Customize the Layers](#)” on page 24.

The *recipes-core* directory should be at the same level as the *conf* directory.

2. Use this procedure to create the “my-application” example recipe used in this procedure:
 - a. Create the *recipes-examples/* directory and change into it.

```
mkdir recipes-examples  
cd recipes-examples
```

- b. Create the *my-application/* directory inside *recipes-examples/*:

```
mkdir my-application
```

In the following steps, the example recipe directory is *my-application*. The recipe filename should end in *.bb* (for example, *my-application.bb*). For the purpose of the example, the file content should be similar to the following:

```
# contents of recipes-examples/my-application/my-application.bb  
DESCRIPTION = "a simple python script for my-application"  
MAINTAINER = "User <user@123.com>"  
DEBIAN_DEPENDS = "python3.5"  
SRC_URI = "file://my-application.py"  
inherit dpkg-raw  
do_install() {  
    install -m 755 -d ${D}/usr/bin  
    install -m 755 ${WORKDIR}/my-application.py ${D}/usr/bin  
}
```

With this recipe, BitBake searches for *my-application.py* in *files/*.

- c. Create a directory for the files shipped by the recipe:

```
cd my-application  
mkdir files
```

- d. Add the *my-application.py* file with the following contents to include in my-application.

```
# contents of recipes-examples/my-application/files/my-  
# application.py  
  
#!/usr/bin/env python3.5  
  
#Hello world python program  
print("Hello world!")
```

3. In the *recipes-core/images/* directory, create the following files:

- a. A new file named *core-image.bbappend* with the following content:

```
include recipes-core/images/my-application.inc
```

This file adds *my-application* to the core image and causes the image to pick up this additions when it is built.

- b. A new file named *development-image.bbappend* with the following content:

```
include recipes-core/images/my-application.inc
```

This file adds *my-application* to the development image and causes the development image to pick up the additions when it is built.

- c. A new file named *my-application.inc* with the following content:

```
IMAGE_PREINSTALL_append = " <additional prebuilt packages> "  
IMAGE_INSTALL_append = " <my_application packages> "
```

IMAGE_PREINSTALL_append includes the prebuilt packages while
IMAGE_INSTALL_append includes the packages generated by Isar.

For example:

```
# Contents of recipes-core/images/my-application.inc  
IMAGE_PREINSTALL_append = " \  
openssh-server \  
vim \  
"  
IMAGE_INSTALL_append = " \  
my-application \  
"
```

4. Add the custom layer to your build recipe.

- a. Note the absolute path of your layer directory.

```
pwd
```

- b. Open the build project's *conf/bblayers.conf* file in a text editor and add the path to the BBLAYERS list.

For example, if `pwd` returned an absolute path of `/home/os_user/MyProject/meta_layer`, the `BBLAYERS` line would look similar to this:

```
BBLAYERS = "\
/home/os_user/MyProject/meta-layer \
/home/os_user/mgc/embedded/omni/mel-omni \
/home/os_user/mgc/embedded/omni/mel-core \
/home/os_user/mgc/embedded/omni/isar/meta-isar \
/home/os_user/mgc/embedded/omni/isar/meta \"
```

Because custom layers typically should have lower priority than default layers, they are listed before them.

5. Enter the following commands at the terminal command line to build the customized images:

```
bitbake development-image
bitbake core-image
```

Results

The built images include the packages defined in the `.bbappend` files and the `my-application.inc` include file.

Related Topics

[Components Customization](#)

Customize the Layers

You can create new metadata layers to extend or customize the existing capabilities of the OS.

Procedure

1. Create the directory for your layer.

System layers are prefixed with “mel-”; for layers you create, prefix them with “meta-”. For example, to create meta-my-application:

```
mkdir meta-my-application
```

2. Inside your new layer directory, create a `conf` directory, and inside it a `layer.conf` file.

It is recommended to copy an existing layer configuration file to your layer's *conf* directory, and then modify the file as needed instead of creating the layer from scratch. For example:

```
# We have a conf and classes directory, add to BBPATH
BBPATH .= ":{LAYERDIR}"

# We have recipes-* directories, add to BBFILES
BBFILES += "${LAYERDIR}/recipes-*/*/*.bb ${LAYERDIR}/recipes-*/*/
*.bbappend"

BBFILE_COLLECTIONS += "my-application"
BBFILE_PATTERN_my-application = "^${LAYERDIR}/"
BBFILE_PRIORITY_my-application = "6"
LAYERDEPENDS_my-application = "mel-core"

# This should only be incremented on significant changes that will
# cause compatibility issues with other layers
LAYERVERSION_my-application = "1"
LAYERDIR_my-application = "${LAYERDIR}"
```

3. Add the relevant layer content depending on the layer type.
 - a. For a machine layer, add the machine configuration in a *conf/machine/* file within the layer.
 - b. If the layer adds a distribution policy, add the distribution configuration in a *conf/distro/* file within the layer.
 - c. If the layer introduces new recipes, put the recipes you need in the *recipes-** subdirectories within the layer.

Results

Your layer is created and ready to use. After creating a new layer, you can add this layer to an existing project by modifying *conf/bblayers.conf* in your build configuration directory.

Customize an OS Package

The majority of the packages required to assemble an OS image exist as binaries. However, it may become necessary to build some of the packages from the source code with the metadata of the layers included into the build project. You can build new packages to ship simple files such as configuration files.

Procedure

1. Create a new directory for a new recipe in your layer.

```
cd meta-my-application/recipes-conf
mkdir my-application
```

2. Create a directory for the files shipped by the recipe.

```
cd my-application
mkdir files
```

3. Add any files you want to ship into the directory you created in the previous step.

For example, add any configuration files required for your application to the *files* directory, such as *my-application.conf*.

4. Create the recipe file in the new recipe directory.

In the examples for this procedure, the new recipe directory is *my-application*. The recipe file should end in “.bb”, for example, *conf-my-application.bb*. The file content should be similar to the following example:

```
# contents of meta-my-application/recipes-conf/my-application/
# conf-my-application.bb

DESCRIPTION = "configuration for my-application"
MAINTAINER = "Your name here <you@domain.com>"
DEBIAN_DEPENDS = "my-application"
SRC_URI = "file://my-application.conf"

inherit dpkg-raw

do_install() {
    install -m 755 -d ${D}/etc/my-application
    install -m 644 ${WORKDIR}/my-application.conf ${D}/etc/myapplication/
}
```

5. Add the recipe to the images by modifying *IMAGE_INSTALL_append* in the *core-image.bbappend* and *development-image.bbappend* files.

Add the name of the recipe (without the .bb suffix):

```
IMAGE_INSTALL_append = " \
conf-my-application \
"
```

6. Enter the following commands to build the configured package:

```
bitbake development-image
bitbake core-image
```

Results

You can build the newly created package using the BitBake command and configure my-application runtime.

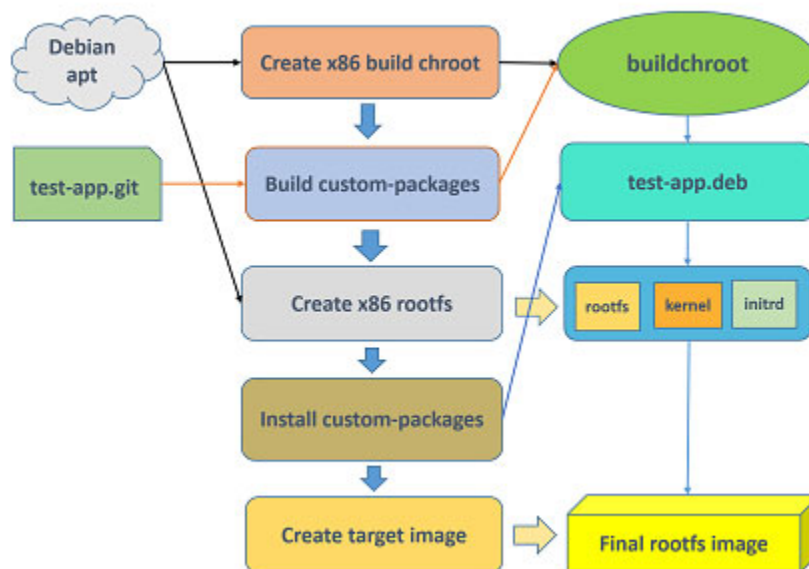
Chapter 4

Debian Packages Creation

It is recommended to install software using packages for Linux distributions that include a package manager. To customize Mentor Embedded Linux Omni OS, you may need to create your own packages. There are common Debian build tools that allow you to manage the packaging infrastructure.

The Debian packages that come with Mentor Embedded Linux Omni OS are managed as standard BitBake recipes. The recipe produces a *.deb* file that contains the compiled binaries for the Debian package. You can modify the rules in the source of the recipe to add specific features to your package. Isar uses BitBake to build and install the packages. The package structure, layering, and workflow are handled by the OpenEmbedded core framework.

Figure 4-1. Debian Packaging Overview



Package Build Files	28
Debian Package Classes.....	29
Debian Package Build Environment	30
Post Install Using Transient Packages.....	30
Build a Debian Package From the Source.....	31

Package Build Files

Debian packaging requires modifying specific files in the *debian* directory¹. The directory is either bundled with the main source file or created from scratch if it does not exist in the source tree.

Main Files

The *debian* directory must contain the following files to build a Debian package:

control

This file contains values required for package management by dpkg and apt.

Sample *control* file:

```
Source:          cpio
Section:         base
Priority:        optional
Standards-Version: 3.9.6
Build-Depends:  ${BUILD_PKG_DEPENDS}
Maintainer:     Mentor Embedded <embedded_support@mentor.com>

Package:        ${PN}
Section:        base
Priority:        optional
Architecture:   ${DISTRO_ARCH}
Depends:        ${shlibs:Depends}
Homepage:       ${HOMEPAGE}
Description:    ${DESCRIPTION}
```

rules

This is a makefile to define the tasks to execute. The *rules* file is different from the makefiles in the upstream source. Unlike other files in the *debian* directory, *rules* must be executable.

Sample *rules* file:

```
#!/usr/bin/make -f
# output every command that modifies files on the build system.
DH_VERBOSE = 1

%:
    dh $@
```

compact

This file defines the debhelper compatibility level. debhelper is a collection of programs that can be used in the *debian/rules* file to automate common tasks related to building binary Debian packages. For Mentor Embedded Linux Omni OS, set the debhelper level to 9.

1. Refer to the *Debian New Maintainers' Guide* for more details.

changelog

This file specifies package information such as version number, revision, distribution, and urgency.

Sample *changelog* file:

```
cpio (${PV}) stable; urgency=medium

* Initial release
```

You can find history for the Debian package releases in this file.

```
boo (1.1) stable; urgency=medium
* new upstream release

boo (1.0) stable; urgency=medium
* initial release
```

copyright

This file allows you to fill in the licensing details for your package. The file must match the format described in the machine-readable *debian/copyright* file. You can also pass the standard licenses to `dh_make` (which converts source archives into Debian package source) to create a template file in the appropriate format.

Other Files

You may need to update one or more of the following additional files under the *debian* directory, depending on your application.

- *dh_install* targets *.dirs, *.docs, *.manpages*
- *source/format*
- *patches/*

Allows you to modify the upstream sources.

Debian Package Classes

Isar provides three base classes for building packages. Additionally, Mentor Embedded Linux Omni OS provides the `dpkg-source` class.

dpkg

This base class builds a source package using the standard Debian source packaging, assuming the source is bundled with required *debian* directory files that are mentioned in “[Package Build Files](#)” on page 28. You do not need to make any additional changes to obtain *.deb* binary files.

Note



This is the recommended approach to build a source package, unless there are special requirements regarding the build and deploy options of the package files.

dpkg-base

This is the primary base class that is inherited by dpkg. It provides all the required functions to build a package (some can be customized, if needed). This can also be used in cases where you want to build a package with custom scripts and then use functions from the class to further deploy the generated *.deb* files in a local apt (mel-apt) repository.

dpkg-raw

This class is used to build non-Debian source packages that are simple to build and deploy without complex build tools. This class creates all the required *debian* directory build files with minimal information and rules, so you do not need to create *debian* directory build files in advance, unless your application requires them.

dpkg-source

This class is only used to build an upstream Debian package from source and make changes to the package before building it. You can identify a Debian package with a bug or a missing feature, then patch and rebuild the upstream Debian source in the local build-chroot environment.

Debian Package Build Environment

The chroot environment provides a minimal Debian rootfs with primary build tools, scripts, and essential package build dependencies.

You need to create a BitBake recipe that defines how to build the package. This is processed in the buildchroot environment. Building the source package in a chroot environment provides an environment based on the target system, regardless of the host configuration.

You can use the `DISTRO_ARCH` variable in your *conf/local.conf* file to specify the build architecture. The package manager in the buildchroot environment is dpkg. This ensures that the package functionalities on the target are similar to what is installed on the host system.

Post Install Using Transient Packages

A transient package is a *.deb* file that performs pre-configuration of the rootfs image and is prefixed with *configure* in the recipe file name.

Add the transient package to your layer by modifying the *layer.conf* file.

Example:

```
IMAGE_TRANSIENT_PACKAGES_append += " foo "
```

You can use the transient package to run a post install script for your programs.

Example:

```
inherit dpkg-raw

DESCRIPTION      = "Amend the system watchdog"
HOMEPAGE         = "https://mentor.com/"
LICENSE          = "MENTOR"
LIC_FILES_CHKSUM = "file://${LAYERDIR}/\
                    LICENSE;md5=2e95391d1963dd13edcc6647e7d054b3"

MAINTAINER       = "Mentor Embedded <embedded_support@mentor.com>"
PV               = "0.1"
SRC_URI          = "file://postinst.in"

do_install () {
    envsubst      \
        <${WORKDIR}/postinst.in \
        >${WORKDIR}/postinst
}
```

Build a Debian Package From the Source

Building a Debian package from the source is recommended if you need to extensively customize the package from its source before the installation instead of directly installing the binary package. Create a BitBake recipe to build a Debian package from the source which enables your to control all the options for the package as needed.

Procedure

1. Navigate to the *debian/* directory and modify the packaging details in the upstream program.
2. Create the non-native source package for your program.

A non-native source package contains the mandatory input files required to build a Debian package.

3. In the *debian/source/format* file, set the formate for the non-native source package to 3.0 (quilt).
4. Trigger the package build.

Example:

```
bitbake mel-cpio
```

Results

Building the source package produces a *.deb* binary file that you can install with the **dpkg** command or deploy into a rootfs with the **apt** command.

Examples

Example of a standard recipe file that builds upstream package in the Isar environment for the Copy in and out (cpio) file archiver utility and its associated file format:

```
DESCRIPTION      = "CPIO 2.12 for MEL"

LICENSE          = "GPLv2"

LIC_FILES_CHKSUM = "file://${LAYERDIR}/
LICENSE;md5=2e95391d1963dd13edcc6647e7d054b3"

MAINTAINER       = "Mentor Embedded <embedded_support@mentor.com>"

DEBIAN_DEPENDS   = "build-essential libc6"

PV              = "2.12"

SRC_URI          = "https://ftp.gnu.org/pub/gnu/cpio/cpio-${PV}.tar.bz2 \
${DEBIAN_FILES} \
"

SRC_URI[sha256sum] =
"70998c5816ace8407c8b101c9ba1ffd3ebbecba1f5031046893307580ec1296e"

DEBIAN_FILES = "file://debian_control file://debian_rules file://
debian_changelog file://debian_compat"

inherit dpkg

S = "${WORKDIR}/cpio-${PV}"

do_install_builddeps_prepend()
{
    DEBIAN_DIR=${S}/debian
    export BUILD_PKG_DEPENDS="$(echo ${DEBIAN_DEPENDS} | sed 's/
[[:blank:]]/, /g')"

    export PV="${PV}" PN="${PN}" DISTRO_ARCH="${DISTRO_ARCH}"
    HOMEPAGE="${HOMEPAGE}" DESCRIPTION="${DESCRIPTION}"

    install -m 0755 -d ${DEBIAN_DIR}
    install -m 0644 ${WORKDIR}/debian_compat ${DEBIAN_DIR}/compat
    install -m 0755 ${WORKDIR}/debian_rules ${DEBIAN_DIR}/rules
    envsubst <${WORKDIR}/debian_changelog >${DEBIAN_DIR}/changelog
    envsubst <${WORKDIR}/debian_control >${DEBIAN_DIR}/control
}
```


Example of the generated *.deb* file:

```
mel-cpio_2.12_amd64.deb  
mel-cpio-dbgsym_2.12_amd64.deb
```

Example of using the `dpkg` command to deploy the package:

Note



The runtime dependencies should be installed on the host prior to running the **dpkg** command.

```
dpkg -i mel-cpio_2.12_amd64.deb
```

Alternatively, you can deploy the package from the *conf/local.conf* file:

```
IMAGE_INSTALL_append = " mel-cpio"
```


Chapter 5

Application Build Environment

The application build environment consists of an Eclipse-based integrated development environment (IDE) named Sourcery CodeBench and an application development environment (ADE). The ADE acts as an archive, shipping headers and libraries to the target system.

Generate an ADE Image	35
Launch the IDE	36
Install the ADE on the IDE	36
Create Sample Applications	37
Build and Run Applications on the Target	38

Generate an ADE Image

The ADE image provides an archive that you can install into the Sourcery CodeBench IDE to build applications for the target system.

Prerequisites

- A build project has been created on the host system.

Procedure

1. On the host system, open a terminal and change to your build project directory.

```
cd <build-project-directory>
```

2. Run the *setup-environment* script.

```
. setup-environment
```

3. Enter the following command to create the ADE image:

```
bitbake development-ade
```

Results

You can find the generated ADE image under the *tmp/deploy/ade/* directory.

Related Topics

[Create a Project](#)

Launch the IDE

The Sourcery CodeBench IDE allows you to create applications that you can run on your target system.

Procedure

1. On your host system, open a terminal and change to the *CodeBench/bin* directory within the Mentor Embedded Linux Omni OS installation directory.

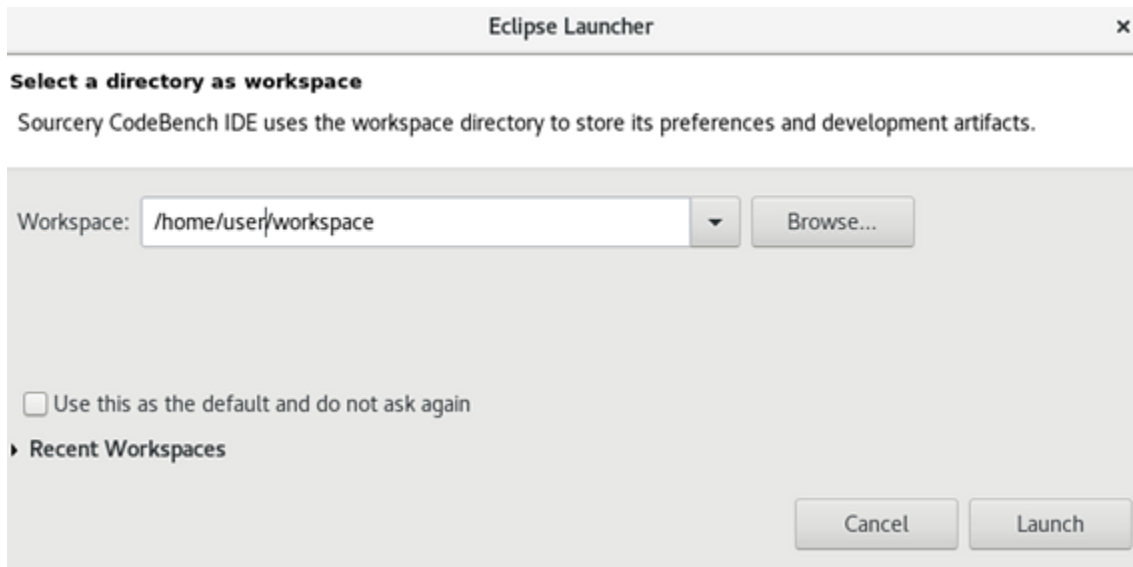
```
cd <install-dir>/codebench/bin
```

The default for *<install-dir>* is *mgc/embedded* in your home directory.

2. Invoke the Sourcery CodeBench IDE launcher.

```
./sourcerygxx-ide
```

3. Specify the workspace path in which you want to save your projects and click **Launch**.



Install the ADE on the IDE

The ADE installation on Sourcery CodeBench enables you to create Mentor Embedded Linux Omni OS applications that you can run on the target system.

Prerequisites

- [Generate an ADE Image](#).

Procedure

1. Open Sourcery CodeBench and go to **Help > Install New Software**.

2. In the Install window, click **Add**.
3. Click **Archive**, then navigate to the location where your ADE image resides.
4. Select the ADE image and click **Add**.

The window lists the archive contents.

5. Click **Select All** to install the entire archive.
6. Click **Next** then **Finish** to start the installation.

You can monitor the progress in the progress window of the IDE. When the installation completes, you will be prompted to restart the IDE.

7. Click **Restart Now** to apply the new changes.

Create Sample Applications

Use the Sourcery CodeBench IDE to create application projects for the target systems running Mentor Embedded Linux Omni OS.

Restrictions and Limitations

- This procedure assumes creating a C++ project.

Prerequisites

- [Install the ADE on the IDE](#).

Procedure

1. Open Sourcery CodeBench and go to **File > New > C++ Project**.
The C++ Project creation window displays.
2. Specify the Project name.
3. Under Project Type, expand the **Executable** category, select an example project, then click **Next**.

The project configuration window displays.

4. From the ADE dropdown list, select the ADE you have installed on Sourcery CodeBench.
5. Click the **New** button next to the Target connection dropdown list.
The connection settings window displays.
6. Select SSH Only, and then click **Next**.
7. In the Host name, provide the target IP address, then click **Next** and **Finish** to close the connection settings window.

8. In the project configuration window, click **Next**.
The Debug Settings window displays.
9. From the Toolchain dropdown list, select the proper toolchain for your target architecture.
10. Click the **Add** button next to the Launch target dropdown list.
The New Linux GDB Server Target window appears.
11. In the Host address, provide the target IP address.
12. Check the “This target allows remote access (e.g. via SSH)” check box to allow remote access.
13. Select your target IP address from the Connection dropdown list and click **Finish**.

Results

Sourcery CodeBench creates a new C++ project and loads the project into the Project Explorer panel.

Build and Run Applications on the Target

You can build and run applications on the target system through Sourcery CodeBench.

Prerequisites

- [Create Sample Applications](#)

Procedure

1. Open Sourcery CodeBench, right-click the project name in the Project Explorer panel, and click **Build Project**.
The project starts the build process. You can monitor the progress in the Sourcery CodeBench **Console** tab.
2. When the build completes successfully, right-click the project name and click **Run As > Run Configurations**.
The Run Configuration window displays.
3. In the Run Configurations window, click **Run**.
A new window displays and requests the User ID and Password.
4. Enter your User ID and Password for the target, and then click **OK**.

Results

The application runs on the target with the previously specified settings. You can monitor the output on the Sourcery CodeBench **Console** tab.

Chapter 6

Supported IoT Protocols

Mentor Embedded Linux Omni OS supports common Internet of Things (IoT) protocols that you can use to program your target system connectivity with other devices.

Protocols and Definitions	39
Building Images With IoT	40

Protocols and Definitions

Connecting a single device to a wide range of cloud systems requires a standard protocol to organize the communication between the connected devices.

The following IoT protocols are supported:

- **AMQP** — The Advanced Message Queuing Protocol (AMQP) is an application layer protocol that supports a wide variety of messaging applications and communication patterns.
- **CoAP** — The Constrained Application Protocol (CoAP) is a lightweight, application layer, web transfer protocol for constrained nodes and constrained networks. CoAP enables document transfer and provides smaller packets than standard network protocols (HTTP and TCP). Clients make requests to servers, and servers send back responses. Clients can perform GET, PUT, POST and DELETE operations on the resources.
- **MQTT** — Message Queuing Telemetry Transport (MQTT) is a lightweight publish-subscribe messaging protocol that runs on top of TCP/IP. It is an application layer protocol for small footprint devices where network bandwidth is constrained. The Publish-Subscribe messaging pattern requires a message broker. The broker distributes messages to specific clients based on the message identifier.
- **XMPP** — Extensible Messaging and Presence Protocol (XMPP) is an application layer protocol for instant messaging. XMPP is an open standard that enables the near-real-time exchange of structured data (in XML format) between devices. XMPP features, such as federation across domains, publish and subscribe, and authentication, are useful for IoT applications.

You can use the following with the XMPP protocol:

- **ejabberd** — This package provides a robust, massively scalable XMPP server.
- **finch** — This packages enables you to verify whether XMPP is installed or not. Finch is a console client that supports XMPP and additional protocols.

Protocols Mapping to Packages

The following table shows the IoT protocols mapping to the packages you can use in Mentor Embedded Linux Omni OS:

Table 6-1. IoT Protocols Mapping to Packages

Protocol	Package	Package Description
AMQP	rabbitmq-server	A widely used open-source message broker.
CoAP	libcoap	The C library for the CoAP protocol.
MQTT	mosquitto	MQTT broker for version 3.1 of the MQTT protocol
XMPP	ejabberd	The ejabberd package provides a robust, massively scalable XMPP server.

Building Images With IoT

You can enable IoT support for the standard Mentor Embedded Linux Omni OS images. This automatically adds the required IoT packages to the generated root file system image.

Prerequisites

- You may need to install the command line utilities for the protocols you want to use.
- You have created a project and set up the build environment. Refer to the Getting Started Guide for your target for the specific instructions.

Procedure

1. Navigate to the `conf/local.conf` file within your project directory and add the IoT protocols to the image features.

```
IMAGE_FEATURES += " amqp coap mqtt xmpp "
```

2. Build the image.

Only standard images (core-image or development image) are supported.

```
bitbake <target-image>
```

Results

The system generates the target image and includes the IoT packages for the protocols you have specified.

Related Topics

[Prepare the Host System](#)

[Create a Project](#)

Chapter 7

System Update Using SWUpdate

SWUpdate is a framework for secure system updates either from a storage media or over the network. For Mentor Embedded Linux Omni OS, the SWUpdate encrypted binaries are packed into an image that you can create and deploy into the target. During the installation of an SWUpdate image, the system decrypts the image binary packages and deploys them on the target.

System update through SWUpdate is compatible with secure boot and disk encryption. It is flexible enough to support installation from a minimal or recovery system, and it can also be used on a running system.

Create the SWUpdate Image	43
Overview of the sw-description File.....	44
Update the System From a USB.....	45
Update the System Over the Air.....	47
Install and Set up the hawkBit Server.	47
Update the System Using the hawkBit Server	48

Create the SWUpdate Image

SWUpdate images enable you to add the system update settings to the OS image before installing the OS on the target.

Prerequisites

- You have created a project directory and set up the build configuration. Consult the Getting Started Guide for your target for the specific instructions.

Procedure

1. Navigate to the *conf/local.conf* file within your build configuration directory and modify it by adding SWUpdate to the image features.

```
IMAGE_FEATURES += "swupdate"
```

2. Return to the build configuration directory and build the image.

The SWUpdate image name must end with -swu.

```
cd ..  
bitbake <target-image>-swu
```

Results

The required image is built and ready to be used on the target system. It is located with the other image files in the following location:

`<build_directory>/tmp/deploy/images/<target>/<type>-image-mel-omni-target.swu`

Overview of the sw-description File

The SWUpdate deployment properties are defined in a description file named *sw-description*, which is part of the SWUpdate image.

The following is the *sw-description* file structure:

```
software = {
    embedded-script = "
        require ("swupdate_handlers")
    ";
    version = "1.4.0"; hardware-compatibility: [ "v1.0" ]; stable: {
        platform1: {
            images: (
                {
                    filename = "...";
                    sha256 = "...";
                    type = "...";
                    hook = "...";
                }
            ); files: (
                {
                    filename = "...";
                    sha256 = "...";
                    compressed = ...;
                    encrypted = ...;
                    type = "...";
                    path = "...";
                },
                ...
            ); uboot: (
                {
                    name = "...";
                    value = "...";
                },
                ...
            );
        }; platform2: {
            ...
        };
    };
};
```

Any modifications to the *sw-description* file occur during the SWUpdate image customization. By default, the SWUpdate image uses Libconfig, a standard library, to parse the *sw-description* file. The SWUpdate image also supports adding extra update functionality using the Lua programming language.

When using Libconfig, note that all keywords are case-sensitive and that the file must be complete — `#include` is not allowed.

sw-description File Keywords

- **software** — The top level of the hierarchy. It contains all of the update details.
- **embedded-script** — A place holder in which you can add Lua handlers that you created for specific updates. This keyword can also contain parameters that are resolved at runtime.
- **version** — A string representing the version of the software update components.
- **hardware-compatibility** — A string representing the hardware versions that are compatible with the software update image.
- **stable platform1/platform2** — A group of software tools to invoke in case of update failure.
- **images** — The rootfs images to install during the system update. These provide a root file system.
- **hook** — The name of a function to call or a script to run when a specific entry is parsed. For example, Lua hooks can automatically add rootfs partitions to prevent system partitions hard-coding.
- **files** — Details about single files pertaining to the software update, such as *kernel*, *dtb*, and *initrd*.
- **uboot** — Details about the bootloader environment to update. The SWUpdate image supports multiple bootloaders such as u-boot and efibootguard.

Update the System From a USB

You can add the SWUpdate image to a USB storage device and use it to update the system.

Prerequisites

- Install Mentor Embedded Linux Omni OS on the target.
- [Create the SWUpdate Image](#).

Procedure

1. Prepare a USB device with enough space to store the SWUpdate image you created.
2. Format the USB using an appropriate file system.
3. Copy the SWUpdate image into the USB.
4. Plug the USB into the target.

5. Mount the USB to run the software update.

Use one of the following commands depending on the target, substituting “development” or “core” for *<type>*:

Note



The target device must allow root access in order to proceed with mounting the USB drive.

```
swupdate -H zynqmp-ultrazed-iocc:v1.0 \  
-f /etc/mel-os/swupdate/swupdate.cfg \  
-i /mnt/<type>-image-mel-omni-zynqmp-ultrazed-iocc.swu \  
-e stable,platform$(grep PLATFORM_ID /boot/swupdate.env | \  
cut -d "=" -f 2) -p "/usr/bin/swupdate-helper"
```

Or:

```
swupdate -H turbot-dual-e:v1.0 \  
-f /etc/mel-os/swupdate/swupdate.cfg \  
-i /mnt/<type>-image-mel-omni-turbot-dual-e.swu \  
-p "/usr/bin/swupdate-helper"
```

Or:

```
swupdate -H industrial-pc:v1.0 \  
-f /etc/mel-os/swupdate/swupdate.cfg \  
-i /mnt/<type>-image-mel-omni-industrial-pc.swu \  
-p "/usr/bin/swupdate-helper"
```

Update the System Over the Air

You can update the system through the network using the over-the-air (OTA) method. This requires that the system connects to the update server. Mentor Embedded Linux Omni OS supports system updates using the Eclipse hawkBit™ server.

The hawkBit server provides a web interface for updates. The server must be set up to allow the embedded devices to connect and retrieve software.

Install and Set up the hawkBit Server	47
Update the System Using the hawkBit Server.....	48

Install and Set up the hawkBit Server

Install and set up the hawkBit server on your development host.

Procedure

1. Install the hawkBit server using the instructions at <https://www.eclipse.org/hawkbit/gettingstarted/>.

If the host has sufficient space (generally not true when the host is a virtual machine), the most convenient way is to install the hawkBit server using a Docker container.

- a. Download a Docker CE version that is compatible with your development host.

<https://download.docker.com/linux>

- b. Install Docker CE.

For example:

```
sudo dpkg -i docker-ce_18.09.3~3-0~ubuntu-xenial_amd64.deb
```

Note



To run Docker CE without sudo, add your user account to the “docker” user group. You need to do this only one time per installation.

- c. Download and install the hawkBit server.

```
docker run -d -p 8080:8080 hawkbit/hawkbit-update-server
```

- d. Obtain the Docker container ID.

```
docker ps -a
```

Note



To run Docker CE without “sudo”, add your user account to the “docker” user group. You need to do this only one time per installation.

2. Use a web browser to connect to the hawkBit server. The URL is the development host's IP address, with the socket specified in Step 1.c if you used a Docker container.

For example:

```
http://134.86.61.4:8080/UI/login
```

Where 134.86.61.4 should be replaced with your development host IP address.

3. Log in to the hawkBit server.

The default user name and password is “admin”.

4. Click the **System Config** tab on the left panel.
5. Under Authentication Configuration, check the “Allow a gateway to authenticate and manage multiple targets through a gateway security token” check box.
6. Note down the 32-character gateway security token, as you will need to provide it to the embedded device.
7. Configure “Polling Time” with the time on which the server polls on the device ID to start the SWUpdate.

Update the System Using the hawkBit Server

You can use hawkBit to run network-based system updates. The hawkBit server provides a web interface for over-the-air system updates.

Prerequisites

- [Create the SWUpdate Image](#)
- [Install and Set up the hawkBit Server](#)

Note



Consult the hawkBit documentation for more details.

<https://www.eclipse.org/hawkbite>

- Obtain the IP address and the port for the hawkBit server.
- Use the following details when needed during the update:

Table 7-1. hawkBit Configuration Details

Attribute	Description	Default Value
tenant ID	The hawkBit user ID for the device.	default.

Table 7-1. hawkBit Configuration Details (cont.)

Attribute	Description	Default Value
device ID	The device ID to use for communication with the hawkBit server.	The device MAC address.
gatewayToken ID	A 32-character string you can get from the hawkBit server interface.	No default value.
hardwareRevision ID	The version that appears in the <machine-name>:<swupdate-hardware-compatibility-version>	The following are the default versions depending on your hardware: <ul style="list-style-type: none"> • turbot-dual-e:v1.0 • zynqmp-ultrazed-iocc:v1.0 • industrial-pc:v1.0

Procedure

1. On the development host, open a web browser and log in to the hawkBit server.
2. On the hawkBit server, click the **Distributions** tab on the left panel.
3. Create a distribution and a software module for your system.
4. Click the **Upload** tab on the left panel and attach your SWUpdate image to the recently created software module.

Make sure the attachment completes successfully before proceeding to the next step.

5. Click the **Distributions** tab again, then drag the software module that has the image attached and drop it on the distribution.
6. During the initial Mentor Embedded Linux Omni OS setup, configure the hawkBit server using the details shown in [Table 7-1](#) then reboot the target.
7. When the target reboots, check the swupdate-hawkbit.service status.

```
systemctl status swupdate-hawkbit.service
```

8. Go to the hawkBit server, and click the **Deployment** tab.
9. Under Targets, locate your target ID, and then drag the distribution you created above and drop it on the target ID.

Results

The hawkBit interface shows the update progress. Once the update is complete, your target will boot automatically.

To confirm the update on the target, use the following command:

```
cat /proc/cmdline
```

Because SWUpdate uses a dual-partition fallback technique for safely performing updates, the target system is now running on a different partition than before.

Chapter 8

Security Modules

A secure system environment requires a strong security module on top of any operating system. An OS without a security module relies on the kernel, all privileged applications, and user accounts to be correctly configured. An error in any of them can compromise the security of the whole system. A correctly configured security module can confine security breaches to just an individual application or user account.

The Linux kernel supports a variety of computer security models and modules. Mentor Embedded Linux Omni OS implements the Security-Enhanced Linux (SELinux) module for system security.

The SELinux module provides the tools to set up security domains for multiple concurrent users and automated services. Domains can be arranged as hierarchies, distinct categories, or a mix of the two.

Introduction to SELinux	51
SELinux Commands Overview	53
Extend the Security Policy	56
Recommendations for Security Policy Creation	59

Introduction to SELinux

SELinux uses the Linux Security Module (LSM) framework. SELinux provides tools to manage the system security policy and the associated security modules.

How SELinux Operates

SELinux implements mandatory access controls, which allow a system administrator to define how applications and users can access different resources (for example, networks or files). It can differentiate a user from the programs the user runs, and it also allows more fine-grained permissions on files, such as “append only.”

SELinux has two possible states in target images:

- Installed but not enabled; referred to as “permissive mode.”
- Installed and enabled; referred to as “enforcing mode.”

By default, when the SELinux feature is installed and enabled, the system denies all access attempts and logs them in a log file to be audited by the policy of the security module later.

The SELinux module augments the unmodified Linux security module. The security policy of the SELinux module must explicitly permit access before the standard permissions, such as file system read, write, and execute are considered.

When you install Mentor Embedded Linux Omni OS with the SELinux module, the SELinux default security policy runs in permissive mode. The permissive mode is a set of configurations that loads the security policy into the system, evaluates permissions, and logs all violating accesses without enforcing the policy. The permissive mode is ideal for initial system setup and configuration. However, it is not recommended for deployment scenarios because it does not enforce a strong security policy.

SELinux Security Policy

SELinux provides a component that enforces the generic security policy. This component loads the policy during the system boot-time to make sure the policy applies on a system-wide level, and it enables the auditing system to receive reports related to policy and security events.

The security policy is not part of the LSM; instead, it is one or more files describing the actions permitted on the system. If the action is not described and permitted in the security policy files, the system denies it and stores the details of the action in the log files.

Mentor Embedded Linux Omni OS Default Security Policy

Mentor Embedded Linux Omni OS provides a default security policy that is based on the SELinux project reference policy. The default policy provides configurations for general-purpose computing platforms that may have multiple concurrent interactive users or may provide services to local or remote clients, such as web or database servers.

Note



You can find the security policy files within your Mentor Embedded Linux Omni OS installation directory under the *etc/selinux* directory.

The default policy provides a user experience similar to UNIX. The administrator user (root) has root access (full permissions), other privileged users have a staff context, and regular users are in a third user context. Most system services execute in an unconfined context that is isolated from all other user contexts but otherwise allows the same system access they would have without SELinux in place.

Isolating the services from the user context increases the system security by ensuring even if the service obtains a higher privilege, it cannot obtain additional user access because the root, staff, and regular user contexts are isolated and independent of the system services. For example, if a service attempts to switch to user access such as root, this may cause the service to lose some of its current privileges because root users are not allowed within an unconfined domain.

When to Enforce Policy

Mentor Embedded Linux Omni OS installs SELinux in permissive mode, logging and allowing all access. This enables you to profile your system's typical behavior and tune the configuration policy. Do not switch to enforcing mode until you are confident that the security policy allows your system to operate as desired. It is possible to entirely lock yourself out of an enforcing system if the policy is overly restrictive.

SELinux Commands Overview

You can check the current state of the SELinux module to identify whether the module is installed on your system. Also, you can use the SELinux commands to get information about the installed features and the security related events.

SELinux Current State

You can identify whether SELinux is installed on an embedded device running Mentor Embedded Linux Omni OS using the following command:

```
grep selinux /etc/mel-os/features
```

Example:

```
root@omnios:~# grep selinux /etc/mel-os/features
selinux
```

The command result confirms that SELinux is installed.

SELinux Mode

Check the mode of SELinux using the following command:

```
grep selinux /proc/cmdline
```

Example:

```
root@omnios:~# grep selinux /proc/cmdline
root=/dev/mel/platforma None rootwait console=tty1
security=selinux
selinux=1 enforcing=0 splash plymouth.ignore-serial-consoles quiet
```

The command results show that SELinux operates in permissive mode (enforcing=0).

SELinux Installation Details

Show the details of the installed SELinux module using the **sestatus** command:

Example:

```
root@omnios-~# sestatus
SELinux status:                enabled
SELinuxfs mount:              /sys/fs/selinux
SELinux root directory:       /etc/selinux
Loaded policy name:            default
Current mode:                  permissive
Mode from config file:         permissive
Policy MLS status:             enabled
Policy deny_unknown status:    allowed
Max kernel policy version:    31
```

Security Events Log

View the system log for security events by using the **ausearch** command and specifying the Access Vector Cache (AVC) module. The AVC module is responsible for logging SELinux events.

Example:

```
root@omnios:~# ausearch -m avc -c ntp -ts recent | tail -1
type=AVC msg=audit(1556730609.091:428): avc: denied { getattr } for
pid=1462 comm="ntpd" path="/run/ntp.conf.dhcp" dev="tmpfs" ino=23595
scontext=system_u:system_r:ntpd_t:s0
tcontext=unconfined_u:object_r:dhcpc_var_run_t:s0 tclass=file
permissive=1
```

The command result shows that during the last 10 minutes, the system logged an attempt by the **ntp** command that should be denied if the current security policy is enforced. The attempt is logged and allowed because the system is in permissive mode. The log shows a denied “getattr” access on the */run/ntp.conf.dhcp* file.

Auditing Log Events

Check the root cause for the denied actions in the logs using the **audit2why** command.

Example:

```
root@omnios:~# ausearch -m avc -c ntp -ts today | tail -1 | audit2why
type=AVC msg=audit(1556730609.091:428): avc:  denied  { getattr } for
pid=1462 comm="ntpd" path="/run/ntp.conf.dhcp" dev="tmpfs" ino=23595
scontext=system_u:system_r:ntpd_t:s0
tcontext=unconfined_u:object_r:dhcpc_var_run_t:s0 tclass=file
permissive=1
```

Was caused by:

Missing type enforcement (TE) allow rule.

You can use audit2allow to generate a loadable module to allow this access.

The command result shows that the access was denied because there is no valid rule for the transition between the source context (system_u:system_r:ntpd_t) and target context (unconfined_u:object_r:dhcpc_var_run_t).

The suggested action in the example above is to create an additional policy rule that allows the ntpd daemon to execute successfully. If you run the **audit2allow** command, it generates a module fragment that you can load into the system.

Note



The **audit2allow** command is suitable for adding to an existing policy file, but the raw policy is usually not installed on an Omni OS platform, only the loadable binary policy, so it will be necessary to create a separate module for this new component by extending the security policy.

```
root@omnios:~# ausearch -m avc -c ntp -ts today | tail -1 | audit2allow

#===== ntpd_t =====
allow ntpd_t dhcpc_var_run_t:file getattr;
```

Permissive and Enforcing Modes

By default, the system operates in permissive mode until the system administrator changes it to enforcing mode. However, on the next system reboot, the system defaults back to permissive mode.


Use the **getenforce** and **setenforce** commands to detect the current mode and switch between the two modes.

Example:

```
root@turbot:~# getenforce
Permissive
root@turbot:~# setenforce 1
root@turbot:~# getenforce
Enforcing
```

Once the security policy is defined and complete, the system administrator can change the default mode settings to enforcing. To do this, edit the `/etc/selinux/config` file by setting the “SELINUX” variable to “enforcing”.

Caution

 Do not change to the enforcing mode until you make sure the security policy definition is complete and you understand how it works. Changing to the enforcing mode while the policy is incomplete can lock you out of the enforcing system.

You can also check and configure the policy parameters using the **getsebool** and **setsebool** commands. Any parameter modifications you make using the **setsebool** command remain unchanged even after the system reboot.

Example:

```
root@omnios~# getsebool -a | head -10
allow_cvs_read_shadow --> off
allow_daemons_dump_core --> off
allow_daemons_use_tty --> off
allow_execheap --> off
allow_execmem --> off
allow_execmod --> off
allow_execstack --> off
allow_ftpd_anon_write --> off
allow_ftpd_full_access --> off
allow_ftpd_use_cifs --> off
```

Related Topics

[Extend the Security Policy](#)

Extend the Security Policy

Mentor Embedded Linux Omni OS implements a default full-featured security policy to reduce the customization effort for the system administrator. The policy does not provide a broad collection of per-service modules but advanced users or system developers can extend it. They can also implement their own policy, if needed.

Restrictions and Limitations

- Developing a security policy from scratch is not covered in this manual. It can be complex. Most utilities lag behind the current SELinux module policy language if they are maintained at all. It is recommended to either work from the current SELinux Reference Policy release, or to extend the default security policy using the core tools provided.
- The default policy exists as a loadable binary file. Do not extend the policy using the **audit2allow** command directly because there is no raw policy file that you can edit.

Instead, create a separate module defining the policy update then load the new module into the system.

- It is recommended that you are fully aware of any changes you make to the security policy before loading them into the system. Be aware that some actions are performed only infrequently and may have not yet occurred, such as opening a socket to connect to a database.

Prerequisites

- You are on a device running a Mentor Embedded Linux Omni OS target image and any intended applications.
- You are logged in with administrator privileges.

Procedure

1. Create a separate module for the component you want to add to the policy using **audit2allow**.

Example:

```
root@omnios:~# ausearch -m avc -c ntp -ts today \  
| tail -1 | audit2allow -M omnios-ntpd-example
```

The system responds with:

```
***** IMPORTANT *****  
To make this policy package active, execute:  
semodule -i omnios-ntpd-example.pp  
root@omnios:~#
```

The above command uses **ausearch** to check the AVC for NTP access events in the past 24 hours, then sends the output to **audit2allow**.

The **audit2allow** command creates two files in the current directory:

- *omnios-ntpd-example.te*
- *omnios-ntpd-example.pp*

The *.te* file contains the update policy details and infrastructure, while the *.pp* file is the binary policy module that you can load into the existing SELinux module. A policy package file (*.pp*) can contain one or more modules.

2. Enter the **semodule -i** command to activate the created module.

The command may take time to complete. Also depending on your system, you may see some warning messages.

3. Enter the **semanage** command to load the activated module into the system.

Results

The new module is permanently added to the running security policy and keeps loading even after system reboot.

Examples

The *.te* file in this procedure looks similar to this:

```
module omnios-ntpd-example 1.0;
require {
    type ntpd_t;
    type dhcpc_var_run_t;
    class file getattr;
}

#===== ntpd_t =====
allow ntpd_t dhcpc_var_run_t:file getattr;
```

While the *.pp* file looks similar to this:

```
SE Linux modular policy version 1, 1 sections, mod
version 17, MLS, module name omnios-ntpd-example\003
```

Another method to allow the denied NTP access is to feed the *audit.log* file to the **audit2allow** command:

```
root@omnios:~# grep 'comm="ntpd"' /var/log/audit/audit.log \
| audit2allow > -M omnios-ntp
```

Examine everything in *omnios-ntp.te* and the related files and make any changes you want (for example, to allow additional access that is not present in the log but which you know will be required in the future) and then use the *checkmodule* and *semodule_package* utilities to compile the raw files into a binary *.pp* file:

```
root@omnios:~# checkmodule -M -m -o omnios-ntpd.mod omnios-ntpd.te
checkmodule: loading policy configuration from omnios-ntpd.te
checkmodule: policy configuration loaded
checkmodule: writing binary representation (version 17) to omnios-
ntpd.mod
root@omnios:~# semodule_package -o omnios-ntpd.pp -m omnios-ntpd.mod
```

Repeat the above steps for any command or utility that generates AVC denials until the system allows all of the activities required for proper operation.

Related Topics

[Recommendations for Security Policy Creation](#)

Recommendations for Security Policy Creation

Developing a security policy from scratch can range in complexity from relatively easy to very complex. This document does not provide any specific information about how to create a security policy from scratch. However, it lists some guidelines and recommendations to consider before creating your own security policy.

- Most of the available utilities that enable you to edit the SELinux module from scratch are not regularly maintained and may not be compatible with the SELinux module latest updates. Make sure you choose a utility that is compatible with your security module.
- Many examples suggest that you feed the *audit.log* file directly to the **audit2allow** command to make the policy accept all the denied attempts on your running system. This approach has two critical disadvantages:
 - It does not guarantee a complete security policy for the system during runtime. It only reflects the system state when the policy was created. The policy will deny actions that had not occurred at the time of policy creation, for example, opening a socket connection to a billing database.
 - If you do not make sure the policy files reflect proper security settings, the policy will allow all the events logged since the system boot without enforcing any restrictions. This may include harmful activities.
- It is not recommended to create a new policy with a new name to modify the behavior of an existing policy.
- When you upgrade SELinux, new modules replace the existing ones. Create a backup of your customized *.te*, *.fc*, and all related files and store them in a safe location. This enables you to make changes later then load them to the updated security module, if needed.
- When upgrading an existing policy to a new SELinux module version (for example, from 1.0 to 1.1), use the **semanage -r** command to remove the existing module before inserting the updated one.
- When you are compiling a new security module, remove the *.mod* files then run the **checkmodule** command to ensure that the new module is created.

Appendix A Examples

Perform common setup, customization, and OS tasks by following these examples.

Example: Creating a Build	61
Example: Adding a Debian Package Using local.conf	61
Example: Creating a Custom Layer	62
Example: Customizing the development-image Recipe	62
Example: Building With a Customized Layer	63
Example: Including Configuration Files and Scripts	63
Example: Using Post-Install Scripts	64
Example: Managing sudo Users	64
Example: Setting the root Password	65
Example: Enabling or Disabling systemd Services	65
Example: Creating a Read-Only root File System	65

Example: Creating a Build

This example demonstrates the command flow for creating a new project and build configuration.

```
# Create a new build project
. <install_dir>/omni/mel-core/setup-environment -b $HOME/build-pc \
<target-name>

# Build the development image
bitbake development-image

# Other targets include
# - core-image: "production image", root login not permitted
# - service-stick-image: USB installation image
```

Example: Adding a Debian Package Using local.conf

You can customize the build using the *local.conf* file.

```
# Change into your project directory
cd $HOME/build-pc

#Edit the local.conf file
gedit conf/local.conf

# Add the following line to the end of conf/local.conf:
IMAGE_PREINSTALL_append = " rt-tests "

# Build and run the image
bitbake development-image
```

Example: Creating a Custom Layer

The *local.conf* file enables quick and basic changes to an image. Adding configuration changes to a layer can increase efficiency because the changes can be reused in later operations.

```
# Change into your project directory
cd $HOME/build-pc

# Create directories for your layer
mkdir new-layer
mkdir new-layer/conf

# Create a new layer.conf files in new-layer/conf
```

The following is an example of the *layer.conf* file:

```
# There is a conf and classes directory.
# Add them to BBPATH
BBPATH .= ":{LAYERDIR}"

# There are recipes-* directories.
# Add them to BBFILES
BBFILES += "${LAYERDIR}/recipes-*/*/*.bb ${LAYERDIR}/recipes-*/*/*.bbappend"

BBFILE_COLLECTIONS += "new-layer"
BBFILE_PATTERN_new-layer = "^${LAYERDIR}/"
BBFILE_PRIORITY_new-layer = "6"
LAYERDEPENDS_new-layer = "mel-core"

# This should only be incremented on significant changes that will
# cause compatibility issues with other layers
LAYERVERSION_new-layer = "1"
```

Example: Customizing the development-image Recipe

A customized layer can augment recipes from OS layers through the use of *.bbappend* files.

```
# Change into your project directory
cd $HOME/build-pc

# Create the required directories to overlay the image recipes
mkdir -p new-layer/recipes-core/images

# create a new bbappend file in your layer directory. For example:
# ./new-layer/recipes-core/images/development-image.bbappend
```

The following is an example of the *development-image.bbappend* file:

```
# Add the following prebuilt packages to the development-image
IMAGE_PREINSTALL_append = " \
rt-tests \
"
```

Example: Building With a Customized Layer

The build process with customized layers is the same as the regular build process. The only difference is that you must include the layer as an argument to the *setup-environment* script.

```
# Create a build-pc directory for the <target-image> machine

sudo rm -rf $HOME/build-pc
. <install-dir>/omni/mel-core/setup-environment \
  -b $HOME/build-pc -l new-layer <target-name>

# Build and run the development image
bitbake development-image
```

Example: Including Configuration Files and Scripts

This example demonstrates how to add custom scripts to your image.

```
# Create a dpkg-raw recipe if you need to ship configuration/scripts or
# prebuilt binaries

# Create recipes-examples/hello-python/hello-python.bb
```

The following is an example of the *hello-python.bb* file:

```
DESCRIPTION = "a simple Python script"
DEBIAN_DEPENDS = "python3.5"

SRC_URI = "file://hello.py"
inherit dpkg-raw

do_install() {
install -m 755 -d ${D}/usr/bin
install -m 755 ${WORKDIR}/hello.py ${D}/usr/bin
}
```

```
# Create recipes-examples/hello-python/files/hello.py

#!/usr/bin/env python3.5
print("hello python!")

# Add hello-python to the image using IMAGE_INSTALL_append
```

Example: Using Post-Install Scripts

Packages built by the `dpkg-raw` class can run a script after they are installed. This is often used to activate systemd services or amend existing configuration files. This example demonstrates how to enable a systemd service.

```
# Create a file named recipes-examples/postinst-example/postinst-
# example.bb
```

where the *postinst-example.bb* file contains the following:

```
DESCRIPTION = "enable the xxx service"
DEBIAN_DEPENDS = "xxx" # make sure the service is installed
SRC_URI = "file://postinst"

inherit dpkg-raw

# Create recipes-examples/postinst-example/files/postinst
```

where the *postinst* file contains the following:

```
#!/bin/sh

systemctl enable xxx

# Add postinst-example to the image using IMAGE_INSTALL_append
```

Example: Managing sudo Users

You can add any number of users to the `sudo` group at build time.

```
# create recipes-conf/images/configure-development-image.bbappend
# (or recipes-conf/images/configure-%-image.bbappend to match all images
# where sudo is available)
```

The following is an example of the *configure-development-image.bbappend* file:

```
# Add the following users to the "sudo" group
SUDO_USERS_append = " \
mel \
"
```



```
# Build the image
bitbake development-image
```

Example: Setting the root Password

The root password may be changed at build time.

```
# amend recipes-conf/images/configure-development-image.bbappend
# (or recipes-conf/images/configure-%-image.bbappend)
```

The following is an example of the *.bbappend* file:

```
# Do not force a root password change, and set a custom password
ROOT_PASSWD_CHANGE = "0"
PASSWORDS_append = " root:new "

# Build the image
bitbake development-image
```

Example: Enabling or Disabling systemd Services

You can control which services are enabled or disabled at build time.

```
# amend recipes-conf/images/configure-development-image.bbappend
# (or recipes-conf/images/configure-%-image.bbappend)
```

The following is an example of the *.bbappend* file:

```
# Disable the "mel-setup" service
SYSTEMD_DISABLE_SERVICES_append = " \
mel-setup \

# Build the image
bitbake development-image
```

Example: Creating a Read-Only root File System

You can configure the root file system to be read-only. The Omni OS uses overlays to implement this property.

```
# create recipes-conf/images/configure-development-image.bbappend
```

The following is an example of the *.bbappend* file:

```
OVERLAYROOT_TMPFS = "1"
```

Example: Creating a Read-Only root File System

```
# create recipes-images/images/development-image.bbappend

IMAGE_PREINSTALL_append = " overlayroot"

# Build the image
bitbake development-image
```

Appendix B

Linux Commands

This section gives brief descriptions of the bash and Linux commands used in the examples. Most Linux installation offer more detailed reference information through man pages. To retrieve a man page for a command, enter “man <command>” (without quotation marks or brackets) at the shell prompt.

Note



All Linux, bash, other command-line keywords, and filenames are case-sensitive.

- *command1 | command2*

Directs the output of *command1* to the input of *command2*. “|” is read as “pipe.”

- *. script_name*

Run the script. Scripts are text files containing shell commands, and must have the permissions set to allow execution. “.” is a bash alias for the base Linux command “source”.

“.” has other significant uses in Linux shells:

- A filename that begins with “.”, such as *.bashrc*, is not shown by default when listing the contents of a directory.
- When part of directory path, such as “*./bin/*,” the period refers to the current directory. This can be useful when you are trying to refer to a local copy of a utility instead of one that is in your shell’s \$PATH. You can run applications by typing “*./application*”, without a space.

If you are entering a command such as “*. setup-environment*” and the shell responds with “file not found,” verify you have a space between the period and the filename.

- *bitbake*

This is a separate application and not a shell or Linux command.

- *bzip2 -c filename*

Compresses *filename* and streams the compressed form to the terminal.

- *cat filename*

Prints the contents of *filename* to the terminal window. “cat” refers to “concatenate” and is more typically used to send the contents of a file into another command.

- *cd directory_name*

Change directory. If specified without a directory name, you return to your home directory.

- `cp filename new_filename`

Creates a copy of the first file with the name specified. Although there are certain conventions regarding file suffixes, the OS does not use file extensions to determine type.

Note



For ease of future reference, restrict filenames to letters, numbers, period (.), hyphen (-), and underscore (_). You can use any other ASCII character, including whitespace, but will need to “escape” the characters to reference the file on the command line later.

- `dd if=input_file of=output_file ...`

Reads the input file, performs any specified conversions (a common one is block size), and writes it to the output file. Useful for transferring images from one hardware device to another.

- `dmesg`

Displays Kernel messages, such as recently connected devices. Requires root privilege.

- `dpkg -i filename`

Installs filename or returns an error if *filename* is not a Debian package. Typically *filename* ends with *.deb*.

- `export VARIABLE=value`

Set the environment variable *VARIABLE* to *value* and make it available for any applications called from this terminal window. (To make the settings permanent, add the line to your *.bashrc* file.) Environment variables can be referenced by prefacing with a \$, for example, \$PATH.

- `gedit filename`

A desktop editor that opens *filename*. This is an optional program and may not be on your host.

- `grep search-string`

Searches the input for search-string. This utility has many variants and dozens of command line options. Use ^ to match start of line.

- `losetup`

Sets up a loop device, a way of representing a file (typically an image) as a hardware device such as a USB drive or CD.

- `ls -l directory`

Lists the contents of the directory with details about permissions, owners, modification dates, and so on. To see the contents of the current directory without the details, type “ls” with no arguments. Use “ls -a” to include hidden files.
- `mkdir directory`

Creates a directory with the name you specify. You can create a set of nested directories with the -p switch.
- `mount`

Lists all mounted file systems.
- `partprobe /dev/file`

Reports the partition table changes in the system, such as device content modifications.
- `pwd`

Prints the absolute path of the directory you are working in. This command takes no arguments.
- `rm -rf directory`

Removes the directory and all its contents. Although it is not recommended, you can run it with sudo to avoid confirming all the deletions.
- `sudo command`

Execute *command* with administrator privileges.
- `sync`

Writes any pending changes to their intended destination; flushes the OS buffer.
- `tail`

Print the last ten lines of its input. Can be performed on a file to print the last ten lines of it. (This can be useful for checking a log to see if the application successfully completed, exited with an error, or is still running.)
- `tee filename`

Copies the input to a file while also allowing it to pass through. This is useful to capture transcripts from the terminal.
- `umount directory`

Unmount the directory.

board support package (BSP)

A set of configuration files allowing the Mentor Embedded Linux Omni OS tools to build for a particular type of CPU and board.

directory

The Linux term for what Microsoft Windows refers to as a folder.

host

The computer running the Mentor Embedded Linux Omni OS tools and Sourcery CodeBench ADE.

image

A binary file that contains the contents of a file system, including executable programs.

install_dir

The directory in which the *omni* folder resides. By default, this is *\$HOME/mgc/embedded*.

project_dir

The directory containing one or a set of related build configurations. This directory should be outside of the installation directory.

script

A text file containing shell commands, similar to a Windows *.bat* file. Also referred to as a shell script.

shell

A command line interpreter that most Linux systems run on top of the OS to provide utilities. The default shell in the Debian distribution, and the one used for all examples in this manual, is *bash*.

target

The device intended to run the compiled OS and application images.

End-User License Agreement with Embedded Software Supplement

Use of software (including any updates) and/or hardware is subject to the End-User License Agreement together with the Embedded Software Supplement Terms. You can view and print a copy of this agreement at:

mentor.com/embeddedeula

